Automatic Anti-Pattern Detection in Microservice Architectures based on Distributed Tracing

Hübener, Tim

Award date:
2021

Automatic Anti-Pattern Detection in Microservice Architectures based on Distributed Tracing

August 3, 2021

Author:
Tim Hübener | 0979287

Supervisor:
Yaping Luo

Mathematics and Computer Science Department
Software Engineering and Technology

M.Sc. Computer Science and Engineering
Eindhoven University of Technology

Declaration concerning the TU/e Code of Scientific Conduct

I have read the TU/e Code of Scientific Conduct[i].

In carrying out research, design and educational activities, I shall observe the five central values of scientific integrity, namely: trustworthiness, intellectual honesty, openness, independence and societal responsibility, as well as the norms and principles which follow from them.

Date

26.07.21

Name

Tim Hübener

ID-number

0979287

Signature

Submit the signed declaration to the student administration of your department.

[i] See: https://www.tue.nl/en/our-university/about-the-university/organization/integrity/scientific-integrity/
The Netherlands Code of Conduct for Scientific Integrity, endorsed by 6 umbrella organizations, including the VSNU, can be found here also. More information about scientific integrity is published on the websites of TU/e and VSNU

Acknowledgments

First of all I would like to thank my family for their love and support throughout my studies.

Next, I want to thank Michel Chaudron, Pieter Vallen, Jonck van der Kogel and Tom Liefheid for their extensive time and effort. I greatly appreciate their commitment and contributions to this project. Their presence during our weekly meetings and availability throughout the weeks is something I don't take for granted and has been fundamental in the success of my graduation project.

Finally, I want to give a special thanks to Yaping Luo for her tremendous support throughout the last year. From the day I applied for this graduation project to its completion she has gone above and beyond to ensure the success of my graduation project. For this I am deeply thankful.

Abstract

The successful migration to microservice based applications by large companies such as Netflix and Amazon popularized the idea of this architectural style and it has led to an industry wide adoption starting around 2015. The problem with microservice architectures is the fact that identifying architectural anti-patterns has become increasingly difficult. This is due to the continuous change of the architecture and the overall growth of applications in size and complexity. Current visualization and anti-pattern detection techniques are not adequate enough to help developers in identifying anti-patterns. Therefore, the goal of this study is to develop a methodology for detecting anti-patterns in microservice architectures based on execution traces.

Our approach is to first reconstruct the microservice architecture from execution traces and represent the architecture as a graph. Then we compute node level metrics based on graph algorithms. Finally, we link the computed metrics to a set of anti-patterns.

The contribution of this research is threefold. The first contribution is the approach of distributed tracing for architecture recovery. The second contribution is the application of concepts and algorithms, developed within the field of graph theory for automatic identification of architectural anti-patterns. The third contribution is the application of our approach on a large scale (400+ services) industry application.

Contents

List of Figures

List of Tables

Glossary

APM Application Performance Management. 9, 41, 44

ATD Architectural Technical Debt. 4, 10, 11, 13

BVC Business Value Chain. 23–25, 30, 31, 44

DT Distributed Tracing. 4, 8, 9, 12–14, 21, 41, 42, 46, 48

MSA Microservice Architecture. 1, 2, 4, 9, 11–14, 19, 21, 26, 42, 45, 47, 48

RPC Remote Procedure Call. 4

SDG Service Dependency Graph. 4, 9, 13, 35

SOA Service Oriented Architecture. 12

TD Technical Debt. 10, 13

Chapter 1

Introduction and Research Question

This chapter introduces the thesis' motivation and provides the context for the defined problem of this graduation thesis. Furthermore, our contributions are highlighted and the research question is defined. Finally, this chapter provides an overview of this thesis' structure.

1.1 Introduction

The successful migration to microservice based applications by large companies such as Netflix and Amazon popularized the idea of this architectural style and has led to an industry wide adoption starting around 2015. The idea of the microservice architectural style is to develop a single application as a suite of small components called microservice or simply service. Each service runs in its own process and communicates often via a REST-ful HTTP connection. These services are built around business capabilities, emphasizing loose coupling and high cohesion, and are independently developed and deployed by a single team [1]. The switch towards a Microservice Architecture (MSA) has enabled companies and organizations to deliver software more efficiently, as smaller developer teams can work independently from one another. Additionally, it can increase the resilience and the reliability as well as improve the scalability and maintainability of the overall application.

Even though microservices provide several advantages compared to a monolithic application, this architectural style introduces its own set of challenges and potential drawbacks. For example, applications need to be able to tolerate the failure of services, since a service might be unavailable due to high load, maintenance or other factors. Hence, all other services that make calls to this service have to handle the situation in which they do not get a response. This leads to increasing complexity in each individual service. Consequently, the chances of suboptimal implementations increase, not only within the individual services, but possibly also in how the services communicate with each other. In MSAs the inter-service communication is closely linked to the application's overall architecture. Any suboptimal inter-service communication solutions are suboptimal architectural solutions (also called architectural anti-patterns, or simply anti-patterns). The creation of anti-patterns is not always a conscious decision, but can be the result of teams independently working on separate parts of the application. In practice, the presence and impact of anti-patterns might not always be immediately detectable. Nonetheless, it is important to find and remove anti-patterns. If implemented incorrectly, a microservice based application can suffer from numerous and significant drawbacks. For example, failing to maintain loose coupling and high cohesion means services cannot be deployed separately offsetting any scalability, maintainability or ease of deployment gains. Additionally, it can decrease the application's resilience as well as runtime performance. Due to the communication complexity debugging becomes harder compared to monoliths. The problem is that identifying anti-patterns within MSAs becomes increasingly difficult as their architectures are continuously changing and applications are ever increasing in size and complexity.

Currently, visualization tools have been a popular approach to get a holistic understanding of an application. Open-Source and commercial tools such as Vizceral [2] Buoyant [3] and AHAS [4] are currently being used and developed within the industry. The academic community has developed several visualization tools for recovering an application's architecture [5, 6, 7, 8]. The larger an application becomes, the less information these types of tools provide. This is due to the visualization becoming too cluttered and therefore making it difficult to identify anti-patterns. Thus, for large scale applications it is necessary to filter and selectively show only parts of the overall architecture to prevent the aforementioned cluttering. Another approach to better understand a MSA is to use concepts and algorithms developed within the field of graph theory [9, 10, 11]. These algorithms compute metrics for each service which can be used for evaluating the architectural design or to detect architectural anti-patterns. However, the computed metrics are difficult to interpret and do not provide definitive results. In the mentioned studies the metrics are computed for relatively small (< 70 services) applications which are created as a testing benchmark and therefore do not organically evolve over time.

1.2 Research Question

The contribution of this research is threefold. The first contribution is the approach of using distributed tracing for architecture recovery. The second contribution is to utilize concepts and algorithms, developed within the field of graph theory for the automatic identification of architectural anti-patterns. The third contribution is the application of these on a large scale (> 400 services) industry application. The results of this research can help developers gain a better overall understanding of their application as well as help pinpoint areas within the application's architecture with suboptimal structures. Furthermore, developers can use the tool to reason about architectural changes and make better estimates of the potential impact to the architecture. To verify our contribution we formulate the following research question.

Can anti-patterns in microservice architectures be automatically identified from distributed tracing?

1.3 Thesis Structure

The thesis has seven chapters, and it is organized as follows:

Chapter 1: Introduction
This chapter introduces the thesis' problem and its context. Furthermore, the contribution and research question are defined.

Chapter 2: Background and Related Work
This chapter presents the background knowledge and fundamental concepts used in this thesis. It also highlights related work in the field of automatic anti-pattern detection.

Chapter 3: Methodology
This chapter presents the approach used for the automatic detection of anti-patterns within microservice based applications.

Chapter 4: Visualization Tool
This chapter presents the dashboard developed alongside the anti-pattern detection approach for presenting and visualizing the results. Additionally, this chapter presents the results from applying the approach in a large scale industry application.

Chapter 5: Case Study and Results
This chapter outlines the performed case study and presents the results.

Chapter 6: Discussion, Future Work and Threats to Validity
This chapter discusses the results from our approach and their implications for ING. Furthermore, future work possibilities are discussed and the threats to validity are addressed.

Chapter 7: Conclusion
This chapter summarizes this graduation thesis by drawing a final conclusion.

Chapter 2

Background and Related Work

This chapter presents the background knowledge and fundamental concepts used in this report. Furthermore, related work in the field of anti-pattern detection in MSAs is highlighted. In the first section the concept of distributed tracing is explained. In the second section the Service Dependency Graph is introduced and how it can be recovered from different data sources. The third section explains the concept of Architectural Technical Debt (ATD) and the difference between ATD and anti-patterns. In the last section the concept of network analysis is presented and the related work regarding automatic anti-pattern detection is highlighted.

2.1 Distributed Tracing

Distributed Tracing (DT), also called distributed request tracing, is a method used to profile and monitor applications. Distributed applications, such as ones based on the microservice architecture, make root cause analysis of application failures more complex compared to a monolithic application. For example, a front-end service might query a back-end service to fetch data requested by the user. The back-end service in return might query potentially hundreds of other services, which again might call more services, to aggregate the information requested by the user. This stream of requests continues until all data is gathered and processed. Continuing the example, we assume the first back-end service encounters a failure and cannot serve the front-end the requested data. The failure is detected by the team responsible for the first back-end service. However, the failure is not caused by their service, as it does not receive data from its downstream service. But the team has no access to the downstream service that does not return the requested data. As a result, the team cannot verify if that service is the root cause or if the failure arises further downstream. DT solves this problem by keeping a record and providing an overview of all service to service calls.

While not being the first tracing tool [12, 13, 14], the design paper for Google Dapper [15] was a significant publication for popularizing the idea of DT. In the paper Siegelman et al. describe the development of a production-grade tracing tool, with three key goals in mind.

1. **Low overhead:** The tracing system should have negligible performance impact on running services. In highly optimized services small monitoring overheads might compel the deployment teams to turn the tracing system off.
2. **Application-level transparency:** Tracing should not require deployment teams to actively maintain the tracing infrastructure.
3. **Scalability:** It needs to handle Google's scale for at least a few years.

Dapper and most of its descendants work with an annotation-based scheme. This means applications or middleware have to explicitly tag every call between services with a global identifier that links these message records back to the originating request. All messages belonging to the same request are grouped in a trace. In the case of Dapper, a trace is a tree of Remote Procedure Calls (RPC).

The data model is also suited to trace SMTP sessions as well as HTTP requests. Traces are modeled using trees, spans and annotations. In a trace tree, as shown in Figure 2.1, the tree nodes are basic units of work which are referred to as spans and the edges represent a relationship between a span and its parent span. Spans without a parent id are called root spans.

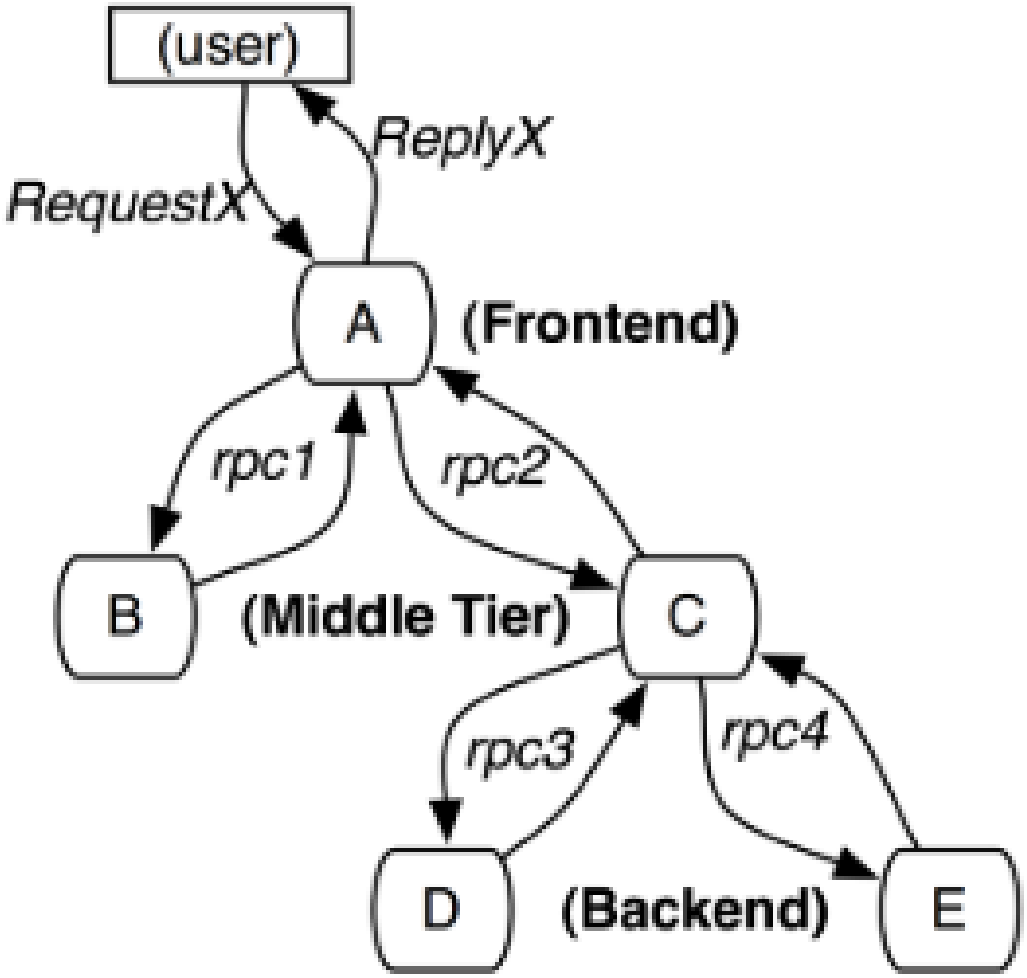

Figure 2.1: The path taken through a simple serving system on behalf of user request X. The letter-labeled nodes represent processes in a distributed system. | Reprinted from B. H. Sigelman et al., "Dapper , a Large-Scale Distributed Systems Tracing Infrastructure," Google Res., no. April, p. 14, 2010, [Online]. Available: link.

Figure 2.2 shows the structure of a trace. The trace is made up of five spans. The first span called *Frontend.Request*, starting at timestamp 20, has no parent-id and span-id one. The second span is called *Backend.Call* and has span-id two and parent-id one. This span starts at timestamp 21 and ends at timestamp 23. The next span is *Backend.DoSomething* starting at timestamp 24. This span itself invokes two new spans with span-id four and five both running in parallel. At timestamp 29, spans four and five have both returned back to span three. At that timestamp, span three is complete and can return back to span one. The *Frontend.Request* span now has performed its unit of work and stops at timestamp 31. For each span, the span's name and its start as well as end times need to be recorded by Dapper's RPC library instrumentation. Each service has to implement the RPC library separately and ensure the correct annotations are made for each span. This is a drawback of Dapper but also allows the service's developers to extend their spans with annotations to provide more information about the trace.

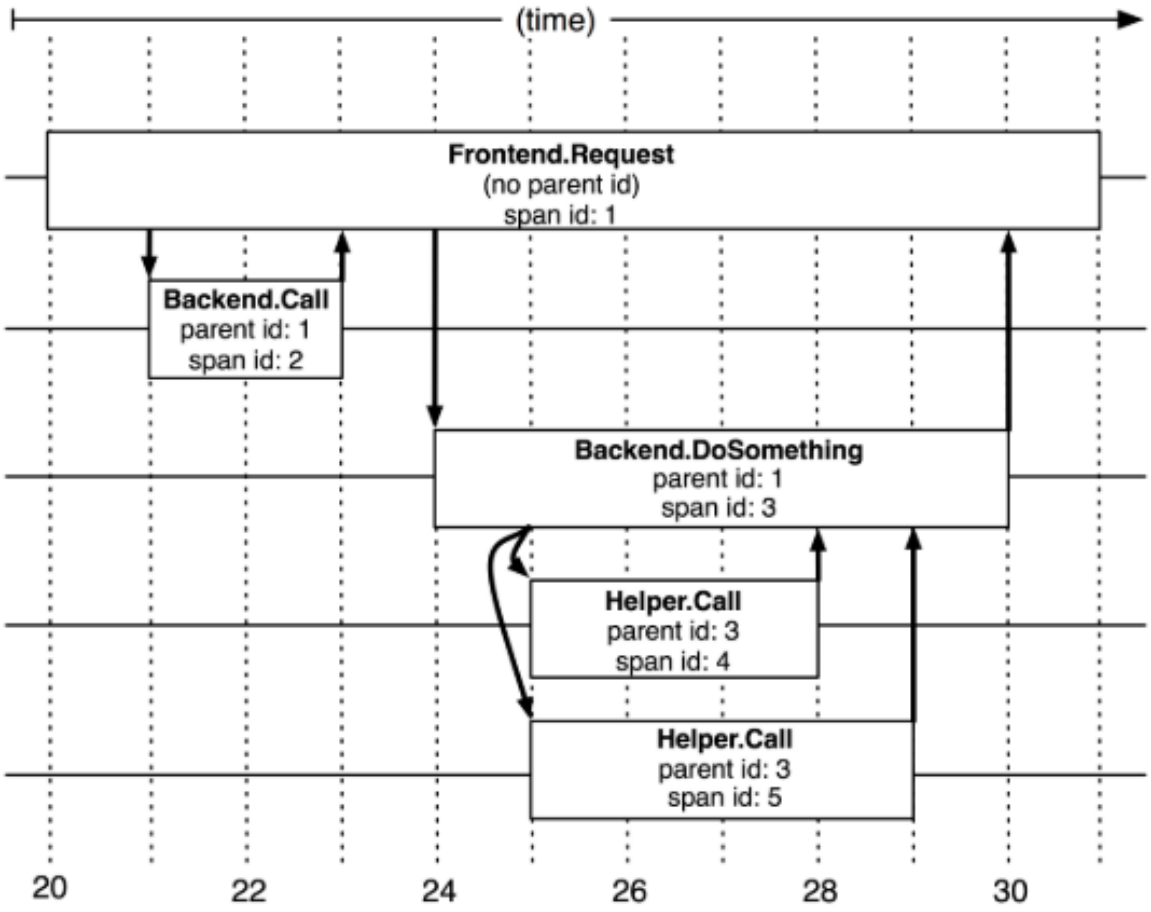

Figure 2.2: The causal and temporal relationships between five spans in a Dapper trace tree. | Reprinted from B. H. Sigelman et al., "Dapper , a Large-Scale Distributed Systems Tracing Infrastructure," Google Res., no. April, p. 14, 2010, [Online]. Available: link.

Dapper stores the recorded spans in local log files of the service. Dapper daemons then pull the information from all services that log their spans and store the information in a Dapper Bigtable (See Figure 2.3). In a Dapper Bigtable, a trace is stored as a single row, with each column corresponding to a span. Bigtable supports sparse table layouts, as individual traces can have an arbitrary number of spans. Figure 2.3 shows a small table representing a Dapper Bigtable. There are three traces and five spans stored in the table. If a span is not part of a trace, its cell for that trace is set to nil. This example shows that most cells in the table will be nil, which is referred to as a sparse table.

One descendant of Dapper is Zipkin. It is an open source version of Google's Dapper that was further

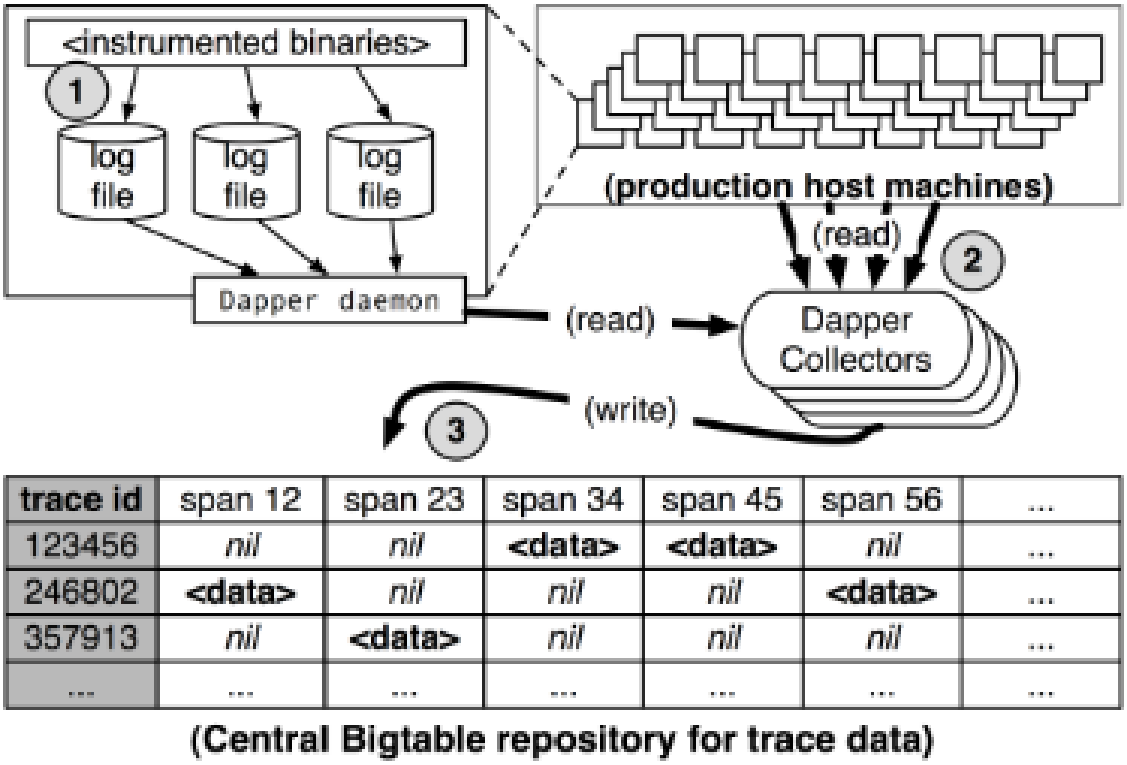

Figure 2.3: An overview of the Dapper collection pipeline. | Reprinted from B. H. Sigelman et al., "Dapper , a Large-Scale Distributed Systems Tracing Infrastructure," Google Res., no. April, p. 14, 2010, [Online]. Available: link.

developed by Twitter. Similar to Dapper, services need to implement a tracer or instrumentation library to report trace data to Zipkin. The most popular ways to report data to Zipkin are via HTTP or the Kafka event streaming platform [1]. The data is then sent to the user interface and stored in-memory. Data can also be persisted with a supported backend such as Apache Cassandra or Elasticsearch. An alternative to Zipkin is Jaeger, which was created at Uber. Jaeger's architecture differs from Dapper and Zipkin as it includes a client that emits traces to an agent. The agent listens for inbound spans and routes them to the collector, which then validates, transforms and persists the spans. Jaeger uses a distributed architecture which makes it more scalable compared to Zipkin. Jaeger also differs in the way it collects data. It does not collect every trace and span generated, but takes a sample of the monitored data. This approach not only allows Jaeger to handle sudden surges in traffic, but increases Jaeger's overall performance. However, currently the future of the Jaeger project is somewhat uncertain. In 2019 OpenTelemetry was announced as a new CNCF [2] sandbox project resulting from a merger of OpenTracing [3] and OpenCensus [4]. OpenTelemetry and Jaeger have some overlap in the problems they solve. These are the client libraries, agent and collector. For these areas the Jaeger team is planning on collaborating with OpenTelemetry and ideally deprecating the respective Jaeger components to avoid redundant software [16].

[1] https://kafka.apache.org/
[2] https://www.cncf.io/
[3] https://opentracing.io/
[4] https://opencensus.io/

The Distributed Tracing implementation within ING is based on the OpenTracing specification. The OpenTracing specification defines a span as follows. A **span** is a 6-tuple (n, s, f, T, L, sc):

- **n:** the name of the operation
- **s:** the start timestamp
- **f:** the finish timestamp
- **T:** a set of span tags T as key value pairs $< k, v >$, where k is a string and v is a string, Boolean or numeric type.
- **L:** a set of span logs L as key value pairs $< t, < k, v >>$, where the key t is a timestamp and the value is another key value pair $< k, v >$. k must be a string and v can be of any type.
- **sc:** the SpanContext sc is a tuple (R, B). R is a set of references as key value pairs $< k, v >$, where k is $\in \{'trace\_id', 'span\_id'\}$ and v is a string. B is a set of Baggage items as key value pairs that apply to the given span, its SpanContext, and all spans which directly or transitively reference the local Span.

2.2 Service Dependency Graphs

To better understand microservices based architectures, researchers and companies build tools to visualize microservices and their dependencies. Generally, the MSA is represented in a Service Dependency Graph (SDG) that visualizes services as nodes and their dependencies as edges. The graphs can be generated from several different data sources, the most common being a service discovery mechanism and tracing data.

2.2.1 Tracing Data

One SDG recovery method is to use trace information from a Distributed Tracing system. For this all traces need to be assembled. This is done by grouping all spans by their trace-id and then connecting the spans according to the parent span-id recorded in each span. Once this is done, relations can be set according to the following pattern. If a client span from service $s1$ is followed by a server span from service $s2$ a relation between $s1$ and $s2$ can be set. These server to server relations can then be exported to a graph database and be used to visualize the SDG. Since the SDG can be recovered from Distributed Tracing data, several Application Performance Management (APM) tools such as AWS X-Ray [5], Datadog [6], Elastic APM [7] offer such visualizations as part of their license-based product. An open source alternative is Netflix's Vizceral [2] which provides an intuitive overview of an application's architecture as well as it's real-time traffic flow.

2.2.2 Service Discovery Mechanism

Another SDG recovery method is to use a service discovery mechanism. Within a MSA, services have to communicate with each other. However, since services can be dynamically added or removed the IP and port numbers of services can change frequently. Therefore, to allow services to find each other and communicate is a problem that needs to be addressed. In 2012 Netflix open-sourced its service registry tool called Eureka [17] which originally solves the need for mid-tier load balancing but also is a popular service discovery mechanism. Figure 2.4 shows a typical Eureka deployment, representative for also many other service registry tools. Per region there is one Eureka cluster which only knows about the service instances in its region. Each service registers with Eureka and then sends a heartbeat every 30 seconds to the Eureka cluster. If a service fails to send three heartbeats, it is taken out of the service registry. The registration information and the renewals are replicated to all the Eureka nodes in the cluster. Services from any zone can look up the registry information to locate their required services and make remote calls. Since the service registry knows which services are currently active and which services communicate with each other, retrieving the SDG is relatively simple.

[5] https://aws.amazon.com/xray/
[6] https://www.datadoghq.com/
[7] https://www.elastic.co/apm/

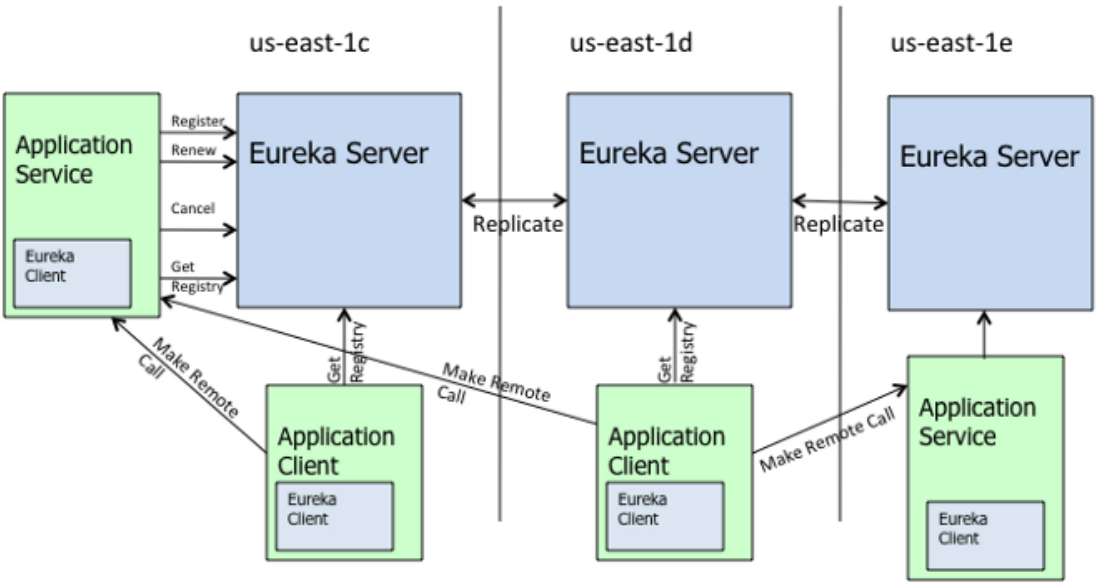

Figure 2.4: The high level architecture of Eureka. Taken from Netflix Tech Blog.

2.3 Architectural Technical Debt

Architectural Technical Debt (ATD) is a type of Technical Debt (TD) consisting of suboptimal architectural solutions, that provide short-term development benefits but increase overall development costs in the long run. TD is comprised of three main concepts, which are the debt, its interest and the principal [18]:

- **Debt:** A sub-optimal solution that provides short-term benefits but which creates future interest payment.
- **Interest:** The extra cost that needs to be paid due to the accumulated debt. This extra cost can be of any form. For example, longer development time, deployment dependencies, performance decrease, etc.
- **Principal:** The principal can be two things. On the one hand it can be the cost of developing a better solution that prevents debt. On the other hand it can be the cost of refactoring an existing sub-optimal solution, removing the existing debt.

In some cases it can be cheaper to accumulate debt instead of removing it [19]. This is the case when the interest is less than the principal. If the interest is more costly than the principal, the debt should be removed. However in practice, it is hard to estimate the actual costs of the principal and the interest. Therefore, generally speaking it is desirable to identify and remove ATD, since problems in the architecture may slowdown new functionalities and raise the related costs.

2.3.1 Anti-Patterns

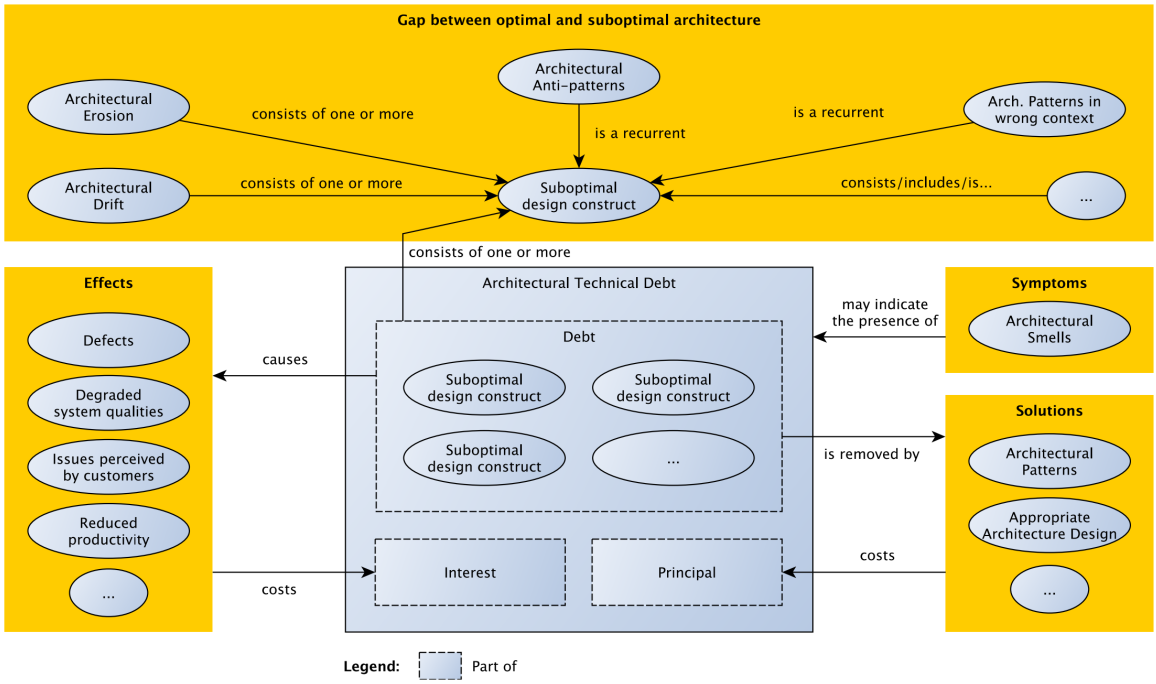

Figure 2.5: ATD and the related concepts | Reprinted from S. S. de Toledo, A. Martini, and D. I. K. Sjøberg, "Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study," J. Syst. Softw., vol. 177, no. April, p. 110968, Jul. 2021, doi: 10.1016/j.jss.2021.110968. link.

The research on anti-patterns in MSA is still in its infancy and there is no well-defined taxonomy [20]. Mo et al. [21] define architectural anti-patterns as repeatable suboptimal design constructs that violate design principles and increase the likelihood of having bugs and changes. Toledo, Martini and Sjøberg [22] follow the definition of Mo et al. [21] by defining anti-patterns as repeatable suboptimal design constructs. However, they expand on the definition and define suboptimal design constructs to consist of one or more occurrences of Architectural Technical Debt. Their definition of ATD and related concepts is shown in Figure 2.5.

Tighilt et al. [23] aggregate a set of 16 anti-patterns for MSAs from previous literature. They organize their set of anti-patterns into four categories, based on the development cycle of a microservice-based system. The categories are: design, implementation, deployment and monitoring. For each anti-pattern they explain among other things its general form, list the symptoms, mention the consequences and highlight a refactoring solution. For example, a design anti-pattern is the Hardcoded Endpoints anti-pattern. Its general form is: Microservice IP addresses, ports, and endpoints are explicitly/directly specified in the source code. The symptoms are: Hardcoded endpoints anti-pattern show via the presence of IP addresses or fully qualified domain names in source code, configuration files, or environment variables. The consequences are: When there are many microservices in a system, it becomes more difficult to track all the endpoints and URLs. Running multiple instances of a microservice with a load-balancer becomes impossible. Changing the IP address or port number of a microservice requires changing and redeploying other microservices. The recommended solution is: Service discovery prevents hardcoding IP addresses and port numbers. It tracks microservices endpoints and ease the communications among microservices [23].

2.4 Network Analysis

Networks are used to denote a relational perspective on data and are used in a large variety of domains (e.g., electrical circuits, communication networks, bioinformatics, etc.) and are commonly used for modeling complex systems. The notation of graphs fit this conceptualization well and is therefore commonly used to formalize network analysis concepts. A graph is defined as follows $G = (V, E)$ with V being a finite set of vertices (nodes) and E being a set of edges (links). Edges are defined by pairs of vertices u and v, such that $u \in V$ and $v \in V$. In a directed graph, the edges are specified by ordered pairs $\langle u, v \rangle$ where u is the source vertex and v is the target vertex. In an undirected graph, the edges are specified as sets such that $\{u, v\} = \{v, u\}$. There are various metrics that quantify the properties of a vertex. For example, the degree centrality is defined to be the number of edges in E that connect to a vertex v and is denoted by $deg(v)$. In a directed graph a further distinction can be made between the in-degree, being the number of edges with v as its target, and an out-degree, being the number of edges with v as its source.

Based on these vertex properties a large number graph algorithms are being developed [24]. Graph algorithms can be categorized into three types: path-finding, centrality, and community detection. Centrality algorithms are are used to identify the most critical nodes within a graph. For example, the degree centrality algorithm does this by counting the number of edges a node has. Another centrality algorithm is the betweenness centrality which counts the number of shortest paths passing through a node. Community detection algorithms look at the groups and partitions. For example, the weakly connected communities algorithm finds groups where each node is reachable from every other node in that same group. Path-finding algorithms try to find the shortest path between two nodes. However, we do not use any path-finding for our anti-pattern detection methodology.

2.4.1 Service Metrics for Architecture Evaluation

From these graph algorithms a large number of metrics are being developed for measuring and evaluating services and service-based systems. Bogner, Wagner and Zimmermann [25] provide an overview of the metrics in current literature with focus on maintainability and analyze their applicability for microservice systems. Except for the centralization metrics the majority of the identified metrics are found to be also applicable for microservice systems. Bogner, Wagner and Zimmermann present a maintainability model for service-oriented and microservice systems [26]. They first identify quality attributes desired for the system architecture and then determine applicable metrics to measure the maintainability. For example, the coupling degree of a service can be measured considering the number of consumed services, the number of consumers and the number of pairwise dependencies in the system. The granularity is measured with the number of exposed interface operations.

2.4.2 Microservice Anti-Pattern identification

Similar to the service metrics mentioned above, graph algorithms can also be used for identifying patterns that are undesirable in the structure of software (so-called 'anti-patterns'). Anti-patterns are poor solutions to recurring design problems not limited to microservices. There is a large body of research in anti-pattern detection [27] [28] [29] [30], however, these approaches are not suited to detect anti-patterns in MSAs. Previous work on Service Oriented Architecture (SOA) also proposes the automatic detection of anti-patterns. Nayrolles, Moha and Valtchev [31] propose the equivalent to our approach for SOA based applications. They use execution trace mining to compute a set of metrics that they link to so called rule cards. These rule cards capture the expected symptoms of possible anti-patterns. The conceptual approach of our work is similar, however, we implement this for MSAs. Furthermore, we compute our metrics with graph algorithm and provide a visualization of the architecture. Pigazzini et al. [10] extend the Arcan [32] tool for the automatic detection of Shared Persistence, Hard-Coded Endpoints, and Cyclic Dependency anti-patterns. Our study differs from this study as we detect a different set of anti-patterns. Furthermore, we conduct our evaluation on a single large application (> 400 services) instead of five smaller applications (< 20 services), use graph algorithms for anti-pattern detection and we recover the architecture from Distributed Tracing data. Kobyliński and Sobczak [9] use a very similar set of graph algorithms as we do to detect the distributed monolith and the cycling dependency anti-pattern. However, they perform a manual

analysis to link the metrics to the anti-patterns. In this thesis we detect a different set of anti-patterns and implement a computation based approach for automatically linking graph algorithm metrics to anti-patterns.

2.5 Summary

Distributed applications, such as ones based on the microservice architecture, make root cause analysis of application failures more complex compared to a monolithic application. To allow developers to perform root cause analysis, Distributed Tracing, also called distributed request tracing, has been developed. DT is used to profile and monitor applications, but lately it has also been used to recover Microservice Architectures.

Microservice Architectures are commonly represented in a Service Dependency Graph that visualizes services as nodes and their dependencies as edges. The graphs can be generated from several different data sources. The most common is a service discovery mechanism and Distributed Tracing data. SDG recovery with Distributed Tracing is done by grouping all spans by their trace-id and then connecting the spans according to the parent's span-id recorded in each span. Relations can then be set according to the following pattern. If a client span from service $s1$ is followed by a server span from service $s2$, a relation between $s1$ and $s2$ is set.

Architectural Technical Debt is a type of Technical Debt (TD) consisting of suboptimal architectural solutions, that provide short-term development benefits but increase overall development costs in the long run. Toledo, Martini and Sjøberg [22] follow the definition of Mo et al. [21] by defining anti-patterns as repeatable suboptimal design constructs. However, they expand on the definition and define suboptimal design constructs to consist of one or more occurrences of Architectural Technical Debt. Tighilt et al. [23] have aggregated a set of 16 anti-patterns for MSAs from previous literature. They organized their set of anti-patterns into four categories, based on the development cycle of a microservice-based system. The categories are: design, implementation, deployment and monitoring.

Networks are used to denote a relational perspective on data and are used in a large variety of domains. The notation of graphs fits this conceptualization and therefore is commonly used to formalize network analysis concepts. The research body on graphs is extensive and over the years many graph algorithms have been developed to gain insights into graphs. Graph algorithms can be categorized into three types: path-finding, centrality, and community detection. In recent years, graph algorithms have been used to detect anti-patterns in MSAs.

Chapter 3

Methodology

This chapter describes the methodology used to answer the research question. The structure of our methodology is based on a general data collection, data processing and analysis approach. We first describe the data collection process. Section 3.2 highlights the data processing. In Section 3.3 the matching of metrics to anti-patterns is explained. Finally, Section 3.4 details the process of the expert validation.

3.1 Data Collection

The first step of our methodology is the data collection to recover the application's architecture. The practice of architecture recovery is extensively studied and several different approaches are compared [33]. These architecture recovery approaches are all developed for monolithic applications and use a static analysis of the application's source code. However, these approaches cannot be used for the Microservice Architecture, as a static analysis cannot recover the dependencies between services. For example, MicroART [8] uses a combination of static and dynamic architecture recovery. In a first step, given a GitHub repository link, it searches for: (i) a Docker-compose file specifying the system components interactions (e.g., container name and build-path), and (ii) a Docker-file for each microservice from which it retrieves specific listening ports and exposed ports. In a second step, MicroART queries the Docker environment at runtime in order to retrieve the IP address and the network interface used by each microservice to recover the service dependencies.

This two-step approach highlights a fundamental problem of static architecture recovery for microservice based applications. Since it is not able to recover service dependencies, it requires an analysis step at runtime. Additionally, the architecture of microservice based applications is continuously changing as teams add and remove services, which a static analysis is not able to detect without rerunning the analysis step. Another problem that arises in larger companies, is the fact that the source code is distributed throughout many teams and might not be accessible in a central way. Therefore, it might be challenging to find all source code artifacts for the analysis. MICROLYZE [7] uses a fully dynamic approach and rebuilds the microservice infrastructure that is registered in a service discovery tool like Eureka [1] or Consul[2]. Next, it uses Distributed Tracing to recover the service dependencies. This approach is better suited for continuously recovering the application's architecture, however it requires a service discovery tool to be present and needs two steps to recover the architecture.

ING's code base consists of several thousand repositories spread over multiple subsidiary companies with multiple version control providers. This means getting access all source code artifacts is impractical and time consuming. Furthermore, there does not exist a central service discovery tool for ING's MSA. Therefore, our approach needs to solely rely on distributed tracing data to recover the application's architecture. In the first step we continuously listen for two days for new spans on a Kafka [3] event stream and group spans into buckets by their trace-id. Each bucket has a timer that is

[1] https://github.com/Netflix/eureka
[2] https://www.consul.io/
[3] https://kafka.apache.org/

reset whenever a new span is added to the bucket. When the timer for a bucket runs out we assume that the trace is complete and we continue to the next step.

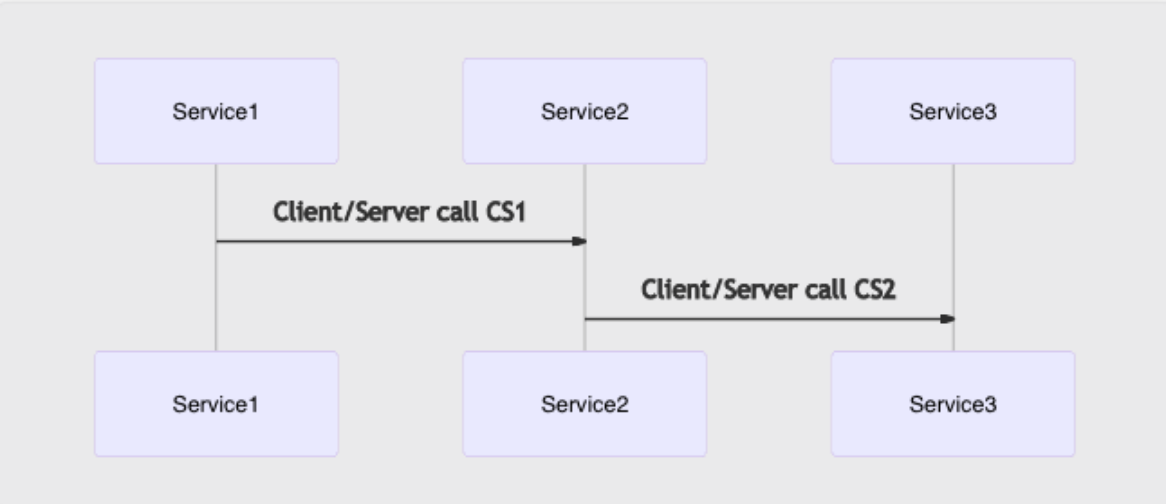

Figure 3.1: Sequence of three Client-Server-Spans between services forming a single trace.

Given a trace as shown in Figure 3.1, we have a total of four spans. Each service to service call has a client span and a server span summarized as *Client/Server call CS*. In this example, the client span in *Client/Server call CS*1 has id 1 and no parent id and the server span has id 2 and parent id 1. Similarly, the client span in *Client/Server call CS*2 has id 3 and parent id 2 and the server span has id 4 and parent id 3. Each server span in every *Client/Server call* contains the service's name as well as the name of the operation that is called. Thus we can now set the link *Operation*1 of *Service*1 consumes *Operation*2 of *Service*2 and store this information in a graph database. The database schema is shown in Figure 3.2

Figure 3.2: Graph database schema after trace aggregation.

3.2 Data Processing

After the data collection we perform two processing steps. In the first step we add direct service to service relations. That is, if *Service*1 $-HAS\_OPERATION-$ *Operation*1 and *Operation*1 $-CONSUMES-$ *Operation*2 and *Service*2 $-HAS\_OPERATION-$ *Operation*2 we set a *CALLS* relation between *Service*1 and *Service*2. The final schema of our database is shown in Figure 3.3.

Figure 3.3: Final graph database schema after second processing step.

The second processing step is to compute a set of metrics, based on a subset of the service-based maintainability metrics [25], and graph theory based metrics [34]. For each service in the database we compute the metrics shown in Table 3.1

Table 3.1: Abbreviation and description of metrics

Abbrev.	Name	Description
AIS	Absolute Importance of the Service	Number of services which depend on a service.
ADS	Absolute Dependence of the Service	Number of other services a service depends on.
WSIC	Weighted Service Interface Count	Weighted number of exposed interfaces or operations per service (all weights = 1.0).
SIUC	Service Interface Usage Cohesion	Quantifies cohesion of a service based on the number of operations invoked by every client.
BTW	Betweenness Score	Calculates unweighted shortest paths between all pairs of nodes in a graph. Each service receives a score, based on the number of shortest paths that pass through the node.
LCC	Local Clustering Coefficient	The local clustering coefficient C_n of a node n describes the likelihood that its neighbors are also connected.
CSD	Cyclic Service Dependencies	Indicates if a service has a cyclic dependency

For the following metric definitions we consider a graph $G = (V, E, L)$ where V is the set of all nodes in our graph database, $E \subseteq V \times V$ is the set of all edges in our graph database and L is a labeling function which maps from a node or an edge to the corresponding label. The set of node labels is defined as follows $L(node) = \{Service, Operation\}$ and the set of edge labels is defined as follows $L(edge) = \{CALLS, HASOPERATION, CONSUMES\}$.

We define s to be a node such that $s \in V$ and $L(s) = Service$. Furthermore, we define the following functions, where $v \in V$ and $L(v) = Service \vee Operation$:

- $deg_{in\_services}(v)$ returns the set of nodes N in V that have a directed edge $e = \langle n, v \rangle$ with n as source and v as target such that $n \in N, n \in V$, $L(n) = Service$ and $L(e) = CALLS \vee HASOPERATION$.
- $deg_{out\_services}(v)$ returns the set of nodes N in V that have a directed edge $e = \langle v, n \rangle$ with v as source and n as target such that $n \in N, n \in V$, $L(n) = Service$ and $L(e) = CALLS \vee HASOPERATION$.
- $deg_{in\_operations}(v)$ returns the set of nodes N in V that have a directed edge $e = \langle n, v \rangle$ with n as source and v as target such that $n \in N, n \in V$, $L(n) = Operation$ and $L(e) = HASOPERATION \vee CONSUMES$.
- $deg_{out\_operations}(v)$ returns the set of nodes N in V that have a directed edge $e = \langle v, n \rangle$ with v as source and n as target such that $n \in N, n \in V$, $L(n) = Operation$ and $L(e) = HASOPERATION \vee CONSUMES$.

AIS is computed as follows:

$$AIS(s) = deg_{in\_services}(s) \tag{3.1}$$

ADS is computed as follows:

$$ADS(s) = deg_{out\_services}(s) \tag{3.2}$$

WSIC is computed as follows:

$$WSIC(s) = deg_{out\_operations}(s) \tag{3.3}$$

SIUC is computed as follows:

$$SIUC(s) = \frac{\sum_{i=1}^{WSIC(s)} deg_{out\_operations}(o_i)}{\sum_{i=1}^{WSIC(s)} deg_{in\_operations}(o_i) * WSIC(s)} \tag{3.4}$$

Where $\sum_{i=1}^{WSIC(s)} deg_{out\_operations}(o_i)$ is the sum of outgoing operation dependencies of all operations of s and $\sum_{i=1}^{WSIC(s)} deg_{in\_operations}(o_i)$ is the sum of incoming operation dependencies of all operations of s.

BTW is computed as follows [35]:

$$BTW(s) = \sum_{n \neq v} \delta(n, v|s) \tag{3.5}$$

Where $\delta(n, v|s)$ is the length of the shortest path connecting the services n and v that goes through s.

LCC is computed as follows:

$$LCC(s) = \frac{2T(s)}{deg_{services}(s)(deg_{services}(s) - 1)} \tag{3.6}$$

Where $T(s)$ is the number of triangles of service s [36] and $deg_{services}(s) = deg_{in\_services}(s) \wedge deg_{out\_services}(s)$.

CSD is computed as follows:

$$CSD(s) = \begin{cases} 1 & \text{if } s \text{ has a cyclic dependency} \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

3.3 Metrics to Anti-Pattern Matching

For our data analysis step we compute the weighted euclidean distance between each service and the following set of anti-patterns.

- **Mega-Service:** A service that serves multiple purposes and has high number of lines of code, modules or files, as well as a high in degree.
- **Nano-Service:** A service that is too fine-grained such that its communication and maintenance efforts outweigh its utility.
- **Bottleneck-Service:** A service that is being used by too many consumers and therefore becomes a bottleneck and single point of failure.
- **Ambiguous-Service:** A service that implements a large amount of functionality, but provides only one public operation to invoke all of it. Accepted requests are internally forwarded to various methods.

To compute the euclidean distance we perform a min-max normalization of all metrics for all services and then define vectors for each anti-pattern based on the anti-pattern symptoms characterized in [23] and [37]. An alternative for defining the anti-pattern vectors can be a multiple regression on

Table 3.2: Anti-Pattern Vectors

	AIS	ADS	WSIC	SIUC	BTW	LCC	CSD
Mega Service	1*	1	0	0	-	-	-
Nano Service	0	0	0*	-	-	-	1*
Bottleneck	1	1	0	-	1*	1	-
Ambiguous Service	1	1	0*	0*	-	-	-

the attributes or similarly the GainRatioAttributeEval from the WEKA [4] tool. We do not use this approach as we do not have a ground truth of which services' metrics express an anti-pattern.

Tighilt et al. [23] describe the mega microservice as a service that serves multiple purposes and has high number of lines of code, modules or files, as well as a high fan-in. Since the high fan-in is explicitly described as one of the symptoms we expect the number of incoming dependencies (AIS, see Table 3.1) to be high and we increase the weight for this metrics. Furthermore, as a mega service serves multiple purposes we expect the number of outgoing dependencies (ADS) to be high and the usage cohesion (SIUC) to be low.

Tighilt et al. [23] state that a nano-service anti-pattern exists if the system has a large number of microservices; (2) microservices exchange a lot of information; (3) cyclic dependencies exist. Bogner [37] describes the nano-service anti-pattern as a small or too fine grained service with only one or very few operations. Therefore, we expect the number of operations metric (WSIC) to be low and the CSD metric to be 1 as this indicates cyclic dependencies. We give both of these metrics a higher weight factor. To approximate the small or fine grained functionality of a nano-service we set the expected AIS and ADS to 0 as we expect a service with little functionality to be called by fewer other services.

Bogner [37] states that an ambiguous service has a large amount of functionality, but provides only one public operation to invoke all of it. We therefore expect the WSIC metric to be low and give it a higher weight. Again we approximate the large functionality with high AIS and ADS but also expect the usage cohesion (SIUC) to be low as probably many different services will invoke different functionality of the service.

The bottleneck anti-pattern is defined by Bogner as a service that is being used by too many consumers and therefore becomes a bottleneck and single point of failure. We therefore expect AIS, ADS, BTW and LCC to be high. Additionally, to better capture the single point of failure aspect we give the betweenness score a higher weight factor.

The vectors for the anti-patterns can be found in Table 3.2. Cells with a zero or one indicate that we expect the value of that metric to be low or high respectively after the min-max normalization. The weight for the distance computation of each metric is one. Cells that are also marked with a * have twice the weight when computing the euclidean distance, as these metrics are identified to be characteristic of the given anti-pattern as mentioned above. Cells marked with a - indicate that the respective metric is not applicable for the given anti-pattern. Thus, in the distance computation, the weight of the metric is set to zero. The distance computation can be found in Formula 3.8, where $pattern_i$ and $service_i$ is the $i-th$ metric of the anti-pattern vector or the microservice respectively. w_i is the metric's assigned weight as just detailed.

$$d(pattern, service) = \sqrt{\sum_i w_i * (pattern_i - service_i)^2} \tag{3.8}$$

As the anti-pattern distances are computed with different vectors, which have different value ranges, we cannot directly compare the distances with each other. A microservice might have a distance

[4] https://waikato.github.io/weka-wiki/

of 1 to the mega-service anti-pattern and a distance of 0.5 to the nano-service anti-pattern. Then even though it appears to be closer to the nano-service anti-pattern it might still be a mega-service. For example, this can be the case if the value range for mega-service goes from 1 to 10 and the nano-service range goes from 0 to 0.5. Therefore, the relative ranking for each anti-pattern expresses better our confidence level that a service's metrics express an anti-pattern. Thus as a final step we do a min-max normalization on the computed anti-pattern distances.

To detect the Cyclic Dependencies anti-pattern we use the query capabilities of the graph database. We chose to limit length of cyclic dependencies to 6 edges. We find this to have the best trade-off between the number of detected cycles and runtime performance.

3.4 Expert Validation

To validate our methodology we perform two rounds of expert validation. The first round is after the data processing to verify that the computed metrics represent the actual architecture and can be used for further analysis. As the aspect of continuous monitoring of the architecture is central to this study we take two snapshots of the architecture, with two months between them, and compute the relative change in metrics. We then take a sample of the services with the highest relative change in metrics and interview the responsible teams.

In total we interview nine ING developers from six teams. With each team we first show them the screenshots of the two architecture snapshots and then present the computed metrics. Next, we ask them to confirm whether the recovered architecture is correct for both snapshots and what the reason is for the change in architecture. After the developers confirm the snapshot, we discuss the computed metrics for their service. For this we first explain how we compute the metric and what the metric indicate. The developers can then ask any questions about the metrics. Lastly, we discuss if the computed metrics are a valid representation of their service.

The second round is after matching the metrics to the anti-patterns to validate our results. For this we talk to three ING developers from two teams, where the team is developing a service that is flagged for an anti-pattern. Again we show them our snapshot from the architecture, explain the metrics and how we link them to anti-patterns. Then they can ask any questions about our approach. Finally, we ask them whether they agree with the result of our methodology and why they designed the service and its dependencies in this specific way. The results from our expert validation are discussed in Section 6.1.

3.5 Summary

This chapter presents the methodology used to automatically detect anti-pattern within MSAs. The structure of our methodology is based on a general data collection, data processing and analysis approach. The first step is to reconstruct the application's architecture from distributed trace data. The distributed trace data is collected by listening to an event stream of all traced spans. The spans are then grouped into traces from the span-ids. The reconstructed traces allow to recover the service to operation and operation to operation relations. The next step is the processing of the data. The first processing step is to set the service to service relations based on the data recovered in the data collection step. In the second processing step, the metrics are computed for each service. The last step of the approach is the linking of the service's metrics to anti-patterns. This is done by computing the weighted euclidean distance between the services' metrics vector and the anti-pattern vector. The approach is validated by sampling a set of flagged services and interviewing the respective teams to verify the results.

Chapter 4

Visualization Tool

This chapter provides the motivation for the developed ING/Viz tool in Section 4.1, the defined requirements in Section 4.2 and the tool's architecture in Section 4.3. Section 4.4 presents the data visualization and discusses how the tool meets the defined requirements.

4.1 Motivation

During the expert validation teams are not able to derive meaningful insights from only the computed metrics. Without spending more time to understand the metrics the teams don't know how to interpret the metrics and draw actionable conclusions. Linking the metrics to anti-patterns allows teams to quickly understand what the problem is. However, without a visual representation, providing more context information, it is difficult for teams to get an overview and understanding of the abstract architecture.

Visualizations are an outstanding tool to get a holistic understanding of complex and abstract concepts, which is why they are frequently used for representing microservice architectures. However, for large scale applications the visualization becomes too cluttered to provide meaningful insight. By combining the metrics and detected anti-patterns with a visualization tool we want to overcome the shortcomings of both approaches. By visualizing the flagged services within the overall architecture, teams not only get a better understanding of why a service is flagged as an anti-pattern but also get a visual overview of the surrounding architecture of the application. Additionally, filtering for flagged services ensures that the visualization does not become too cluttered. Furthermore, starting from a single flagged service teams are be able to expand the number of services step by step.

4.2 Requirements

To build a tool that can add meaningful insights to our computed metrics, together with a team from ING, we define the following set of requirements.

- **R1**: Provide an overview of the entire architecture.

  Motivation: The tool must be able to provide an Overview+Detail view, as discussed by Cockburn, Karlson and Bederson [38]. This way users can get a better understanding of the architectural context of a single service and a group of services.

- **R2**: Provide possibilities to filter out specific areas of the architecture.

  Motivation: To prevent cluttering of the visualization the tool must be able to filter out services that are not relevant.

- **R3**: Provide possibilities to dynamically add and remove parts of the architecture.

  Motivation: To help users explore the architectural context of a single service or a group of services the tool must be able to add and remove service to the current visualization.

- **R4**: Provide an overview of the computed metrics.

  Motivation: The tool must be able to display an overview of the metrics to allow users to find dependencies or patterns within the metrics and compare them with each other.

- **R5**: Provide a summary of identified anti-patterns.

  Motivation: The tool must give an overview of the identified anti-patterns such that users can quickly gain insights from the tool.

- **R6**: Provide additional context and information for services.

  Motivation: Metrics and anti-pattern are not able to capture the whole situation of a service. Therefore, to help users decide if a flagged service needs more attention, the tool must provide more context information for each service.

4.3 Architecture

After defining the requirements for the tool we build ING/Viz. ING/Viz provides a visualization of ING's Microservice Architecture within an interactive dashboard. Users can see the computed metrics, a report of the detected anti-patterns and a graph representation of the MSA. It also provides extra context information for users, fulfilling the aforementioned requirements. Figure 4.1 shows the architecture of ING/Viz. The components in orange are already present within ING and the components in blue are added as part of this graduation thesis. The only ING component that is adapted for this graduation thesis is the *REST-API*, as it needs to be able to handle the new request from the *Backend API*.

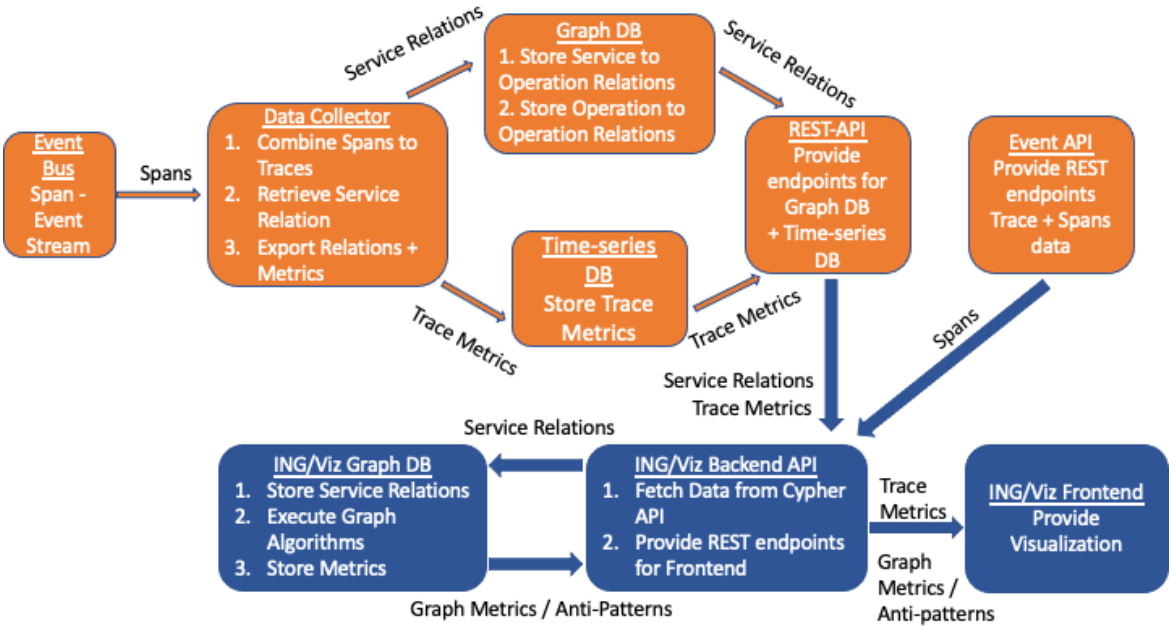

Figure 4.1: ING/Viz architecture overview.

The first component in the architecture is the event bus. This is a central event bus within ING. The event bus provides a topic that all microservices that are connected to the Distributed Tracing can use to publish their spans. In the next step the data collector subscribes to the same topic that all spans are being published on. The data collector then groups all spans it receives into traces. From the trace annotations it extracts the service to operation and operation to operation relations and stores these into the *Graph DB* as described in Section 3.1. Furthermore, the data collector exports

meta data about the traces to the *Time-series DB*. The meta data contains whether the trace was successful, the duration time and a counter of the number of requests.

The *REST-API* connects to the *Graph DB* and the *Time-series DB* to provide REST endpoints to access the relational data and trace-metrics data. As part of this graduation thesis two new endpoints are added to the *REST-API* component for the *Backend API* to connect to. The endpoints are:

- POST /query - accepts a JSON object with a *query* key and a Neo4j Cypher query as value. Returns a JSON object with the data requested in the *query* from the *Graph DB*.
- POST /trace-metrics - accepts a JSON object with a *query* key and a Prometheus-QL query as value. Returns a JSON object with the data requested in the *query* from the *Time-series DB*.

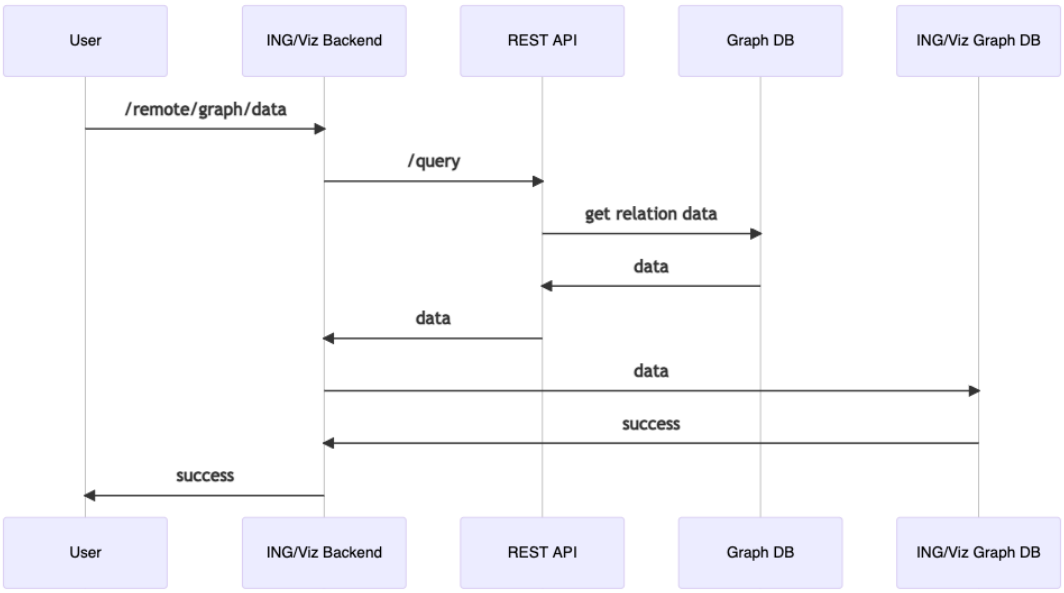

Figure 4.2: The event sequence for fetching the relational data from the Graph DB and inserting it into the ING/Viz Graph DB.

The *ING/Viz Backend API* is a Kotlin [1] Vert.x [2] application. We choose to use Kotlin with Vert.x as this technology stack is popular within ING and therefore makes it easier for ING developers to continue working on and maintaining this project. The *ING/Viz Backend API* fetches the all relational data from the *Graph DB* and stores it in the *ING/Viz Graph DB* as shown in Figure 4.2. The *ING/Viz Backend API* also connects to the *Event API* to fetch trace and span data. The *ING/Viz Graph DB* is a Neo4j [3] graph database. The *ING/Viz Backend API* then calls the *ING/Viz Graph DB* and computes the metrics as described in Section 3.2 and does the linking of metrics to anti-patterns as described in Section 3.3. Again we choose a technology that is already present within ING for the graph database. The Neo4j Graph Data Science Library [4] provides two algorithms that we use for the metrics computation. All computed metrics and anti-pattern information is subsequently stored in the *ING/Viz Graph DB*. The *ING/Viz Backend API* itself also provides a REST API for the *ING/Viz Frontend* to connect to. The endpoints are:

- GET /remote/graph/data - accepts a JSON object with a *query* key and a Neo4j Cypher query as value. Returns a JSON object with the data from calling the /query endpoint of the *REST API*.

[1] https://kotlinlang.org/
[2] https://vertx.io/
[3] https://neo4j.com/
[4] https://neo4j.com/docs/graph-data-science/current/

- POST /remote/trace/metrics - accepts a JSON object with a "query" key and a Prometheus-QL query as value. Returns a JSON object with the data from calling the /trace-metrics endpoint from the *REST API*.
- GET /remote/trace/search - requires a trace-id as *id* query parameter. Returns an array of the trace's spans as JSON objects. The JSON objects contain the span's operation name and the name of the span's service.
- GET /remote/bvc - Returns a string array of all Business Value Chain names.
- GET /remote/bvc - Given a BVC query parameter returns a string array of trace-ids for the given BVC.
- POST /local/graph/query - accepts a JSON object with a *query* key and a Neo4j Cypher query as value. Returns a JSON object with the data requested in the *query* from the *ING/Viz Graph DB*.
- GET /local/graph/metrics/compute - Calls the *ING/Viz Graph DB* to compute all metrics. Returns "Success" if the computation was successful.

The *ING/Viz Frontend* is a React [5] application. For the menus and layout the ant.design [6] component library is used. The graph visualization is implemented with the Graphin [7] library. We choose this technology stack for the frontend mainly because of the Graphin library. The Graphin library provides powerful graph visualization functionalities as React components. Therefore, we choose to use the React framework for the frontend. Through the API of the *ING/Viz Backend API*, the *ING/Viz Frontend* can access all data stored in the *ING/Viz Graph DB* and all data stored in the *Time-series DB*. The *ING/Viz Frontend* is responsible for the visualization of the dependency graph and the presentation of the metric results. The data and metric visualization is described in Section 4.4.

We deliberately choose to develop ING/Viz as a separate standalone web application with its own database, backend and frontend. One key factor for this decision is development speed. Applications that are part of the ING ecosystem must to undergo rigorous security and data protections audits. However, since ING/Viz is an internal tool that is not exposed to the internet the security and data protections audits are not necessary at the moment and would take too long for the rapid prototyping approach we use. Once ING/Viz goes into production it has to pass the security and data protections audits. Furthermore, it is not clear whether ING/Viz will be continued within ING, thus integrating it into the ecosystem is not desired at this point. Also in a future development stage ING/Viz could be used outside of ING. Therefore, we choose to develop ING/Viz as a separate standalone web application.

Figures 4.3 and 4.4 show two example sequences for user interaction with the *ING/Viz Frontend*. Figure 4.3 details the use case of a user requesting trace-metrics data to be overlaid on the graph's edges as shown in Figure 4.5. In this case the *ING/Viz Frontend* calls the */remote/trace/metrics* endpoint of the *ING/Viz Backend API*. The *ING/Viz Backend API* in return calls the */trace − metrics* endpoint of the *RESTAPI* which fetches the requested data from the *Time-series DB*.

[5] https://reactjs.org/
[6] https://ant.design/
[7] https://graphin.antv.vision/en-US/

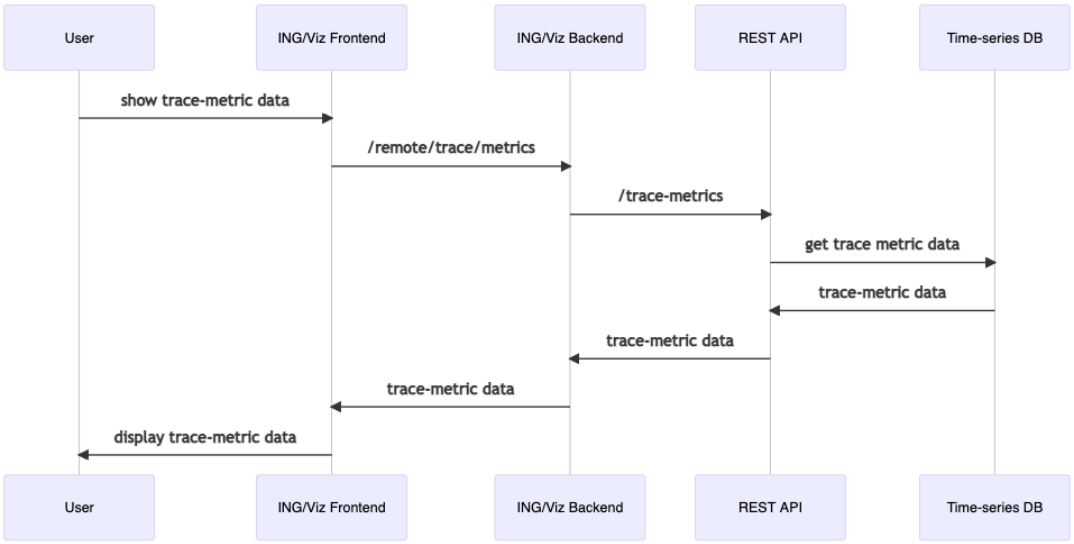

Figure 4.3: The event sequence for fetching the trace-metrics data and displaying it in the ING/Viz Frontend

Figure 4.4 details the use case in which a user wants to see the trace of a Business Value Chain (BVC). The sequence starts when the user navigates to the BVC page. The *ING/Viz Frontend* then calls the */remote/bvc* endpoint of the *ING/Viz Backend*. Next, the *ING/Viz Backend* calls the */lov/bvc* endpoint of the *REST API*. The *REST API* sends back a list of all BVC names, which is subsequently displayed in the *ING/Viz Frontend*. From this list the user can select a BVC. The *ING/Viz Frontend* then calls the */remote/bvc* endpoint of the *ING/Viz Backend* again, but this time provides the BVC-id as query parameter. The *ING/Viz Backend* extract the BVC-id from the query and calls the */spans/search* endpoint of the *Event API* and provides the BVC-id as filter. The *Event API* returns a list of spans that belong to the provided BVC-id from the last two hours. The *ING/Viz Backend* processes this list and only sends the distinct trace-ids to the *ING/Viz Frontend*. The user can then select a trace-id of which she wants to see the entire trace. The *ING/Viz Frontend* then calls the */remote/trace/search* endpoint of the *ING/Viz Backend*, which in return continuously calls the */spans/search* endpoint of the *Event API* until it has received all spans of the requested trace. Once the *ING/Viz Backend* has received all spans, it extracts the service names and operation names of all spans and sends this data to the *ING/Viz Frontend*.

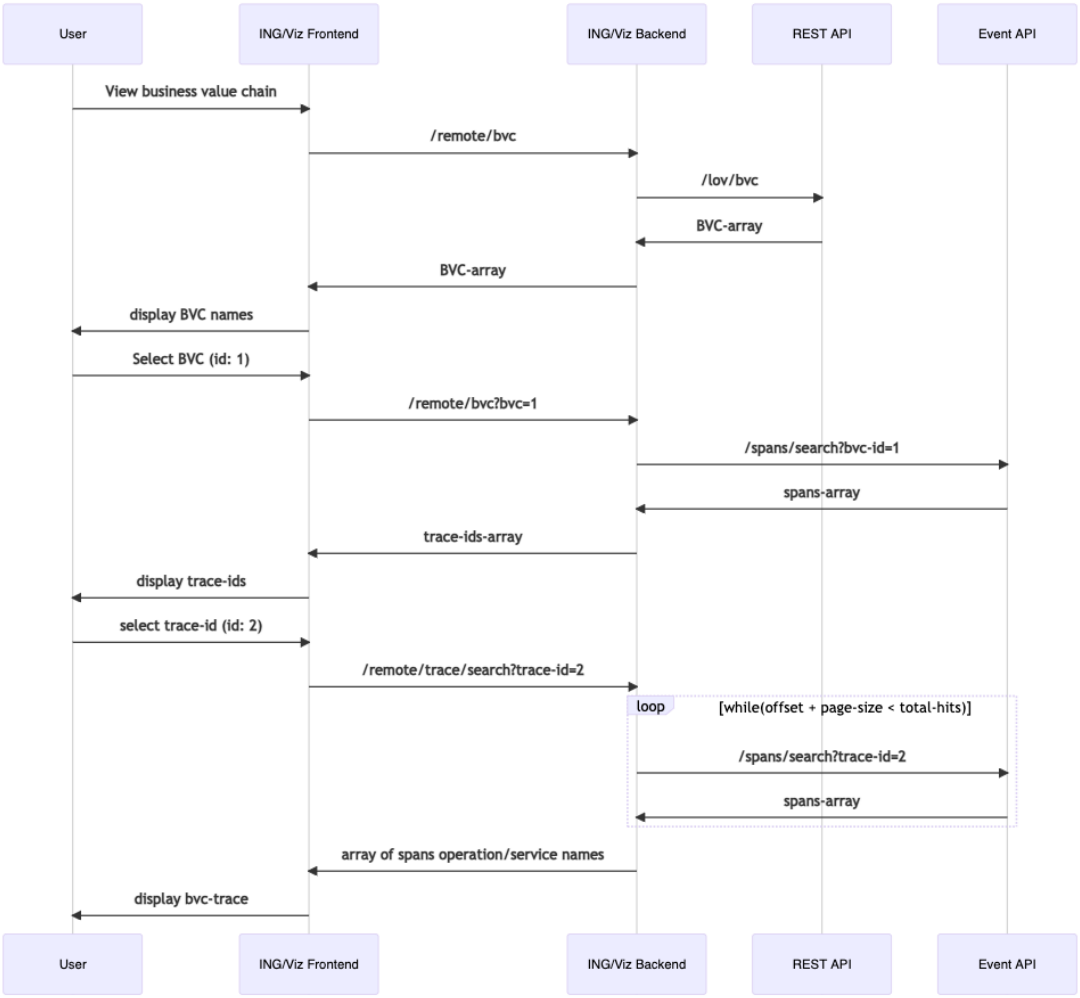

Figure 4.4: The event sequence for displaying a Business Value Chain trace and displaying it in the ING/Viz Frontend

4.4 Data Visualization

The Microservice Architecture is visualized as dependency graph, where nodes represent microservices and edges represent dependencies. Figure 4.5 shows an overview of the entire architecture with trace-metrics data projected onto the edges. The thickness of edges represents the throughput of traces per minute between the two services in the last hour. The thickness scales in five steps between two and ten pixels in width and depends on the logarithmic scaling from 10 to 100000. The color of an edge can be one of four states: green, orange, red and black. An edge is red if the success-rate of the traces is below 95% or the average response time is greater than 500ms. An edge is orange if the success-rate is below 98% or the average response time is greater than 100ms. Otherwise the edge is green. An edge is black if no throughput is measured within the last hour. The trace metric information can help users assess the impact of an identified anti-pattern. For example, if a service's metrics are flagged as an anti-pattern, but the trace-metrics are green, the users could decide to postpone a refactor. However, if the trace-metrics are red on most of the service's edges, this could mean that refactoring this service needs a high priority. This overview satisfies requirements **R1**, **R3** and **R6**.

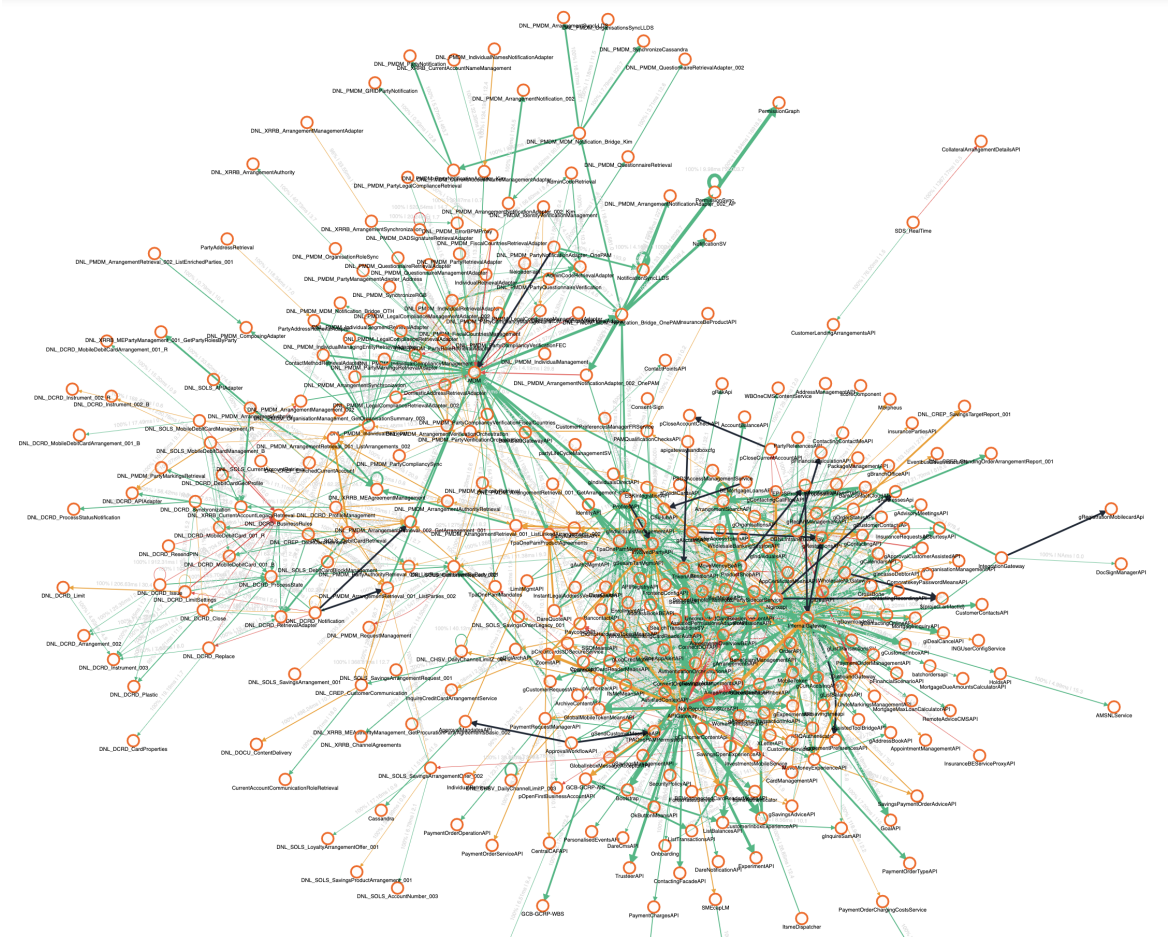

Figure 4.5: ING/Viz tool displaying the whole architecture with trace-metrics projected onto the edges.

Going from a high level overview to a detailed view, Figure 4.6 shows a mega microservice identified by the anti-pattern metrics and its dependencies. This detail view gives users the possibility to get a clear understanding of the dependencies of a part of the architecture and allows them to make a better judgment whether an anti-pattern is identified correctly. Each service in the view can be removed or expanded to show its operations as well as its in and outgoing service dependencies. This allows for an easy exploration of the architectural context, letting the user seamlessly switch from the detail view back to a higher level overview. This satisfies requirements **R2**, **R3** and **R6**.

Figure 4.6: ING/Viz tool displaying a flagged mega-service surrounded by its depending services.

Figure 4.7 shows dual axis charts such that the user can compare computed metrics with each other. In the top left corner the user can select the metric she wants to compare and then the dashboard shows a dual axis chart for all other computed metrics with the selected metric. For example, in Figure 4.7 we can see AIS plotted against SIUC and WSIC, where AIS is shown in dark blue sorted from high to low. This view can help users find dependencies or correlations between metrics. For example, AIS does not appear to be correlated with SIUC or WSIC.

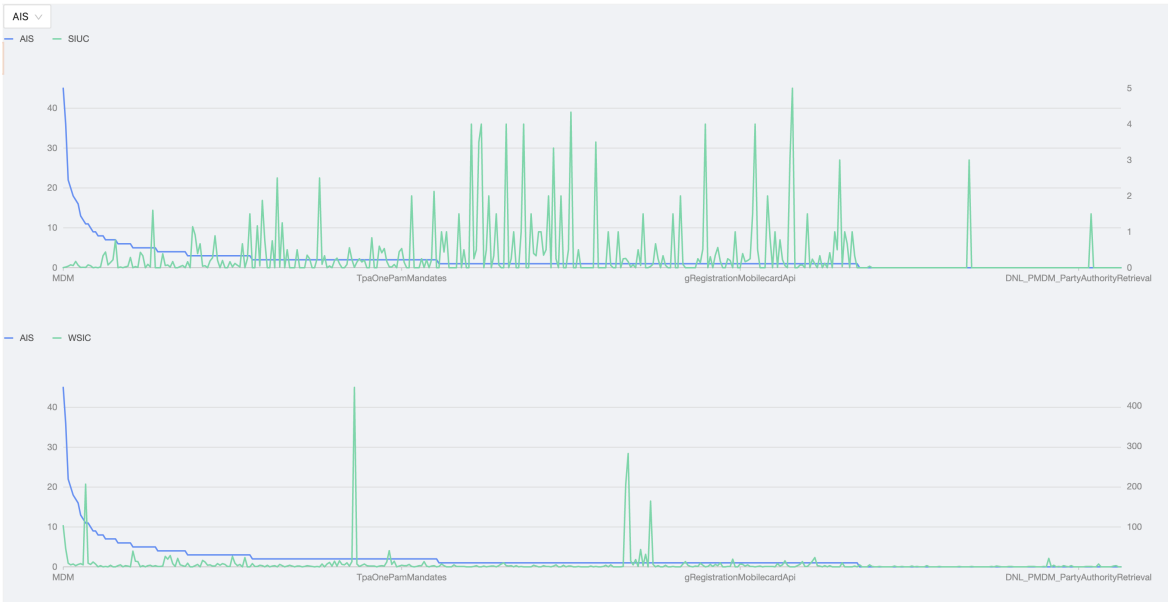

Figure 4.7: ING/Viz tool's diagram page displaying dual axis line diagrams of the computed node metrics.

All metric information can also be displayed in a sortable table as shown in Figure 4.8. The table gives an easy overview of a service's metrics. Furthermore, the sorting allows users to quickly get the highest or lowest scoring services for a particular metric. Additionally, there is a metrics table view for all operations. This helps users identify operations with bad metric scores. The diagrams and table views satisfy **R4**.

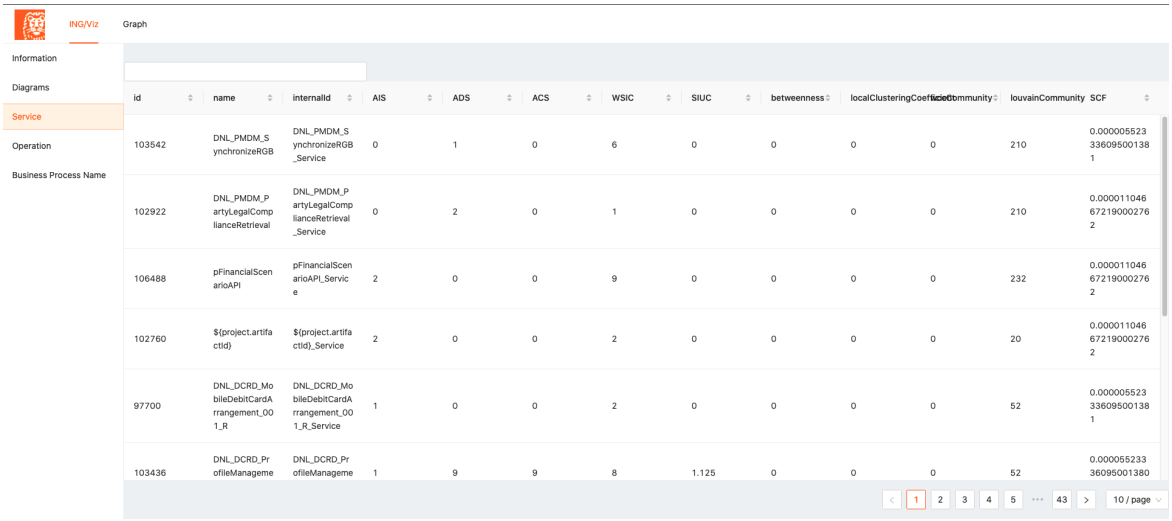

Figure 4.8: ING/Viz tool's metrics table listing the metrics for all services.

Figures 4.10,4.9,4.11 show how annotation based distributed tracing can add additional context information. The collected traces do not only include service name and operation name information as described in Section 3.1. Some spans also contain information about the Business Value Chain they belong to. Business Value Chain are also called customer journeys within ING and describe a series of events that belong to a certain action an ING customer might perform. For example, signing up as a new ING customer or changing his home address. This means if all services annotate their spans with the BVC their spans belongs to, we can explore the overall graph from a BVC perspective. Combined with the computed metrics and identified anti-patterns this can help improve BVC and therefore the application's architecture. ING/Viz provides the possibility to list all BVCs currently traced within ING. Once a user selects a BVC a list of all services and their operations that belong to the BVC trace are listed as shown in Figure 4.9. If the user wants, she can also change the view and have the list as well as a graph representation of the trace presented side by side as shown in Figure 4.10. This side by side view gives a better understanding of how the trace passes between services and operations. This gives a developer a different angle for finding refactoring possibilities not only on an architectural level but also from a BVC level. Additionally, to provide users with a better understanding of the context of a BVC, ING/Viz can highlight all services of a BVC within an overview of the entire architecture. Figure 4.11 shows the same BVC trace as in Figure 4.9 but in relation to the entire dependency graph. These overviews satisfy requirements **R2** and **R6**.

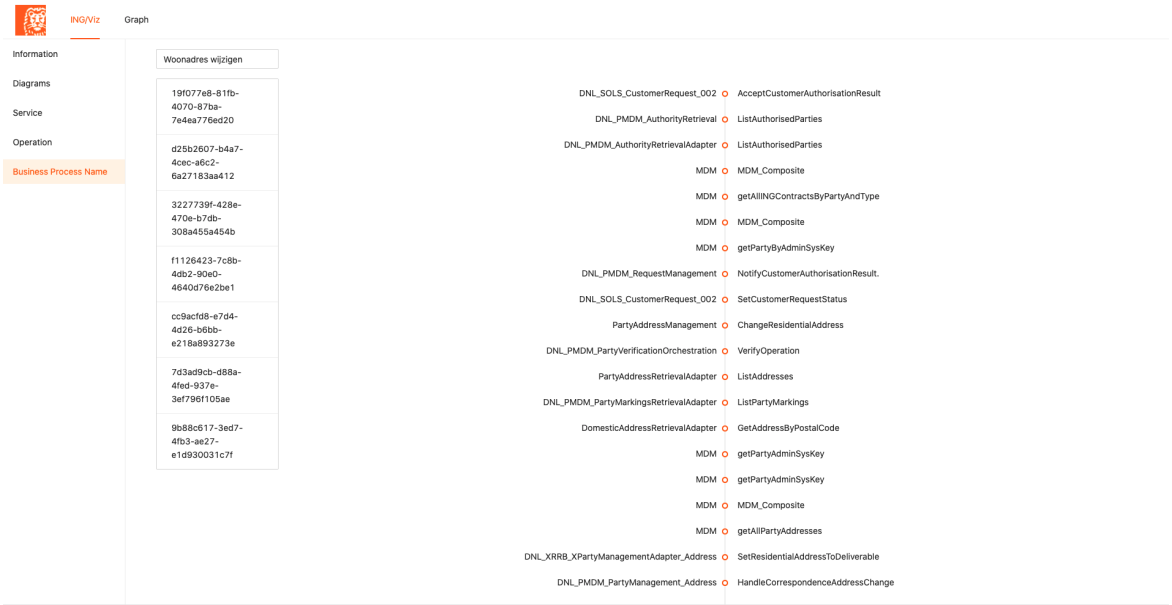

Figure 4.9: ING/Viz tool displaying a BVC list of services and operations.

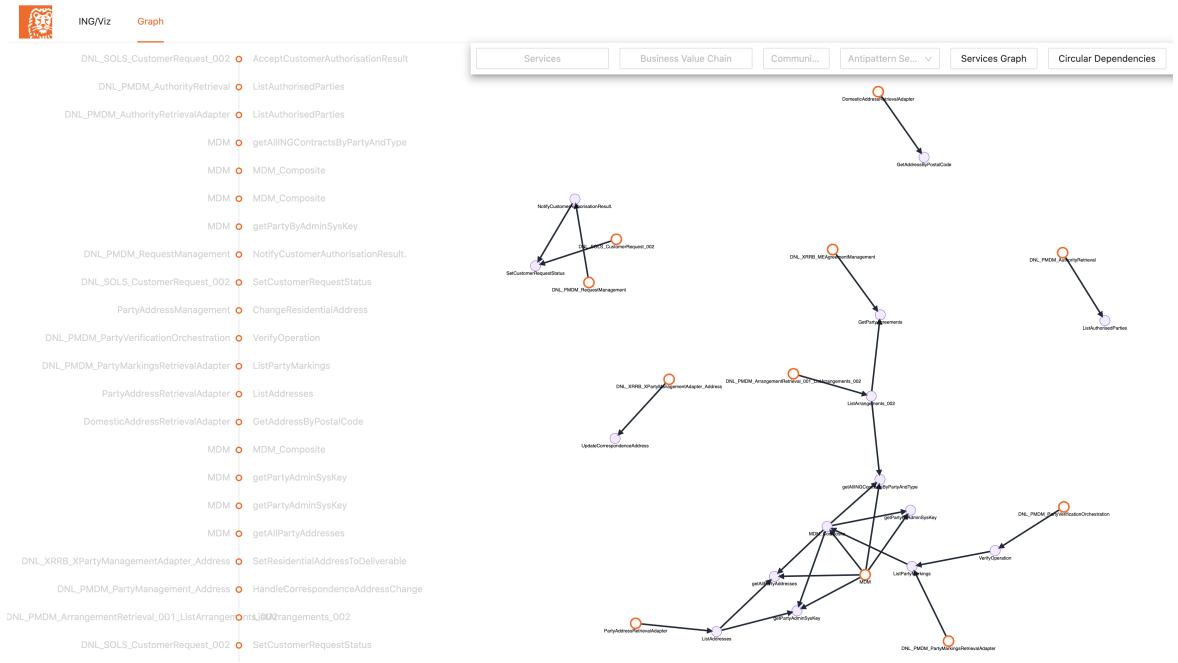

Figure 4.10: ING/Viz tool displaying a BVC in a list and graph format.

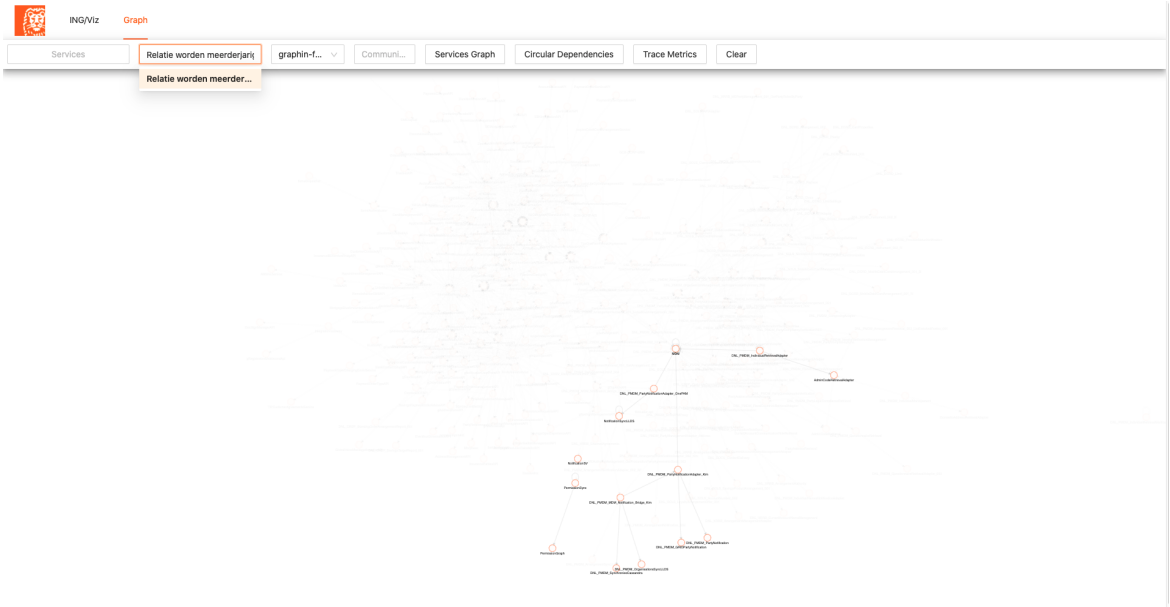

Figure 4.11: ING/Viz tool highlighting a BVC services within the overall architecture.

Figure 4.12 shows four Louvain communities [8] within the overall architecture. We compute the communities with the weakly connected communities algorithm and Louvain community algorithm provided by the Neo4j Data Science Library [9]. In her GOTO 2019 talk Nicki Watt [39] discusses how community detection algorithms can be used to confirm logical groupings of services. Users can verify if the communities confirm their understanding of logical groupings of services. If this is not the case, this can be an indication of bad domain separation within the application. Furthermore, if a service has dependencies with services from many different communities this can be an indication of bad separation of functionality. This view satisfies requirement **R6**.

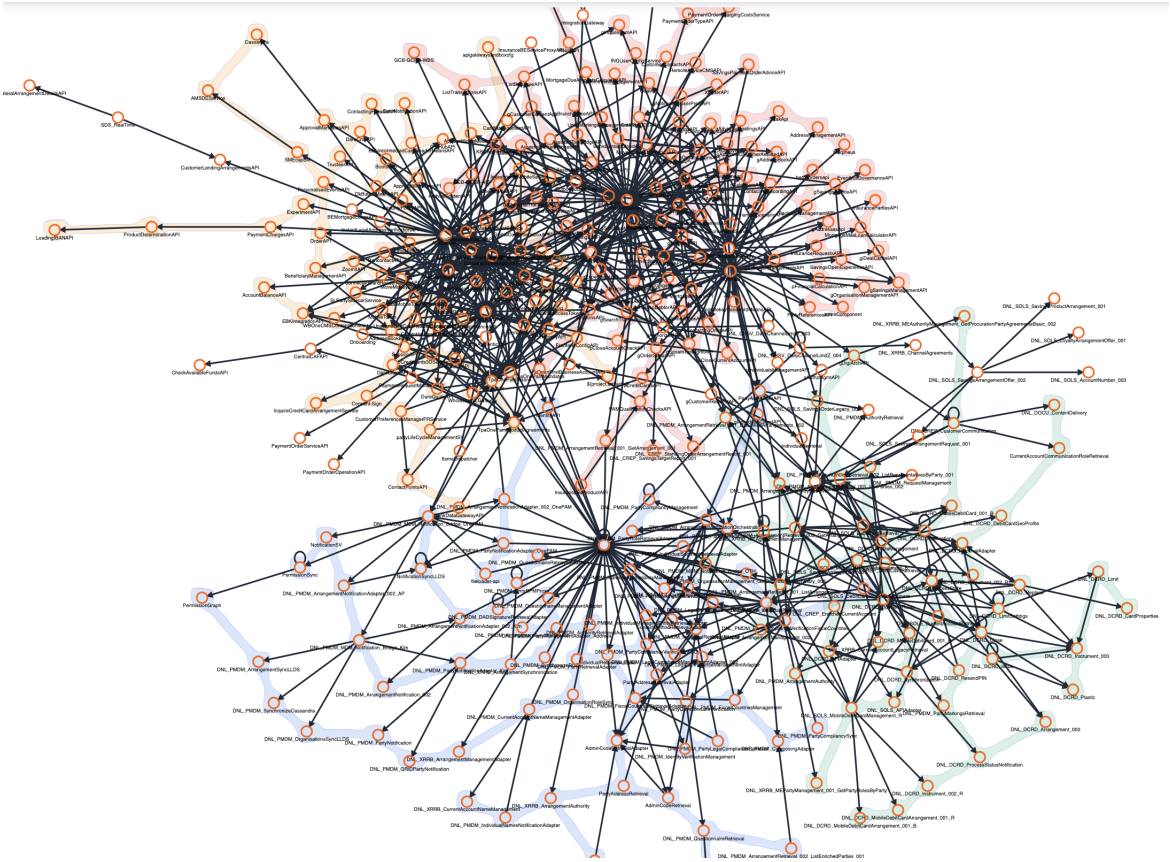

Figure 4.12: ING/Viz highlighting the four largest Louvain communities within ING's architecture. In total 13 Louvain communities are detected.

[8] https://neo4j.com/docs/graph-data-science/current/algorithms/louvain/
[9] https://neo4j.com/docs/graph-data-science/current/

Figure 4.13 shows the results page of the dashboard. On this page the user gets a small rundown for each computed anti-pattern. For each anti-pattern there is a small description of the anti-pattern's symptoms. Furthermore, the top 5 services that are closest to the given anti-pattern are listed, which is followed by a possible refactoring solution. At the end of the page there is a sortable table with all results. Additionally, the user can dynamically include or exclude gateways and ngnix services and can set the distance thresholds for each anti-pattern. These thresholds are used to compute the occurrence percentage for each anti-pattern. For example, in Figure 4.13 the distance threshold for the mega-service anti-pattern is set to 0.62. Given this threshold, 4% of services are flagged, which is displayed next to the name of the anti-pattern. This is especially helpful for the nano-service anti-pattern as a symptom of the anti-pattern is a high number of nano-services.

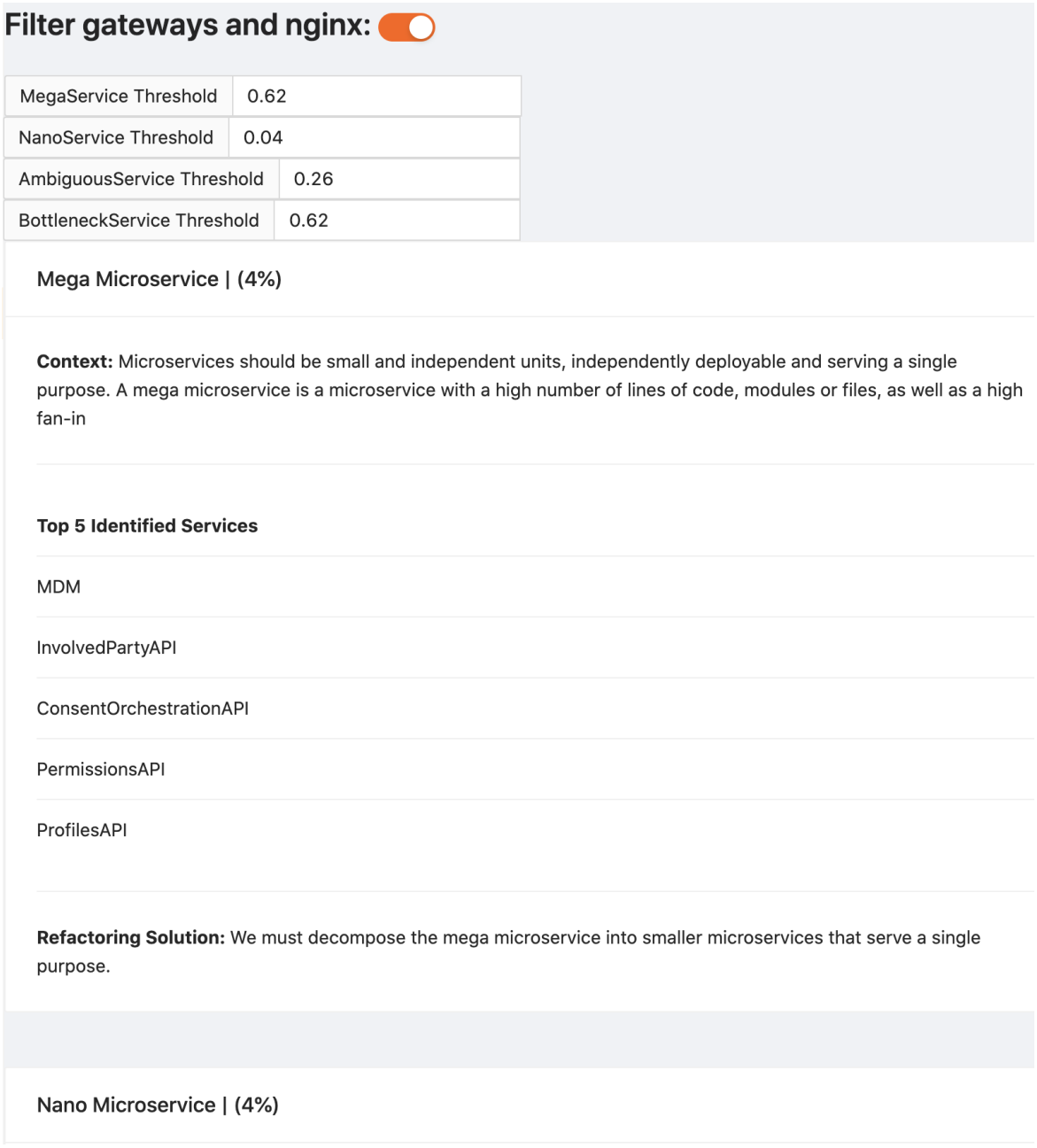

Figure 4.13: ING/Viz tool's report page listing the flagged services for each anti-pattern.

In Figure 4.14 the top 5 services that are closest to the mega-service anti-pattern are highlighted within the architecture. However, the user can also have the other anti-patterns highlighted. This provides the user with additional context information and an intuition of where the anti-patterns are situated within the architecture. Furthermore, this view could possibly reveal areas of the architecture that have significantly more flagged services. Thereby, revealing not only suboptimal services but also problem areas in the architecture.

Figure 4.14: ING/Viz highlighting 5 services flagged as mega-service.

Figure 5.1 shows the results from the anti-pattern matching as a diagram. This is an easy way to see the distribution of the results and helps the user get an understanding of the computed results. These three views satisfy requirement **R5**.

4.5 Summary

This chapter presents the ING/Viz dashboard developed as part of this graduation thesis. In Section 4.1 the motivation behind the dashboard is highlighted. In Section 4.3 the architecture of the dashboard is presented and in Section 4.4 the dashboard itself is presented.

The dashboard provides an interactive visualization of the application's architecture as a Service Dependency Graph. It allows the user to filter, expand and explore ING's architecture. Furthermore, it gives an overview of the computed metrics as well as a report on the detected anti-patterns. Additionally, it provides the user with several additional data sources to gain a better understanding of the application's architecture. The user can use the additional context information to make a better judgment whether the detected anti-patterns need additional in depth investigation or can be ignored.

Chapter 5

Case Study and Results

In Section 5.1, this chapter details the case study that was performed with the developed anti-pattern detection approach and the developed visualization tool. Furthermore, the results of this case study are presented in Section 5.2. In Section 5.3 we answer the research question.

5.1 Case Study

For evaluation purposes, we integrate the tool within the ING Group (ING). ING is a Dutch multinational banking and financial services corporation. Its primary businesses are retail banking, direct banking, wholesale banking, private banking, asset management, and insurance services [40]. From our data collection we register 426 services within ING's microservice architecture for retail and small and medium businesses (NL/BE). However due to the collection process as described in Section 3.1 there might be more services. These could be services that use distributed tracing but did not create any traces in the collection period or there might be microservices not using the tracing client. Nonetheless, we are certain that we have registered the vast majority of services within the application and our dataset is large enough to confidently demonstrate our approach. There is another set of ING's code base that is not captured in our approach, namely the legacy services that connect to the microservice architecture but have no tracing client installed. Since this paper focuses on the analysis of microservice architectures, we would have excluded these parts of ING's application from our results regardless.

5.2 Results

5.2.1 Methodology Results

In Table 5.1 we present the percentage of services that are flagged for each anti-pattern given a set threshold. The threshold is chosen by us after manually testing different thresholds and verifying that the flagged services meet our expectations. Therefore, the thresholds can be very different within the same application, depending on who sets them. Also the thresholds will be very different for other applications. With the selected thresholds we can see that the two most common anti-pattern are the mega-service and the nano-service. The least common anti-pattern is the ambiguous-service anti-pattern with 2%.

Table 5.1: Percentage Flagged Services

Anti-Pattern	Threshold	Flagged %
Mega Service	0.61	4
Nano Service	0.04	4
Ambiguous Service	0.26	2
Bottleneck Service	0.63	3

Figures 5.1, 5.2, 5.3, 5.4 show the distribution of distances for each anti-pattern. The orange line indicates the selected threshold for each anti-pattern. The distributions of all four anti-patterns follow a similar trajectory. For small values the slope is steep and for higher values the slope becomes flatter. We expect this distribution as we only expect a small percentage of microservices' metrics to express an anti-pattern. However, from these diagrams we can also see that there is no clear distinction in the slope to identify services whose metrics express an anti-pattern. For example, in Figure 5.2 the slope drops abruptly for distances < 0.7. However, we choose a threshold of 0.04, as services above this threshold do not meet our expectations of a nano-service. The threshold we choose is half way between an almost linear increase from 0 to 0.06 and is therefore impossible to detect automatically. An automated approach could detect the change in slope at 0.7 but this is an incorrect threshold.

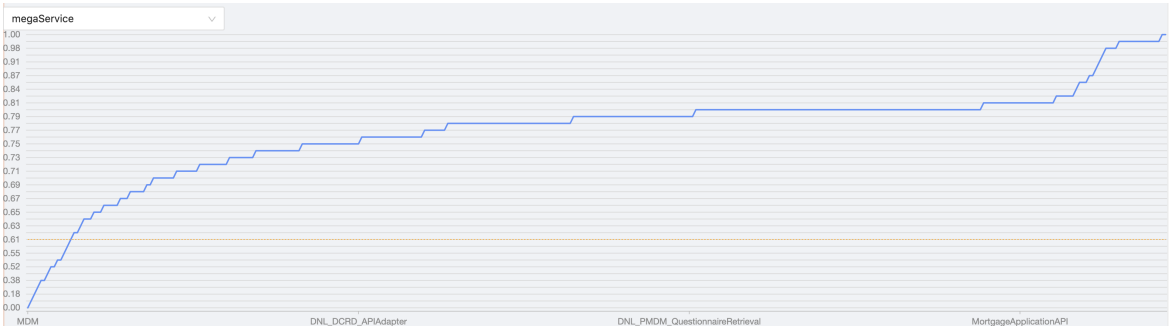

Figure 5.1: ING/Viz mega-service anti-pattern matching results as diagram with the threshold marked with an orange line.

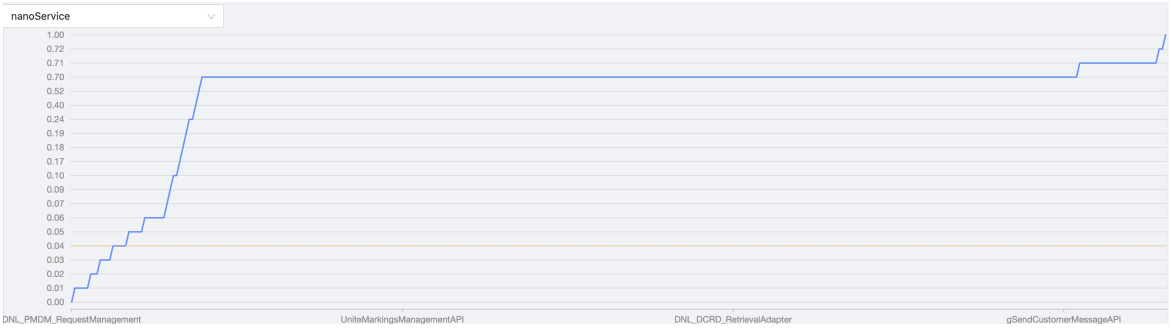

Figure 5.2: ING/Viz nano-service anti-pattern matching results as diagram with the threshold marked with an orange line.

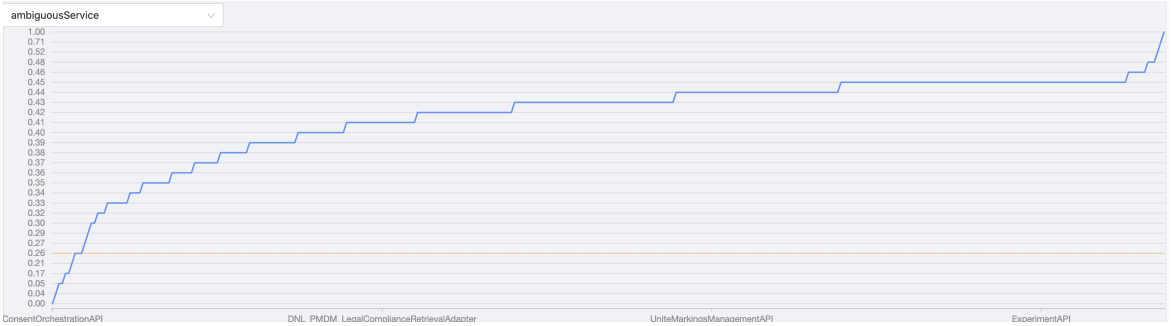

Figure 5.3: ING/Viz ambiguous-service anti-pattern matching results as diagram with the threshold marked with an orange line.

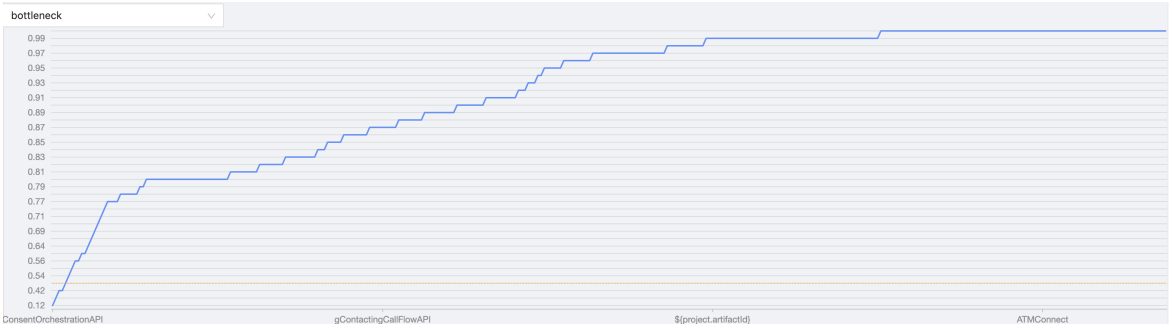

Figure 5.4: ING/Viz bottleneck-service anti-pattern matching results as diagram with the threshold marked with an orange line.

In Tables 5.2, 5.3, 5.4, 5.5 we list for each anti-pattern the top 5 closest services. We filter out any gateways/load-balancer services as these are listed in the top 5 for Mega-Service, Bottleneck and Ambiguous-Service. After consulting with domain experts, we conclude that these gateway/load-balancer services cannot be seen as a microservice in this application. We will discuss this further in Chapter 6.1. We can see that MDM, InvolvedPartyAPI, ConsentOrchestrationAPI and PermissionsAPI are listed for mega-service, bottleneck-service and ambiguous service.

Table 5.2: Distance service to mega-service anti-pattern

Name	MegaService
MDM	0.00
InvolvedPartyAPI	0.09
ConsentOrchestrationAPI	0.18
PermissionsAPI	0.21
AuthenticationOrchestrationAPI	0.38

Table 5.3: Distance service to nano-service anti-pattern

Name	NanoService
DNL_PMDM_RequestManagement	0.00
ApprovalWorkflowAPI	0.01
AgreementPreferencesAPI	0.01
DNL_PMDM_ArrangementNotificationAdapter_002_OnePAM	0.01
DNL_PMDM_ArrangementNotificationAdapter_002_Kim	0.01

Table 5.4: Distance service to bottleneck anti-pattern

Name	Bottleneck
ConsentOrchestrationAPI	0.12
MDM	0.25
InvolvedPartyAPI	0.42
PermissionsAPI	0.42
DNL_PMDM_MDM_Notification_Bridge_OnePAM	0.48

Table 5.5: Distance service to ambiguous-service anti-pattern

Name	AmbiguousService
ConsentOrchestrationAPI	0.00
PermissionsAPI	0.04
MDM	0.05
InvolvedPartyAPI	0.50
AuthenticationOrchestrationAPI	0.17

As described in Section 3.2 we identify cyclic dependencies via a graph query and the CSD metric only indicates whether a service is part of a cyclic dependency. Therefore, it is easier to present the results via a figure. Figure 5.5 shows all cyclic dependencies that we detect. In total 38 services have a cyclic dependency. Our query identifies 4 clusters of services that have cyclic dependencies between the services. Most cycles are of length 2, but there are also larger cycles of length 3.

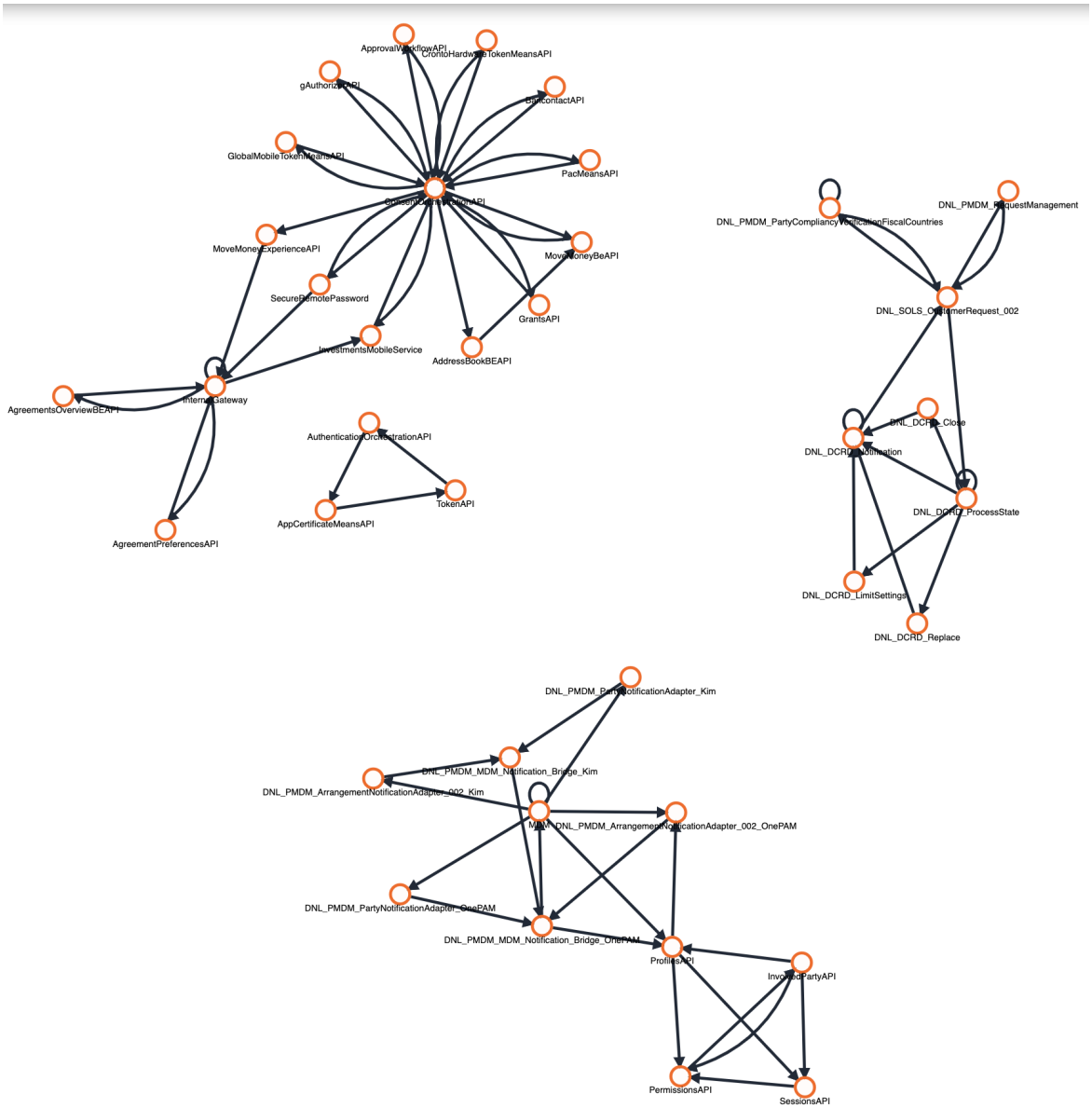

Figure 5.5: ING/Viz tool displaying all cyclic dependencies.

5.2.2 Expert Validation Results

The results from the expert validation are that the service to service and service to operation relations are correct but not complete. For some services relations are missing. The number of missing services for Java based applications is less than 10%. However, the number of inconsistencies and missing services is significantly higher for services that are based on the TIBCO software [1] compared to Java based services. The TIBCO services belong to the legacy code and are developed as a suite of services

[1] https://www.tibco.com/

but originally not with the microservice architecture in mind. The services do not use HTTP calls as the Java based applications but communicate via JMS [2] messaging which causes some side effects. For example, the DNL_DCRD_Notification has two operations where one calls the other according to our DT data. However, these are JMS calls which are considered to be internal procedure calls for TIBCO services. Thus, the information that is sent to the DT does not represent the real world correctly. Furthermore, as some relations are missing we do not detect all anti-patterns. According to one expert the DNL_SOLS_DebitCardBlockManagement should be flagged as an anti-pattern. In our database the DNL_SOLS_DebitCardBlockManagement has four service dependencies. However, it actually has 16 service dependencies, which would have caused it to be flagged as mega-service by our methodology. We can also confirm that the DNL_DCRD_BusinessRules service which is flagged as mega-service is in fact a mega-service. The DNL_DCRD_BusinessRules centralizes a lot of functionality which causes it to perform a large number of checks.

The feedback for ING/Viz is mostly positive with some remarks on the usage of ING /Viz. This first remark is that developers do not require this dashboard in their day to day work. The reasoning being, that a service's team knows which other services they connect to. The second remark is, to have a clear separation between our tool and Application Performance Management tools such as AWS X-Ray [3] and Datadog [4] as these also provide architecture overviews. The positive feedback is that the tool raises awareness for anti-patterns within teams as these are not usually taken into consideration during development. Furthermore, the tool creates an observability for architecture design and "stimulates or scrutinizes" teams to spend more time on architecture design as it is measured and checked in possible architecture audits.

5.3 Result Research Question

From our results we conclude that we can automatically identify anti-patterns in microservice architectures from distributed tracing. However, the output from our approach is not a definitive identification of anti-patterns but has to be evaluated by domain experts. Based on our computed metrics some desirable architectural designs have metric properties that classify them as an anti-pattern. For example, a central authentication service will have many dependencies with other services as it is undesirable for all services to implement their own authentication logic. Therefore, additional domain knowledge is needed to get a definitive result. But the metrics can prioritize and reduce the number of services that need to be manually checked and can therefore provide valuable automated support for developers in identifying anti-patterns in microservice architectures.

5.4 Summary

In this chapter we present the results from the anti-pattern detection approach and the developed visualization tool. To verify our approach and visualization tool we perform a case study within the ING Group (ING). We register 426 services within ING's microservice architecture for retail and small and medium businesses (NL/BE). With the selected thresholds the two most common anti-patterns are the mega-service and the nano-service. The least common anti-pattern is the ambiguous-service anti-pattern. Furthermore, we detect 38 services with cyclic dependencies grouped in 4 clusters. From our results we conclude that we can automatically identify anti-patterns in microservice architectures from distributed tracing. However, the output from our approach is not a definitive identification of anti-patterns but has to be evaluated by domain experts.

[2] https://en.wikipedia.org/wiki/Jakarta_Messaging
[3] https://aws.amazon.com/xray/
[4] https://www.datadoghq.com/

Chapter 6

Discussion, Future Work and Threats to Validity

This chapter is made up of four sections. Section 6.1 discusses the results from the case study. Section 6.2 presents possible future work directions. Section 6.3 highlights potential threats to validity of this thesis. In Section 6.4 we provide personal recommendations for ING based on our results.

6.1 Discussion

In this thesis we present the methodology for automatic anti-pattern detection in microservice architectures and we present the ING/Viz tool for visualizing the results from the methodology. The visualization tool requires the anti-pattern detection methodology to work, however, the methodology does not require the visualization and can be used by itself. Therefore, we discuss the two topics separately.

6.1.1 Automatic Anti-Pattern Detection

From the results we conclude that our approach for Microservice Architecture recovery via Distributed Tracing works but is impeded by the reality of an industrial application. The expert validation confirms that the detected service dependencies are correct, however, several services and dependencies are missing due to the inconsistent way that spans are traced and annotated. Therefore, in the case of ING further work is required to recover the full and correct application architecture. But for simpler applications or applications with more consistently implemented DT our approach is able to recover the application's architecture.

Furthermore, we conclude that our approach for automatic anti-pattern detection can identify anti-patterns within MSAs as demonstrated with the DNL_DCRD_BusinessRules service. However, additional domain knowledge is needed to definitively determine whether a tagged service's metrics express an anti-pattern. We need an expert to confirm any flagged service as the metrics cannot account for some deliberate design decisions. For example, in Tables 5.2,5.4,5.5, the MDM service is ranked first for the mega-service anti-pattern, ranked second for bottleneck and ranked second for ambiguous-service. Thus, based on the metrics we expect an anti-pattern. However, this architecture is by design, as the MDM service provides customer data, which is the central point for authentication and consent processes. This design decision is made to prevent repetition in implementing the complex authentication logic. Therefore, the service is currently not a candidate for redesign. Though one could argue, from an availability perspective, to use sharding and have MDM services per geographical region. Nevertheless, our approach reduces and prioritizes the number of services that need to be manually checked, which is a significant improvement over manually searching for anti-patterns. Our approach provides valuable automated support for teams in identifying anti-patterns in MSAs.

Additionally, our approach can help to confirm or disprove developer's intuition of architectural

refactoring possibilities. For example, several developers we talked to expect the number of mega-services to be high but not the number of nano-services. To verify this we manually determine the threshold for the mega-service and nano-service anti-patterns such that the flagged services satisfy our expectations. Having set the threshold we find that 4% of services are flagged for both anti-patterns. This result is very interesting and not expected by the developers we interviewed. Such biases in teams might result in decreased performance as possible improvement possibilities are ignored. In this example, a high number of very small services can cause a severe decrease in maintainability. This is a result of teams having to deal with a comparatively large development overhead compared to the provided functionality of the service. This shows the strength of a data driven identification compared to intuition as it is less prone to biases.

We can also conclude that more work is needed to improve the results. This is especially the case for the ambiguous-service anti-pattern. The results for the ambiguous-service are similar to the mega-service and bottleneck anti-pattern results. However, after analyzing the results we conclude that the listed services do not represent what we expect an ambiguous service to be. The flagged services have clearly defined functionality and have several operations to interface with the service. The commonality between the flagged services is the fact that they provide essential functionality (e.g. authentication) which many different services use. From a metrics perspective, the flagged service appears to serve multiple purposes. Therefore, we conclude that the metrics we use are not suitable to detect the ambiguous-service anti-pattern. In the case of the ambiguous-service a different set of metrics is needed, as the current metrics fail to correctly capture the symptoms of the anti-pattern.

Likewise, the results for the bottleneck-service anti-pattern are very similar to the mega-service results. This is to be expected as the dominant metric of the mega-service is the number of incoming dependencies and the dominant metric of the bottleneck anti-pattern is the betweenness score. This leads to a clear dependency. The more incoming dependencies a service has the higher the chance that a shortest path leads through this service increasing the betweenness score. However, we do see some differences. For example, the ConsentOrchestrationAPI is closer to the bottleneck anti-pattern and therefore it might make more sense to use a refactoring solution for the bottleneck anti-pattern. However, more work is needed to better define or expand the anti-pattern vectors, to ensure the results have a clearer separation.

The results for the cyclic-service anti-pattern are the most definitive. Visualizing the flagged services provides the most added value of all the anti-patterns. In this use case, it allows us to easily see that there are clusters of cyclic dependent services. This could mean that the services are highly dependent on each other, which closely resembles the distributed monolith anti-pattern. Thus, the visualization adds valuable context information.

From our evaluation with teams within ING we conclude that the metrics are correct. But from the discussion we notice that the metrics are very difficult to quickly understand. Furthermore, the feedback from the interviewed teams is that it is hard to develop an intuition of the insights they provide. In our second validation round the team finds it a lot easier to understand the results. We are therefore convinced that the matching of metrics to anti-patterns provides a valuable step in communicating the results to a broader audience. The matching gives teams a starting point for possible refactoring solutions for the detected anti-patterns.

6.1.2 ING/Viz tool evaluation

From the feedback the ING teams provide and from personal testing we conclude that the developed ING/Viz tool is a valuable addition to our developed anti-pattern detection methodology. The results in form of the metrics are very abstract. This means the metrics provide very little actionable insights for people and teams that do not know how to interpret the metrics. The linking of metrics to anti-patterns makes the results more understandable, however, a simple list of flagged services does not spark any discussion. After showing the teams the flagged services in the tool, they immediately start discussing the results. The fact that the architecture is tangible with the visualization makes

it far easier for teams to reason about the results and weigh pros and cons of refactoring solutions. Therefore, we are convinced of the tool's value.

We are not able to fully assess the additional context data ING/Viz provides. From personal usage the trace-metrics data is especially interesting. It makes it possible to estimate the criticality of a detected anti-pattern without having domain expertise of the particular service. If a service is flagged, but the metrics are green for all edges this is a lesser problem as this means the percentage of successful spans is greater than 97% and the average response time for all spans is smaller than 100ms. If a flagged service has many red edges, this might be a serious problem as this means that either the percentage of successful spans is lower than 95% or the spans average response times are greater than 500ms. This is valuable information. However, at the moment the trace-metrics are based on the traffic of the previous hour. Having historic data at this point makes more sense and would provide more reliable information. Unfortunately, at the moment trace-metrics data is only stored for two weeks and therefore unsuitable.

We are not able to verify the value the community detection provides as we have no ground truth to compare it to. Without an upper limit to the number of communities, the Louvain community detection algorithm divides the application into 13 communities. Interestingly, the largest four communities seem to be grouped around three mega-services (InvolvedPartyAPI, MDM, DNL_DCRD_BusinessRules) and the InternalGateway. This is shown in Figure 4.12. However, we cannot confirm whether these communities make sense, as we are not able to find a single overview of ING's application with all services grouped by their logical business domains. Therefore, we cannot compare the detected communities to the intended groups by ING.

So far the Business Value Chain visualization has not been used by teams in the intended purpose but the feedback from two developers is very positive. They like the ability to focus on a single BVC, which significantly reduces the number of services in the visualization. This makes it easier for a software architect to single out the weakest link in the BVC and work on improving that service first. This provides a clear angle for improvement, compared to looking at the entire architecture. Finding a refactoring possibility, that has a positive impact on the application's architecture, is difficult. But for a chain of services this becomes a lot simpler.

As discussed in Section 5.2 the feedback from the expert validation is mostly positive with some remarks on the usage of ING /Viz. Regarding the first remark, that developers do not need this tool in their day to day work, we agree that it is correct. However, we argue that this tool is not intended for the day to day work of developers. We rather see this tool being used by software architects and product owners in cross team evaluations of the architecture. Having a tangible overview of the complete architecture makes it easier for teams to discuss possibilities for improvement. Therefore, we think the remark is valid, however it does not apply to our intended use case of the tool. Regarding the second remark to have a clear separation between our tool and Application Performance Management tools such as AWS X-Ray [1] and Datadog [2] as these also provide architecture overviews. This is surely something to consider when continuing work on this project. But we see a clear distinction between APM tools and ING/Viz. APM tools focus on real time metrics and incident analysis. ING/Viz focuses on historic metric data and long term effects of the application's architecture. We are certain that ING/Viz has a clear use case that does not overlap with APM tools.

The discussion highlights the main strength of the developed tool. It can spark discussion about architectural design decisions and allow teams to potentially find better solutions. We envision the tool being used in regular architecture audits similar to security audits. In these audits teams use the tool to identify weaknesses in their architecture and develop mitigation strategies to ensure high availability for essential services. For example, using sharding for the MDM service as discussed above. Furthermore, this could allow teams to monitor anti-patterns in the architecture development over a long period and help them prevent bad architectural solutions. For example, our approach

[1] https://aws.amazon.com/xray/
[2] https://www.datadoghq.com/

flags all gateway services and NGINX [3] services as mega-service, bottleneck and ambiguous-service. After consulting domain experts we learn that all gateways are thin infrastructure layers and while being a key element in a MSA, they should not be seen as a microservice. However, many enterprise applications add increasingly more logic, such as circuit breakers, to their gateways. These solutions might offer short term gains but once too much logic is put in the gateways they should be regarded as microservices and consequently show anti-pattern symptoms. Recently, this exact problem has been slowing down Netflix, which addresses the problem with graph-federation [41].

6.2 Future Work

6.2.1 Automatic Anti-Pattern Detection

There are four future work possibilities for the automatic anti-pattern detection methodology. Firstly, in future work we want to verify in depth the results from the methodology. That is to contact a majority of the teams within ING and discuss whether their service is rightfully flagged as an anti-pattern. Any findings from this work we then want to use to improve the matching of metrics with anti-patterns. This could be achieved by changing the definition of the anti-pattern vectors to get more accurate results. Here a multiple regression could be used to find the best combination. Another possibility for improving the anti-pattern detection is by changing the linking approach. At the moment a simple euclidean distance is computed but the application of machine learning techniques could lead to better results [29]. Additionally, we want to expand the methodology to include more metrics and anti-patterns. For this research we choose the metrics and anti-patterns that can be detected with the available data and that would provide the most insight for ING. This research shows the potential of the developed methodology and that it is worth expanding on this research. Thus, in future work, by adding more metrics, we expect to be able to better differentiate and detect more anti-patterns. Moreover, for the final prioritization of anti-patterns, we look to linking with the strategic software development roadmap which includes prioritization of feature-areas as well as development-hot-spots [42]. In future work we want to look at measuring the metrics development over an extended period of time. Analyzing the computed metrics over an extended period of time could provide insight into whether the overall architecture is improving or deteriorating. This could help teams to prioritize their efforts to maintain or improve an application's architecture. Finally, we expect knowing more about the business role of each microservice could lead to great improvements in identifying anti-patterns. However, as shown in this research it is not always given that the business roles of a service are known. Therefore, we want to look at automatic approaches for business role extraction. One possible direction for automated extraction of knowledge about the roles of services in the overall design is the use of role-stereotypes [43].

6.2.2 ING/Viz tool

Parties from the TU/e as well as ING show interest in continuing the work on ING/Viz. The source code for the project is transferred to two teams within ING. They maintain and continue to work on the project via new graduate interns. At the end of this thesis we get in contact with a new team from ING, that retrieves the Microservice Architecture and additional information from several data sources within ING. They are interested in collaborating on this project to provide more data points for the dashboard. For example, they link services to their respective teams. This information could be used in combination with the community detection algorithms and could be an interesting overlay in the dashboard. Thus, in future work we can look into how to improve the current functionality and update ING/Viz to take advantage of any improvements from the automatic anti-pattern detection.

6.3 Threats to validity

Wohlin et al. [44] provide a list of possible threats that researchers can face during a scientific research. In this section, we describe the actions taken in order to increase the validity and decrease the threats.

[3] https://www.nginx.com/

External validity concerns how the results and findings can be generalized. As discussed in Section 5.2 the computed distances between services and anti-pattern cannot be generalized as these are relative for each application. However, the presented methodology is generalizable and can be used to implement a similar tool in other applications. To increase the external validity we provide as much detail as possible for the methodology section.

Internal validity is defined as the extent to which the observed results represent the truth in the studied population. We mitigate this threat by rigorously checking our approach and verifying that each step in our process produces correct results. Furthermore, we verify the collected data by manually validating samples with the responsible teams.

Construct validity concerns how the selected studies represent the real population to answer the research questions. This is one of the biggest challenges of our studies because the provided span annotations are not uniform throughout all teams. Some spans do not provide all information or provide wrong information. Also, not all services within ING implement the tracing client and are therefore not in the dataset. We are not able to mitigate these two threats as this would exceed the scope of this study.

Conclusion validity concerns the relations between the conclusions that we draw and the analyzed data. Even though we are not able to completely mitigate the construct validity threat, we document our methodology as detailed as possible and discuss any decisions made. Therefore, we believe that if other researchers were to duplicate this work, given the same application, they would draw the same conclusions.

6.4 Personal Recommendations for ING

My graduation internship at ING was kicked off by the question of how ING could improve the service dependencies within its microservice architecture. After a literature study preceding this graduation internship the scope and research question for this graduation internship was determined. Finalizing my internship, I want to conclude my research with two recommendations regarding the original problem statement, by combining the insights from my research with the insights and impressions I gained working at ING.

Firstly, I think that more emphasis needs to be put on observability. The first step to solve a problem is to know and understand the current situation. However, until now there has been no visualization of the architecture of ING. And there is still no overview of the entire ING architecture as this thesis only collects data on a subset of business domains of ING. No one knows all existing services and how they are connected. This makes it challenging to understand the current situation, monitor progress and therefore makes it hard to determine solutions for current dependency problems. This graduation thesis shows that a visualization can greatly improve discussion regarding architecture and therefore I recommend continuing the work on visualizing ING architecture. Furthermore, there needs to be a greater push towards widespread usage of Distributed Tracing. At the moment not all services use DT leading to blind spots in the architecture. Not all spans are annotated correctly making it more difficult to trust the data collected from DT. However, as every team manages its span's annotations a company wide push for improving DT is needed. Another part of observability is accountability. At the moment it is possible, but too difficult, to know which teams develop which service. This makes simple tasks, like sending emails, unnecessarily difficult. Consequently, hampering productivity as communication between teams is challenging.

Secondly, ING has a large code base and therefore is still in the process of migrating to the microservice architecture. I think this is a necessary migration to ensure the growth of the application. However, I would recommend siding on initially creating fewer but larger services. Many small services inherently create many dependencies and a lot of maintenance overhead. If a service turns out to be a mega service, it can still be split into smaller services. From the results of this research we see that currently

there are as many nano-services as mega services. But the developers interviewed did not expect nano-services to be a problem and were focused on the number of mega-services. Thus, there might be a bias towards creating many small services. I would recommend creating an awareness that creating services is not a goal in itself.

6.5 Summary

In this chapter we discuss the results of this graduation thesis, outline future work possibilities, address threats to validity and provide recommendations for ING.

From the results we conclude that our approach for automatic anti-pattern detection can identify anti-patterns within MSAs. However, additional domain knowledge is needed to definitively determine whether a tagged service's metrics express an anti-pattern. Additionally, our approach can help to confirm or disprove developer's intuition of architectural refactoring possibilities. We find that our developed tool can spark discussion about architectural design decisions and allow teams to potentially find better solutions. We envision the tool being used in regular architecture audits similar to security audits. In these audits, teams use the tool to identify weaknesses in their architecture and develop mitigation strategies to ensure high availability for essential services.

There are numerous future work possibilities for the automatic anti-pattern detection methodology. We want to perform an extensive verification of our results, improve the matching of metrics with anti-patterns, add more metrics and anti-patterns to the approach, measure our metrics over an extended period of time and combine our metrics with business roles of services. Additionally, we want to continue work on the ING/Viz tool within ING and add new functionality as well as adapt it to work with changes made to the automatic anti-pattern detection methodology.

Chapter 7

Conclusion

In this thesis, we present our approach for automatically recovering microservice based application architectures via distributed tracing and detecting architectural anti-patterns within the recovered architecture. Furthermore, we present a tool for visualizing the system's architecture in an interactive dashboard. The dashboard allows the filtering of specific services and enables developers to explore the application's architecture. Additionally, the tool provides extra information such as the success-rate of traces between services as well as linking traces to business processes.

From the results we conclude that our approach for Microservice Architecture recovery via Distributed Tracing works but is impeded by the reality of an industrial application. The expert validation confirms that the detected service dependencies are correct, however, several services and dependencies are missing due to the inconsistent way that spans are traced and annotated. Furthermore, we learn that linking the computed service metrics to anti-patterns provides more insight for developers than solely the metrics. We also conclude that our approach for automatic anti-pattern detection can identify anti-patterns within MSAs. However, additional domain knowledge is needed to definitively determine whether a tagged service's metrics express an anti-pattern. Additionally, our approach can help to confirm or disprove developer's intuition of architectural refactoring possibilities. We find that our developed tool can spark discussion about architectural design decisions and allow developers to potentially find better solutions. We envision the tool being used in regular architecture audits similar to security audits. In these audits developers use the tool to identify weaknesses in their architecture and develop mitigation strategies to ensure high availability for essential services. However, more research is needed on improving the detection anti-patterns based on metrics. In the case of the ambiguous-service the results are similar to the mega-service and bottleneck anti-pattern results. From analyzing the results, we conclude that a different set of metrics might be needed as the current metrics fail to correctly capture the symptoms of the anti-pattern. Likewise, the results for the bottleneck-service anti-pattern are very similar to the mega-service results. Therefore, more work is needed to better define or expand the anti-pattern vectors, to ensure the results have a clearer separation.

In short, the combination of ING/Viz and our methodology for automatic anti-pattern detection in microservice architectures based on distributed tracing is a significant step in helping developer teams to identify anti-patterns in large and complex Microservice Architectures. Furthermore, we provide ample possibilities for future research directions.

Bibliography

[1] M. F. J. Lewis. Microservices. [Online]. Available: http://martinfowler.com/articles/microservices.html

[2] C. R. Justin Reynolds. Vizceral open source. [Online]. Available: https://netflixtechblog.com/vizceral-open-source-acc0c32113fe

[3] bocoup. Microservice networks. [Online]. Available: https://bocoup.com/work/buoyant

[4] A. Cloud. Visualizing a microservices architecture with ahas. [Online]. Available: https://alibaba-cloud.medium.com/visualizing-a-microservices-architecture-with-ahas-d763167013b7

[5] S.-P. Ma, C.-Y. Fan, Y. Chuang, I.-H. Liu, and C.-W. Lan, "Graph-based and scenario-driven microservice analysis, retrieval, and testing," *Future Generation Computer Systems*, vol. 100, pp. 724–735, nov 2019. [Online]. Available: https://doi.org/10.1016/j.future.2019.05.048https://linkinghub.elsevier.com/retrieve/pii/S0167739X19302614

[6] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, and L. Su, "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*. New York, NY, USA: ACM, nov 2020, pp. 1387–1397. [Online]. Available: https://dl.acm.org/doi/10.1145/3368089.3417066

[7] M. Kleehaus, Ö. Uludağ, P. Schäfer, and F. Matthes, "MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-Based Environments," in *Lecture Notes in Business Information Processing*, 2018, vol. 317, no. June, pp. 148–162. [Online]. Available: http://link.springer.com/10.1007/978-3-319-92901-9_14

[8] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, no. April. IEEE, apr 2017, pp. 298–302. [Online]. Available: https://ieeexplore.ieee.org/document/7958510/

[9] A. Kobyliński and A. Sobczak, *Perspectives in Business Informatics Research*, ser. Lecture Notes in Business Information Processing, R. A. Buchmann, A. Polini, B. Johansson, and D. Karagiannis, Eds. Cham: Springer International Publishing, 2020, vol. 398. [Online]. Available: http://link.springer.com/10.1007/978-3-642-40823-6http://link.springer.com/10.1007/978-3-030-61140-8

[10] I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi, "Towards microservice smells detection," in *Proceedings of the 3rd International Conference on Technical Debt*. New York, NY, USA: ACM, jun 2020, pp. 92–97. [Online]. Available: https://dl.acm.org/doi/10.1145/3387906.3388625

[11] T. Engel, M. Langermeier, B. Bauer, and A. Hofmann, "Evaluation of Microservice Architectures: A Metric and Tool-Based Approach," in *Lecture Notes in Business Information Processing*. Springer International Publishing, 2018, vol. 317, pp. 74–89. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-92901-9_8

[12] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic Internet services," in *Proceedings International Conference on Dependable Systems and Networks*, no. February. IEEE Comput. Soc, 2002, pp. 595–604. [Online]. Available: http://ieeexplore.ieee.org/document/1029005/

[13] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Proceedings of the 4th USENIX Conference on Networked Systems Design; Implementation*, ser. NSDI'07. USA: USENIX Association, 2007, p. 20.

[14] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and performance-aware systems," *Proceedings of HotOS 2003 - 9th Workshop on Hot Topics in Operating Systems*, pp. 85–90, 2003.

[15] B. H. Sigelman, L. Andr, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper , a Large-Scale Distributed Systems Tracing Infrastructure," *Google Research*, no. April, p. 14, 2010. [Online]. Available: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36356.pdf

[16] Y. Shkuro. Jaeger and opentelemetry. [Online]. Available: https://medium.com/jaegertracing/jaeger-and-opentelemetry-1846f701d9f2

[17] K. Ranganathan. Netflix shares cloud load balancing and failover tool: Eureka! [Online]. Available: https://netflixtechblog.com/netflix-shares-cloud-load-balancing-and-failover-tool-eureka-c10647ef95e5

[18] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)," *Dagstuhl Reports*, vol. 6, no. 4, pp. 110–138, 2016. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6693

[19] T. Besker, A. Martini, R. Edirisooriya Lokuge, K. Blincoe, and J. Bosch, "Embracing technical debt, from a startup company perspective," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 415–425.

[20] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, and A. Zimmermann, "Towards a collaborative repository for the documentation of service-based antipatterns and bad smells," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019, pp. 95–101.

[21] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 1008–1028, 2021.

[22] S. S. de Toledo, A. Martini, and D. I. Sjøberg, "Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study," *Journal of Systems and Software*, vol. 177, no. April, p. 110968, jul 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121221000650

[23] R. Tighilt, M. Abdellatif, N. Moha, H. Mili, G. E. Boussaidi, J. Privat, and Y.-G. Guéhéneuc, "On the Study of Microservices Antipatterns," in *Proceedings of the European Conference on Pattern Languages of Programs 2020*, no. 1. New York, NY, USA: ACM, jul 2020, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3424771.3424812

[24] S. Even, *Graph Algorithms*, 2nd ed., G. Even, Ed. Cambridge University Press, 2011.

[25] J. Bogner, S. Wagner, and A. Zimmermann, "Automatically measuring the maintainability of service- and microservice-based systems," in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, vol. Part F1319, no. October. New York, NY, USA: ACM, oct 2017, pp. 107–115. [Online]. Available: https://dl.acm.org/doi/10.1145/3143434.3143443

[26] ——, "Towards a practical maintainability quality model for service-and microservice-based systems," in *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, vol. Part F1305, no. September. New York, NY, USA: ACM, sep 2017, pp. 195–198. [Online]. Available: https://dl.acm.org/doi/10.1145/3129790.3129816

[27] F. Palomba, A. De Lucia, G. Bavota, and R. Oliveto, "Anti-pattern detection: Methods, challenges, and open issues," in *Advances in Computers*. Elsevier, 2014, vol. 95, pp. 201–238.

[28] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Guéhéneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 466–475.

[29] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aimeur, "Support vector machines for anti-pattern detection," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 278–281.

[30] R. Fourati, N. Bouassida, and H. B. Abdallah, "A metric-based approach for anti-pattern detection in uml designs," in *Computer and Information Science 2011*. Springer, 2011, pp. 17–33.

[31] M. Nayrolles, N. Moha, and P. Valtchev, "Improving soa antipatterns detection in service based systems by mining execution traces," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 321–330.

[32] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 282–285.

[33] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 486–496.

[34] E. Gaidels and M. Kirikova, "Service dependency graph analysis in microservice architecture," in *Perspectives in Business Informatics Research*, R. A. Buchmann, A. Polini, B. Johansson, and D. Karagiannis, Eds. Cham: Springer International Publishing, 2020, pp. 128–139.

[35] U. BRANDES and C. PICH, "Centrality estimation in large networks," *International Journal of Bifurcation and Chaos*, vol. 17, no. 07, pp. 2303–2318, 2007. [Online]. Available: https://doi.org/10.1142/S0218127407018403

[36] Neo4j. Local clustering coefficient. [Online]. Available: https://neo4j.com/docs/graph-data-science/current/algorithms/local-clustering-coefficient/

[37] J. Bogner. Service-based antipatterns. [Online]. Available: https://xjreb.github.io/service-based-antipatterns/

[38] A. Cockburn, A. Karlson, and B. B. Bederson, "A review of overview+detail, zooming, and focus+context interfaces," *ACM Comput. Surv.*, vol. 41, no. 1, Jan. 2009. [Online]. Available: https://doi.org/10.1145/1456650.1456652

[39] N. Watt. Explore your microservices architecture with graph theory & network science. [Online]. Available: https://www.youtube.com/watch?v=0G5O1ffYIPI&t=1578s

[40] Wikipedia. Ing group. [Online]. Available: https://en.wikipedia.org/wiki/ING_Group

[41] Netflix. How netflix scales its api with graphql federation (part 1). [Online]. Available: https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2

[42] M. R. V. Chaudron, B. Katumba, and X. Ran, "Automated prioritization of metrics-based design flaws in uml class diagrams," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2014, pp. 369–376.

[43] A. Nurwidyantoro, T. Ho-Quang, and M. R. V. Chaudron, "Automated classification of class role-stereotypes via machine learning," in *Proceedings of the Evaluation and Assessment on Software Engineering*, 2019, pp. 79–88.

[44] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.