Architectural Anti-Pattern Identification and Mitigation in Microservice Applications Based on Telemetry

Amund Lunke Røhne
amundlrohne@gmail.com

October 16, 2023, 44 pages

Academic supervisor: Dr. Benny Åkesson, k.b.akesson@uva.nl
Daily supervisor: Dr. Ben Pronk, ben.pronk@tno.nl
Host organisation/Research group: TNO-ESI, https://esi.nl/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Features of microservice architectures such as scalability, separation of concerns and their ability to facilitate rapid system evolution have made them popular with large organizations employing hundreds if not thousands of software developers. The same features that make them attractive also demand a non-trivial amount of complexity and maintenance over the evolution of an application. One of these complexities is the decomposition of microservice applications, which can have a direct effect on both the maintainability and performance of a given system. Microservice architecture decomposition evolves together with the application and is prone to errors referred to as architectural anti-patterns over its lifetime. Researchers have been successful in detecting simple anti-patterns through the use of both static and dynamic analysis of service dependency relationships. However, there have not been any attempts at creating formal mathematical definitions of more complex architectural anti-patterns that usually depend on developer domain knowledge. Furthermore, no work has been done to help developers in resolving these architectural anti-patterns. In this work, we introduce a new Granular Hardware Utilization-Based Service Dependency Graph (GHUBS) model based on multi-graphs that allow for the detection of complex architectural anti-patterns in an implementation-agnostic manner. We have created formal mathematical definitions of four architectural anti-patterns and devised algorithms to detect them in the GHUBS model. Lastly, we attempt to guide developers in their mitigation efforts by suggesting changes to their application architectures driven by hardware utilization metrics. All of this was implemented in a tool by the name of Televisor and validated in two case studies on well-known open-source microservice benchmarking applications, where we found 10 anti-pattern instances divided among two of our four formalized anti-patterns.

Contents

1 Introduction **4**
- 1.1 Problem Statement . . . 4
  - 1.1.1 Research Questions . . . 5
  - 1.1.2 Research Method . . . 5
- 1.2 Contributions . . . 5
- 1.3 Outline . . . 5

2 Background **7**
- 2.1 The Rationale Behind Microservices . . . 7
- 2.2 Microservice Challenges . . . 8
- 2.3 Observability in Microservice Applications . . . 8
  - 2.3.1 Traces and Distributed Tracing . . . 8
  - 2.3.2 Metrics . . . 9

3 Related Work **10**
- 3.1 Observability Models . . . 10
- 3.2 Anti-Pattern Detection . . . 12
  - 3.2.1 Static Analysis . . . 12
  - 3.2.2 Telemetry-Based Analysis . . . 12
  - 3.2.3 Anti-Pattern Definitions . . . 13
- 3.3 Microservice Decomposition . . . 13

4 Establishing a Basis for Automated Detection and Mitigation **14**
- 4.1 Goals and Motivation . . . 14
- 4.2 The GHUBS Model . . . 14
  - 4.2.1 Increasing Granularity . . . 14
  - 4.2.2 Adding Utilization Metrics . . . 16
  - 4.2.3 Formalizing the GHUBS Model . . . 17
  - 4.2.4 Generating the GHUBS Model . . . 17

5 Formalization and Detection of Architectural Anti-Patterns **18**
- 5.1 Inappropriate Intimacy . . . 18
- 5.2 Microservice Greedy . . . 19
- 5.3 Megaservice . . . 20
- 5.4 Cyclic Dependency . . . 21

6 Utilization Based Mitigation Recommendations **23**
- 6.1 Merging Services . . . 23
  - 6.1.1 Inappropriate Intimacy . . . 23
  - 6.1.2 Microservice Greedy . . . 24
  - 6.1.3 Performing a Merge in the GHUBS Model . . . 25
- 6.2 Splitting Services . . . 26
  - 6.2.1 Megaservice . . . 26
  - 6.2.2 Performing a Split in the GHUBS Model . . . 27
- 6.3 Removing Operations . . . 27
  - 6.3.1 Cyclic Dependency . . . 27
  - 6.3.2 Removing an Operation in the GHUBS Model . . . 28

7 Prototype and Validation **29**
7.1 Implementation . . . 29
7.1.1 Televisor Backend Module . . . 29
7.1.2 Televisor Frontend Module . . . 30
7.2 Case Studies . . . 30
7.2.1 Social Network . . . 30
7.2.2 Media Application . . . 32

8 Discussion **35**
8.1 The GHUBS Model . . . 35
8.2 The Anti-Pattern Detection and Formalization . . . 35
8.3 The Mitigation Suggestions . . . 36
8.4 Threats to Validity . . . 37
8.4.1 Telemetry Based Methodologies . . . 37
8.4.2 Predicting Hardware Utilization Metrics . . . 37
8.4.3 Coverage of Anti-Patterns . . . 37
8.4.4 The Applicability of the Tool . . . 37

9 Conclusion and Future Work **39**
9.1 Conclusion . . . 39
9.2 Future Work . . . 40

Bibliography **42**

Chapter 1

Introduction

Microservice application architectures have become more and more popular over recent years because of their unrivalled support for big organizations with large teams of developers, as opposed to traditional monolithic designs. Modelling organizations and software after business processes has become the norm, and microservice architecture excels at facilitating this kind of workflow [1]. However, while microservice applications provide these benefits, they also tend to grow very large and complex over the evolution of a system. The size and complexity of microservice applications make it difficult for developers to get a solid understanding of an application in its entirety. This lack of understanding is non-trivial, as it can be very detrimental to quality assurance processes, such as debugging, performance optimization and overall maintainability across multiple services and teams [2]. This is exasperated by the fact that microservice applications allow for rapid iteration and development of new software. The faster an application evolves, the faster it can become chaotic and difficult to maintain.

Ensuring that a system remains scalable, maintainable and performance-optimized at every step of the development process is crucial, as the costs associated with a project are directly dictated by the quality of the application. The more effort a team of developers has to dedicate to application maintenance, the less time they will spend developing new features and possible avenues of income. Furthermore, microservice architectures are already susceptible to high degrees of network overhead, which can be further exasperated by poor microservice application decomposition. Performance problems like this one and others stemming from bad decomposition lead to higher infrastructure costs and smaller margins for the businesses that deploy large-scale applications [3].

One of the overarching problems with decomposition in microservice applications is the reliance on individuals with domain knowledge about business processes to design a prudent architecture. This is also reflected in the literature on microservice architecture anti-patterns (also referred to as bad smells) [4, 5], where these anti-patterns in many cases are indicative of poor decomposition. While mostly reliant on domain knowledge for both detection and mitigation, the same literature also identifies anti-patterns that are strictly related to microservice dependency relationships. Examples of these are the Inappropriate Intimacy, Microservice Greedy, Megaservice and Cyclic Dependency anti-patterns. The existing work defines these in natural language, which leaves much up to interpretation and makes it difficult to identify them in microservice applications.

Furthermore, while observability tooling has become quite abundant in the industry, few solutions make use of it for the explicit purpose of improving architecture design based on telemetry gathered from production systems. Telemetry from observability tooling usually comes in the form of metrics, logs and traces. The work that does exist is primarily concerned with automatic service deployment schemes and dynamic resource allocation [6–8]. That is optimizing applications on the platform level, rather than the application level. Platform level refers to the hardware, server or servers that the application is running on. Additionally, the work that detects potential issues with architectural designs on the application level makes little to no effort to guide developers in mitigating these problems.

1.1 Problem Statement

As touched upon, efficiently managing the complexity of microservice systems during their lifetime is challenging. This is especially true for rapidly evolving systems involving several different programming languages and technologies. While there is research that addresses problems regarding optimal resource allocation and deployment, very little work addresses architectural anti-patterns of microservice applica-

tions in an implementation-agnostic manner. Both detection and mitigation of architectural anti-patterns are time and resource-consuming activities. As such, not having well-defined methodologies for how to perform such work is detrimental.

As we do not have methodologies for performing identification and mitigation of architectural anti-patterns, there is also a general lack of tooling. There are several tools for visualizing microservice applications, and other tools for detecting simple anti-patterns. However, they lack support for more complex anti-patterns such as Inappropriate Intimacy, Microservice Greedy, Megaservice and Cyclic Dependency and none of the tools support both visualization and detection, not to mention mitigation guidance [9]. Furthermore, the creation of such tooling has been hampered by informal architectural anti-pattern definitions [4], lack of consensus on universal microservice decomposition practises [2, 10] and insufficient observability models.

1.1.1 Research Questions

To tackle these issues, we have set out to answer the following questions:

RQ1 How can we utilize telemetry data to generate a model for automatically detecting instances of the four architectural anti-patterns, Inappropriate Intimacy, Microservice Greedy, Megaservice and Cyclic Dependency in microservice applications?

RQ2 How can we provide formal mathematical definitions of known microservice architectural anti-patterns, and apply them to create automated detection procedures?

RQ3 To what extent can we suggest mitigation techniques for solving the detected architectural anti-patterns, and determine the viability of said mitigation techniques based on hardware utilization metrics?

1.1.2 Research Method

The methodology and tooling in this thesis have been inspired by a literature review of existing work in the area of microservice performance optimization, architectural anti-pattern definition, detection and telemetry-based analysis. To validate our work we have performed a case study where we apply our methodology and tooling on two microservice applications from the open-source DeathStarBench benchmarking suite [11].

1.2 Contributions

In this thesis, we will be looking at how we can make formal mathematical definitions of these anti-patterns, detect them programmatically in a telemetry-driven model of our creation and finally give suggestions for mitigation techniques, based on utilization metrics. As we intend to alleviate the guesswork for developers it is all done in a fully visualized and highly configurable manner. This involves:

1. Creation of a new Granular Hardware Utilization-Based Service-Dependency Graph (GHUBS) model based on directed multi-graphs generated through the use of telemetry data.
2. Formal mathematical definitions of microservice architectural anti-patterns, and algorithms for detecting them in the GHUBS model.
3. Automated mitigation techniques for anti-pattern correction in microservice applications, and viability determination based on hardware utilization metrics.
4. Implementation of the proposed methods in a proof-of-concept tool called Televisor.

1.3 Outline

In Chapter 2, we describe the background of this thesis including concepts that will be needed to understand the topic matter. Chapter 3 describes related work and the gaps we wish to fill. Chapter 4 introduces a new GHUBS model for microservice architecture analysis. Chapter 5 goes into detail on how we formalize anti-patterns using directed multi-graphs, and how we utilize the formalization to create detection algorithms. Chapter 6 explains the process of taking the detected anti-patterns and creating appropriate mitigation suggestions based on the GHUBS model and the hardware utilization metrics contained within. Our case study, proof–of-concept tool Televisor implementation details and

the accompanying results are shown in Chapter 7 and discussed in Chapter 8. Finally, we present our concluding remarks in Chapter 9 together with future work.

Chapter 2

Background

Microservice applications are complex systems that require solid infrastructure and a non-trivial amount of tooling to operate efficiently. Here we will have a look at the rationale for their use, some of the associated challenges, and the concepts needed to understand how we can tackle this complexity. More importantly, we will need to know these concepts to understand the key contributions of this thesis.

2.1 The Rationale Behind Microservices

The microservice architecture arose from a need to decompose monolithic systems into smaller, more maintainable individual components or services. A monolithic architecture can be defined as: "a software application whose modules cannot be executed independently" [1], meaning that changes made to the system will affect the system in its entirety. The definition of the microservice architecture is: "a distributed application where all its modules are microservices", and the definition of a microservice is: "a cohesive, independent process interacting via messages" [1].

In comparison to the popular monolithic architectures of the past, microservices have many attributes that make them attractive to large-scale businesses and organizations. Smaller services offer flexibility, in the sense that individual components can be changed without affecting the system as a whole. Business processes today change rapidly, and employing techniques (such as the microservice architecture) to ensure that the digital side of a business can keep up with its market is critical. Furthermore, microservices are modular, meaning that the individual services contribute to the overall system behaviour rather than having a single system that offers full functionality. This modularity also makes them reusable, potentially opening the doors for more effective use of business resources. Perhaps more importantly, the microservice architecture facilitates the evolution of a system. Individual components can be altered without necessarily creating the need for compounding changes in other components. The microservice architecture increases and encourages maintainability if utilized correctly [1]. Organizations are also able to utilize several more programming languages than they would with most monolithic architectures. Introducing this kind of diversity into a system can be very beneficial as individual services can be purpose-built from top to bottom. For instance using higher level garbage collected languages for developer speed in some services, while using lower level languages with manual memory management for performance in others. Rather than depending on generic "jack-of-all-trades" solutions [2].

As mentioned, a monolithic architecture consists of modules which cannot be executed independently. This restriction introduces coordination challenges in big teams of hundreds of developers as the interdependencies grow stronger. As there is no separation of concerns, the areas of responsibility might become muddled between teams and productivity could plummet as a result due to integration errors, a need for reworks and miscommunication. With the microservice architecture, however, teams can be structured after business processes and the services they develop mirror these areas in a highly cohesive and lightly coupled manner [1]. This facilitates the creation of well-defined boundaries for team members to work within and is commonly referred to as Domain Driven Design [12]. The separation of concerns created by these boundaries mirrors the way Conway describes the development of systems in [13], and one might argue that leaning into the structure he observed is beneficial.

2.2 Microservice Challenges

It is important to note that while microservices bring several advantages and help with scaling in most businesses, they also increase complexity. The effectiveness of the microservice architecture is determined by the thoroughness of the decomposition activity at design time. Furthermore, the decomposition has to be maintained over the evolution of a microservice application to ensure that the benefits are not lost. From a technical perspective, a bad decomposition can lead to poor performance. This is due to the network communication overhead being far larger in distributed microservice architectures than in monolithic architectures with in-memory communication [1, 2, 14].

Problems stemming from a bad application decomposition have definitions and are formally known as anti-patterns or bad smells. Some anti-patterns are caused by aspects of a microservice application other than the decomposition, but these are not the focus of this thesis. Generally, architectural anti-patterns for microservice applications are defined in natural language [4, 5]. This makes them more approachable for developers and easier to discuss but also makes them more challenging to detect in actual systems. For instance, the *Wrong Cuts* anti-pattern has been given the definition: "Microservices should be split based on business *capabilities*, not on technical layers (presentation, business, data layers)" [5]. This anti-pattern describes the preferred decomposition of a microservice application. The approach of splitting microservices based on business functions suggests that a degree of domain knowledge is required to architect a solution. As *capabilities* vary significantly between businesses, a natural language description is somewhat appropriate. Other anti-patterns could arguably be a symptom of Wrong Cuts, for instance, the Inappropriate Intimacy smell: "The microservice keeps on connecting to private data from other services instead of dealing with its own data" [5]. That is, if the Inappropriate Intimacy smell is apparent, it might be a sign of a bigger architectural issue. It should be noted that the Inappropriate Intimacy smell is also described in natural language, but has a stricter definition that describes a relationship between individual microservices.

These challenges and others have prompted the industry to create many tools and solutions to facilitate and streamline the usage of microservices. Popular examples are Kubernetes for microservice orchestration, Jaeger [15] and Prometheus [16] for observability and telemetry gathering, and Docker [17] which has become a staple of containerization technologies. The rise of these technologies has also prompted a need for talented developers who are experienced with them, meaning that for a business to effectively employ a microservice architecture they need staff who can manage and develop such an environment.

2.3 Observability in Microservice Applications

We call the group of techniques and tools used in the process of gaining application insight, observability tools or techniques. Microservice architectures arguably need more of this kind of tooling than monolithic applications because of their inherent distributed nature [2]. Static analyses and end-to-end testing of microservice applications are usually not feasible because of their scale and diversity. As such, we employ tooling to monitor applications and gather telemetry at runtime. There are several different types of information we can derive from such tooling, four common ones being: traces, metrics, logs, and baggage. These four types, also referred to as Signals in the OpenTelemetry documentation [18], each represent different information about our application. We will be looking at traces and metrics.

2.3.1 Traces and Distributed Tracing

Traces describe a sequence of process execution in an application and is composed of several *spans*. A span is a collection of information about the process it describes. According to the OpenTelemetry definition [18], a span contains at a minimum: a trace ID, representing which trace it belongs to; a unique span ID, for identification; trace flags, such as start and end timestamps; and trace state, which can contain vendor-specific information. While spans by themselves can be useful, for instance for spotting latency bottle-necks in an application, they shine when we introduce distributed tracing. Distributed tracing allows us to see the causal connections between spans [19], and annotates every span with a link to their parent's span ID. With this new information, a trace can cover many microservices and we can use this to gain more insight about our system. The tools Jaeger and Zipkin [20] allow developers to query spans and traces and use them to troubleshoot their applications or develop tools to do it for them.

Figure 2.1 shows a trace from the Jaeger UI [15]. On the left-hand side, we can see the names of the services that have been invoked followed in black text, and to their right the name of the operation

that is being executed. The Gantt chart on the right-hand side shows the individual spans inside of this trace. All of the spans have a timestamp and a duration, making it possible to reason about temporal behaviour in the application. They are composed hierarchically and the indentation of the service names on the left-hand side describes the parent-child relationships between the spans. By looking at the black line through the spans we can see the system flow, and how the result is eventually returned to the first service. Note that this black line does not always take concurrent execution across multiple services into account.

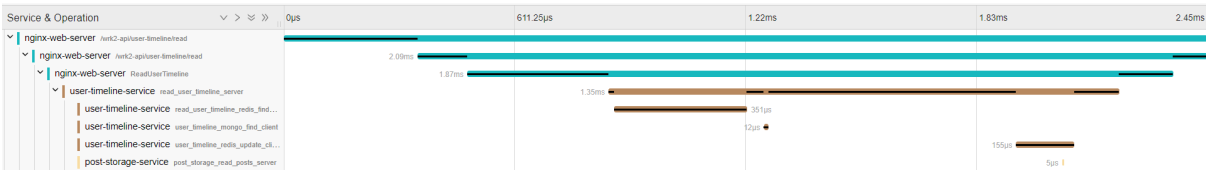


Figure 2.1: Trace from the Jaeger UI

2.3.2 Metrics

Metrics or more concretely *time-series metrics* is a "measurement of a service captured at runtime. The moment of capturing a measurement is known as a metric event, which consists not only of the measurement itself but also the time at which it was captured and associated metadata" [18]. There are many different kinds of time-series metrics in a microservice application. However, some of the more common metrics to gather are hardware utilization metrics. For instance, CPU, memory, network input and output usage. While the means of exposing this information is highly dependent on the application infrastructure, we can utilize tools such as Prometheus to store the data in time-series databases. Prometheus, being the most prominent time-series metric database, also provides a query language for retrieving and manipulating the data. Making it convenient for developers to create further tooling around.

Figure 2.2 shows the Prometheus UI [16] executing a query to collect the mean CPU utilization of several microservices over the course of an hour. The query field at the top of the figure contains the query that has been executed. The colourful lines in the graph are individual microservices, where the Y-axis shows the percentage mean CPU utilization and the X-axis shows timestamps. Even with a relatively small number of services and only the last five minutes displayed as in this graph, it is relatively difficult to decipher.

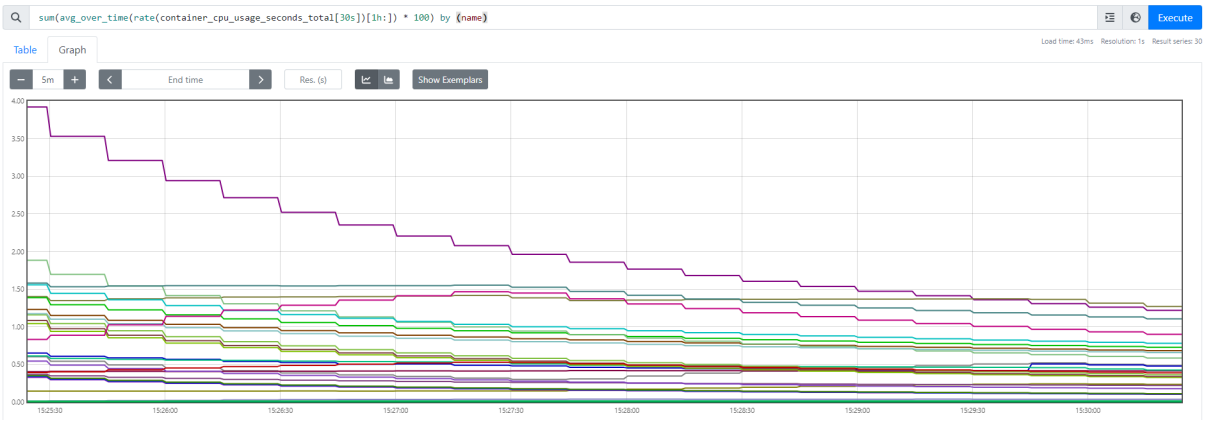


Figure 2.2: Graph Visualization of Prometheus Vector Metric

Chapter 3

Related Work

We divide the related work into the following categories: Observability Models, Anti-Pattern Detection and Microservice Decomposition. We intend to compare existing solutions and research to our own goals of creating a model, detecting and formalizing architectural anti-patterns and providing mitigation suggestions. This should illuminate the gaps in the current research and explain our positioning in the academic landscape.

3.1 Observability Models

To perform anti-pattern detection and mitigation of microservice applications we need data. While representative of a system, raw telemetry data is not particularly well suited to reason about architectural design. Therefore, we would like to base our work on a model that gives us the proper abstractions.

The Y-Chart Model [21] contains a separation of application, platform and a mapping between the two. This is similar to what we need, in the sense that we also need to know something about the application structure and the platform it is running on. Figure 3.1 shows the relationship between the elements in a Y-Chart and how insight from the analysis is used to improve the applications and platform [22]. However, the Y-Chart Model does not require that hardware utilization metrics be included. Furthermore, the mapping layer has a many-to-one relationship between the application and platform layer. We want to know the hardware utilization metrics on a per-microservice basis. The state of the platform is not needed and is usually more prominent in redeployment work. The GHUBS model in Chapter 4 addresses this by abstracting away the platform layer and only focusing on service-specific utilization metrics.

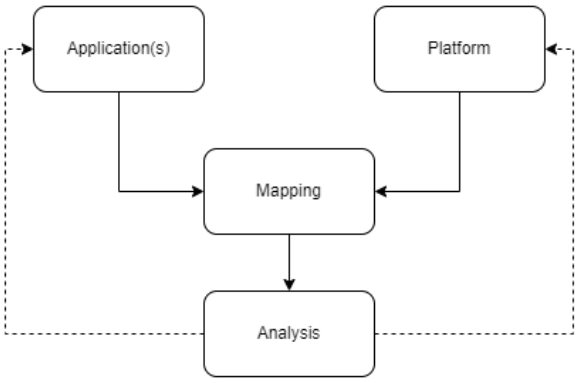


Figure 3.1: Y-Chart

Service Dependency-Graphs, also known as SDGs are a popular way of visualizing the topology of a given microservice application. SDGs have several features that will be beneficial for us in the detection of anti-patterns, however, they are somewhat lacking in granularity. One example of this is the lack of separation between application traces. SDGs are typically represented as a directed graph, including every single relationship and a count for the number of times they have been called on each edge. Figure 3.2 shows the SDG that is present in the Jaeger UI [15], visualizing service dependency relationships

and the number of times they have been invoked. Furthermore, SDGs prevent us from distinguishing individual spans or operations between microservices. Without this kind of information, the traditional SDG makes it difficult to say much about the relationships between individual microservices, making it more suitable for application-wide analysis.

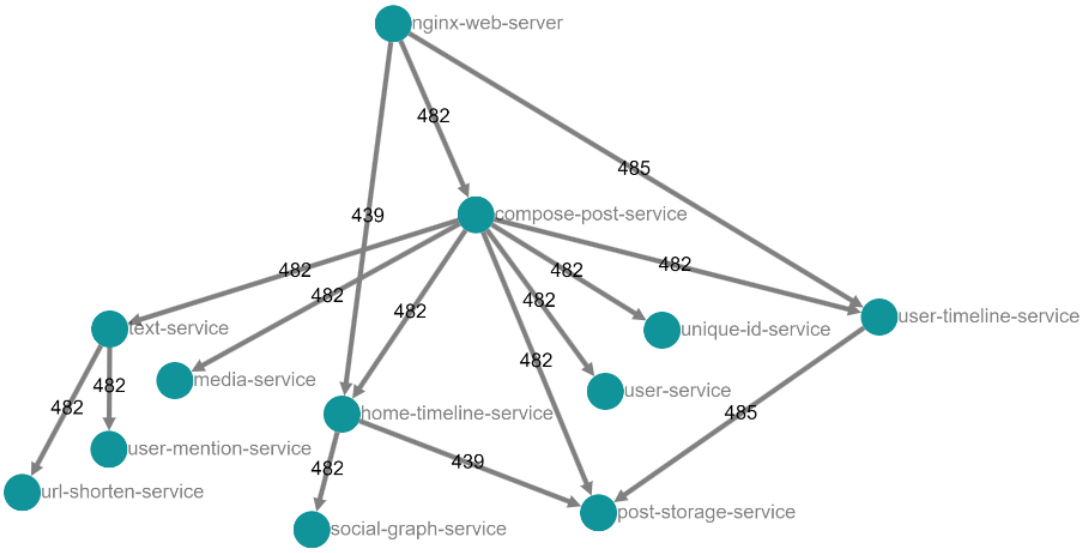


Figure 3.2: Jaeger SDG

Additionally, the traditional SDG does not visualize the separation of concerns on individual requests. While it can give us insight into the coupling in a microservice application [23, 24], it is not quite granular enough to say anything about what happens in the application on a per-client request basis. The lack of this perspective makes it so we cannot reason about system behaviour based on external requests and the services they invoke, it only offers application-wide composition. It should be noted, that Jaeger [15] provides Deep Dependency Graphs in their GUI that show an entire trace as an SDG branch. We see this view in Figure 3.3, showing the same trace as in Figure 2.1. This Deep Dependency Graph is successful in visualizing the dependency relationships between the services on this request. However, the anti-patterns we wish to detect and mitigate, require this abstraction in the context of the traditional SDG and is therefore not sufficient on its own. In Chapter 4 we go into detail on how to solve this problem.

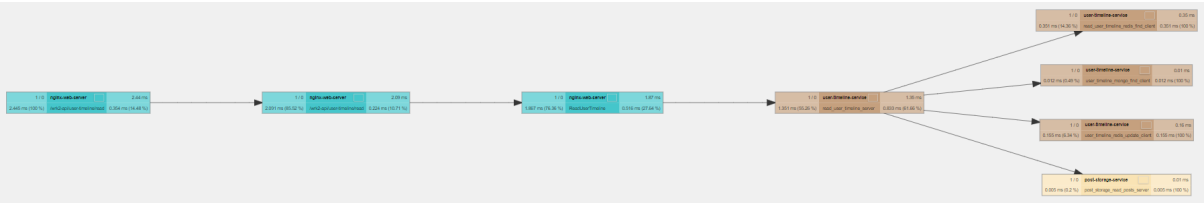


Figure 3.3: Jaeger Deep Dependency Graph

As the name suggests, SDGs give us information about the dependency relationships between microservices. This implies that there is not a whole lot of information about the actual microservices in the application. We wish to validate our mitigation suggestions and to do so will require some information about the state of the microservices, specifically hardware utilization metrics. While we do not believe that hardware utilization metrics should be a standard feature of the conceptual SDG, it will be necessary for our work. We will go into detail on how we validate our anti-pattern mitigation suggestions in Chapter 6.

Neither of the models includes anything about how they should be created, and leave the developers to their own devices to figure it out. This is another thing that is addressed by the GHUBS model in Chapter 4.

3.2 Anti-Pattern Detection

When looking into the existing literature on microservice anti-patterns and bad smells [4, 5], we see that while several of them describe structural patterns in microservice applications, they are all defined in natural language. There are also quite a few that deal with problems that can only be found on the microservice level in the code, which we will not address. The analysis approaches we are about to discuss attempt to find a wide range of anti-patterns, and several of them have interpreted the natural language anti-patterns to do so.

3.2.1 Static Analysis

There are several approaches for detecting anti-patterns in microservice applications. Approaches such as the one discussed in [25] analyses compiled Java projects to derive a service-dependency graph. While this approach is successful in creating a service-dependency graph outside of system runtime, it is limited to microservice applications written purely in Java.

There has also been research [26] on utilizing reflection, a feature in the Java programming language (and others) that "allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program" [27], to create SDGs. The researchers are primarily interested in using the SDG to calculate test coverage over entire microservice applications, however, they also detect cyclic dependencies between microservices. In [28], researchers also detect the cyclic dependency smell, however through extending an abstract syntax-tree-based tool created for detecting technical debt in monolithic Java applications. They were also able to detect instances of the Hardcoded Endpoints and Shared Persistence smells [4].

In [23] the authors calculate structural coupling in Java-based microservice applications by gathering coupling metrics found in the service classes. The researchers also chose to represent and visualize this in a directed graph, named a Structural Coupling graph. The microservices in the graph are coloured: green for services with a high degree of connections (hubs); yellow for services working as bridges between two or more services; blue for services with more outbound than inbound connections; and the rest coloured red. This graphical representation aids developers in spotting decomposition issues, which services are critical to the operation of the application, and which are not.

Another approach by Baresi et al. [10] utilizes monolith API interface definitions (OpenAPI schemas) to derive decomposition suggestions with adjustable levels of granularity when it comes to size and number. This is done by matching natural language identifiers on the API routes to an existing reference dictionary that groups related concepts. This creates cohesive microservice decompositions in about 80% of the cases they tested. The same methodology could arguably be used to solve the *Wrong Cuts* anti-pattern, if the correct interface definition files are present.

These static analysis methods are all limited in terms of their applicability. They are usually restricted to a single programming language, an architectural design pattern or some other technology. As such, they are not generally applicable. Despite these limitations, they do have the advantage in that they are usually more performant, and can guarantee complete coverage of the application in question. This is not always the case with our next topic, telemetry-based analysis.

3.2.2 Telemetry-Based Analysis

Telemetry-based approaches such as the ones in [29, 30], utilize distributed tracing built into microservice applications to generate a service-dependency graph at runtime. This approach is preferable as it allows us to analyze microservice applications regardless of the programming languages used in their creation. This dramatically increases the versatility of the tooling and methodology, which is something we would like to incorporate in our work. Especially in regard to our model creation.

Another way of dynamically creating an SDG at runtime is through service meshes such as Istio or Linkerd. In [24], an SDG is created based on service mesh logs, provided by telemetry tooling, containing information about inbound and outbound requests in order to determine dependency relationships. The researchers use the generated SDG in order to detect five anti-patterns, of which three are calculated using "anti-pattern metrics". The metrics are the number of in- and out degrees of each individual service and using them they calculate the following smells: Absolute Importance of the Service (AIS), Absolute Dependence of the Service (ADS) and Absolute Criticality of the Service (ACS). The severity of AIS, ADS, and ACS anti-patterns are relative to the application in question, and concern the total number of inbound, outbound and combined dependencies for a service, respectively. This in turn means that a system will always exhibit these anti-pattern, regardless of size, and leaves it up to the developer to

evaluate their severity as they are ranked in a descending fashion. These anti-patterns that depend purely on application metrics without any implication of their severity through limiting results with boundary values, make it difficult to tell if the microservice application actually is unhealthy. Furthermore, they encourage the developer to compare individual microservices relatively when analysing the data. The more homogenous the results are, the more difficult it will be for the developer to make effective decisions regarding the architecture of their system.

Because of the inherent complexity of microservice applications, there are many different optimization vectors to be considered. In the field of resource management and service deployment, runtime telemetry is heavily used to aid in and automate performance optimization in microservice applications [6–8]. Some solutions also use telemetry to evaluate the effectiveness of their findings or perform comparative analysis [31]. While our goal is not to perform such optimizations, we can see that they garner good results and accurate representations of the microservice applications in question. This increases our confidence in the use of telemetry for microservice observability.

3.2.3 Anti-Pattern Definitions

As we have alluded to several times by now, anti-patterns are generally defined in natural language. A notable exception is the Cyclic Dependency anti-pattern, which is familiar from graph theory and the ones that can be detected through static code analysis such as Hard-Coded Endpoints [4] which refers to a lack of service discovery in an application. This lack of formal definitions of anti-patterns allows for subjective interpretation from the researchers working in this field and is quite detrimental to the detection and mitigation process as a whole. It creates inconsistencies between different research addressing the same or adjacent anti-patterns, and while facilitating discussion, does little in the way of facilitating the creation of universal good practises across microservice applications. In Chapter 5 we will address how we can begin to resolve this.

3.3 Microservice Decomposition

Most work around microservice decomposition and prevention of the Wrong Cuts anti-pattern revolve around some sort of developer input, expertise or domain knowledge [12, 32]. For our work, we attempt to see if there are decomposition patterns or practices that can be proposed and validated without such input. We are willingly leaving organizational structure out of the decomposition equation, to realize system decompositions based on objective metrics that might not be readily apparent.

Chapter 4

Establishing a Basis for Automated Detection and Mitigation

Now that we have a better idea of the related work, we can proceed with the creation of a model we believe will facilitate in-depth anti-pattern analysis. Before we can come to a conclusion about what kind of data structure we will need, we first need to have a look at what we are attempting to do. Thereafter, explain how we address the problems standing in the way of our goals, and finally what the resulting model looks like and how it can be created.

4.1 Goals and Motivation

Our methodology consists of three distinct steps: 1) detecting architectural anti-patterns; 2) generating mitigation suggestions of said anti-patterns and 3) validation of the mitigation suggestions against metrics collected from the microservice application. Each of the steps requires a certain infrastructure to be set in place and a way of utilizing that infrastructure. We design our methodology on the basis that the microservice application in question is instrumented with distributed tracing as one of its infrastructure requirements. This is in order for us to extract the traces that will be necessary for the creation of our model, and ultimately steps 1) and 2). Furthermore, step 3), the validation step, will be based on microservice utilization metrics. This requires monitoring of the hardware the services are running on and knowledge of to which degree the services are utilizing it.

The detection and mitigation work is dependent on our ability to collect some information. Firstly, we need to know what services exist in our microservice application. Secondly, we need to know the relationships between those services. Finally, we need to be able to perform computations that allow us to detect specific patterns and rectify them in the application. SDGs, allow us to do these things to a certain degree while utilizing well-known graph-based techniques. However, they are not a silver bullet and lack the necessary abstractions that we need for all of the anti-patterns we wish to detect, mitigate and run validation on.

4.2 The GHUBS Model

In order to perform the computations we need, we have extended the traditional SDG with a few features to accommodate for the limitations discussed previously. This includes increasing granularity and mapping the microservices to their respective hardware utilization metrics. The result is the Granular Hardware Utilization Based SDG (GHUBS) model.

4.2.1 Increasing Granularity

In order to first tackle our granularity issue, we need to increase the number of edges that usually exist in an SDG. Instead of restricting the number of edges between two services to 1 per direction, we create an edge for each unique span between two services. The resulting graph will show not only which microservices are dependent on each other, but the actual individual operations that cause the dependencies. Essentially, we look at every span in the traces and create an edge between their source and destination service for every unique one that we come across. As opposed to creating a single edge

between services, regardless of the number of unique spans between them. We can tell if a span is unique by looking at its source and destination service, as well as its human-readable name and identifier that indicates its function.

This increase in granularity gives us more information to work with. However, there is still no separation of concerns and we cannot identify what actions cause the microservice application to operate the way it does. In other words, we need to be able to group the microservices and their operations by the requests they are triggered by. Similar to the same way that Jaeger can show us SDG branches of individual traces in their Deep Dependency Graph, but in the context of the rest of the GHUBS model. The traces we are primarily interested in, are the ones that we can relate to an action happening outside of the system (a request).

While the microservice application architecture does not provide any obvious ways for how this should be handled, there are best practices. The most common of which is the API Gateway architectural design pattern [33]. In fact, not having an API Gateway is considered an anti-pattern [4]. An API Gateway serves as a single point of entry for a microservice application. Translating external requests into internal operations on the communication protocols chosen for the microservice application. The API Gateway could for instance expose an HTTP REST API to the world, and translate incoming requests to gRPC [34] for use in the internals of the application. While the API Gateway pattern can have a few other features, they are not relevant to our intentions. We are interested in the fact that the API Gateway pattern establishes points of entry into the system that the internal execution depends on. This way we can group our traces to these points of entry, and thereby see the separation of concerns between individual traces in the system. In other words, we can link an external request to a set of microservices (a system flow) with their respective operations and distinguish them from each other, without removing the ability to see if the services in the group have uses outside of the current request. This facilitates the detection of architectural smells on a per-request basis as well as the traditional application-wide perspective, as we can easily move between the two.

Figures 4.1 and 4.2 show the difference in an example with three external requests coming into a microservice application. Figure 4.1 is the traditional SDG, where the three requests are grouped from the *API Gateway* to *Service A* and we can see the number of executions on each span. Figure 4.2 on the other hand, shows us that the three requests are unique and labelled *R1*, *R2* and *R3*. In fact, it can show us the trace in its entirety in the GHUBS and how it is distinct from the other traces.

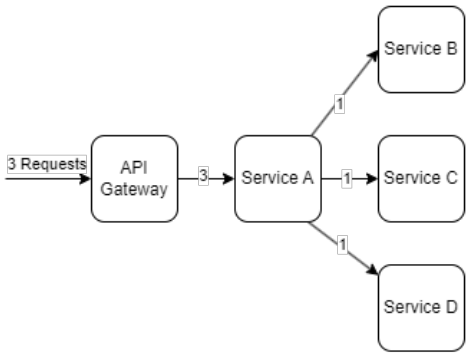


Figure 4.1: Traditional SDG Level of Granularity

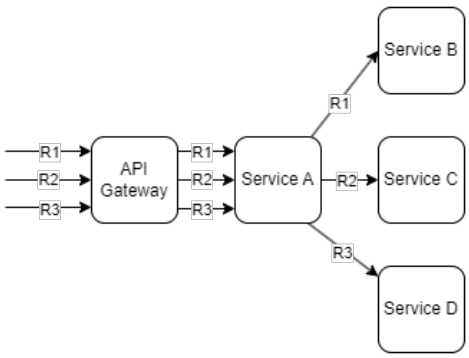


Figure 4.2: GHUBS Level of Granularity

The definition of the API Gateway architectural design pattern says that there should only be a single point of entry to the application. While our approach utilizes the existence of the pattern, it is not limited to applications that only have a single point of entry. Our approach does however require the application to limit external communication to a set of known services and keep the rest closed off from the greater internet. Fundamentally, what we are looking to find are external stimuli that lead to a process in the microservice application. Whether that stimuli are data coming from a sensor in an embedded application, or a user following someone in a social network application is irrelevant for the types of computations we wish to perform. With that being said, for this thesis, we are primarily looking at microservice applications with a single API Gateway for the sake of simplicity. With our current approach, examining multiple points of entry would require multiple iterations of the methodology and thereafter some manual analysis. However, extending the method to work with more points of entry would simply be a matter of stitching the resulting GHUBS models together.

4.2.2 Adding Utilization Metrics

We are using hardware utilization metrics to evaluate the impact and viability of our mitigation strategies. As we discussed about the Y-Chart in Chapter 3, there are a few different approaches for how to do this. We can measure the utilization metrics of the platform the microservices are running on; the utilization metrics of the cluster the microservices are contained within, or we can measure the utilization metrics on a per microservice basis. Following the theme of the GHUBS model so far, we wish to achieve a high degree of granularity for our utilization metrics as well. Meaning, that we wish to record the utilization metrics on a per-microservice basis. While this will be an abstraction of the hardware platform, it will serve as a more accurate representation of the state of the microservices in an application before and after a mitigation technique has been employed. As opposed to the hardware layer in the Y-Chart.

We wish to record six utilization metrics in the same time frame that we gather the traces for the construction of the SDG. The metrics are the 99.7% percentile, mean and standard deviation of the CPU and memory utilization. However, the GHUBS model is not limited to these metrics, and can in fact incorporate whatever metrics are most relevant to the work at hand. Some immediate examples could be network input and output for each microservice, span latency between microservices on a request, and end-to-end latency for an entire request. Capturing the 99.7% percentile, mean and standard deviation is not a given either. In our case, we want a pessimistic view of the system when performing validation, making 99.7% percentiles suitable. However, in use cases where metrics are collected over long stretches of time, it might not prove as useful. Going with a 100% percentile would give us the most pessimistic metrics, but this would most likely not be very representative of the system over time.

It is important that the metrics are gathered in the same timeframe as the traces, as in large applications there could be rarely used processes that would end up misrepresenting the application. There are two variables that come into play when capturing data from the microservice application. The first is the capture period, which is the timespan in which we poll the application for metrics. The second is the polling rate, which is the frequency at which we gather metrics in the capture period. For this work, we have utilized a capture period of 1 hour and a polling rate of 30 seconds. The capture period can be considered fairly short, and thereby frequent if executed consecutively. We do this for both traces and utilization metrics and wish for rare events to be captured in their entirety as they happen. Not as a blip on the radar. For instance, there could be irregular CRON jobs performing costly computations in the microservice application, significantly skewing the utilization metrics. The optimal capture period of telemetry gathering is likely to vary between different applications, because of examples such as this. However, the intention is to use a capture period that is granular enough to record such events in an encapsulated timeframe, while still capturing a realistic state of the application. A correct configuration should result in an accurate snapshot of the running system and is therefore suitable for the types of mitigation strategies we wish to apply. When it comes to the polling rate within the capture period, it is usually limited by the platform the tooling is run on. Generally, one can expect that the more frequently we poll the application, the more accurate utilization metrics can be derived.

Figure 4.3 shows the updated version of the GHUBS model. This version includes hardware utilization metrics mapped to each microservice. Note that the individual metrics (percentile, mean and standard deviation) have been abstracted away for better clarity.

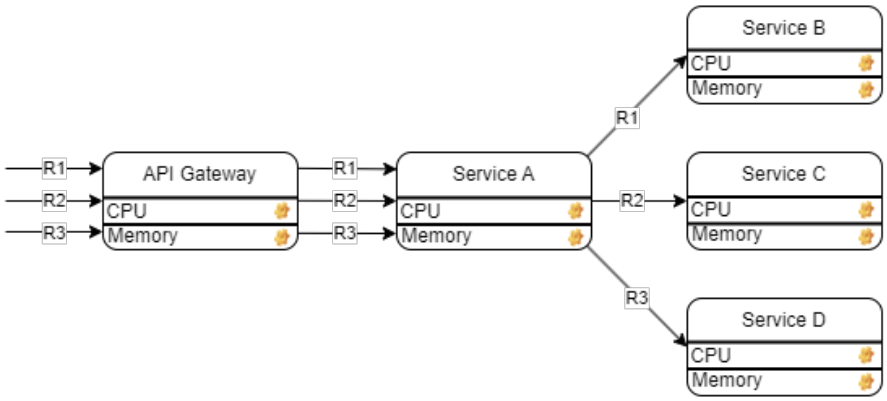


Figure 4.3: GHUBS With Hardware Utilization Metrics

4.2.3 Formalizing the GHUBS Model

As our GHUBS model is essentially a directed graph, we can take advantage of the established language for directed graphs to define our anti-patterns as well. That is, define our bad smells according to patterns of edges and vertices found in the GHUBS model. With our addition of potentially parallel edges to the traditional SDG, the proper formalism would be a directed multi-graph, more commonly found in the domain of networking. We let V be the set of all nodes in the directed multi-graph, where each node $v \in V$ represents a service. Let E be a set of all edges on a request, where each edge $u, v, r \in E$ represents a dependency from service u to service v, and r represents the redundancy count of the edge from u to v. Then we let $i, E \in R$, where i is an identifier for the request, and R is a set of all requests in the GHUBS model. The GHUBS model can be defined as a direct graph $G = (V, R)$, with nodes V and requests R. As opposed to multi-graphs used in the context of networking, the redundant edges in the GHUBS model do not serve as failovers. Meaning that they do not increase the resilience of the application, as the redundant edges serve unique functions. As such, they are not necessarily a boon for the microservice application in question.

4.2.4 Generating the GHUBS Model

As implied, the GHUBS model is generated based on traces and spans from a microservice application that has been configured with distributed tracing. Furthermore, it utilizes hardware utilization metrics in order to capture the state of the application and facilitate the validation of anti-pattern mitigation techniques. The hardware utilization metrics are stored in a lookup table, where the microservice names act as keys. This way we can perform queries against the table and retrieve the specific metrics we need. To create the granular SDG part of the GHUBS model we employ an algorithm similar to the simplified pseudo-code in Listing 4.1. Our function takes in the service name of the API Gateway service on line 1, and then we extract the names or identifiers of the requests on line 2. Line 3 defines R, the resulting dictionary of edges grouped by request identifiers. On line 5 we iterate over the request names and extract the trace from each of the requests on line 6. Line 7 converts the traces to edges, for use in our model in R. The edges are assigned to an identifier on R on line 8. Finally, we return the R in our GHUBS where $G = (V, R)$ on line 10. The set of vertices V or services can thereafter be derived from the same requests and composed together with R to make the GHUBS model G. After which, utilization metrics can be queried based on the service names and inserted into the aforementioned lookup table.

```
1  function createGHUBS(apiGatewayService):
2      requestNames = getAPIGatewayRequests(apiGetwayService)
3      R = {}
4
5      forEach(i of requestNames):
6          trace = getTraceOnRequest(i)
7          E = convertSpansToEdges(trace.spans)
8          R[i] = E
9
10     return R
```

Listing 4.1: Creation of GHUBS Model

Apart from deriving the GHUBS model from an existing system through telemetry, we can also create a mock GHUBS model. This is done by defining the microservices, their utilization metrics, the requests in the application, and the dependency relationships in those requests. Having the ability to do so is quite important as it gives developers the opportunity to evaluate their design before any development begins, or test a particular idea using artificial data. In fact, the testing and development of this methodology was done on a mock GHUBS model meant to inhibit the anti-patterns we have sought to detect. Before finally being validated on actual applications.

Chapter 5

Formalization and Detection of Architectural Anti-Patterns

Having defined our new fine-grained GHUBS model, we will proceed in this chapter by formalizing architectural anti-patterns and showing how they can be detected in the model with pseudo code. The four anti-patterns we will be looking at are *Inappropriate Intimacy*, *Megaservice*, *Cyclic Dependency*, and *Microservice Greedy*. Table 5.1 shows the definition of each smell as stated in [5].

Table 5.1: Selected Microservice Architectural Anti-Patterns

Microservice Anti-Pattern	Description
Inappropriate Intimacy	The microservice keeps on connecting to private data from other services instead of dealing with its own data.
Microservice Greedy	Teams tend to create new microservices for each feature, even when they are not needed. Common examples are microservices created to serve only one or two static HTML pages.
Megaservice	A service that does a lot of things. A monolith.
Cyclic Dependency	A cyclic chain of calls between microservices.

While these anti-pattern descriptions help us facilitate a common language to discuss them, they are rather vague and leave room for interpretation. This quality is detrimental to the creation of detection algorithms. In order for us to programmatically detect architectural bad smells we will need formal patterns which can be repeatably detected in multiple systems, using the telemetry data we have available to us.

It is important to note that a single anti-pattern definition can materialize in several different ways. This is due to the inherent vagueness of the natural language definitions. With that in mind, we have found four instances of the four anti-patterns according to our interpreted understanding of the source material [4, 5]. While we do believe that the formal definitions we have created are correct, they do not exclude other instances of the same anti-patterns from existing.

Furthermore, the set of anti-patterns described here should serve as a way of thinking about architectural smells in general. This means that by utilizing the GHUBS model formalism, developers can create more detection patterns and potentially develop bespoke patterns serving the needs of their applications. This idea makes the GHUBS model and associated anti-pattern detection methodology far more versatile.

With this in mind, we set forth to create formal definitions of each smell.

5.1 Inappropriate Intimacy

Following the definition in Table 5.1, we are looking for microservices that are dependent on each other or on a specific service in such a way that the separation of concerns between the services becomes blurred. However, the definition does not tell us anything about the number of microservices that are included. Neither does it define the interval of these connections or if it matters. The definition of what can be considered as private data of a microservice is also ambiguous.

The instance of the pattern that we seek to detect in this case is when we have multiple services dependent on a single service on the **same request**. If there are several microservices that backwards propagate this anti-pattern, we also want to know which microservices they are. The consequences of the

Inappropriate Intimacy anti-pattern are that it can introduce race conditions and unnecessary network communication. If the processes are synchronous, it could also cause a tail latency bottleneck. Examples of services with the Inappropriate Intimacy anti-pattern can be seen in Figure 5.1.

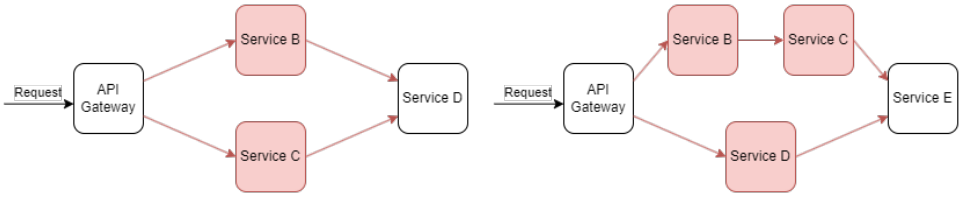


Figure 5.1: Instances of the Inappropriate Intimacy Anti-Pattern

In Figure 5.1 we can see two instances of the Inappropriate Intimacy anti-pattern we wish to detect. In formal terms, we are first looking for a node that has 2 or more inbound edges. We call this node, the *converging node*. If we find such a converging node, we want to detect all other nodes on the paths between the converging node, and a *diverging node* with two or more outbound edges. The number of inbound and outbound edges from the converging and diverging nodes does not have to be equal. If they are not, it is possible that there are multiple nested inappropriate intimacy smells. In which case the top level one will be considered. The nodes between the diverging and converging nodes are marked as responsible for the anti-pattern. In formal notation, the Inappropriate Intimacy smell is on a single request, $r, E \in R$, for some request identifier r. Where two or more sequences of nodes S contain $d, v_1, v_2, ..., v_k \in V$ such that $(v_i, v_{i+1}, 1) \in E$ for $i = 1, ..., k-1$ and $(v_k, c, 1) \in E$, where $d, c \in V$ and is the diverging and converging node respectively.

Listing 5.1 shows a simplified version of the algorithm used to detect instances of the Inappropriate Intimacy anti-pattern. The pseudo-code shows on line 1 the function definition and our parameter, the request in question. On lines 2 and 3, we find all converging and diverging nodes respectively. On line 5 we define our result, an array about to become nested. On lines 7 and 8 we iterate over the converging and diverging nodes, checking if there are paths between them on lines 9 and 10. If we find two or more paths between them, we add the paths to our result, excluding the diverging and converging nodes c and d. What we end up with is a nested array where each element of the outer array is an array of the edges responsible for the anti-pattern. We can use this nested array to retrieve the individual services included later.

```
1  function detectInnappropriateIntimacy(request):
2      convergingNodes = findConvergingNodes(request) // Nodes with >= 2 inbound edges
3      divergingNodes = findDivergentNodes(request) // Nodes with >= 2 outbound edges
4
5      result = []
6
7      forEach(c of convergingNodes):
8          forEach(d of divergingNodes):
9              S = getPathsBetweenNodes(c, d) // Returns nested array of edges
10             if(paths.length >= 2):
11                 result += [paths...] - [c,d] // Spread and remove c and d from the
                       result
12
13     return result
```

Listing 5.1: Detection of the Inappropriate Intimacy Anti-Pattern

5.2 Microservice Greedy

For the *Microservice Greedy* smell we are trying to find services that only serve a singular purpose, and might be redundant in the microservice application architecture. In [5], the example given is: "Teams tend to create new microservices for each feature, even when they are not needed. Common examples are microservices created to serve only one or two static HTML pages." However, as our data model does not give us insight into what the microservices are actually doing it does not make much of a difference for our detection. Such services are usually a result of poor planning or inconsiderate additions of new

functionality [5]. Whatever the case for introduction, they increase complexity and decrease application maintainability needlessly. An example of the anti-pattern can be seen in Figure 5.2.

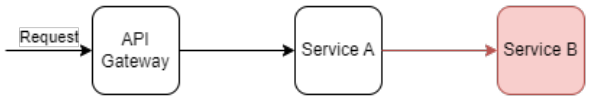


Figure 5.2: Instance of the Microservice Greedy Anti-Pattern

As opposed to our other detection methods, in the case of the *Microservice Greedy* anti-pattern we look at all requests aggregated. This is to prevent flagging microservices that are only responsible for a singular function in an individual request but have responsibilities in other requests. Every node that is detected as such will be marked as responsible for the anti-pattern. In terms of our GHUBS model, create a union of all edges in every request, $e_1 \cup e_2 \cup e_3 \cup ... \cup e_n = E$, where $i, e \in R$. We define a function $g : V \to Z^+$ that takes a node as input and returns the count of inbound connections to that node.

$$g(v) = |\{(u, v, r) \in E\}|$$

To check if a node has only a single inbound connection, evaluate $g(v)$. If $g(v) = 1$, then we can conclude that v has only a single inbound connection. In turn, v also has the *Microservice Greedy* anti-pattern.

Listing 5.2 shows the pseudo code for the Microservice Greedy detection algorithm. First, we define our function and take in a set of all the requests R and another set of all the services V as parameters. Thereafter, we take the union the requests and retrieve all of the edges on line 2. On line 3 we define a variable that will be an array containing the edges responsible for the anti-pattern. On lines 5 and 6 we iterate over all of the services $v \in V$ and return the edges that end in v. If we find that exactly one edge ends in v on line 7, we append that edge to our results, and finally return the result on line 10. Once again, we can derive the involved services from the responsible edges.

```
1  function detectMicroserviceGreedy(R, V):
2      E = unionRequestEdges(R)
3      result = []
4
5      forEach(v of V):
6          inboundEdges = g(v, E)
7          if (len(inboundEdges) == 1):
8              result += inboundEdges
9
10     return results
```

Listing 5.2: Detection of the Microservice Greedy Anti-Pattern

5.3 Megaservice

The Megaservice anti-pattern denotes services that have become monolithic in nature and thereby serve too many unrelated functions to be considered a microservice. The number of functions a microservice has to serve before it is considered a Megaservice is not clear following the definition stated in [5]. This instance of the anti-pattern shown in Figure 5.3 clearly shows the redundant execution between the "API Gateway" and "Service A". This implies that the microservice has several exclusive operations, that require this kind of dependency relationship even on a single request. This could for instance be a remnant of a legacy monolithic application that has been transitioned into a microservice application, where the old API is not conforming to the microservice best practices. The redundant execution in Figure 5.3 is meant to describe a **single request** that requires executing two operations, in order to complete the functionality of the new microservice application the monolith has been placed in. This kind of relationship can cause data races and other unwanted behaviour in the subsequent microservices. In the figure, we can imagine that the operation from *Service A* to *Service C* is dependent on all operations being executed on *Service A* first. If this does not happen, a data race can occur. Furthermore, this anti-pattern can potentially lead to a service becoming a single point of failure, or performance bottleneck in the overall microservice application.

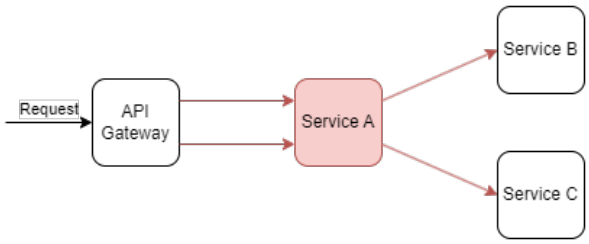


Figure 5.3: Instance of the Megaservice Anti-Pattern

Formally, the instance of the Megaservice anti-pattern we are looking to detect is when a node has two or more inbound edges coming from a second node. In order to check if we have such redundant edges, we defined a function $f : V \times V \to Z^+$ that takes in a pair of nodes and returns the redundancy count between them.

$$f(u, v) = \sum_{(u,v,r) \in E} r$$

Where $i, E \in R$ for some request identifier i. To check if there are two or more redundant edges between two services u and v, we evaluate $f(u, v)$. If $f(u, v) \geq 2$, then we can conclude that there are two or more redundant edges between u and v. Which in turn means that node v suffers from the Megaservice anti-pattern according to our definition.

Listing 5.3 shows us the pseudo-code for our Megaservice detection algorithm. On line 1 we find our function definition and the parameter, which is the request in question. We define our result variable on line 2, which is a nested array containing arrays with the responsible edges. Lines 4-6 iterate over all services and retrieve the edges we have between service u and v. If we find that we have more than two edges between them on line 7, we will add service v to our result array on line 8. Finally, the function returns the result array which will contain all edges, pointing to a service that inhibits the Megaservice anti-pattern. As with the Inappropriate Intimacy detection pseudo-code, we can derive the service in question based on this.

```
1 function detectMegaservice(request):
2     result = []
3
4     forEach(u of request.services)
5         forEach(v of request.services)
6             edges = f(u,v)
7             if (len(edges) >= 2):
8                 result += edges
9     return result
```

Listing 5.3: Detection of the Megaservice Anti-Pattern

5.4 Cyclic Dependency

The Cyclic Dependency anti-pattern is stated to be "a cyclic chain of calls between microservices" and can cause the microservices involved to be difficult to maintain or reuse in isolation [5]. The cycle could also cause perpetual loops of computation, although this is something we would expect most developers to notice before using a method such as this. This definition also leaves out the number of microservices up to interpretation, however, this is not a deciding factor for the existence of the smell. Figure 5.4 shows an example of the anti-pattern, with the responsible services and operations marked in red. Again, this is on a single external request.

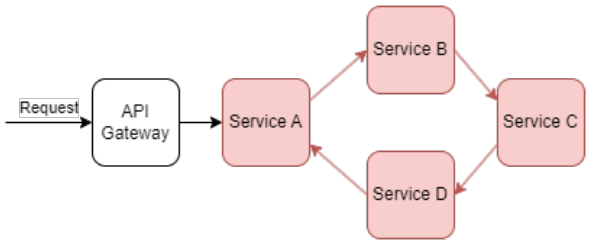


Figure 5.4: Instance of the Cyclic Dependency Anti-Pattern

Of the four smells we are formalizing the Cyclic Dependency smell is the one with the most well-defined pattern from the literature. A cycle in our SDG is a sequence of nodes S containing $v_1, v_2, ..., v_k$ such that $(v_i, v_{i+1}) \in E$ for $i = 1, ..., k-1$ and $(v_k, v_1) \in E$. Where $r, E \in R$, for some request identifier r. Note that the elements of E have been reduced to a tuple of two elements, as the number of redundant edges is irrelevant.

This method allows us to detect cycles with an unlimited amount of microservices in them. Nested cycles will also be detected, however, they will be identified as several different instances of the smell in the microservice application.

Listing 5.4 shows the pseudo code for the Cyclic Dependency detection algorithm. We start with our function definition, taking a parameter containing the request in question on line 1. Thereafter, we define a variable to hold our array of results on line 2. On line 4 we iterate over the services in the request and find cycles that start and end in service v on line 5. If we have found a non-nil cycle on line 6, we add it to our results on line 7. Finally, we return the resulting edges on line 9.

```
1 function detectCyclicDependency(request):
2     result = []
3
4     forEach(v of request.services):
5         S = findCycleEndingInService(v, request.edges)
6         if(len(S) != 0):
7             result += S
8
9     return result
```

Listing 5.4: Detection of the Cyclic Dependency Anti-Pattern

Chapter 6

Utilization Based Mitigation Recommendations

Now that we have formal definitions for how to detect the chosen anti-patterns, we would like to use this information to suggest mitigation techniques that will resolve the anti-patterns.

Our approach for doing this will be through altering operation paths and merging or splitting services. However, while we could simply suggest alterations based on the formal definitions of the anti-patterns, we want the recommendations to reflect the state of the actual microservice application. In order to achieve this, we utilize the hardware utilization metrics that we introduced in the GHUBS model. We use the term state here as the metrics one may choose to collect could consist of many things in addition to hardware utilization metrics, although these additional metrics will not be our focus here. Recall that we pay special attention to the frequency of our metric capture in order to prevent edge cases from skewing our results.

When we execute recommendations on the granular SDG, we need to be careful that we do not create new smells as a result of the resolutions. Furthermore, we also have to be attentive to the data paths in the microservice application. That is, we do not wish to sever communication paths in the microservice application that would prevent a particular microservice access to the data it depends on. After executing a recommendation we also need to evaluate the application again and ensure that additional anti-patterns are not introduced.

6.1 Merging Services

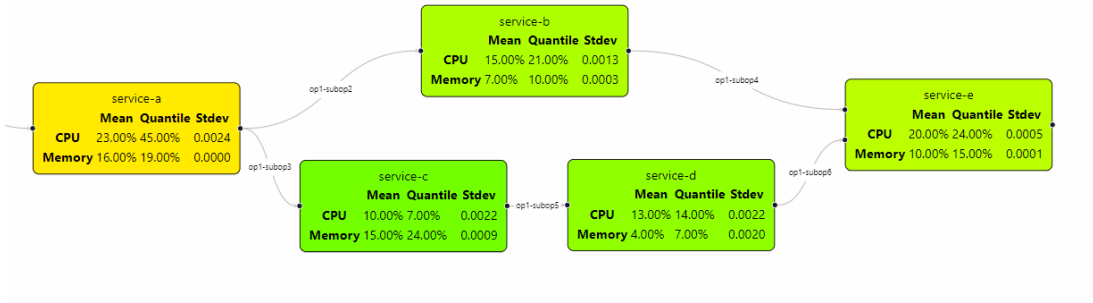
The first mitigation technique we utilize is the merging of two or more services in the GHUBS model. When merging services we are intentionally absolving separation of concerns in order to create bigger services. It is important we make sure that we do not create services that become too big or monolithic.

6.1.1 Inappropriate Intimacy

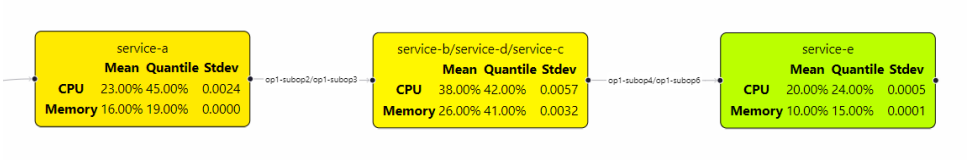
For the Inappropriate Intimacy anti-pattern there are two mitigation options with regards to our formalized smell pattern. The first option would be to split the data service that our responsible services converge upon and divide the resulting services among the responsible services. However, as we do not have any knowledge of what data that service is responsible for, or if it even can be logically split, this would not yield a useful suggestion. Furthermore, if we make the assumption that the designers of the microservice application have divided their data boundaries on the principles of Domain Driven Design [12], the data service is most likely only responsible for a collection of data that logically belongs together and thereby should not be split. Assuming that a system is properly decomposed while still inhibiting anti-patterns such as this is somewhat contradictory. However, while a system might have had proper data boundaries at its conception, it could have deteriorated over its evolution. Ultimately, we cannot see what the data boundaries are, and as such we cannot provide guidance on such a split. This leads us to the alternative solution to resolve the Inappropriate Intimacy smell. By merging the services that are responsible for the smell, we will be able to resolve the smell and still maintain data accessibility for the services involved.

However, we cannot simply merge services together. We also need to account for the performance

implications such an action could induce. To do this, we attempt to create a pessimistic scenario, by summing the 99.7% percentile (quantile) CPU utilization metrics of the services involved. This means, that if we have three services that are marked for merging with 10%, 15%, and 20% quantile CPU utilization respectively, the resulting merged service would have a quantile CPU utilization of 45%. To see if the resulting merge is acceptable, we compare the utilization metrics to a single or a set of boundary requirements specified by the developer and reject or accept the recommendation based on this. A developer could, for instance, decide that the limit for their application is a quantile CPU utilization below 60%, in order to make it easier to reason about the temporal behaviour of the microservices. The resulting mitigation technique can be seen in Figure 6.1a and 6.1b, showing the before and after respectively. Service A, B and C have been merged, as well as the relevant operations.



(a) Before



(b) After

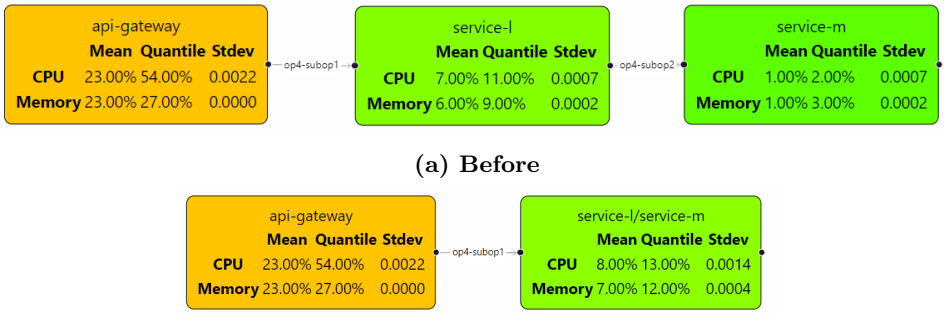
Figure 6.1: Mitigating the Inappropriate Intimacy Anti-Pattern

Merging the services in this way helps us consolidate the microservices into a cohesive unit and should increase maintainability and reduce coupling as a result. However, there are some cases where this might not be true. If the number of microservices is exceedingly large, it might be more feasible to merge them into several microservices instead of one, creating a pipeline pattern. This could be the better choice even if the number of services is small but the utilization metrics are large. To arrive at how to compose the new merged services, the developer should look to consolidate small services with highly dependent workloads. For instance, one service could cause a significant amount of latency because of the network overhead alone. Other services might be halted waiting for a callback that could be avoided with better composition. Alternative options involve following practices from Domain Driven Design [12], or a dictionary reference method such as the one in [10]. Furthermore, if the microservices serve as unique components of parallel execution, merging them could simply degrade the performance of the application. However, if they do in fact serve such a purpose we believe it to be prudent to merge the services into a cohesive unit and rather scale the number of instances of that microservice. This way maintaining low coupling while retaining the performance-oriented architecture.

6.1.2 Microservice Greedy

The second smell where we potentially want to merge services together is the Microservice Greedy smell. If we detect a service that has this smell, we wish to merge the greedy service with its parent service as the greedy service is most likely redundant. However, we have to account for the utilization metrics of both services in question. The service that has been annotated as greedy could previously have been the result of a split, creating the current structure, because it had a particularly high workload. Therefore, in the same manner, as with the Inappropriate Intimacy smell, we check if the summed quantile utilization metrics of the two services under question exceed the requirements set by the developer. If they do, the merge will not be recommended. Furthermore, we also make sure that the parent service is not the API

Gateway, as we do not want to introduce business logic there. Figure 6.2a and 6.2b show the merging of the "Greedy Microservice" with its parent microservice.



(a) Before

(b) After

Figure 6.2: Mitigating the Microservice Greedy Anti-Pattern

As we cannot say anything about the operations happening in a particular microservice, this anti-pattern is the one that is most susceptible to wrongful mitigation suggestions. Services that are not highly coupled could be chosen. In order to combat this, we suggest that the utilization metric requirements for a merger to take place should be very strict. In Figure 6.2b we can see that the merged microservice L/M only has a 99.7% percentile CPU utilization of 13% which is comparatively low to the rest of the microservices in the application. Despite this weakness, the detection pattern and mitigation technique are successful in making the developer aware of potentially long branches of unnecessary microservices in the application. It should be noted, that a parent service might have several greedy services as its children. In this case, we will consider the summed utilization metrics across all of the microservices involved.

6.1.3 Performing a Merge in the GHUBS Model

When performing a merge in the GHUBS model we need to create a new service, composed of the merged services and update the edges that used to refer to the merged services. Listing 6.1 shows the steps we take to merge services in simplified pseudo-code. We define our function on line 1 and take the responsible (bad) services and edges as parameters. Thereafter, we continue by creating a new service and then assigning it a name derived from the service to be merged, as well as utilization metrics summed from the same services on lines 2-4. Then on line 6, we iterate over the bad edges. For each edge, we check if it connects with a service outside the set of bad services. We do this for both the source and destination service on lines 7-9 and 11-13 respectively. If it has a source outside of the bad services we add the source to the new service. If the destination is outside, we add it as a dependency to the new service. We also make sure to point the edges to the new service. Finally, we remove all edges that refer to one of the now merged services as well as duplicates on lines 15 and 16. We also remove duplicates from the dependents and dependencies of the new service on lines 18 and 19. The function finally returns the new service and a new set of edges on line 21. The old ones can be removed, and replaced with the new ones. Note, that we also have to adjust healthy edges that used to have a dependency relationship with our bad services to now point to or from our new merged service. If not we might end up with a graph that is not strongly connected.

```
1  function mergeMicroservices(badServices, badEdges):
2      mergedService = new Service()
3      mergedService.utils = sumServiceUtilizationMetrics(badServices)
4      mergedService.name = joinServiceNames(badServices)
5
6      forEach(e of badEdges):
7          if(!badServices.contains(e.from)):
8              mergedService.dependents += e.from
9              e.to = mergedService.name
10
11         if(!badServices.contains(e.to)):
12             mergedService.dependencies += e.to
13             e.from = mergedService.name
```

```
14
15      updatedEdges = removeEdgesReferringToServices(badEdges, badServices)
16      updatedEdges = removeDuplicates(updatedEdges)
17
18      mergedService.dependents = removeDuplicates(mergedService.dependents)
19      mergedService.dependencies = removeDuplicates(mergedService.dependencies)
20
21      return mergedService, updatedEdges
```

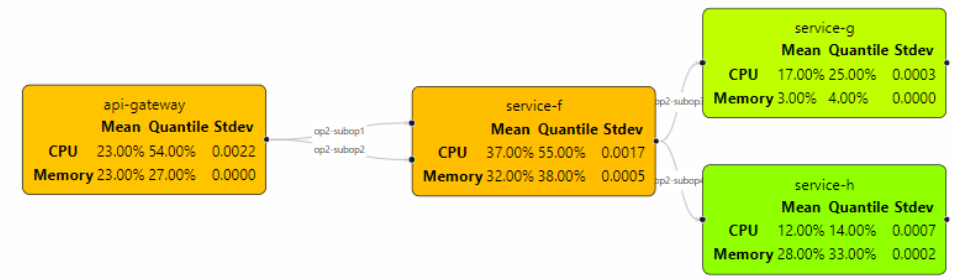
Listing 6.1: Merging Services

6.2 Splitting Services

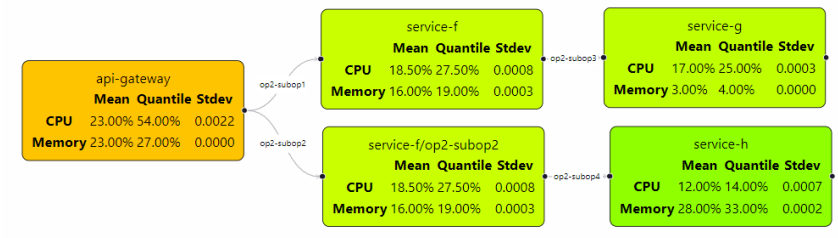
For the second mitigation technique, we employ the splitting of microservices. Splitting a service is done with the intention of creating new separations of concern and limiting the responsibility of an existing microservice. When splitting a service we need to make sure that we do not sever any data connections unnecessarily. Furthermore, splitting a service leaves us susceptible to introducing the Inappropriate Intimacy smell in our microservice application if the resulting services converge into a single data service.

6.2.1 Megaservice

In the case of the Megaservice anti-pattern, we wish to split the responsible service into several smaller services. The number of resulting services is determined by the number of incoming operations to the service on the request in question. As opposed to our merging technique, splitting a service will result in the split services retaining their respective fraction of the total utilization. This approach will most likely not give us realistic predictions, and an argument could be made for increasing the metrics by some factor to make the predictions more pessimistic. The microservices that were previously dependent on the responsible microservice will in turn be allocated to the newly created microservices in a round-robin fashion, preventing the creation of a new Inappropriate Intimacy smell. Figure 6.3a and 6.3b shows a visualization of the applied mitigation technique for the Megaservice anti-pattern.



(a) Before



(b) After

Figure 6.3: Mitigating the Megaservice Anti-Pattern

Allocating the dangling microservices in this way could be disadvantageous. In the case that there are fewer dependent services than resulting services after the split we will potentially sever important data connections. However, we do not know which dependent services would logically go together with the resulting services. Therefore, this is considered to be a best-effort approach and represents the new structure of the system regardless of incorrect labelling. With that being said, the developer should

be able to identify if the split would result in severance of the data connections and adjust the data boundaries and scope of the split accordingly. Let us look at the example from section 5.3 with the application that is transitioning from a monolith to a microservice architecture. In this case, the developer intends to transition the application into cohesive and lightly coupled microservices. The process of doing so includes defining new data boundaries and ensuring the separation of concerns. As such, we believe that the suggested structure made by the mitigation technique should serve as a basis for further design iterations. Rather than an absolute law.

6.2.2 Performing a Split in the GHUBS Model

As mentioned, we need to be wary of creating instances of the Inappropriate Intimacy anti-pattern when splitting services in the GHUBS model. The pseudo-code in Listing 6.2 shows a simplified version of our algorithm. We start by iterating over the redundant edges (bad edges) we detected during our detection step on line 5. For each bad edge, we create a new service on line 6, with a fraction of the old utilization metrics of the old responsible service (bad service) on line 7, and a new name on line 8 based on the name of the bad edge. We then append our newly split service to an array and point the bad edge to the new service, making it "healthy" on lines 11 and 12.

Then we need to handle the dependencies of the bad service. We iterate over the dependencies and create new edges for each of them on lines 14 and 15. The new edges are assigned one of the new split services in a round-robin fashion on lines 16 and 17, ensuring that we will not create an instance of the Inappropriate Intimacy anti-pattern. Finally, we return the new services and the updated edges on line 20. We also have to make sure to redirect any other edges that still refer to the bad service to the new services, but this is not included in the pseudo-code.

```
1  function splitService(badService, badEdges):
2      splitServices = []
3      updatedEdges = []
4
5      forEach(e of badEdges)
6          service = new Service()
7          service.utils = divideServiceUtilizationMetrics(badServices, badEdges.length)
8          service.name = e.operationName + badService.name
9          splitServices += service
10
11         e.to = service.name
12         updatedEdges += e
13
14     forEach(i = 0; i < badService.dependencies.length; i++):
15         edge = new Edge()
16         edge.from = splitServices[i % splitServices.length]
17         edge.to = badService.dependencies[i]
18         updatedEdges += edge
19
20     return splitServices, updatedEdges
```

Listing 6.2: Splitting Services

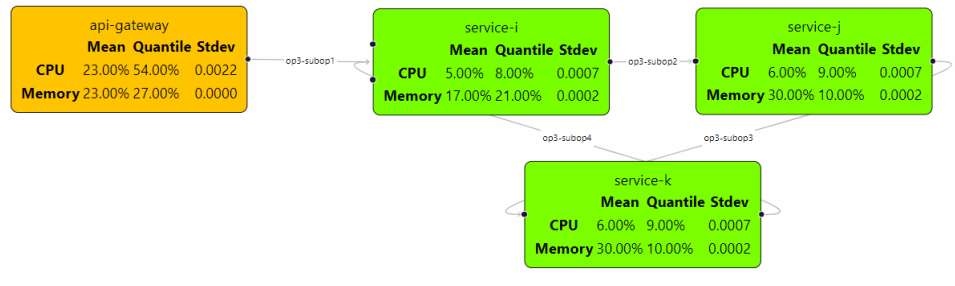
6.3 Removing Operations

The third and final mitigation technique we employ is the removal of operations or edges from the GHUBS model. By removing an operation from a request path in a microservice application, we are signalling to the developer a transfer of functionality. That is, we want the functionality of the dependent service to be transferred to its parent on the same request. The developer could also choose to clone the functionality, effectively creating two forks or externalising it into a library or something similar if it is needed on another independent request. The most appropriate approach would depend on the functionality in question and how good the communication between the parties responsible for the services is.

6.3.1 Cyclic Dependency

For our *Cyclic Dependency* this is a viable solution. In order to find the operation to be removed, we simply attempt to remove individual operations until we find one which can be removed without leaving

any isolated services. This is the $(v_k, v_1) \in E$ edge from the formalism in Section 5.4. Doing it this way means that the data will still be transferred through all of the microservices and they should be able to continue serving their purposes. This mitigation technique can be seen in Figure 6.4a and 6.4b.



(a) Before

(b) After

Figure 6.4: Mitigating the Cyclic Dependency Anti-Pattern

The alternative to removing an operation would be to merge the microservices that are involved in the cyclic dependency. However, that is not always desirable in this situation. While doing so would resolve the cycle, it could lead to consolidating more microservices than necessary. If we were to merge the microservices shown as an example in Figure 6.4b, we would end up with a service that would be a union of services I, J and K, with a 99.7% percentile CPU utilization metric of 26%. In this case, it might be completely viable to merge the services. In cases with many more microservices, however, it might not be feasible. In contrast with the Inappropriate Intimacy mitigation technique where we could do several smaller merges, the Cyclic Dependency mitigation technique would require us to merge all the responsible services in order to remove the cycle.

6.3.2 Removing an Operation in the GHUBS Model

When removing an operation in the GHUBS model we want to avoid leaving any dangling services and ensure that our graph is still strongly connected. That is, we prevent leaving services that do not have a parent service. Listing 6.3 shows the pseudo-code of a simplified algorithm for achieving this. First, we iterate over the responsible edges (bad edges) on line 2, and create a copy of the edges without the edge currently being iterated over on lines 3 and 4. Thereafter, we create an array of the destination services in our set of bad edges on line 5, the destination service being the v in $u, v \in E$. To check for dangling services, we check if any of the services are not included in this array by iterating over them on line 8 and performing the check on line 9. If any service is excluded from the array we flag this on line 10 and repeat with removing another edge. However, if they are all included in the array we return the services and the set of edges, excluding the currently iterated over edge on lines 13 and 14.

```
1  function removeOperation(badServices, badEdges):
2      forEach(be of badEdges):
3          badEdgesCopy = badEdges
4          badEdgesCopy.remove(be)
5          badEdgesDestinations = spreadEdgeDestinations(badEdgesCopy)
6
7          leavesIsolatedService = false
8          forEach(s of badServices):
9              if(!badEdgesDestinations.contains(s.name)):
10                 leavesIsolatedService = true
11                 break
12
13         if(!leavesIsolatedService):
14             return badServices, badEdgesCopy
```

Listing 6.3: Removing Operations

Chapter 7

Prototype and Validation

In order to verify the viability of our research we have created a proof-of-concept tool that automatically applies the methodology on microservice applications. We used the same tool to perform two case studies on two microservice applications from a well-known open-source microservice application suite.

7.1 Implementation

The methodology described in Chapters 4, 5 and 6 was implemented as a two-module application by the name Televisor (Telemetry-Advisor). Our first module, which we will refer to as the backend module, is responsible for the creation of the GHUBS model (both mocked and real), detection of the anti-patterns, creation of mitigation suggestions and validation against utilization metrics. The second module, which we will refer to as the frontend module, is responsible for displaying the results to the developer and conveying the information in a human-friendly manner. In fact, the before and after figures shown in Chapter 6 are from the frontend module. Figure 7.1 shows the architecture of Televisor, the dependency relationships between the different services and how the developer can use insight from Televisor to improve their microservice application.

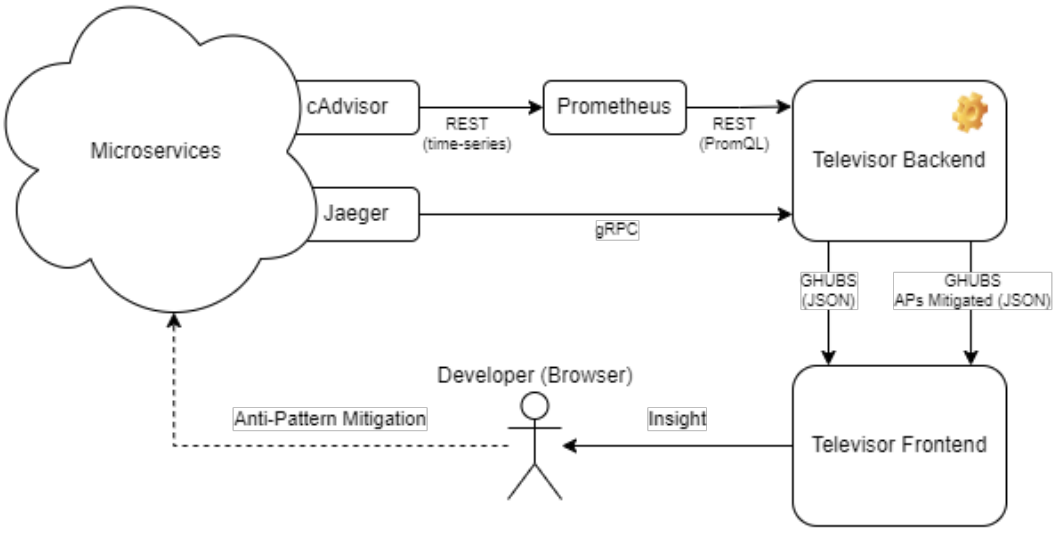


Figure 7.1: Televisor Architecture

7.1.1 Televisor Backend Module

The backend module was written in Go [35] and interfaces with a Jaeger and Prometheus client running on the target system. In order to retrieve application traces we query a Jaeger client running on the target system. The backend implements the Jaeger gRPC ProtoBuf specifications and executes requests towards the client on the :16685 port by default. In order to retrieve utilization metrics, the backend

queries the Prometheus REST API instance on the target machine. The queries are written in PromQL, which is a versatile querying language supplied by Prometheus. However as Prometheus only provides us with a time-series database and a querying language, we still need software to monitor our microservices. We use cAdvisor to monitor Docker containers running on a system and configure Prometheus to use the cAdvisor instance as a collection target.

The program can be configured and then compiled as an executable. This means that it is easy to use it wherever a developer may please, for instance in a Continuous Integration pipeline with some form of benchmarking procedures. When executing the program, two JSON files are produced. The first file describes the system-as-is and annotates the services and operations with detected anti-patterns. The second file describes the system in an "anti-pattern free" state and will simulate the implementation of the mitigation techniques on the microservice application. Both of these JSON files can then be fed into the frontend module for visualization to the developer. This entire flow can be seen in Figure 7.1.

7.1.2 Televisor Frontend Module

The frontend module was written in Typescript utilizing the React and ReactFlow libraries for visualizing the GHUBS model as well as providing interactivity. By supplying the JSON files generated by the backend module, developers have two modes of visualization, which can be easily toggled between through a button. The first shows the system-as-is, with anti-patterns annotated in a list on the left-hand side. The second mode of visualization shows the "anti-pattern free" state, in which mitigation techniques have been applied. This "before and after" look has been shown in Figures 6.1, 6.2, 6.3 and 6.4. Effectively showing a before and after view of the system and the suggested mitigation techniques. Furthermore, the services are coloured in a heatmap fashion, depending on the CPU quantile utilization metrics of the services. As discussed, the GHUBS model provides granularity in the context of the rest of the application. As such we also give the developers the option to filter the visible edges based on requests. If the developer wishes to inspect a particular anti-pattern, on a particular request, they may do so without the noise of other requests. Additionally, there are visualization features such as zoom, pan and functionality to move services around to further help with visibility. All of these features were used to create Figures 6.1, 6.2, 6.3 and 6.4, and can be contrasted against the upcoming Figures 7.2 and 7.4.

7.2 Case Studies

In order to validate our findings, we have run experiments on two of the benchmark applications found in the DeathStarBench microservice application suite [11]. The chosen applications were the Social Network and the Media Application and both of them exhibited anti-patterns. The number and types of anti-patterns detected can be found in Table 7.1.

Table 7.1: Anti-Patterns Detected in DeathStarBench Applications

	Inappropriate Intimacy	Megaservice	Cyclic Dependency	Microservice Greedy
Social Network	0	0	0	6
Media Application	1	0	0	3

The process of validating our mitigation techniques in the cases above could be used as an example of how a developer may reason about an architectural change in their own system. While we have not spent time analyzing the source code of these applications, we can still make quite a few observations based on the information we have and come to certain conclusions about the viability of the mitigation suggestions generated by the Televisor tool.

7.2.1 Social Network

The scope of the Social Network application as described by the paper introducing the DeathStarBench is as follows: "The end-to-end service implements a broadcast-style social network with uni-directional follow relationships" [11]. Across the three requests we gathered from running the benchmark workloads provided by the DeathStarBench suite [11] on the Social Network application we found six instances of the Microservice Greedy anti-pattern. Surprisingly, all six instances were to be found on the same request (/wrk2−api/post/compose). Figure 7.2 shows the services flagged with the Microservice Greedy

smell in red for the /wrk2−api/post/compose request in the Social Network application. The services that are marked in the same box are grouped as they would be merged into the same parent service. Note that as all of the anti-pattern instances were apparent on a single request, we have used the filter feature of the Televisor frontend to abstract away the other edges for visibility purposes. The black dots apparent on the left-hand side of the post−storage−service, home−timeline−service and user−timeline−service without an incoming edge are points of ingress on other requests.

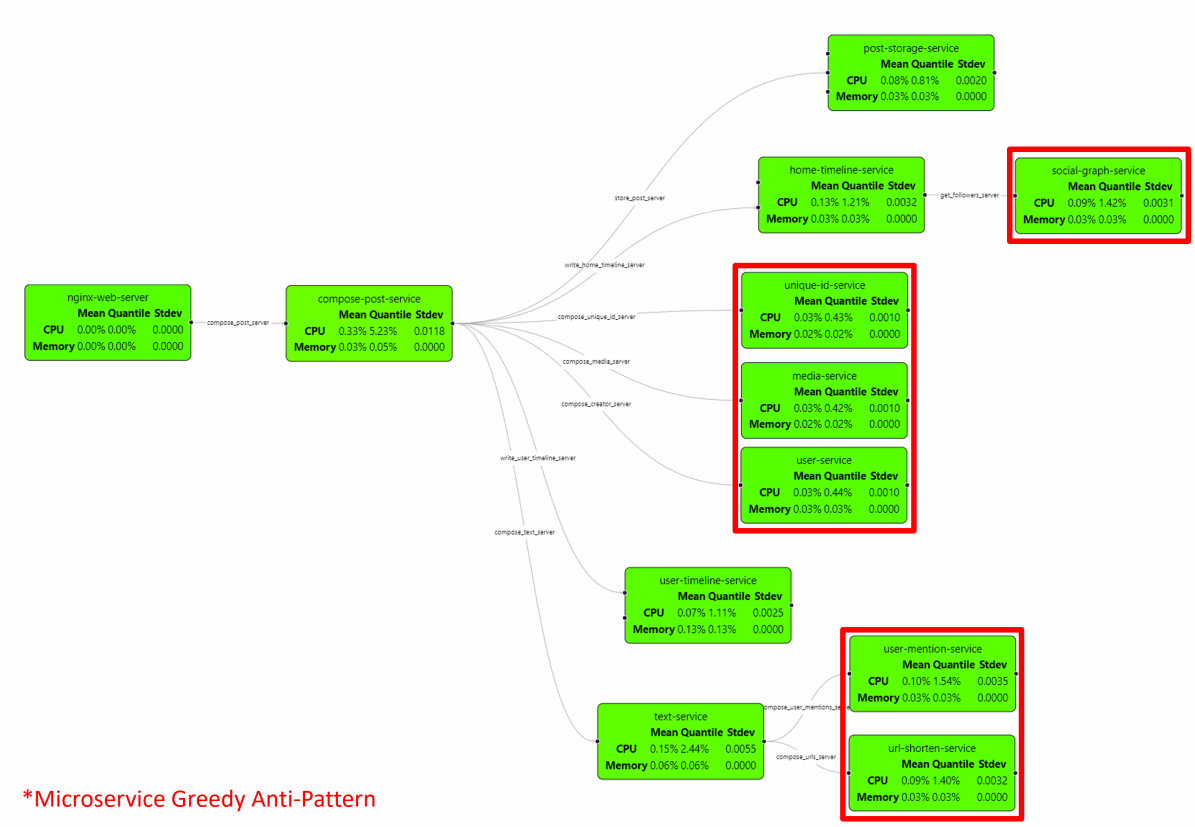


Figure 7.2: Social Network Anti-Patterns

Before we can attempt to validate our mitigation suggestions, we need to have a look at the utilization metrics and see if merging the services with their parents is feasible. For our experimental setup, we have set our requirements to be that the summed CPU 99.7% percentile utilization metric across the microservices in question should be below 25%, and the corresponding metric for memory to be below 15%. We are being generous with our requirements here to facilitate some validation, although as Figure 7.1 shows, the utilization metrics are quite low. The first merge 1) involving microservices unique−id−service, media−service and user−service into compose−post−service, has summed 99.7% percentile utilization metrics of 6.51% and 0.12% for CPU and memory respectively. The second merge 2) with services user−mention−service and url−shorten−service into text−service, has summed 99.7% percentile utilization metrics of 5.38% and 0.12% for CPU and memory respectively. The third and final merge 3) involves the social−graph−service into the home−timeline−service and has summed 99.7% percentile utilization metrics of 2.63% and 0.06% for CPU and memory respectively. As the utilization metrics are within our requirements for the mitigation technique we can go ahead with the validation.

Validation and Reasoning

To evaluate whether or not the suggested merges are feasible we will be looking at dependency relationships illustrated by the GHUBS model, hardware utilization metrics, and analyze a Jaeger trace of the /wrk2−api/post/compose request to reason about temporal behaviour. The trace in question can be found in Figure 7.3. In merge 1) we can see that the unique−id−service, media−service and user−service all perform minuscule amounts of work. In fact, most of the time compose−post−service is waiting for a callback so that it can proceed to execute on the text−service. In merge 2) we can see that the text−service also performs very little work and that most of the processing time happens in the user−mention−service and

the url−shorten−service. From looking at Figure 7.2 we can also see that the text−service is not invoked on any other requests either, and is effectively serving as a wrapper for its child services. We see the same story for merge 3), however in this case the home−timeline−service does serve a purpose on another request.

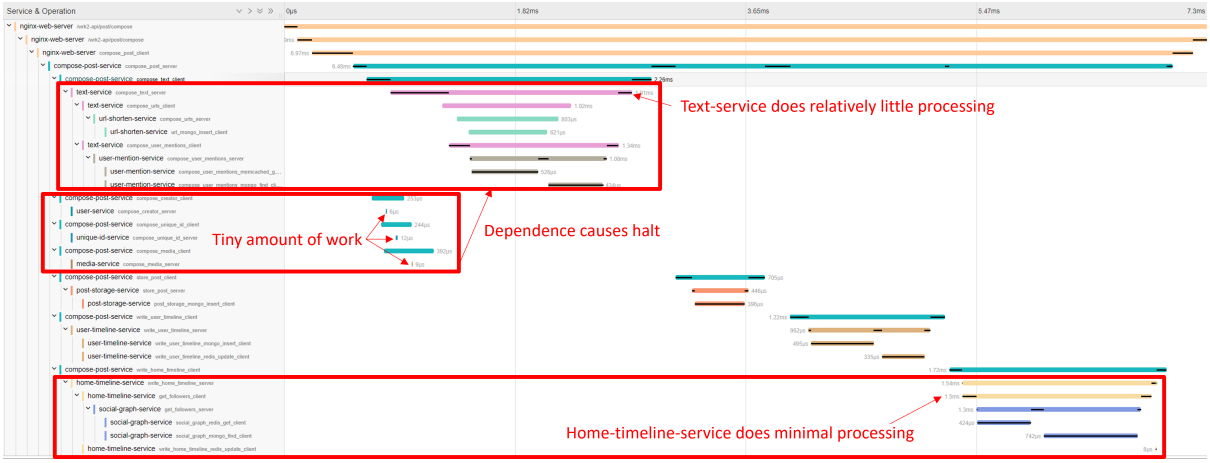


Figure 7.3: Social Network Jaeger Request Trace

With this information, we can draw a few conclusions about each of the proposed merges. In merge 1), we can tell that the child services are already tightly coupled with the compose−post−service because of the callback dependency. We can also tell that their workloads and hardware utilization metrics are nominal, and most of the latency is due to network communication overhead. Merging these services could help alleviate this. For merge 2), the microservices are also tightly coupled and their only path of execution is on this request. This indicates that we can safely merge these services, without creating problems for other requests in the application. Merge 3), while exhibiting the same symptoms as merge 2), is more complicated. There is some latency going from the home−timeline−service to the social−graph−service however it is minuscule. Furthermore, the home−timeline−service is invoked on another request. As such, the question is whether or not merging them would be detrimental to the separation of concerns. By their names and domain, we can perhaps argue that the home−timeline−service is sufficiently dependent on the social−graph−service to justify a merge. Ultimately, for merge 3), the developer has to be the one to investigate the feasibility of the mitigation technique and it is not something we can make a good decision about solely based on the information provided by the Televisor tool. However, the tool is successful in alerting the developer to a potential issue and does provide information that can be invaluable in the beginning of such an investigation.

7.2.2 Media Application

The scope of the Media Application, as described in the DeathStarBenchPaper is as follows: "The application implements an end-to-end service for browsing movie information, as well as reviewing, rating, renting, and streaming moves" [11]. The Media Application has six requests to analyze after running a script for seeding the database, as well as the provided wrk2 benchmark. We found anti-patterns in one of the six requests. Namely, three instances of the Microservice Greedy anti-pattern, and one instance of the Inappropriate Intimacy anti-pattern. The request in question is on the \wrk2 −api\review\compose route. Figure 7.4 shows the GHUBS model of the Media Application. Once again, the Televisor frontend filter functionality has been used to hide the edges that are not present on this request. The services that inhibit the Microservice Greedy anti-pattern are highlighted in red, while the services that have the Inappropriate Intimacy anti-pattern are highlighted in purple.

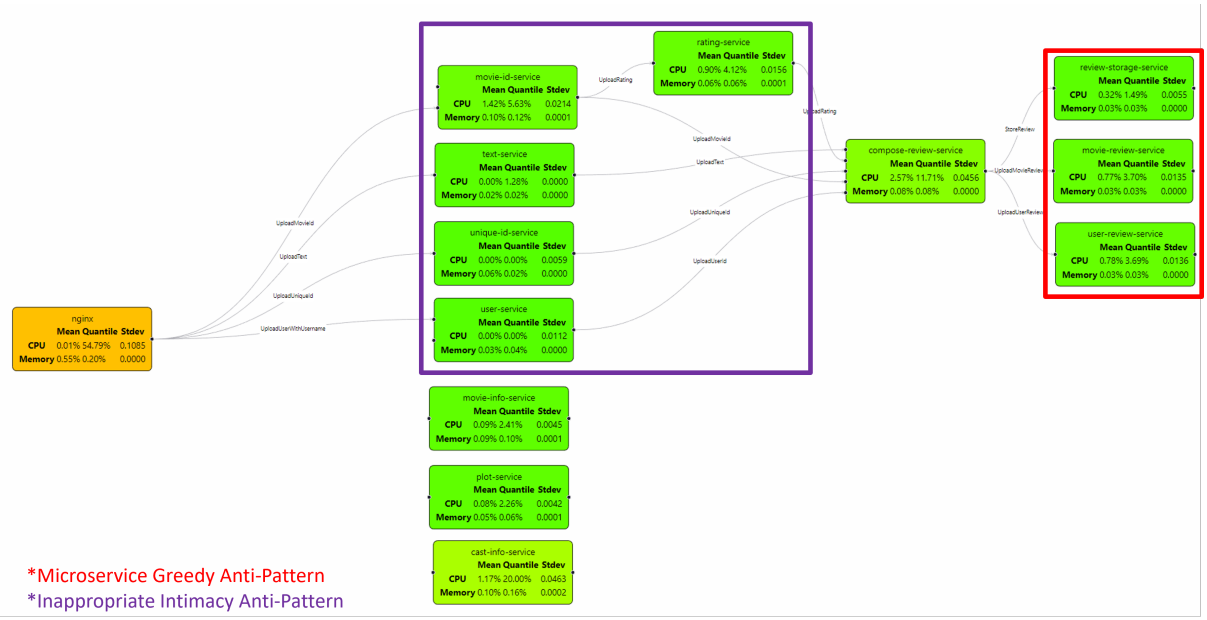


Figure 7.4: Media Application Anti-Patterns

Our utilization metric merging thresholds for the microservices responsible for the Microservice Greedy anti-pattern are the same as for the Social Network application. For the microservices responsible for the Inappropriate Intimacy smell, however, we will accept a merge if the summed CPU 99.7% percentile utilization metric is less than 40%. Our CPU requirement for the Inappropriate Intimacy anti-pattern is more lenient than for the Microservice Greedy anti-pattern, as with the latter we want to avoid a situation where the entire application is suggested to be merged. We are not taking memory utilization into account here, but as alluded to previously this is configurable on a per application and anti-pattern mitigation technique basis. One reason for excluding it could be that it is generally easier to add more memory to a system than to increase the number of cores or the clock speeds of a CPU. In our case, however, it is more a matter of showing off the versatility of the metric requirement functionality. Our first merge 1), consisting of the review−storage−service, movie−review−service and the user−review−service accounts for the Microservice Greedy anti-pattern. The CPU 99.7% percentile utilization is 8.28%, and the corresponding memory utilization is 0.09%. These metrics are acceptable with our requirements for under 25% CPU and under 15% memory, 99.7% percentile utilization metrics respectively. The second merge 2), consisting of the movie−id−service, text−service, unique−id−service, user−service and the rating−service accounts for the Inappropriate Intimacy anti-pattern. Here we have a summed CPU 99.7% percentile of 11.03%, which is also acceptable with our requirement. As such, we can go ahead with the mitigation technique validation.

Validation and Reasoning

Again, to evaluate whether or not the suggested merges are feasible we will be looking at dependency relationships illustrated by the GHUBS model, hardware utilization metrics and analyze a Jaeger trace of the /wrk2−api/review/compose request to argue about temporal behaviour. Figure 7.5 shows the trace. For our first merge 1) marked in red, we can see that both the user−review−service and the movie−review−service are waiting for a callback from the review−storage−service before proceeding with the database updates. This sort of dependency relationship and their temporal behaviour indicates that the services involved are tightly coupled. The temporal behaviour of the second merge 2), does not have any obvious performance problems. However, we can see that a relatively large amount of time is being spent on network communication and waiting for callbacks. By looking at the GHUBS model in Figure 7.4 in conjunction with the Jaeger trace, we can also see that there are two dependencies between the movie−id−service and the compose−review−service. The first one is directly between the two, while the other is a transitive connection through the rating−service. This could potentially be avoided with different data boundaries between the microservices.

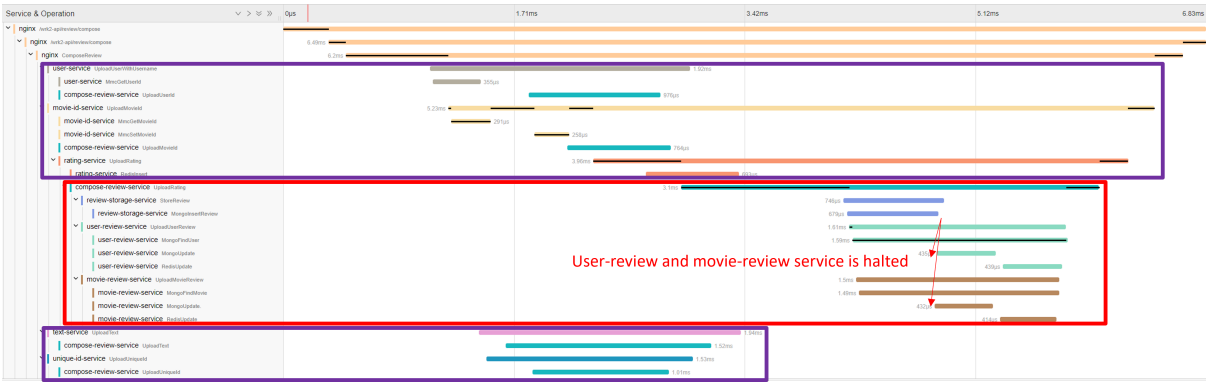


Figure 7.5: Media Application Jaeger Request Trace

For merge 1), it is apparent that the dependency relation between the user−review−service and the movie−review−service waiting for a callback from the review−storage−service is not ideal. The three services perform strictly related tasks and are tightly coupled, making them suitable for a merger. Furthermore, the three services all interact with separate databases, however, they are only invoked on this particular request. As the user−review−service and the movie−review−service are also dependent on data from the review −storage−service, there strictly is no need for separate databases. In fact, merging the services and utilizing a single database with multiple schemas would facilitate the use of database transactions, enhancing data integrity and possibly increasing maintainability by simplifying error handling. With this in mind, we believe that merging the services responsible for the Microservice Greedy smell in the Media Application would be beneficial.

While we did not find much in terms of detrimental temporal behaviour for merge 2), we can still tell that the anti-pattern has introduced unnecessary latency into the application. After all, the incoming data on the request is diverged into five different services and they all converge on the same service. Additionally, the service data boundaries do not seem to follow business processes, and certainly not Domain Driven Design [12]. Because of the number of services involved in this anti-pattern, we do not think that a single merge would be feasible. However, we could make an attempt at recomposing the system after business processes. For instance, merging the services into three new services: a movie−service, user−service and a review−service. Now we could reduce the number of operations on this request to three. We can make a call to the new movie−service to retrieve the movie ID, at the same time retrieve or authenticate the user in the user−service and lastly upload our review in the review−service.

If we were to do this, it would also be prudent to consider the other microservices in the application that are not involved in this request. Say including the movie−info−service and the plot−service in the creation of the new movie−service. While the cast of a movie usually is closely related to the concept of a movie, the cast−info−service should continue to be independent. As we can see, the CPU 99.7% percentile utilization metric is rather high at 20% and we assume that we would like to be able to make changes to actors and such independent of the movies they have played in. Merge 2) is not as clear-cut as the other anti-pattern mitigation suggestions we have looked at so far. While merging all of the services into a single service would probably be detrimental to the separation of concerns in the application, the Televisor tool did provide us with information that could guide us in creating a more maintainable and reliable system. Most importantly it made us think about the application composition by bringing it to our attention.

Chapter 8

Discussion

Now that we have run two experiments on the benchmarking applications from the DeathStarBench [11], we can evaluate how well our model, detection and mitigation suggestions worked. We will discuss the successes and shortcomings of our model, formalization, detection and mitigation based on the results of these case studies.

8.1 The GHUBS Model

We set out to create a model that is granular enough to detect cases of architectural anti-patterns that apply to individual requests, but also anti-patterns that first become apparent when considering multiple or all requests in a microservice application. Keeping the model relatively simple, and being able to formalize it with well-known mathematical notation was also key for facilitating automatic detection of the architectural anti-patterns. Furthermore, we wanted to extend the model with utilization metrics from the microservices to ensure requirements conformance after implementing changes to the model or an application.

Our use of the API Gateway architectural design pattern was useful in determining where to start the trace collection. Furthermore, the requests ingressing the application through the API Gateway are more often than not representative of business processes which helps in reasoning about particular system flows.

In order to evaluate the validity of our formalized anti-pattern detection algorithms, we created a mock GHUBS model reflecting an application that had all of the anti-patterns. This made prototyping and testing of the anti-pattern detection algorithms very efficient. However, we also found that the results of creating a testing model could lead to a false sense of security if we did not test enough variations of the patterns. Feedback from practitioners suggests that the ability to create a mock GHUBS model is very valuable, as it can be used by developers to evaluate a proposed microservice application design before developing it.

8.2 The Anti-Pattern Detection and Formalization

The work presented in this thesis lays the foundation for further formalization and work in the area of automatic detection of microservice anti-patterns. The anti-pattern that has been reported as the most prevalent and harmful by practitioners is the Wrong Cuts anti-pattern [4]. As described in Chapter 3, being able to detect, and mitigate, this anti-pattern is highly dependent on the domain knowledge of the developer and is non-trivial with telemetry data alone. While Wrong Cuts is a standalone anti-pattern, we believe that the four anti-patterns we have worked with could be considered elements of a hypothetical set of Wrong Cuts anti-patterns. Similarly to the Wrong Cuts anti-pattern, the four also lacked formal definitions, however, they did describe microservice dependency relationships in natural language. By formally defining more architectural anti-patterns we might come closer to a set of universal guidelines for proper microservice application decomposition agnostic to domain knowledge. With this in mind, we set out to create formal mathematical definitions of anti-patterns we believe to be instances of these smells.

Our definition of the Inappropriate Intimacy anti-pattern instance is perhaps the most complex of the four and is a result of our interpretation of the available literature. As such, the anti-pattern we

chose to detect is unlikely to be the *only* one that can be considered an instance of the Inappropriate Intimacy anti-pattern. We were able to detect a single instance of this smell on the Media Application from the DeathStarBench [11], and also reach a conclusion about the system decomposition based on the microservices that were highlighted by the detection algorithm. As we saw in that case, a large number of microservices were included in the anti-pattern and we were able to detect that the granularity of the microservice decomposition in the system was very fine. In fact, the Media Application consisted of several services that executed a single function across all of the requests in the application.

The Microservice Greedy anti-pattern was the one we detected the most instances of, which was not a surprise considering the simplicity of the anti-pattern. In fact, before running the experiments in the case studies, we suspected that we would come across several false positives. However, the detection algorithm ended up detecting several cases where we could find performance and maintainability gains by performing a merge. These findings once again made it apparent that the benchmark applications had a very granular decomposition, and as such the anti-patterns did not appear to be false positives. If the services were bigger in terms of responsibility and functionality this could have turned out differently. It should be mentioned that the goal of the DeathStarBench suite is to have applications with enough microservices to make them representative of real-world systems and their inherent complexities [11]. As such it is not surprising if they were eager to create new services, even if they were small.

We were not able to find cases of the Megaservice anti-pattern. As mentioned in Chapter 5, we believe that this smell would be most prevalent in microservice applications that have transitioned from a monolithic architecture, which is not the case for the DeathStarBench [11] applications. A thing to note with several of the microservice application benchmarking suites that are currently out there is that they are mostly created by following best practices. This means that attempting to detect anti-patterns in them can be challenging.

Of our four anti-patterns, Cyclic Dependency is the only one that had a formal definition using graph notation, but in natural language. We did not find any instances of this anti-pattern across our experiments either. However, this is a smell that has been reported to be fairly common by microservice architects and other practitioners [4], and as such it was natural to include it.

We also discovered that the Inappropriate Intimacy and Cyclic Dependency anti-patterns can manifest themselves as nested. With our current algorithms, we would be able to detect the nested anti-pattern instances, however, we do not perform any checks to see if one is a subset of another. In the case of the Inappropriate Intimacy anti-pattern, it would be prudent to subsume the nested instance into the parent when applying a mitigation technique. While for the Cyclic Dependency anti-pattern, we would prefer to treat them separately as we would have to employ multiple transfers of functionality in order to mitigate the anti-patterns.

8.3 The Mitigation Suggestions

Our mitigation technique for the Microservice Greedy and Inappropriate Intimacy anti-patterns was to merge the services flagged as responsible for the anti-pattern. While merging the responsible services into a single new microservice seemed to be feasible for our Microservice Greedy anti-patterns, this was not the case with the Inappropriate Intimacy anti-pattern in the Media Application. We believe this is due to the number of microservices that were flagged as responsible for the Inappropriate Intimacy anti-pattern. The flagged services had use cases on other requests, and their separation of concerns was highly granular. As explained in Chapter 7, we believe that merging them into several services according to business processes would be the better choice. This implies that the reliability of our mitigation suggestion for the Inappropriate Intimacy anti-pattern is weakened for situations involving many services. Perhaps this is something that could be addressed by basing the number of resulting services after a merge on the number of unique requests that invoke the involved services. While we still would not be able to create merges that would conform to business processes, we could get a better estimate of the number of resulting services. This would be more akin to the splitting technique we employ with the Megaservice anti-pattern, where the number of resulting services after a split is dependent on the number of inbound edges to the Megaservice.

Unfortunately, we did not come across the Megaservice or Cyclic Dependency anti-patterns in our case studies. While we did test our detection and mitigation schemes for these anti-patterns in the mock application, it would be highly beneficial to see their impact on a real application.

In hindsight, we see that we were quite dependent on Jaeger to validate our mitigation suggestions for the Microservice Greedy anti-pattern. The ability to see which service on a request demands the most resources is quite useful when reasoning about a suggestion. While we do believe it is beneficial

to use several tools in order to validate potential changes to the application architecture, it is an area in which our model is unnecessarily insufficient. If we were to make a change at this point, we would propose to add latency mean, percentile and standard deviation to the GHUBS model edges between services on each request. With this information, we could extend the functionality of our requirements conformance checking to include network overhead gains from merging services. We also relied on Jaeger for the ability to see in which order executions were made. The Gantt chart view in Jaeger is ideal for this kind of visualization, and while we could add timestamps to the edges in the GHUBS model it would be an inferior solution.

8.4 Threats to Validity

Here, we discuss aspects of our method that are prone to degrading the quality of our research. We are primarily concerned about the implications of telemetry-based methods, our approaches for predicting utilization metrics in microservices after applying a mitigation technique, our coverage of the anti-patterns, and the applicability of our tool and methodology.

8.4.1 Telemetry Based Methodologies

A weakness of any run-time approach is that the workload that is run in an application has to cover most if not all system flows in order to be representative of the application. By this, we mean that the telemetry gathered from an application is only as good as the coverage of executions. For instance in our case, if we were to gather telemetry in a timeframe where one of the API Gateway requests was not invoked, we would be missing edges and potentially be unable to detect anti-patterns on that request. This can be prevented by increasing the length of the telemetry capture timeframe so that we are more likely to gather all possible request executions in a system, however, this weakness needs to be accounted for.

8.4.2 Predicting Hardware Utilization Metrics

A clear weak point of our requirements conformance checking is the way we handle utilization metrics when merging, splitting and moving microservice functionality. Predicting this kind of behaviour is a separate field of research altogether, and our approach is unlikely to be accurate. With that being said, it is successful in conveying to the developer that performing a mitigation technique will come with a change in utilization metrics across the microservice application. We feel fairly certain in the fact that merging services will increase utilization metrics, and splitting them will do the opposite. For the moving of functionality between services, we can expect a rise in computation at the service that ends up being extended. However, guessing the degree to which it will rise is speculative at best and as such we decided against doing so. Another approach would be to check the utilization metrics of the individual services before employing a mitigation technique and ensuring they are within some requirements. Rather than the current check that happens on the predicted utilization metrics of the resulting services after a mitigation suggestion. Thereafter, we could present the predicted values to the developer and allow them to reason about the metrics themselves.

8.4.3 Coverage of Anti-Patterns

The instances of the four natural language anti-patterns we chose to detect depend on our interpretation of the available literature. While we believe that the instances we are detecting are correct, it is important to keep in mind that these are *instances* of anti-patterns and not completely comprehensive. This means that there may be other instances of the same anti-patterns that our formal definitions and software are not able to detect. With that being said, this is a problem that can only be solved by creating more specific anti-pattern definitions and our work lays the foundation for just that.

8.4.4 The Applicability of the Tool

We have tested the tool and methodology against two widely used open-source benchmark applications and one internal mock application. As such, we do not have a very big sample size to determine the applicability of our work. Specifically, we would have liked to test an industrial application that is actually in use. However, the technologies used to develop the Televisor tool are completely open-source,

highly available and already prevalent in many microservice applications. As such, we believe that our tool and methodology could easily be extended and integrated into existing workflows.

Chapter 9

Conclusion and Future Work

Now that we have explained our methodology, demonstrated it in a case study through the use of the Televisor tool and discussed our results we can look at whether or not we were able to answer the research questions we set out on this journey with. Having answered our questions and learned a lot on the way, we have also found several topics that can serve as future work in this area of research.

9.1 Conclusion

We formulated three research questions that aimed to solve concrete problems in the current research on architectural anti-patterns in microservice applications. To answer the research questions we made four novel contributions to the field.

RQ1: How can we utilize telemetry data to generate a model for automatically detecting instances of the four architectural anti-patterns, Inappropriate Intimacy, Microservice Greedy, Megaservice and Cyclic Dependency in microservice applications?

For RQ1, we wanted to know how we could use telemetry data to create a model that would be suitable for detecting architectural anti-patterns, such as Inappropriate Intimacy, Microservice Greedy, Megaservice and Cyclic Dependency in microservice applications. Through researching currently used models and the works they are used in we were able to get an understanding of what makes them good, and what features they are missing to detect these more complex anti-patterns. We found that a lack of granularity and detailed information about individual microservice metrics were detrimental and that if we wanted to attempt detection of more advanced anti-patterns it would be required. We ended up creating the Granular Hardware Utilization-Based SDG (GHUBS), a formal but versatile model which can be used for the detection and eventual mitigation of such anti-patterns. Furthermore, we developed our methodology in a way that would ensure it is agnostic to the technologies used in the microservice applications.

RQ2: How can we provide formal mathematical definitions of known microservice architectural anti-patterns, and apply them to create automated detection procedures?

We created formal mathematical definitions of known architectural anti-patterns and automated detection procedures based on the formalizations, just as we set out to do with RQ2. The four instances of the anti-patterns Inappropriate Intimacy, Microservice Greedy, Megaservice and Cyclic Dependency all seem to be correct and successful in detecting the patterns. We did this by utilizing pre-existing formal notation of directed multi-graphs to define anti-patterns in our GHUBS model. These anti-patterns could then be translated into algorithms and be used for automated anti-pattern detection. Furthermore, our efforts in doing so were successful in creating a platform on which further work can continue. Despite microservice benchmarking suites being designed to comply with best practices, we also managed to detect 10 anti-patterns in the DeathStarBench suite [11] and fruitfully reason about them.

RQ3: To what extent can we suggest mitigation techniques for solving the detected architectural anti-patterns, and determine the viability of said mitigation techniques based on hardware utilization metrics?

Our work with suggesting mitigation techniques and determining their viability, as alluded to by RQ3, was perhaps the most foreign concept in the existing literature. The approach we chose, merging, splitting and moving functionality around in the application is somewhat superficial. As such, the work involved in actually performing these mitigation techniques would be monumental in comparison. However, the method is successful in creating radical change and showing the potential effect of such a change. We did find that when suggesting mitigation, especially a merge, it would be prudent to consider related requests when determining the number of resulting services. Instead of defaulting to merging services into a single new service, regardless of the number of services that are involved. Furthermore, the work of predicting utilization metrics is too simple to be considered realistic. With that being said, we do believe that the state of the microservice should be taken into account when making such a change. Our concept of using simple summations and divisions to predict mitigation outcomes is a starting point and should be treated as such. Recommendations for solving architectural anti-patterns have not been attempted in previous research. As such, taking a first step is important, even if it is not perfect.

Contributions

Overall, we managed to answer the questions we set out to and made four novel contributions: the GHUBS model, formally defined anti-patterns and detection algorithms, mitigation techniques and the proof-of-concept Televisor tool for performing our method automatically. This was then validated in a case study, with accompanying reasoning for how to use the Televisor tool effectively. We used industry-standard technologies in the creation of our method and tool, and such it can be easily adopted in existing workflows without being intrusive or otherwise problematic. In Chapter 3 we said that we would "leave organizational structure out of the decomposition equation", and we did. We found that we can detect decomposition issues without intimate domain knowledge, however, some degree of familiarity is required in validating those findings.

9.2 Future Work

Practitioners have identified a set of 20 microservice anti-patterns [5], and we have covered instances of four anti-patterns. While not all of the 20 anti-patterns are suitable for this kind of dynamic analysis and would require some degree of static analysis, we believe there are several more that could be formalized. The methodology we have developed is as we have shown effective in creating formal anti-pattern definitions, and as such makes for a good opportunity for further research in the field. To facilitate this it would be convenient to have a GUI tool for creating GHUBS models that have novel anti-patterns that need testing, rather than the code-driven approach we are currently using. Such a tool could rather easily be implemented as an extension of our Televisor tool, as it already has an implementation of the GHUBS model and a way of visualizing it.

We noted that the current microservice application benchmark suites that exist are fairly well composed. This brings to question, what would a badly designed microservice application benchmark look like? Such a benchmark could prove useful in testing new tools and methods. With formal anti-pattern definitions creating such an application would be rather trivial, and only a matter of stitching them together in a believable manner.

The utilization metrics we chose to consider in this work are far from the only ones that could be considered. As we alluded to in Chapter 8, adding metrics such as latency and timestamps to the edges could help in creating more accurate mitigation suggestions. Furthermore, they would remove the need for using Jaeger for validating mitigation suggestions, if visualized properly. Predicting utilization metrics when implementing mitigation suggestions is a very difficult task, and the outcome of implementation is highly dependent on the practitioner and the individual microservices. Investigating what would be required to perform such a prediction would be very interesting.

For the splitting mitigation technique, it would also be interesting to look at the internal spans of the responsible service on the function level and see if it is possible to derive more accurate decompositions based on this. One could for instance determine which sequence of executions is responsible for the monolithic behaviour.

Acknowledgements

This work would not have been possible without the help of Dr. Benny Åkesson, Dr. Ben Pronk and the rest of the team from TNO-ESI. Thank you very much for the many discussions, feedback sessions and unyielding enthusiasm for the project.

Bibliography

[1] N. Dragoni *et al.*, "Microservices: Yesterday, Today, and Tomorrow," en, in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds., Cham: Springer International Publishing, 2017, pp. 195–216, ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12 (visited on 12/01/2022).

[2] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," vol. 35, no. 3, pp. 24–35, May 2018, Conference Name: IEEE Software, ISSN: 1937-4194. DOI: 10.1109/MS.2018.2141039.

[3] M. Kolny, *Scaling up the prime video audio/video monitoring service and reducing costs by 90%*, Mar. 2023. [Online]. Available: https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90.

[4] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," vol. 35, no. 3, pp. 56–62, May 2018, Conference Name: IEEE Software, ISSN: 1937-4194. DOI: 10.1109/MS.2018.2141031.

[5] D. Taibi, V. Lenarduzzi, and C. Pahl, "Microservices anti-patterns: A taxonomy," *Microservices: Science and Engineering*, pp. 111–128, 2020.

[6] Y. Gan *et al.*, "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 19–33, ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304004. [Online]. Available: https://doi.org/10.1145/3297858.3304004 (visited on 01/11/2023).

[7] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Firm: An intelligent fine-grained resource management framework for slo-oriented microservices," in *Proceedings of The 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[8] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *11th {USENIX} symposium on operating systems design and implementation ({OSDI} 14)*, 2014, pp. 217–231.

[9] G. Parker *et al.*, "Visualizing anti-patterns in microservices at runtime: A systematic mapping study," *IEEE Access*, 2023.

[10] L. Baresi, M. Garriga, and A. De Renzis, "Microservices Identification Through Interface Analysis," en, in *Service-Oriented and Cloud Computing*, F. De Paoli, S. Schulte, and E. Broch Johnsen, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 19–33, ISBN: 978-3-319-67262-5. DOI: 10.1007/978-3-319-67262-5_2.

[11] Y. Gan *et al.*, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 3–18, ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304013. [Online]. Available: https://doi.org/10.1145/3297858.3304013 (visited on 12/01/2022).

[12] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[13] M. E. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[14] R. Heinrich *et al.*, "Performance Engineering for Microservices: Research Challenges and Directions," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ser. ICPE '17 Companion, New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 223–226, ISBN: 978-1-4503-4899-7. DOI: 10.1145/3053600.3053653. [Online]. Available: https://doi.org/10.1145/3053600.3053653 (visited on 12/01/2022).

[15] [Online]. Available: https://www.jaegertracing.io/.

[16] Prometheus, *Prometheus - monitoring system & time series database*. [Online]. Available: https://prometheus.io/.

[17] Sep. 2023. [Online]. Available: https://www.docker.com/.

[18] [Online]. Available: https://opentelemetry.io/docs/.

[19] A. Bento, J. Correia, R. Filipe, F. Araujo, and J. Cardoso, "Automated analysis of distributed tracing: Challenges and research directions," *Journal of Grid Computing*, vol. 19, pp. 1–15, 2021.

[20] [Online]. Available: https://zipkin.io/.

[21] B. Kienhuis, E. F. Deprettere, P. Van der Wolf, and K. Vissers, "A methodology to design programmable embedded systems: The y-chart approach," *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation—SAMOS*, pp. 18–37, 2002.

[22] J. Lapalme, B. B. Theelen, N. Stoimenov, J. J. Voeten, L. Thiele, and E. M. Aboulhamid, "Y-chart based system design: A discussion on approaches," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:14617463.

[23] S. Panichella, M. I. Rahman, and D. Taibi, "Structural coupling for microservices," *arXiv preprint arXiv:2103.04674*, 2021.

[24] A. Al Maruf, A. Bakhtin, T. Cerny, and D. Taibi, "Using microservice telemetry data for system dynamic analysis," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, IEEE, 2022, pp. 29–38.

[25] 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)*, IEEE, 2016, pp. 433–437.

[26] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, "Using Service Dependency Graph to Analyze and Test Microservices," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, ISSN: 0730-3157, vol. 02, Jul. 2018, pp. 81–86. DOI: 10.1109/COMPSAC.2018.10207.

[27] G. McCluskey, *Using java reflection*, Jan. 1998. [Online]. Available: https://www.oracle.com/technical-resources/articles/java/javareflection.html#:~:text=Reflection%20is%20a%20feature%20in,its%20members%20and%20display%20them..

[28] I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi, "Towards microservice smells detection," in *Proceedings of the 3rd International Conference on Technical Debt*, ser. TechDebt '20, New York, NY, USA: Association for Computing Machinery, Sep. 2020, pp. 92–97, ISBN: 978-1-4503-7960-1. DOI: 10.1145/3387906.3388625. [Online]. Available: https://doi.org/10.1145/3387906.3388625 (visited on 02/01/2023).

[29] S. Luo *et al.*, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 412–426.

[30] X. Guo *et al.*, "Graph-based trace analysis for microservice architecture understanding and problem diagnosis," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1387–1397.

[31] G. Somashekar and A. Gandhi, "Towards Optimal Configuration of Microservices," in *Proceedings of the 1st Workshop on Machine Learning and Systems*, ser. EuroMLSys '21, New York, NY, USA: Association for Computing Machinery, Apr. 2021, pp. 7–14, ISBN: 978-1-4503-8298-4. DOI: 10.1145/3437984.3458828. [Online]. Available: https://doi.org/10.1145/3437984.3458828 (visited on 01/02/2023).

[32] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings 5*, Springer, 2016, pp. 185–200.

[33] C. Richardson, *Microservices pattern: Api gateway pattern*. [Online]. Available: `https://microservices.io/patterns/apigateway.html`.

[34] [Online]. Available: `https://grpc.io/`.

[35] [Online]. Available: `https://go.dev/`.