Tampere University

Mohammad Imranur Rahman

ANALYSIS OF MICROSERVICE COUPLING MEASURES

Faculty of Information Technology and Communication Sciences
Master of Science Thesis
April 2022

ABSTRACT

Mohammad Imranur Rahman: Analysis of Microservice Coupling Measures
Master of Science Thesis
Tampere University
April 2022

Microservices architectures are composed of a collection of modular, fault-tolerant services. In recent years, the software engineering community has published research on viable, recurring, and effective architectural patterns in microservices-based architectures, as they are critical to the maintenance and scaling of microservice-based systems. As well as, ensuring low coupling and strong cohesion among the microservices that comprise the cloud-native application is a crucial property.Services that are loosely connected and highly coherent allow development teams to work in parallel, eliminating communication overhead between teams.

In the first section of this thesis, we attempted to generate a dataset by starting with a selected list of microservice-based projects. The collection is made up of 20 open-source applications that all use certain microservice architecture patterns. Furthermore, the dataset includes information about the aforementioned projects' interservice calls and dependencies.

In the second section, we suggested methods for computing and visualizing the coupling between microservices by expanding and adapting the notions underlying standard of structural coupling calculation. We validate these measures using a case study of 17 projects selected from the aforementioned dataset, and we propose an automated method for measuring them. The findings of this study emphasize how these metrics give practitioners with quantitative and visual views of service architecture, that can be used to design advanced measures to monitor the development of services.

Keywords: SOA, Microservice, Architectural pattern, Service coupling

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

PREFACE

First, I would like to thank my supervisor, Davide Taibi for giving me the opportunity to work on this interesting topic and offering excellent assistance throughout the work. This thesis is the final outcome of about three years of working and studying simultaneously. Again thanks goes to my professor Davide Taibi for his tremendous patience I found in supervising me.

I'd also want to thank my friends and family for their assistance in relieving my tension during the writing process as well as continuously motivating me. My wife Fatima deserves special recognition for her unwavering moral support.

Tampere, 4th April 2022

Mohammad Imranur Rahman

CONTENTS

1 Introduction 1
1.1 Thesis outline 3
2 Background 4
2.1 Service oriented architecture 4
2.2 Microservices 6
2.3 Containerization 11
2.3.1 Docker 13
2.3.2 Docker compose 14
2.4 Microservices architectural patterns 15
2.4.1 API-Gateway pattern 16
2.4.2 Circuit breaker pattern 18
2.4.3 Service discovery pattern 19
2.5 Microservices anti patterns 22
3 Context 27
3.1 Goal and Research Questions 27
3.2 Project selection 28
4 Implementation and dataset 30
4.1 SLOCCount 30
4.2 GraphML 32
4.3 MicroDepGraph 34
4.4 Dataset generation using MicroDepGraph 35
5 Microservices metrics 37
5.1 Proposed matrices 40
5.1.1 Structural coupling 40
5.1.2 Microservice coupling measures 41
6 Results 43
7 Discussion 46
7.1 Threats to Validity 47
8 Conclusion 48
References 50

LIST OF FIGURES

2.1 Simple Service Oriented Architecture. [8] 5
2.2 Architecture of a Microservices system 6
2.3 Advantages of Microservices system [14] 7
2.4 Microservices orchestration [17] 8
2.5 Microservices choreography [17] 9
2.6 Microservices technologies timeline [18] 10
2.7 Monolith microservices architectural design [19] 11
2.8 Container based deployment architecture [21] 12
2.9 A sample Dockerfile 13
2.10 An example of docker compose file [22] 14
2.11 API gateway pattern 17
2.12 Circuit breaker pattern state machine 19
2.13 Client side discovery pattern 20
2.14 Server side discovery pattern 21
2.15 Cyclic dependencies 22
2.16 Shared libraries 23
2.17 Shared libraries 24
2.18 Shared database 25
2.19 Shared database refactored 26

4.1 Dependency graph 32

5.1 Structural Coupling: graph representation 40

6.1 An example of Microservices-based System 43

LIST OF TABLES

3.1 Selected microservices projects 29

4.1 Generated dataset of the projects 35

5.1 The Metrics Proposed in the Literature 39

6.1 Example of metrics for the system in Figure 6.1 43
6.2 SIY, LWF, GWF and SC for the system in Figure 6.1 44
6.3 Generated dataset of the projects 44
6.4 Results of the Coupling Metrics Applied to the 17 projects. 45
6.5 Results of the LWF and GWF of the 17 projects. 45

LIST OF PROGRAMS AND ALGORITHMS

4.1 GraphML file 33

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Program Interface
DRY	Don't Repeat Yourself
ESB	Enterprise Service Bus
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
KLOC	1,000 Lines of Code
LOC	Lines of Code
LXCs	LinuX Containers
MSA	Microservices architecture
OO	Object oriented
OS	Operating system
PaaS	Platform-as-a-service
REST	Representational State Transfer
SOA	Service oriented architecture
URL	Uniform Resource Locator
VM	Virtual machine

1 INTRODUCTION

Like any other discipline Software Architecture is evolving to meet the growing need of shift towards modular, loose coupling and robustness. Service-oriented architecture(SOA) is one of those architectural styles which emerged to support developing of distributed systems where the components or modules are defined as services. SOA has been in use in industry from early 90's. But the problem associated with the development of large scale enterprise applications were first introduced approximately in 1960's[1].

These problems arises mostly by the usage of monolith architecture of the systems, where all the components of the system such as business logic, data access layers and user interfaces are combined in a single and self contained program. In monolith architecture the components are tightly coupled with each other so that failure of one component of the system affects the whole system and keep the system down until the component is fixed.

The drawbacks of monolith architecture has been tried to be solved in SOA. But as the application grows the complexity to maintain the application gets cumbersome in SOA. In the SOA paradigm, to consume services from different consumers there is a need for Enterprise Service Bus (ESB) to integrate different services or systems. ESB also holds some part of business logic which makes ESB's hard to maintain and more coupled. Thus there was a clear need for better architecture which can overcome the drawbacks of SOA/ESB paradigm and conventional monolithic architecture.

With the popularity of cloud applications and cloud computing new architectural styles have emerged. One of those architectural styles is "Microservices". The term microservices was first coined by Martin Fowler [2] in 2011 at the workshop of software architects. The microservice architectural style is not only thought by Martin Fowler, there are also other practitioners who previously thought somehow similar ideas to microservices for instance Werner Vogels from Amazon described this approach as “encapsulating the data with the business logic that operates on the data,with the only access through a published service interface” [3].

According to Fowler, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these

services, which may be written in different programming languages and use different data storage technologies [2]. As mentioned microservices have independent services these services are easy to maintain and test. These systems can be highly fault tolerant and scalable as these are deployed separately and having their own technology stack. Because of the aforementioned benefits microservices architecture is getting popular nowadays.

In a recent survey done by O'Reilly on 1502 participants of whom most of them were of technical background ranging from big enterprises to small companies. In the survey most of the respondents said they are mostly successful with microservices [4]. In the survey the respondents highlighted the benefits they got by using microservices architectural style. The largest portion for the response regarding benefits of using microservices, 45% respondents named "feature flexibility," followed just under 45% respondents mentioned "responding quickly to changing technology and business requirements" as main benefits.

While the survey talks about the benefits of using microservices it also sheds light on pain points of using or migrating towards microservices architecture from a monolith system. The most faced impediment mentioned by the respondents was the mindset of the people in the organization. As each components of microsevices organized around the business capabilities of an organization so its very important to have the right mindset towards MSA. Another most responded migration issue towards microservices is decoupling the services in granular components from monolith systems. This issue is also found by an empirical investigation done by Davide et al. One of the most common issues towards adoption of MSA is the complexity to decouple services from the monolithic system [5].

Designing and implementing microservices can be complex and daunting. After putting each services in place and setting up the communication between the services its not done yet. Maintaining this new system based on MSA can be daunting when requirements and business grows. To help developers maintaining and understanding their MSA systems, in the context of this thesis we focused on understanding different matrices between services in MSA.

1.1 Thesis outline

The rest of this paper is organized as follows.

The theoretical backdrop, basic notions behind this study, and all the theory applied and used in subsequent chapters are covered in Chapter 2.

In contrast, chapter 3 delves deeper into the study's setting, research topic, and methodologies.

The project selection and introduction to the in-house developed tool to produce dataset and visualization of the microservices projects architecture, as well as other tools needed to generate the dataset that will be utilized in subsequent study, are covered in Chapter 4.

The metrics we suggested to measure the connection between services in microservices systems are discussed in Chapter 5. We also offer other matrices from other literatures in this chapter, with one of them being compared to our suggested metrics.

Documenting the findings is the topic of Chapter 6. The results of several matrices used to microservices projects are shown in this section.

In Chapter 7, we addressed the outcomes as well as the constraints that were discussed.

Lastly, chapter 8 finishes this master's thesis by providing a concluding summary of the work and exploring the potential for future work within the scope of this thesis.

2 BACKGROUND

This chapter introduces the main topics related to microservices architectures including a brief introduction to microservices architectural patterns and anti-patterns. Therefore, this chapter focuses on topics that are important for understanding the different concepts of microservices and its matrices.

The first Section 2.1 discusses fundamental of microservices. Then second section 2.2 gives a brief idea about microservices architectural patterns. After that third section 2.3 introduces different microservices anti-pattern.

2.1 Service oriented architecture

Enterprise systems are evolving rapidly from monolithic silos to distributed systems through service oriented architecture. IT companies need to change their legacy processes to address evolving company challenges meeting real time without second chances. In order to run and decouple business processes and underlying structures flexibly, service-oriented architectures (SOAs) have evolved. The term Service Oriented Architecture was first coined by Roy Shulte and Yefim Natis in 1996 to define a model of multi-tier architecture which lets organizations share logic and data across multiple applications [6] [7]. Service-Oriented Architecture (SOA) is a software architectural model that uses services as key elements for the creation of applications/solutions. It is a model which organizes a set of capabilities, often spread across the network and likely under the influence of various domains of ownership. Organized skills can be used to solve business problems. A business problem is generally characterized as any problem, in any domain of interest, encountered by an entity or an organization as it relates to its business. Services are self- describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications [8]. Services accomplishes tasks that can be anything from basic requests to complex business processes. In other words, SOA is using a loosed coupling design theory where each service is an independent entity with a minimal dependence on other common resources such as databases, legacy applications or APIs. Achieving this form of interaction involves a means of putting together a collection of capabilities that may exist at various physical locations and be managed by different spheres of ownership, and integrating them to meet the needs of the user [9].

A system built on the SOA paradigm must provide visibility of needs and capabilities; must include a means for consumers and providers to interact; and must produce real

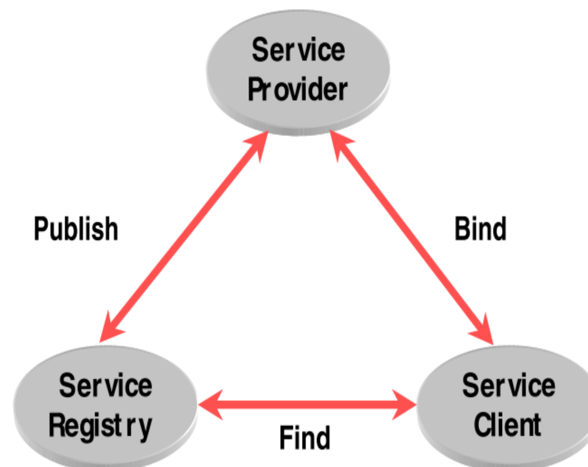

Figure 2.1. *Simple Service Oriented Architecture. [8]*

world effects that address a consumer's business problem [10].

The simple SOA describes a communication among autonomous agents as a transmission of messages between service requesters (clients) and service providers. Consumers are software agents who query the execution of a service. Providers are software entities who deliver the service. Services can act concurrently both as consumers and suppliers. In the basic SOA system Consumers and providers are not the only thing in the architecture, it also consists of service discovery or service registry. As we can see in Figure 2.1 the service provider, who operates, executes, and manages access to the services; a service client, which can be an application, program, or client who is requesting and calling a service; and a service registry acts as a broker that gathers every one of the services together and keeps a registry of services available [8]. The drive for enterprise automation, inspired by possibilities in terms of cost savings and higher performing, more efficient implementations, has created the need for integration of different applications. Integration has been one of the driving forces in the software industry during the late nineties. Integration has been area of research interest in the field of SOA integration. There a number of technologies available to realize SOA. Among them, Web services and the set of related specifications, and also services that are built following the REST (REspresentation State Transfer) architecture (called RESTful services) are gaining the momentum for integration at the data level [11]. The communication amongst client and the service providers may also be built on messaging platforms, such as Apache Kafka, RabbitMQ, Apache ActiveMQ etc. These solutions provide mainly asynchronous message transfers between distributed applications in a point- to-point (sender-receiver) or publish-subscribe manner. SOA has basically two integration approach which are (1) direct point-to-point and (2) hub-and-spoke [12]. Out of these two patterns hub-and-spoke is the most used in enterprise integration, and this brokering system is called ESB (Enterprise Service Bus).

2.2 Microservices

Microservices is an architectural style influenced by service-oriented architecture which has recently began gaining prominence in decomposing monolith systems. Software architectural patterns have been going through progressive transition towards delivery, modularization and loose coupling, with both the purpose of increasing code reuse and reliability [13]. Microservices architecture is getting in demand amongst companies for their new software systems as well as in enterprise systems where big enterprises also adopting this architecture quite fast. Microservices are small and decentralized services deployed independently, for a common and explicitly specified task [2]. Microservices are comparatively small and independent services that operate together, are designed on a company requirement, and have a clear and well specified objective. Microservices facilitate autonomous deployment, encouraging small teams to operate on isolated and oriented services while choosing the most appropriate technology for their job that can be implemented and scaled independently [2]. Currently, there has been significant interest in microservices architectures, across academia and industry communities together. Most definitions of microservices already arisen in recent times as groups seek to establish a definition. Microservices are a modularization concept. Their aim is to divide big software components into smaller bits. Therefore these control the company and development of information applications. Microservices may even be implemented independently regardless of one another. Improvements to one microservice should be put into development regardless of alterations to other microservices. Microservices could be implemented using various frameworks and technologies. There is no limitation on the programming language or the framework for each microservice. Microservices hold their own data storage: a personalized database or an entirely separate schema in a centralized database. In the figure 2.2 we can see a high level architectural view of a microservices system.

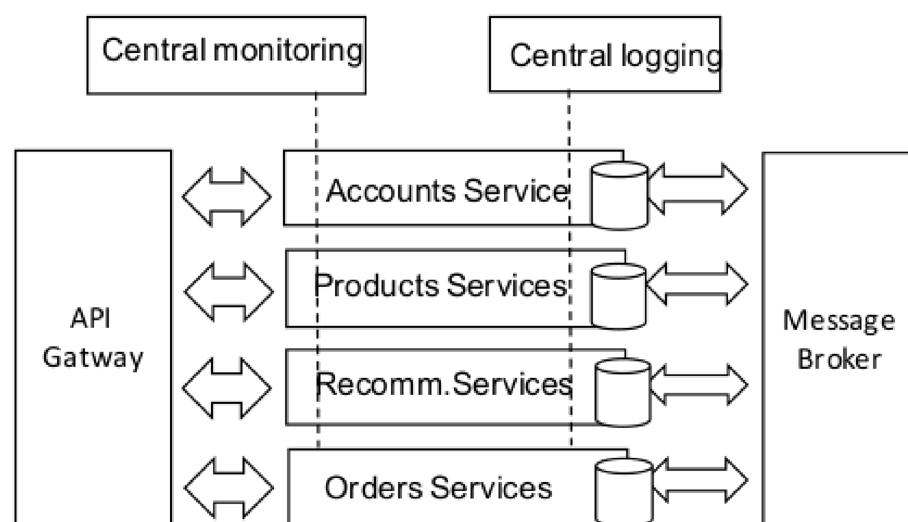

Figure 2.2. *Architecture of a Microservices system*

Microservices brings a great idea of service modularization. When a monolith system is built upon different components using various technologies and frameworks unwanted dependencies can instantly slip in. For instance, someone refers a class or method in such a context where it is not planned to be included. Doing this establishes a dependency that perhaps the designers of the class or method are not conscious of. Any improvements made to the class or feature may trigger unintended problems in some other component of the system. In a short time, too many dependencies would have arisen and the situation has deteriorated too much and the system will not further be maintained and improved. Microservices, in comparison, interact only through specific interfaces, which are implemented utilizing frameworks such as messaging queues or REST. This keeps the technological barriers on use of microservices greater, and therefore unnecessary dependencies are much less probable to appear. Some of the most important advantages of microservices are illustrated in figure 2.3 [14].

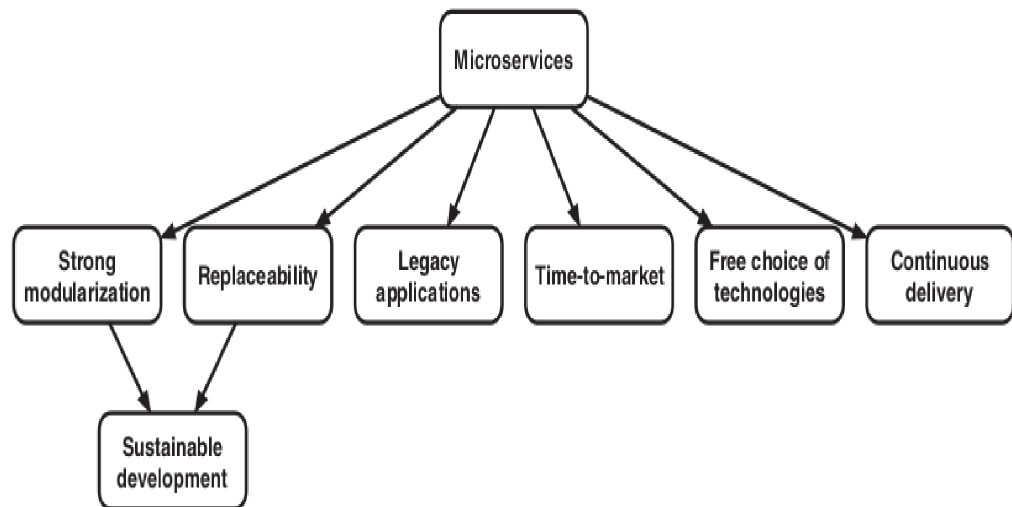

Figure 2.3. *Advantages of Microservices system [14]*

Microservices not only provide technical advantage it also benefits organizational structure as well as communication between different teams. According to an American computer scientist Melvin Edward Conway,

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure [15].

Each services in a microservice system built upon business capabilities where cross-functional teams are responsible for each of the services. In this way the support, development and lifecycle of each services is maintained by each teams.

According to Amazon CTO Werner Vogel he described this as

> You build, you run it.[3]

In this approach each microservices released and updated independently and on its own timetable. Nevertheless, to really leverage the potential of autonomous deployment, one has to employ extremely efficient integration and delivery methods. Microservices architecture is designed in a way that it can be used with continuous delivery and continuous integration, enabling every stage of delivery pipeline automated. While combining automated continuous delivery pipelines and contemporary container tools, it is feasible to release an updated version of a service to production in a matter of seconds.

The communication between microservices is a stateless independent and self-contained by nature [16]. To provide seamless communication between microservices there are two methods to achieve the collaboration orchestration and choreography. Orchestration is on of the most reliable and consistent technique of managing interactions between different services in microservices. In this approach, there is generally one central service that acts as the “orchestrator” of the actual service interactions. This maintains a request/response communication paradigm. Only the central controller is accountable for all communications. Orchestration gives an effective technique for managing the sequence of events whenever there is synchronous processing. There are some disadvantages of orchestration such as service dependency is the concerned point in the centralized environment, as orchestrator is a single point control and single coordinator if it goes down, all processing stops and application fails [17]. In figure 2.4 below it depicts an example of e-commerce application using orchestration method.

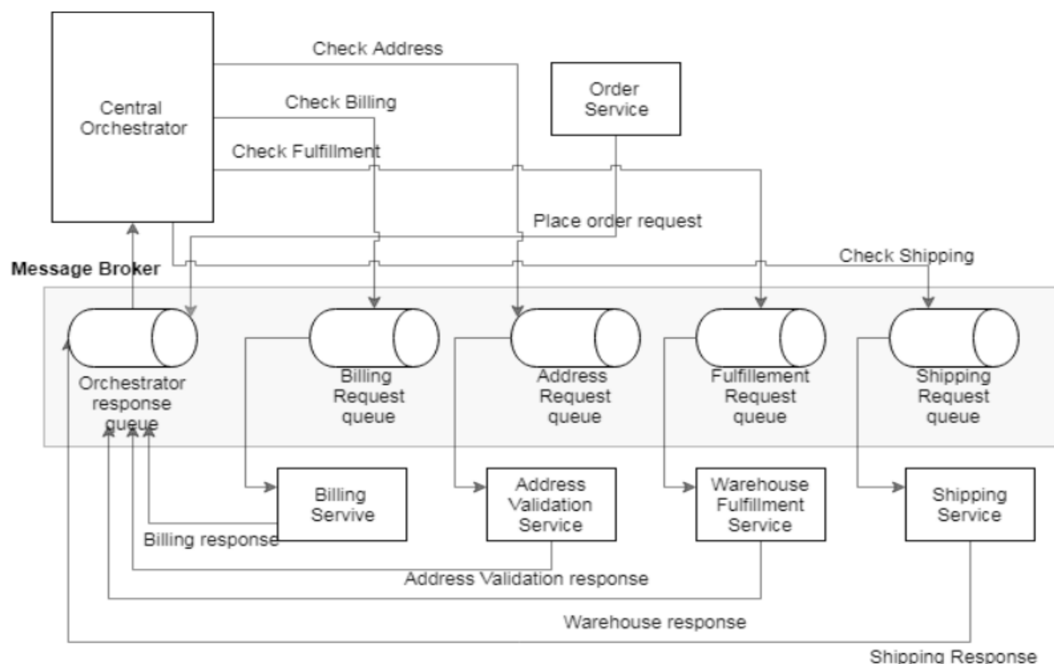

Figure 2.4. *Microservices orchestration [17]*

On the other hand another way to handle the communication in microservices is choreography based where the logic of centralised controller is put inside each service in advance. The services know what to react to and how, beforehand like an autonomous manner. Services utilize an event stream for asynchronous transmission of events. The request/response way of orchestration blocks and increases the wait time whereas in choreography this problem is solved by the asynchronicity of reactive architecture. Utilising event stream with this allows messaging across producer and consumers to be separated.

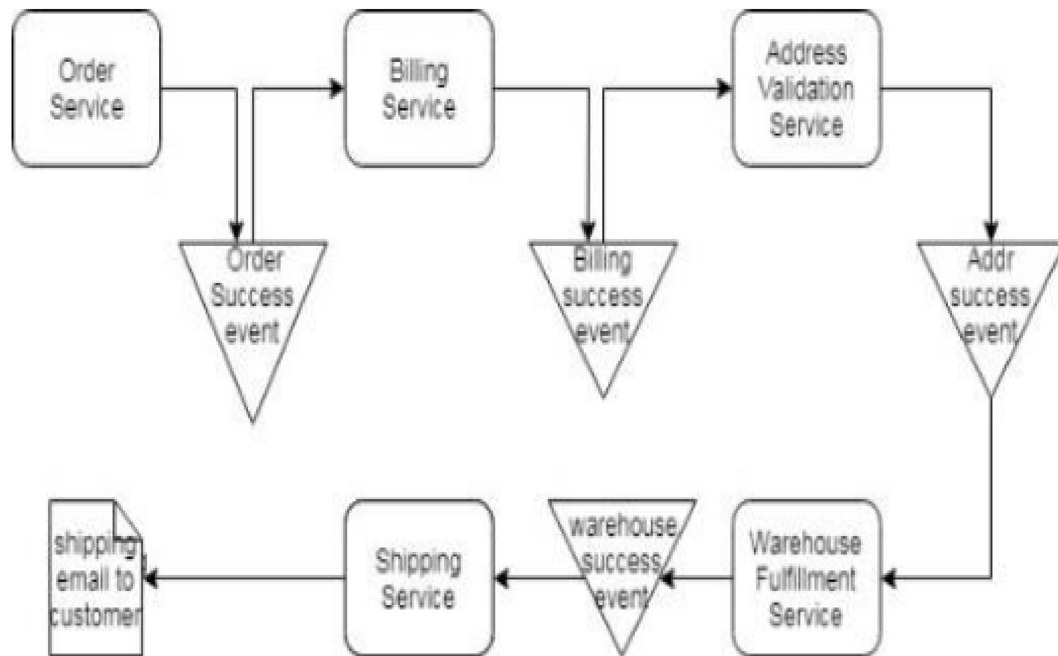

Figure 2.5. *Microservices choreography [17]*

As there are lots of advantages of microservices architecture there are also new technical difficulties need to be tackled while developing an maintaining microservice systems. The division of a system into microservices leaves the system as its own very complicated. Which causes problems at the operational level for instance, high latency periods in the network or the of specific services. There seem to be a lot of aspects to remember also at network infrastructure level for example, it can be difficult to transfer features among various microservices. Eventually, there are more modules to be individually distributed making processes and facilities quite complicated. Such issues need to be dealt with before implementing microservices. While microservice architectures have become more common, such tools began expanding to serve a larger, more varied consumer base, leading to the emergence of even more advanced technologies [18]. In the Figure 2.6 we can see a timeline of 10 waves of technologies, including some of the most popular tools, which have affected microservice technology growth, deployments. The great bulk of these tools illustrated in Figure 2.6 emerged through industry. Even though having industry roots, almost all of these technologies are openly accessible as open source projects. These waves' effect has also been represented over how microservice ap-

plications have progressed from an architectural standpoint. Nevertheless, since these libraries got highly complicated, and since integrating them in a different programming language is not really a straightforward process, developers were frequently constrained to create new services just utilizing the languages for which these libraries were mostly readily accessible.

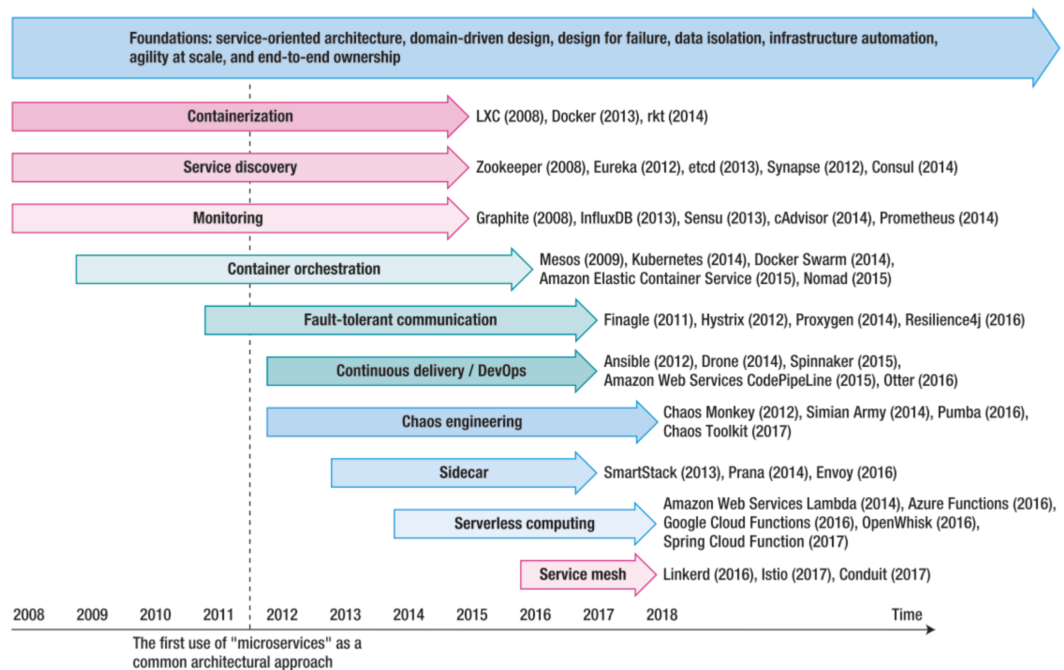

Figure 2.6. *Microservices technologies timeline [18]*

2.3 Containerization

Latest developments in container technologies and the ability to handle difficulties in virtualization have boosted the usage of containers in the cloud platforms for deploying and maintaining microservices based systems. This might also have paved the way for the adoption of a microservice architectural paradigm in cloud-hosted software by lowering infrastructure and maintenance costs [19]. Futhermore, organizing the software to be deployed in the cloud as a group of microservices enables cloud computing providers to give greater scalability promises towards more optimal use of cloud resources, and to proactively and swiftly reorganize software to suit rising client demand. Containerization is frequently explored as a lightweight virtualization technique to deploy and manage microservices systems. Using containers in MSA helps to avoid inefficient use of resources and optimize bundling the subsystems under a separate VM. This is feasible since microservices are generally lighter than traditional software components since they utilize lighter software technologies and platforms. Thus this allows to scale services based on their needs as well as migrating services faster from one vm to another. Figure 2.7 illustrates the comparison of both a traditional monolithic design and a microservice architecture.

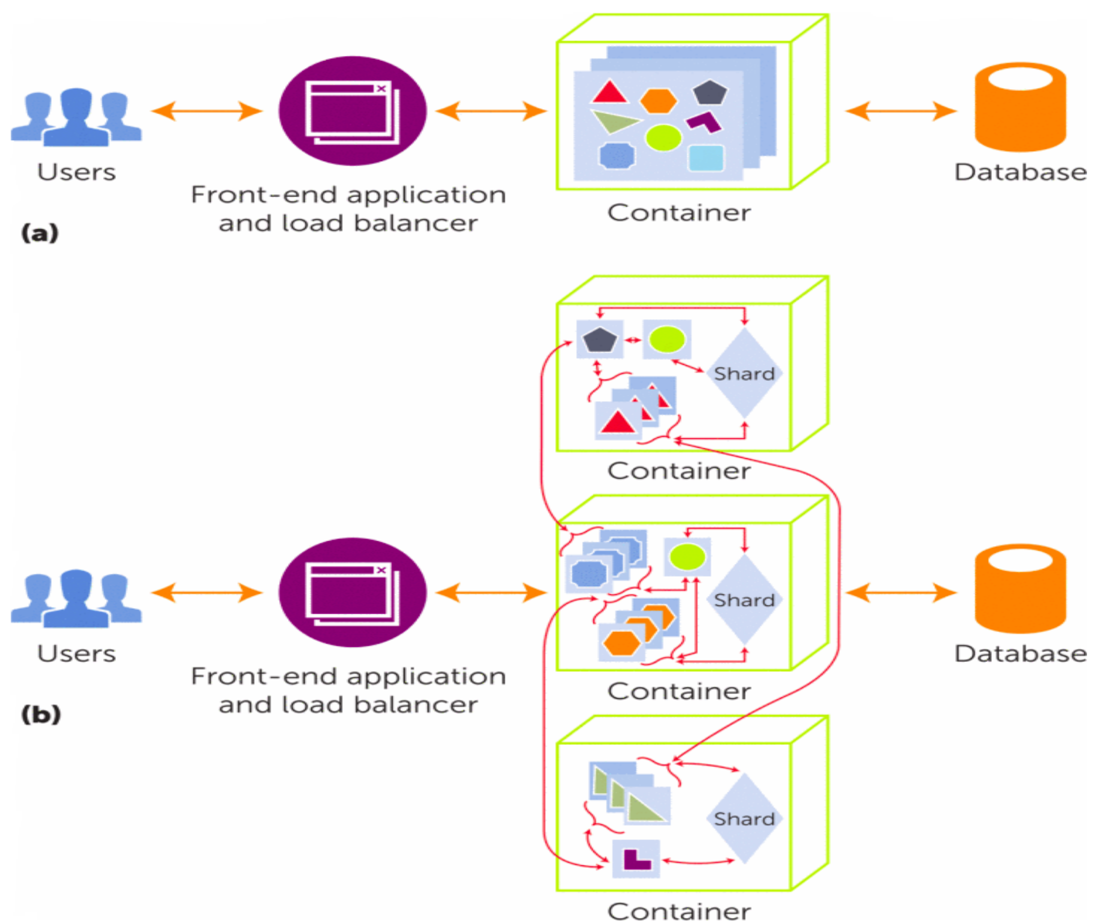

Figure 2.7. *Monolith microservices architectural design [19]*

In the first example, users interact with a front-end application, which sends client request

to instances of the application hosted inside a container and executes all application tasks using the data in the database. In second example, the application is partitioned into several small services, each performing its specific responsibility, distributed in various containers, replicated per system requirement. Cloud computing relies on virtualization system, like virtual machines Containers are a pretty similar but much more ultralight virtualization idea; which consumes less resource and time-consuming, therefore they've been recommended as a method for even more practical and robust application packaging and deployment in cloud platform. Containers are tools for delivering software-that is, they have a platform-as-a-service (PaaS) focus-in a portable way aiming at greater interoperability while still utilizing OS virtualization principles [20]. In containers, systems sharing an OS, binaries and libraries as a consequence the deployments of microservices components will be much lower in size than monolith deployments, making it feasible to maintain several of containers on a host machine. Since containers are using the host OS, rebooting a container doesn't require restarting the host OS. Figure 2.8 depicts the architectural design of a container based system where the applications shares an operating system resources.

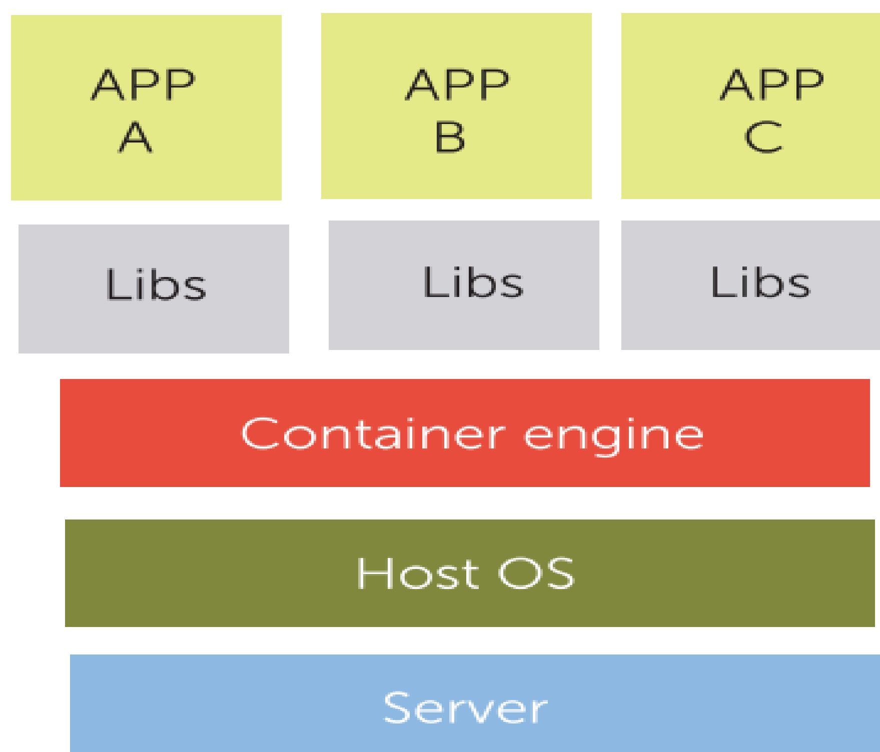

Figure 2.8. *Container based deployment architecture [21]*

2.3.1 Docker

Lightweight Docker containers are increasingly became a technology to deploy and deliver microservice oriented applications, Docker [1] is an open source platform based on container virtualization technology giving a standardized approach to manage the quicker deployment of services within small and compact containers. Docker leverages several strong kernel-level technologies and delivers these at our disposal. The notion of a container in virtualization has been present for some years, but through offering a simple toolbox and a consistent API for controlling various kernel-level technologies, such as LXCs (LinuX Containers) Docker has provided tool set to make building and interacting with containers as straightforward as possible. Containers isolate applications from one another. almost any Docker container is based on an image. The Docker image is essentially a predefined file system that also has a single layer of libraries and binaries which are needed to ensure application operate, and potentially the codebase as well as some additional packages. Containers are constructed on layers built from separate images placed on top of a basic image that may be expanded. An ideal Docker image comprises of portable application containers.

One of Docker's great advantage is its ability to locate, download and run container images that were developed by other developers or open source community rapidly. The repository where images are maintained is known as registry, and Docker Inc. maintains a public registry generally known as Central Index. The repositories play an important part in making it accessible to perhaps large numbers of reusable proprietary and public container images. The public Docker Registry includes images of readily available software, which include databases, content management systems, development environments, Web servers and so forth.

```
1  FROM centos
2
3  MAINTAINER maintainer@docker.com
4
5  RUN mkdir /opt/tomcat/
6
7  WORKDIR /opt/tomcat
8  RUN curl -O https://www-eu.apache.org/dist/tomcat/tomcat-8/v8.5.40/bin/apache-tomcat-8.5.40.tar.gz
9  RUN tar xvfz apache*.tar.gz
10 RUN mv apache-tomcat-8.5.40/* /opt/tomcat/.
11 RUN yum -y install java
12 RUN java -version
13
14 WORKDIR /opt/tomcat/webapps
15
16 EXPOSE 8080
17
18 CMD ["/opt/tomcat/bin/catalina.sh", "run"]
```

Figure 2.9. *A sample Dockerfile*

[1] Docker
https://docker.com

To build a docker image commands are usually executed in Dockerfiles to automate the build process. Each Dockerfile is a script consisting of multiple commands and parameters specified progressively to automatically execute activities on a base image to generate a new image. Figure 2.9 shows an example of how commands looks like in a Dockerfile. They're utilized to manage deployment artifacts and ease the deployment procedure from beginning to end. Whenever the build command is executed, Docker analyzes the Dockerfile to determine which commands to run and afterwards delivers the finished image. Each command adds an additional layer to the final image.

2.3.2 Docker compose

As discussed before, docker has become standard to create and develop container based applications but with this there are other problem arises such as it is required to address the issue of container management, which included their management, collaboration, and coordination of their operations, among other things. Docker Compose is a technology for coordinating several components using Docker.

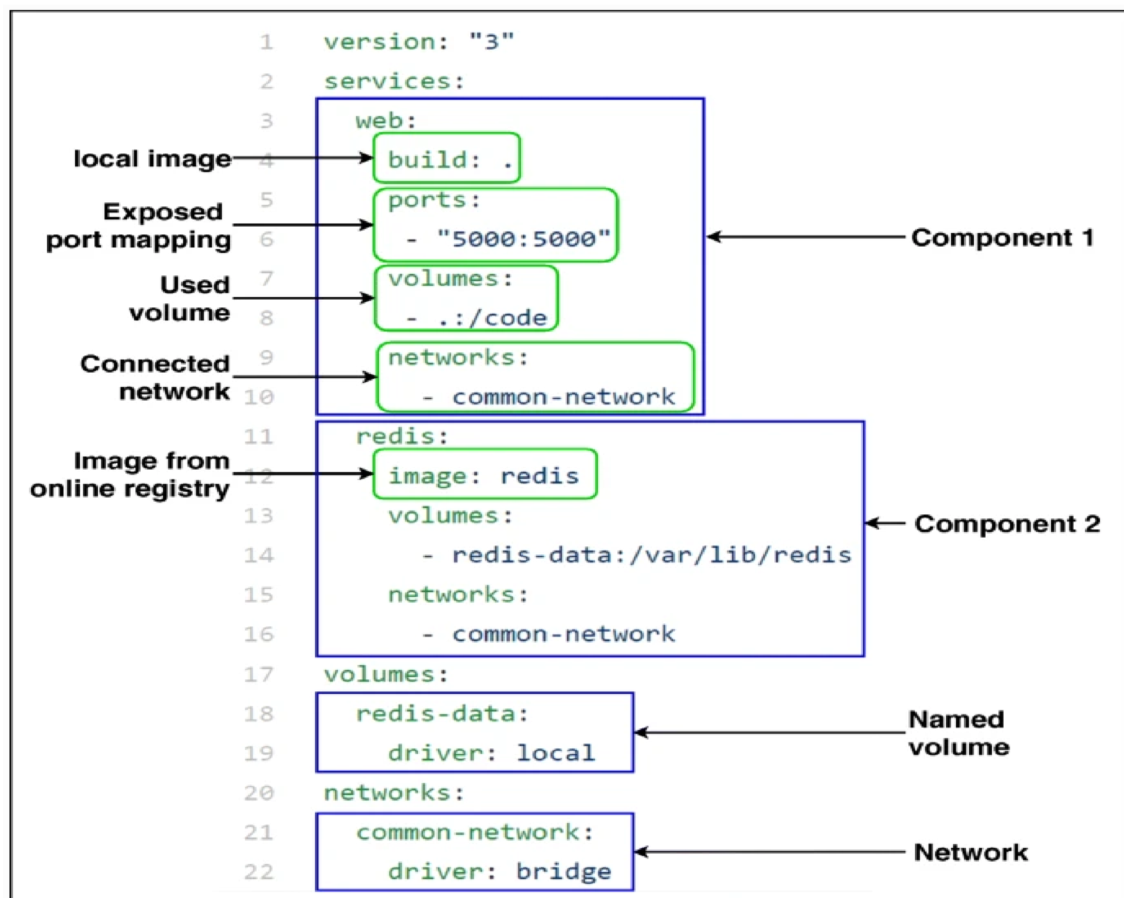

Figure 2.10. *An example of docker compose file [22]*

It enables building orchestration configurations using YAML files, which describe the components that comprise the application, the accompanying images, and how they will be connected to one another, and also volumes for data persistence and networks for service

connections. One can compose a multi-component application using Docker Compose which composes a set of components, each of which is an image and a set of options that specify how the component should behave [22]. A configuration file called docker-compose.yml is used to specify the components structure and properties. Figure 2.10 illustrates an example of docker compose file, where two components are defined as web and redis each having own environment configurations. In docker compose we can define dependencies between services using depends_on, link options to control the order of startup and shutdown of services. Docker Compose always generally begins and ends containers in the order of dependencies.

2.4 Microservices architectural patterns

A software-intensive system's architecture is the foundation upon which the system is constructed to meet the system requirements. In order to influence the requirements, the architecture is built by making a variety of design decisions. To fulfill a number of functional and non-functional criteria the key components and linkages of the system to be developed are represented by architecture patterns. Two types of important design decisions are the application of architecture patterns and tactics [23]. Initial planning considerations, such as how to meet functional needs, non-functional requirements, and physical restrictions, influence the selection of architecture patterns. Two types of important design decisions plays an important role which are the application of architecture patterns and tactics. There are one or more architectural patterns utilize in modern applications. Patterns solves repeating problems in software systems. A software pattern outlines a problem and its background, as well as a generalized solution to the issue. The microservices architectural style is a method of developing single applications as suites of tiny components, each of which runs in its own processes and communicates via lightweight methods. All the architectural patterns comprises of three elements which are [24]:

- Context: Repetitive, common world situations that causes problems.
- Problem: The scenario that develops in the given problem, generalized properly.
- Solution: An effective and efficient architectural solution to the problem, adequately abstracted.

By providing solutions to frequently recurring problems, an architectural style encourages separation of concerns and supports design reuse. Architecture styles are collections of principles and patterns that structure a system. An architectural style is a family of systems in terms of a pattern of structural organization [25]. An architectural style, more particularly, establishes the terminology of modules and connections that can be used in instances of that style, as well as a set of constraints on how they can be combined. One of the advantages of architectural styles is that they provide a standard language for discussing architectural issues in a technology agnostic manner. This allows architects, users, and developers to engage in a discourse focusing on patterns and principles

to analyze and develop architectures without the usage of formal languages. Architecture patterns serve an important in the microservices architectural style. As a result, researchers identify major patterns and their benefits and drawbacks. There are three typically utilized patterns that arise [26]. In the classification, researchers assign different patterns to articles that expressly describe the use of a specific pattern including those where the adopted patterns can be easily determined from the description. In the systematic mapping of Architectural Patterns Taibi et al., 2018 architectural patterns are classified into three categories:

- Orchestration and coordination oriented patterns
- Deployment oriented patterns
- Data storage oriented patterns

In the following sections we will try to describe different architectural patterns.

2.4.1 API-Gateway pattern

An MSA may be required to serve many types of clients and user interfaces, such as the one accessible in web applications and smartphones. Each client's requirements can vary based on its intended use, modular design, and processing capabilities. A client's wants may potentially vary over time. For instance, depending on the strength of its network infrastructure connection, a device may prefer to utilize an API that is more or less network demanding — for example, the description of a product may include more and higher-quality materials, such photographs or integrated instructions. The API Gateway is a service that addresses the problem of having clients of various types. It is a single point of entry that allows access to a variety of APIs. An API Gateway allows users to post numerous APIs, each and every one assigned to a separate set of consumers, and to update the set of published APIs at runtime (since developers may deploy new services during the lifecycle of the MSA). Because an API Gateway serves as an access point for the MSA, it is reasonable for it to have features such as service discovery, load balancing, monitoring, and security.

Its location inside the framework itself is perfect for implementing the proxy circuit breaker pattern, which involves outfitting the API Gateway with circuit breakers for clients and/or services. The mechanism in which these extra functionalities are performed is determined by the underlying technology. With such a single point of entry in and out of the system, it is simple and manageable to enforce runtime governance such as common security requirements, common design decisions , and real-time regulations such as monitoring, reporting, measuring, and throttling. It enables for everything from the flexible addition and removal of microservice components in order to react to the load/demand of microservices. There may have been times when various service users demand certain data and/or have it in a specific format. Figure 2.11 represents a simple example of api gateway pattern where there are two services and two clients from different platforms connected through the api gateway. The API Gateway is indeed the appropriate place to

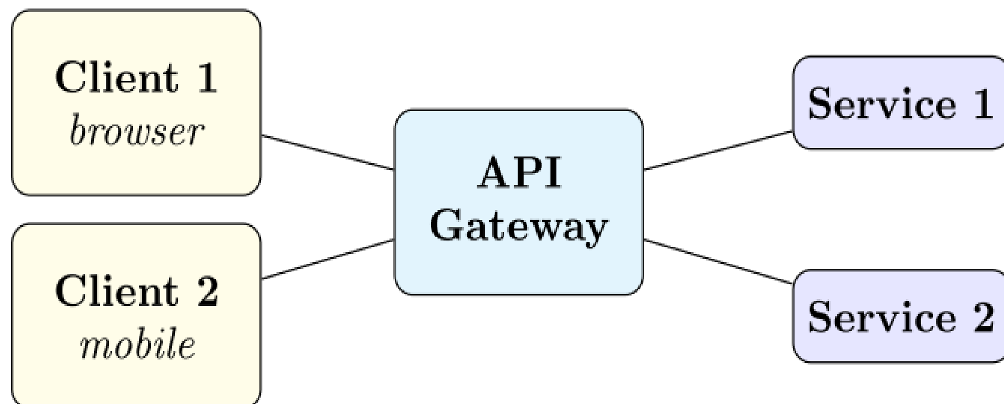

Figure 2.11. *API gateway pattern*

meet these conversion considerations, which may be performed there at gateway level by offering application-specific APIs for the same business functionality. As the degree of diversity tends to increase, a one-size-fits-all strategy would make it more difficult to enhance functionality. The gateway also assists with data acquisition for analytics and monitoring, load balancing, caching, and static response handling.

According to the case study data [27], the API gateway design pattern is the best architecture for an evenly balanced scenario. It also allows us to be more flexible when managing requests from numerous channels. It simplifies the logic for synchronous messaging amongst evenly balanced services and solutions highly scalable artifacts. Segregation of responsibilities, combined with load distribution across numerous service instances, results in the ideal black boxed service experience in a polyglot environment. Polyglot [2] programming benefits from services written in many programming languages and deployed across multiple stacks. Among many other architectures, the API gateway design is commonly utilized as a point-of-contact layer. Clients applications are tied to internal microservices if the API Gateway approach is not used. Client must understand how the system's various parts are divided into microservices. When evolving and reforming internal microservices, those actions have an impact on maintenance since they produce breaking changes to the client apps due to the client apps' explicit connection to the inner microservices. Clients must be updated on a regular basis, making the solution more difficult to evolve. There are too many round trips if a single page/screen in the client app may necessitate several requests to many providers. This method may result in many network round trips between the client and the server, which adds substantial latency. Aggregation handled at an intermediate level may increase performance and user satisfaction.

[2]Polygot https://en.wikipedia.org/wiki/Polyglot$_{(computing)}$

2.4.2 Circuit breaker pattern

Even the most dependable services will eventually exhaust their capacities and fail if there are too many incoming requests. Failure in an MSA is unavoidable and should be approached with concern rather than dismissed. What complicates matters is that in an MSA, a malfunctioning service is likely to be dependent on other services. What happens if our deteriorating service becomes unusable. If we do not properly plan for this event, we risk rendering all other services that rely on it inoperable. This is known as a cascading failure. However, there may be instances where defects occur as a result of unplanned circumstances, which may necessitate a significantly longer repair time. The degree of these failures might range from a partial loss of connectivity to the total breakdown of a service. In some cases, it may be unnecessary for a system to repeatedly attempt a request that is unlikely to achieve success; in the meantime, the program should recognize that even the operation has failed and manage the failure appropriately.

A circuit breaker serves as a proxy for operations that may fail. The proxy must keep track of the amount of consecutive failures and then utilize the information to determine whether it should enable the action to continue or to throw an exception right away.

The circuit breaker pattern is designed to prevent a single component's failure from cascading beyond its boundaries and bringing the entire system down with it. When a service becomes unresponsive, its invokers should cease waiting for it, assume the worst, and begin coping with the possibility that the malfunctioning service will be unavailable. Thus, circuit breakers contribute to the stability and resilience of both clients and services: clients save resources by not attempting to access unresponsive services, while overloaded services can recover by completing part of the tasks they are currently processing. A circuit breaker operates by encapsulating calls to a certain service and monitoring their failure rates. The concept is that if the destination service gets too slow or responds too frequently with errors, the circuit breaker will trip, and subsequent client invocations will immediately return a fault. meeting The pattern can be implemented as a finite-state machine [28], as seen in Figure 2.12. The following is a description of various states.

In **Closed** state requests are routed to the desired service. Faults caused by the requested action, for example exceptions or latencies, raise the relevant failure and timeout counts on the circuit breaker. Whether these indicators surpass a predetermined threshold, or if another predefined criterion is met (e.g., a specific fault was raised), the breaker is tripped and the circuit is opened. On the other hand in **Half open** state only a restricted amount of requests can be processed by the service. If the destination service responds successfully, the circuit breaker is reset towards the closed state, and also the failure and timeout counts are reset. If any of the requests fail while the circuit breaker is in the half-open state, it returns to the open state. **Open** state In requests will not be routed towards the intended service. Rather, a failure message is instantly returned to the client as a response. To resolve the failure, numerous fallback methods can be invoked. The circuit

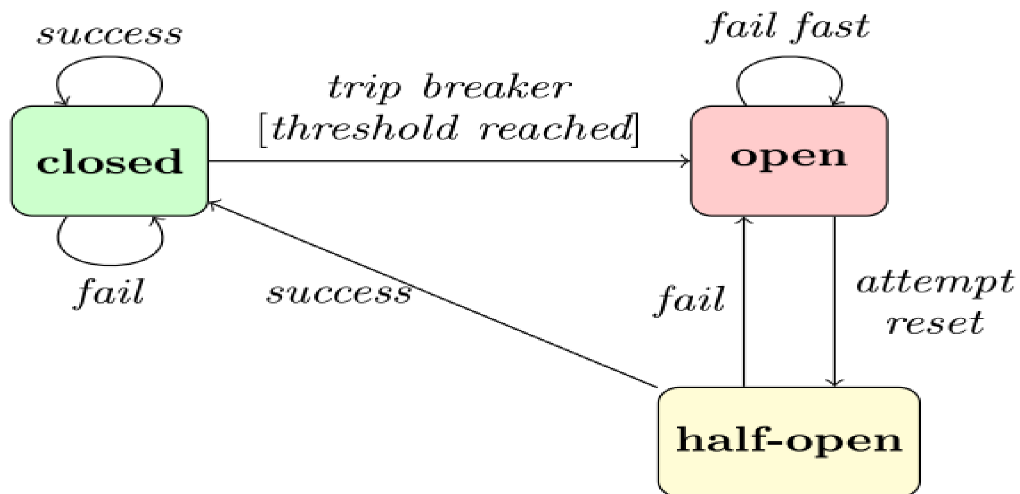

Figure 2.12. *Circuit breaker pattern state machine*
[28]

breaker can switch from open to half-open mode by frequently pinging the service to see whether it becomes responsive afterwards, or after a predetermined duration of time. The Circuit Breaker design ensures system stability when recovering from a breakdown and reduces the impact on performance. It can assist to keep the system's responsiveness up by rapidly denying a request for an action that is likely to fail, rather than waiting for the operation to time out or never respond. If the circuit breaker generates an incident every time it transitions state, this data may be applied to measure the condition of the system segment covered by the circuit breaker or to notify an administrator when a circuit breaker trips to the Open state.

2.4.3 Service discovery pattern

Service Discovery enables how microservices may communicate with one another. In principle, this is a quite straightforward and simple task. For example, a configuration file with the IP addresses and corresponding port number of the microservice can be distributed to all clients. These information can be distributed using standard configuration management systems. Such method, however, is insufficient. Microservices can appear and disappear. This occurs not just as a result of infrastructure outages, but also as a result of new deployments or the growth of the environment through the addition of more servers. Service Discovery must be fluid. A static configuration is insufficient. Because to Service Discovery, the calling microservices are no longer as tightly tied to the called microservice. In theory, configuration techniques may be used to accomplish Service Discovery. Finally, just the data is available that its service is reachable at whatever place is expected to be transmitted. Configuration methods, on the other hand, are the incorrect instruments for the job. Stability and reliability is more necessary for Service Discovery than it is for a configuration server. Under the worst scenario, a breakdown of Service

Discovery might make the connectivity amongst microservices unachievable. As a result, the trade-off between consistency and availability differs from that of configuration systems. As a result, configuration systems should then be utilized for Service Discovery only when they provide enough availability. This may have implications for the required architecture of the Service Discovery system. Service discovery pattern can be achieved in two different ways, one is the client side discovery pattern and another one is the server side discovery pattern.

The client is aware that services do not have fixed placements in the client-side discovery pattern described in Figure 2.13. As a result, it searches the service registry for the location of all the services that it requires. Following that, the client contacts the desired services directly. This architectural design is straightforward, but it necessitates that clients be developed to adhere to this philosophy. Client implementation gets more difficult as a result of having to implement the discovery mechanism. This logic must be repeated for each programming language or framework used in client implementation.

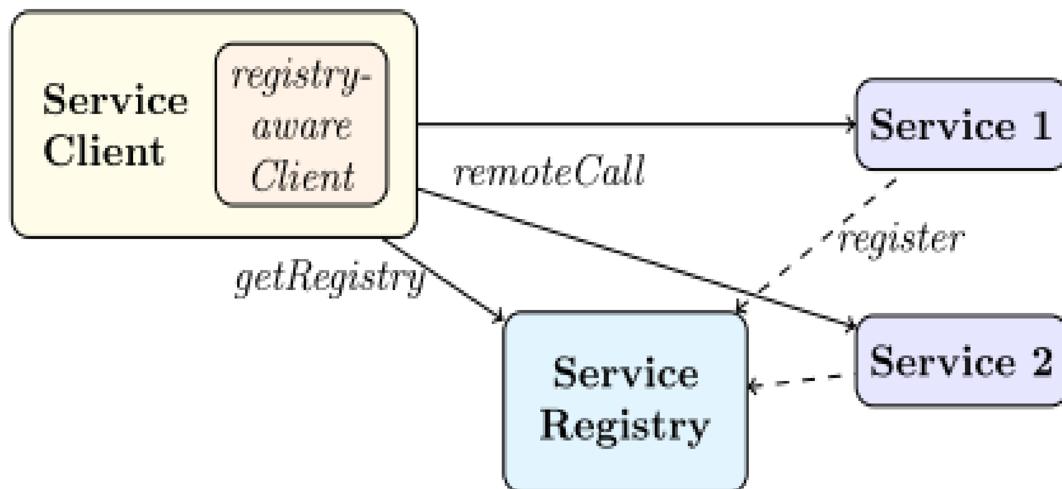

Figure 2.13. *Client side discovery pattern*
[28]

Figure 2.14 depicts an alternate server-side discovery structure in which the discovery functionality is delegated to a specialized router service. The client communicates solely with the router in charge of the services, which is fixed in place. When a request is received, the router contacts the service registry to determine the requested service and then transmits the client request to the latter. This technique, unlike client-side discovery, does not require clients to be aware of the fluid deployment of microservices. The developer must, however, build an extra service that will use resources.

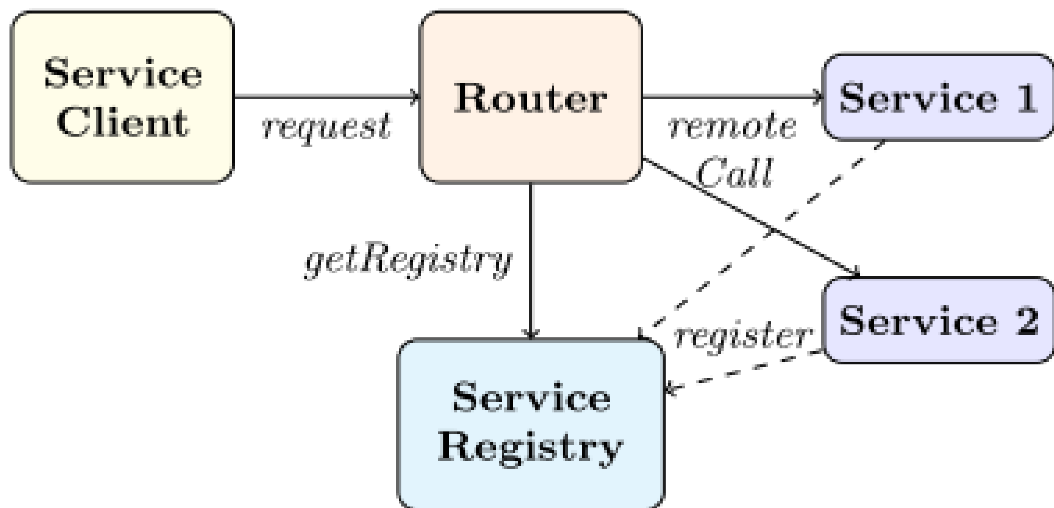

Figure 2.14. *Server side discovery pattern*
[28]

A Service Discovery system should be used in every microservice based design. It serves as the foundation for the management of a significant number of microservices as well as other capabilities such as load balancing. It is possible to get by without service discovery pattern if there are only a few microservices. Service Discovery, on the other hand, is required for huge systems. Because the amount of microservices grows over time, Service Discovery should be built into the design approach from the beginning. Furthermore, almost every system employs at least host lookups, which is already a simple Service Discovery.

2.5 Microservices anti patterns

An "antipattern" is comparable to a pattern, however it is a straightforward but incorrect solution to a problem. Michael Akroyd initially mentioned antipatterns in a talk at the Object World West conference in 1996. An anti pattern is a pattern that describes how to get from a problem to a terrible solution. Identifying harmful habits is just as important as identifying excellent ones. Microservices recently demonstrated to be an effective architectural paradigm for migrating and modernizing monolithic applications. They allow the developers to decompose monolithic applications into small and independent services. Each service is developed by a team; represents a single business capability; and, can be delivered and updated autonomously without impacting other services and their releases.

However, the highly dynamic nature of microservice based systems as well as the continuous integration and continuous delivery of microservices can lead to design and implementation decisions,which might be applied often and introduce poorly designed solutions, called antipatterns. Different researchers have categorized or grouped microservices based on different perspective such as based on the development cycle of a microservice based system, divided the microservices antipattern into four categories [29]. Design: Antipatterns in the specification of a microservice-based system's architectural design. Implementation: Antipatterns in the implementation of microservices. Antipatterns relating to the packaging and deployment of microservice-based systems Monitoring: Antipatterns in monitoring microservice based systems, their behavior, and their modifications. To avoid managing dependencies and the "distributed monolith" issue, microservices should be self-contained functional units that interact via lightweight means. Microservices are cyclically dependent on one another, as seen in Figure 2.15.

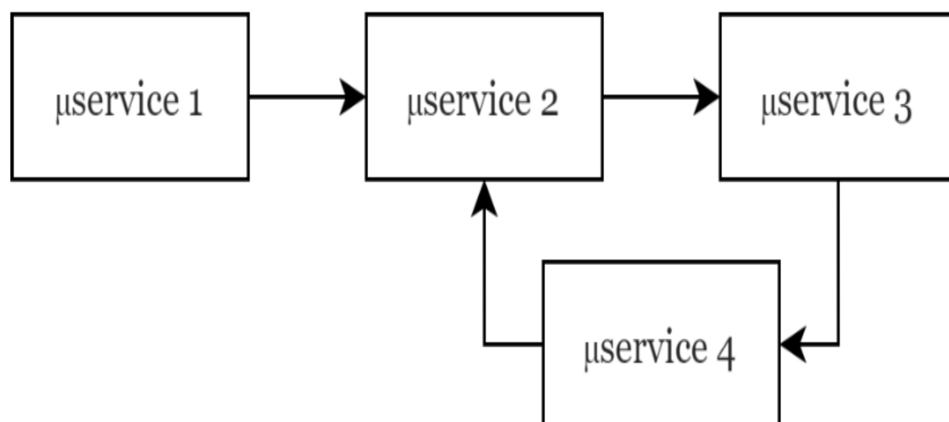

Figure 2.15. *Cyclic dependencies*
[29]

Explicit interactions among microservices, continuous contacts amongst microservices, or the existence of HTTP requests in callbacks are all examples of cyclic dependencies. Microservices are no longer self-contained units. The deployment of a microservice is contingent on the deployment of its dependent microservices. When one of the cyclic-dependent microservices fails, the others fail as well. A microservice that serves multiple purposes known as mega service. A mega microservice is one that has a large number of lines of code, modules, or resources and also a large fan-in. In addition to the complexity of the microservices infrastructure, having a gigantic microservice causes maintenance challenges, poor performance, and difficult testing. We need to break down the mammoth microservice into relatively small, single-purpose microservices. Decomposing a large microservice into smaller pieces makes deployment and testing easier while also isolating business features into well-defined microservices. Developers may focus and organize their efforts by using a single coherent microservice that provides a set of relevant business features. It also prohibits systems from evolving in new directions, microservices from being reused in different situations or systems, and developers from building loosely linked, autonomous teams.

Microservices also shouldn't directly share runtime libraries and code. 2.16 shows how microservices share libraries or files. This antipattern can be detected by the existence of executable files or runtime libraries shared by several microservices and introduced at build or packaging time.

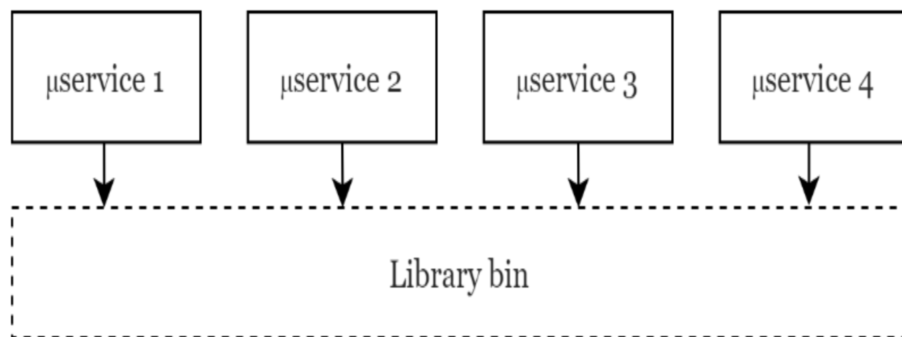

Figure 2.16. *Shared libraries*
[29]

This antipattern connects microservices and dissolves the border between them. Change, evolution, testing, and deployment are also hampered. Even if the DRY principle is violated, runtime assets should not be shared. As indicated in 2.17, shared libraries should have their own microservices. With precise constraints, microservices become self-contained and isolated. Sharing libraries among microservices is beneficial for ensuring consistency and providing a single destination for library updates. When redesigning the Shared Libraries antipattern, furthermore, the libraries must be replicated into the relevant microservices, which makes upgrading and maintaining the libraries and microservices more difficult. One way to refactor is to create a microservice that covers one or more shared libraries.

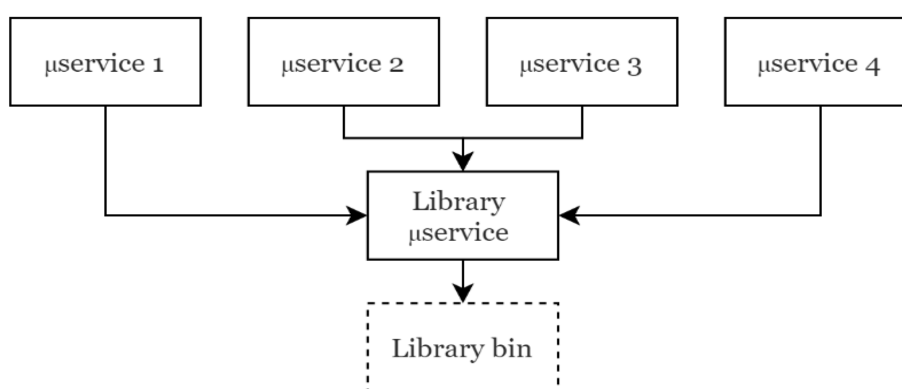

Figure 2.17. *Shared libraries*
[29]

Microservices can communicate with one another using IP addresses and port numbers. IP addresses, ports, and endpoints for microservices are defined explicitly/directly in the source code. The inclusion of IP addresses or fully qualified domain names in source code, configuration files, or environment variables indicates a hardcoded endpoint antipattern. Whenever a system has a lot of microservices, keeping track of all the endpoints and URLs gets increasingly complex. Using a load balancer to run several instances of a microservice becomes unfeasible. Altering a microservice's IP address or port number necessitates modifying and redeploying additional microservices. Hardcoding IP addresses and port numbers is avoided using service discovery. It keeps track of microservice endpoints and makes communication between them easier.

For service discovery, there are two approaches: (1) client-side service discovery and (2) server-side service discovery. Microservice endpoints can change dynamically without affecting other microservices thanks to service discovery. It also makes it easier to deploy microservices on dynamic IP-addressed containers and virtual machines. For all microservice endpoints, it also provides a single, centralized registry. Hardcoding endpoints is a simple way to expedite development, deployment, and testing in a tightly regulated, slow-changing environment. It does, however, make evolution and scalability more challenging. Client-side service discovery, in particular, necessitates building dis-

covery logic within each microservice. Server-side discovery necessitates the addition of a microservice to manage, install, and maintain endpoints, which might slow down the system. A network call is added in both refactoring cases.

Each of these little services in a microservice system may need to persist and access data. Furthermore, in addition to get the most out of the microservices architecture, software architects must manage data storage in a manner that allows each microservice to store and access its own data without interfering with other microservices. In this antipattern multiple microservices can access a same database as illustrated in 2.18. One or more of the following symptoms describe this antipattern: (1) many microservices share configuration files and deployment environments; (2) database tables are prefixed; or (3) databases have a large number of schemas. Microservices grow increasingly intertwined, making maintenance more difficult. Data must also be modified to fit into a single data repository. Microservices are also no longer deployable individually since they share the same database.

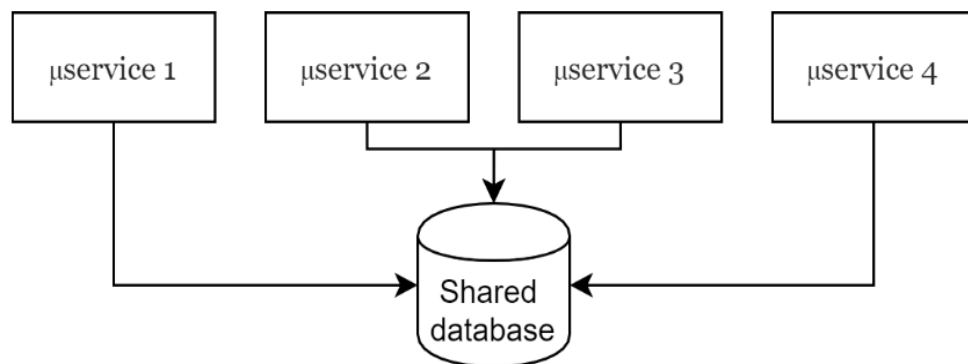

Figure 2.18. *Shared database*
[29]

In terms of how data is accessed utilised, it should always be divided. For each type of data, we must select the right data store and utilize a single database for each microservice. Figure 2.17 illustrates the refactoring solution. The following are some of the benefits of refactoring to the database per service pattern: (1) separation of concerns, as each microservice owns its data; (2) complete management of each microservice by a separate team; and (3) storage technology flexibility.

A persistence service shared by several microservices simplifies deployment and improves speed while assuring data consistency across the microservices. Having one database per microservice necessitates the management of several database systems, the implementation of queries throughout many microservices, and the maintenance of database integrity.

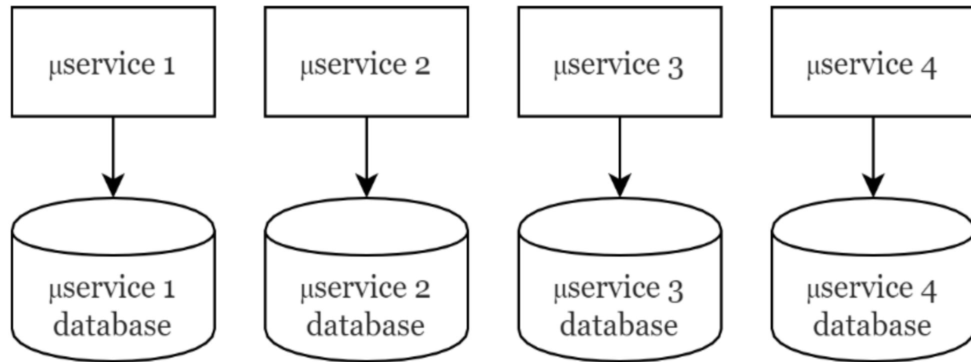

Figure 2.19. *Shared database refactored*
[29]

3 CONTEXT

We organized our study based on the parameters established by [30]. Diverse research procedures are used for different goals, and one approach does not fit everyone. Out of different methodologies our work is Exploratory based research methodology where figuring out what's aiming on, looking for fresh insights, and coming up with new research concepts and theories. An empirical research either collect quantitative or qualitative data. Quantitative data consists of figures and classes, whereas qualitative data consists of sentences, descriptions, drawings, and diagrams, among other things. Statistics are used to examine quantitative data, whereas classification and sorting are used to study qualitative data. Case studies are frequently based on qualitative data since it allows for a fuller and more detailed explanation. However, combining qualitative and quantitative data typically leads to a deeper understanding of the phenomena being examined [31].

3.1 Goal and Research Questions

The software engineering community has recently published research on the prospective, recurring, and successful architectural patterns [26] [5] and anti-patterns [13,17,18] in microservices-based architectures.

Indeed, such systems' organizational structure should be expressed in so-called microservice architectural patterns that best suit the demands of projects and development teams. However, there are few public repositories that share open source project microservices patterns and practices, which might be useful for educational reasons and future study.

According to the goal question metric approach measurement must be defined in a top-down fashion and it must be focused, based on goals and models [32]. The main goal of this work is to determine architectural patterns in microservices and looking into approaches to assess the evolving coupling between services to assist practitioners in understanding how decoupled their services are in the microservices project. The following goals can be developed to help answer the primary research topic.

- Formulate a curated dataset of microservices based systems
- Analyze the projects to determine architectural patterns
- Measure coupling matrices of microservices

3.2 Project selection

We selected projects from GitHub, searching projects implemented with a microservice-based architecture, developed in Java and using docker.

The search process was performed applying the following search string:

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

Results of this query reported 18,639 repository results mentioning these keywords.

We manually analyzed the first 1000 repositories, selecting projects implemented with a microservice-architectural style and excluding libraries, tools to support the development including frameworks, databases, and others.

The microservices projects comprising this study were chosen by a process known as "criterion sampling" [33]. The following were the criteria used to choose projects:

- Having microservices architecture
- Developed in Java
- Docker is used for containerization
- Code is available in Github
- Open source

In this work, we selected the top 20 repositories that fulfill our requirements listed in table 3.1.

In the table 3.1 we listed the name of project used in github repository, the github url to the repository, and lastly the type of the project.

Then, to report the project list, we made a github page [1] and posted multiple inquiries on several forums [2]. Moreover, we monitored replies to similar questions on other practitioners forums[3] [4] [5] [6]. We also looked at responses to similar inquiries on other practitioners' forums to see if they knew of any other relevant Open Source projects that used a microservice architectural approach. The practitioners' forums responded with 19 recommendations to add 6 projects to the list. In addition, four people submitted a pull request to the repository, requesting that new projects be included.

[1] https://github.com/clowee/MicroserviceDataset

[2] Stack Overflow -1 https://stackoverflow.com/questions/48802787/open-source-projects-that-migrated-to-microservices

[3] Stack Overflow -2 https://stackoverflow.com/questions/37711051/example-open-source-microservices-applications

[4] Stack Overflow -3 https://www.quora.com/Are-there-any-examples-of-open-source-projects-which-follow-a-microservice-architecture-DevOps-model

[5] Quora -1 https://www.quora.com/Are-there-any-open-source-projects-on-GitHub-for-me-to-learn-building-large-scale-microservices-architecture-and-production-deployment

[6] Quora -2 https://www.quora.com/Can-you-provide-an-example-of-a-system-designed-with-a-microservice-architecture-Preferably-open-source-so-that-I-can-see-the-details

Table 3.1. *Selected microservices projects*

Project Name	**Project Repository**	**Project Type**
Consul demo	http://bit.ly/2KsGzx6	Demo
CQRS microservice application	http://bit.ly/2YtbtiF	Demo
E-Commerce App	http://bit.ly/2yLqTPW	Demo
EnterprisePlanner	http://bit.ly/2ZPK7je	Demo
eShopOnContainers	http://bit.ly/2YGSkJB	Demo
FTGO - Restaurant Management	http://bit.ly/2M7f8fm	Demo
Lakeside Mutual Insurance Company	http://bit.ly/33iJSiU	Demo
Lelylan - Open Source Internet of Things	http://bit.ly/2TdDfd3	Industrial
Microservice Architecture for blog post	http://bit.ly/2OKY29v	Demo
Microservices book	http://bit.ly/2TeSbI2	Demo
Open-loyalty	http://bit.ly/2ZApXtA	Industrial
Pitstop - Garage Management System	http://bit.ly/2Td7NLY	Demo
Robot Shop	http://bit.ly/2ZFbHQm	Demo
Share bike (Chinese)	http://bit.ly/2YMJgmb	Demo
Spinnaker	http://bit.ly/2YQA2S7	Industrial
Spring Cloud Microservice Example	http://bit.ly/2GS2ywt	Demo
Spring PetClinic	http://bit.ly/2YMVbAC	Demo
Spring-cloud-netflix-example	http://bit.ly/2YOUJxJ	Demo
Tap-And-Eat (Spring Cloud)	http://bit.ly/2yIjXmC	Demo
Vehicle tracking	http://bit.ly/31i5aLM	Demo

4 IMPLEMENTATION AND DATASET

In order to analyze and visualize the selected projects we have use 4 tools. Of these tools 3 are open source projects and 1 have been developed in house solely to analyze the dependencies between services in microservices projects.

4.1 SLOCCount

The term "microservices" implies that the size of the service is important; microservices are, after all, meant to be tiny. Counting the lines of code is one approach to determine the size of a microservice (LOC). However, there are some drawbacks to this approach: it is dependent on the programming language employed.

Microservices are clearly not designed to predetermine the technological stack, and some languages require more code to describe the same functionality as others. As a result, using this statistic to define microservices isn't particularly useful. Finally, microservices are a type of architecture. However, rather than sticking to technical measures like LOC, architectures should match the circumstances in the domain. Attempts to estimate size based on code lines should also be looked up carefully.

SLOCCount (pronounced "sloc-count") is a collection of tools for calculating the number of physical source lines of code (SLOC) in big software systems.As a result, SLOCCount is a "software measurement tool" or "software metrics tool." SLOCCount was created by David A. Wheeler with the intention of counting SLOC in a GNU/Linux distribution, but it may now be used to count SLOC in any software system [34].

Physical SLOC, often known as "non-blank, non-comment lines," are counted by SLOC-Count. A physical source line of code (SLOC) is a line that ends in a newline or end-of-file marker and contains at least one non-whitespace non-comment character, according to the definition. Comment separators are treated as comment characters. Only data lines with whitespace are not included. SLOCCount can count the number of lines of code in a variety of programming languages and categorize them by type. The basic operation of SLOCCount is pretty straightforward. Simply type "sloccount" in a terminal window, followed by a list of source code folders to count. If you simply give it a single directory, SLOCCount will try to be clever and split the source code into subdirectories for reporting reasons.

Running SLOCCount is quite straightforward from commandline/terminal, for instance if we want to measure SLOCCount of Apache 1.3.12 here is the command:

```
sloccount /usr/src/redhat/BUILD/apache_1.3.12
```

The output we'll see shows status reports while it analyzes things, and then it prints out:

```
SLOC Directory SLOC-by-Language (Sorted)
24728   src_modules     ansic=24728
19067   src_main        ansic=19067
8011    src_lib         ansic=8011
5501    src_os          ansic=5340,sh=106,cpp=55
3886    src_support     ansic=2046,perl=1712,sh=128
3823    src_top_dir     sh=3812,ansic=11
3788    src_include     ansic=3788
3469    src_regex       ansic=3407,sh=62
2783    src_ap          ansic=2783
1378    src_helpers     sh=1345,perl=23,ansic=10
1304    top_dir         sh=1304
104     htdocs          perl=104
31      cgi-bin         sh=24,perl=7
0       icons           (none)
0       conf            (none)
0       logs            (none)


ansic:       69191 (88.85%)
sh:           6781 (8.71%)
perl:         1846 (2.37%)
cpp:            55 (0.07%)


Total Physical Source Lines of Code (SLOC)                    = 77873
Estimated Development Effort in Person-Years (Person-Months) = 19.36 (232.36)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Estimated Schedule in Years (Months)                          = 1.65 (19.82)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers  (Effort/Schedule)     = 11.72
Total Estimated Cost to Develop                               = $ 2615760
 (average salary = $56286/year, overhead = 2.4).
```

4.2 GraphML

GraphML is a file format for graphs that is both comprehensive and user-friendly. It comprises of a core language for describing a graph's structural features and a flexible extension mechanism for adding application-specific data. Some of its key feature includes support of different types of graphs:

- directed, undirected, and mixed graphs,
- hypergraphs,
- hierarchical graphs,
- graphical representations

GraphML can be utilized with other XML-based formats because of its XML syntax. On the one hand, it has its own extension method for attaching GraphML elements to <data> labels with sophisticated information. A graphml element with a number of subelements, such as graph, node, and edge, make up a GraphML document. In this study we generated graphml file for each of the projects we analyzed. The goal of this step is to further analyze the graph data as well as visualizing the microservice dependencies in graph.

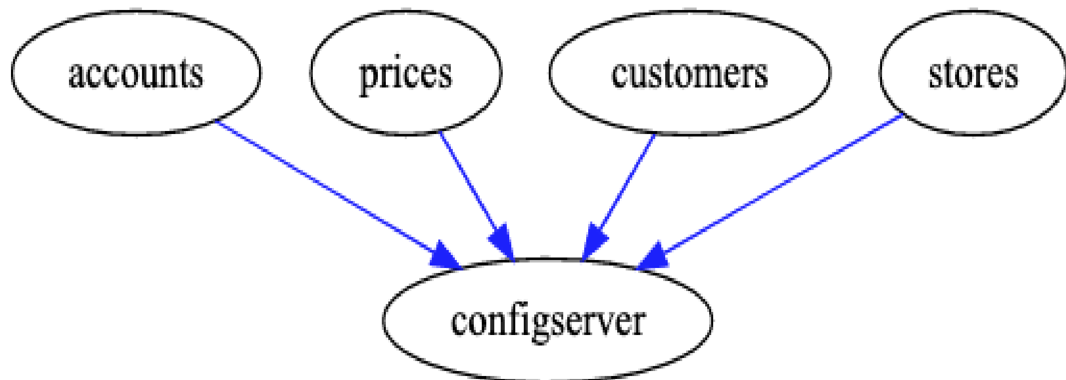

Figure 4.1. *Dependency graph*

Figure 4.1 shows the visualization of dependency graph for the microservice project Tap-And-Eat from collected projects table 3.1. In this project we have 5 services where 4.2 shows contents of graphml file for dependencies of Tap-And-Eat project.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns http://graphml.graphdrawing.org/xmlns/1.1/graphml.xsd">
  <key id="edgelabel" for="edge" attr.name="edgelabel" attr.type="string"/>
  <graph id="G" edgedefault="directed">
    <node id="stores" />
    <node id="configserver" />
    <node id="accounts" />
    <node id="customers" />
    <node id="prices" />
    <edge id="stores-&gt;configserver" source="stores" target="configserver" label="depends">
      <data key="edgelabel">depends</data>
    </edge>
    <edge id="accounts-&gt;configserver" source="accounts" target="configserver" label="depends">
      <data key="edgelabel">depends</data>
    </edge>
    <edge id="customers-&gt;configserver" source="customers" target="configserver" label="depends">
      <data key="edgelabel">depends</data>
    </edge>
    <edge id="prices-&gt;configserver" source="prices" target="configserver" label="depends">
      <data key="edgelabel">depends</data>
    </edge>
  </graph>
</graphml>
```

Listing 4.1. *GraphML file*

4.3 MicroDepGraph

MicroDepGraph is a tool we created in-house for detecting dependencies and plotting microservice dependency graphs.

It analyzes the docker files for service dependencies defined in docker-compose and java source code for internal API calls, starting with the source code of the various microservices. The program is entirely written in Java.

We opted to look at docker-compose files because in microservices projects, the services' dependencies are defined as configuration in the docker-compose file. The tool parses the files from the projects because docker-compose is a YML or YAML file.The services of the microservices project defined in the docker-compose file are first determined by MicroDepGraph. Then it examines dependencies for each service and maps the dependencies for the respective services.

Only looking at the docker-compose file reveals all dependencies' relationships, as there could be internal API calls, for example, via a REST client. As a result, we had to look through the java source code for any API calls to other services. As we look at Java microservices projects, Spring Boot is the most widely used and popular framework for creating microservices in Java. The API endpoints for services in spring boot are setup and defined using various annotations in java source code. As a result, when parsing java source code, we focused on these annotations. The endpoints for each service were first found by processing the java source code and looking for annotations that defined the endpoints. We used an open source library called JavaParser [1] to parse Java source code. After obtaining endpoints for each service, we looked to see if any API calls were made to these endpoints by other services. Then, if one service calls another's API, we map it as a dependency and add it to our final graph. The tool then creates linkages (dependencies) between the services and draws a directed graph after discovering all of the mapping.

MicrodepGraph can be executed in any computer having jre8 installed. It takes two parameters as input: (1) the path of the project in the local disk and (2) the name of the project. Here is the command to execute MicroDepGraph as:

```
1 java -jar microservices-dependency-check.jar <path_of_the_project> <
    project_name>
```

Finally, it generates a graph representation formatted as GraphML file, a neo4j database containing all the relationships and an svg file containing the graph.

Figure 4.1 shows an example of the output provided by MicroDepGraph on the project ”Tap And Eat”.

[1] JavaParser. https://javaparser.org/

4.4 Dataset generation using MicroDepGraph

We started by cloning the repository for each project. Then we ran SLOCcount on each project separately to get the number of lines of code.

Then, to get the dependencies between the microservices, we used MicroDepGraph. For each project, MicroDepGraph provided GraphML and an svg file. We utilized the Apache TinkerPop[2] graph computing framework to create the GraphML file.

In the table 4.4 we listed the dataset collected through SLOCCount and MicrodepGraph, **#Ms** the number of services in the given microservice system, **KLOC** [3] thousands lines of source code, **#Commits** number of commits in the project, **#Dep.** number of dependencies.

Table 4.1. *Generated dataset of the projects*

Project Name	#Ms.	KLOC	#Commits	#Dep.
Consul demo	5	2.343	78	4
CQRS microservice application	7	1.632	86	3
E-Commerce App	7	0.967	20	4
EnterprisePlanner	5	4.264	49	2
eShopOnContainers	25	69.874	3246	18
FTGO - Restaurant Management	13	9.366	172	9
Lakeside Mutual Insurance	8	19.363	12	7
Lelylan - Open Source Internet of Things	14	77.63	2059	11
Microservice blog post	9	1.536	90	7
Microservices book	6	2.417	127	5
Open-loyalty	5	16.641	71	2
Pitstop - Garage Management System	13	34.625	198	9
Robot Shop	12	2.523	208	8
Share bike (Chinese)	9	3.02	62	6
Spinnaker	10	33.822	1669	6
Spring Cloud Microservice	10	2.333	35	9
Spring PetClinic	8	2.475	658	7
Spring-cloud-netflix	9	0.419	61	6
Tap-And-Eat (Spring Cloud)	5	1.418	35	4
Vehicle tracking	8	5.462	116	5

[2]Apache TinkerPop http://tinkerpop.apache.org/
[3]https://en.wikipedia.org/wiki/Source_lines _of _code

The generated graphml file can also be used in alternative graph visualization platforms, such as Gephi[4], to import the GraphML file. We can then use different graph algorithms in these types of graph visualization tools to further analyze the graph. We additionally generated an SVG graphic file as an output, which can be used for additional processing. Afterwards, we saved the findings as graphml files in a Github repository, along with a list of examined microservice applications. The GraphML output from one of the projects examined by MicroDepGraph is shown in 4.1 as visualization of 4.2 graphml generated by MicroDepGraph.

[4]Gephi https://gephi.org/

5 MICROSERVICES METRICS

Microservices should be as decoupled and cohesive as feasible, which is a preferred characteristic [2]. While minimal coupling is crucial in monolithic systems [35], it is much more significant in microservices, because loosely connected services (both static and dynamically) allow developers to make changes to their service without affecting other services [2]. One of the main design ideas in software engineering is "low coupling, high cohesion" [35]. This principle asserts that coupling between modules of a software system should be kept to a minimum while maintaining strong relationships between both the different components of each subsystem.

The strong coupling between the core components of the software is a major impediment to efficient maintenance. Most changes in monolithic systems necessitate changes to numerous sections of the system, and the scale and complexity of the alteration is typically difficult to predict in advance. The goal of such architectural features is to keep maintenance as less time consuming and less complex as possible. To put it another way, updates to the source code should be limited to a single microservice. Microservices can be developed and deployed independently thanks to the decoupled architecture. A loosely coupled service only knows as much as it needs to know about the services it collaborates with [36].

Cohesion is related to decoupling and examines how closely the elements of a given class are related. The degree to which the functionality of separate modules are coupled to one another is measured by cohesion [37]. Low coupling is frequently associated with high cohesiveness [38] [39]. The software components have a high level of cohesiveness, making system understanding easier [39]. As a result, high cohesiveness aids in the system's creation and maintenance. By organizing functionality and components according to business processes, developers aim for strong cohesion and low coupling whenever designing microservice-based systems. Changes to a features and functions should then only affect one microservice [36].

One approach to describe the issue of modularity in traditional and object-oriented (O-O) software engineering is to use software metrics to find a way of making it measurable and quantifiable. For this objective, a number of measures have been created, and a generic framework for desirable qualities of specific types of metrics has been offered [40].

Low-level metrics like as control-flow, functions, size, complexity, or semantic program understandability were employed to quantify software quality in the beginning. Absolute measurements that can be acquired automatically (or at least semi-automatically) are still useful for evaluating a software system's base degree of maintainability. They can be used as a starting point for judging more general qualities of a system with short feedback cycles. The most common traits linked to maintainability in the literature are scale, complexity, coupling, and cohesiveness [41].

In another work Bogner et al. has proposed a maintainability model for service oriented and microservices architecture where they chose a straightforward hierarchical framework. The top-level quality attribute in the first layer is maintainability. The second layer is made up of Service Properties, which reflect some maintainability-related feature of a service or the entire system [42]. In another work Taibi and Systä[43] developed a decomposition framework based on process mining, as well as a set of metrics to assess the decomposition's quality, identifying two size-related measures and a coupling measure [43].

Bogner et al.,[41] in their work for automatically measuring maintainability of services pointed out that the major portion of metrics specifically created for monolithic systems and Service Oriented Architecture (SOA) may be applied to microservices as well . They do, however, point out that different characteristics of microservices might have a big influence on the complexity of autonomous metric gathering, implying the necessity for specific tool support. To propose matrices for microservices based systems in this work, there are four groups of metrics that can utilized to measure microservices

- **Service Size** The size of a microservice system is equal to the sum of its underlying services. A large microservice system is more difficult to maintain than one that is tiny. Although size as a system feature – and notably its most commonly used metric Lines of Code (LOC) – is sometimes viewed as questionable, a basic estimate of it is sometimes important [41]. Nonetheless, size metrics are seldom accurate enough on their own. Furthermore, defining "acceptable" value ranges for these indicators is frequently difficult. Size measurements, on the other hand, can be very valuable in a relative sense, i.e. when comparing the various components of a system.
- **Service Complexity** In microservice-based systems, the amount and variety of internal work carried out by a service, as well as the degree of interaction between its services required to do this. Maintainability suffers as a result of high complexity. Bogner's [41] study proposes three metrics that can be applied to microservices and were initially developed for SOA.
- **Service Cohesion** The amount to which a service's operations contributes to only one task or capability. Maintenance tasks are aided by a high level of cohesion. It's a metric for how closely each element of a software module's functionality is related [37]. High cohesiveness simplifies thinking and reduces dependencies [39]. There are no explicit measurements for cloud-native systems or SOA.

- **Service Coupling** The degree to which a service's interdependencies and interconnections with other services are strong. It is easier to maintain a microservice system made up of loosely connected services with a minimal number of couplings. According to Bogner et al., [41] Coupling is the most straightforward of the four design properties discussed. It lacks a semantic component and may be easily examined using graph theory.

  Together with Panichella and Taibi [44] we proposed a list of metrices in 5.1 for each of the group discussed above. This work expands and complements the proposed coupling metrics by giving straightforward measurement processes, a tool to help identify the matrices automatically in microservices, and a method to visualize them.

Table 5.1. *The Metrics Proposed in the Literature*

Group	Metric
Service Size	- *Number of synchronous cycles* [45]
	- *Distribution of synchronous call per microservice* [45]
	- *Number of synchronous dependencies* of each microservice [45]
	- *Average size of asynchronous messages* [45]
	- *Longest synchronous call trace* [45]
	- *Number of classes per microservice* [43]
	- *Number of classes that need to be duplicated* [43][46]
	- *Weighted Service Interface Count (WSIC* [47])*: number of exposed interface of a service be weighted on the number of parameters.
	- *Component Balance* [48][42]*: number and size uniformity of components (or services). Very big or very small components could be candidates for refactoring.
	- *Number of Operations* [49]*: number of exposed interface of a service.
Service Complexity	- *Total Response for Service* [50]*: adaptation of Response for Class (RFC) [51] to the service level
	- *Number of Versions concurrently used in a Service**
	- *Service Support for Transactions**
Service Cohesion	- *Service Interface Data Cohesion (SIDC)* [50]*, the similarity of the parameters data-types between two services
	- *Service Interface Usage Cohesion (SIUC)* [50]*: $(used\ operations\ per\ client/(clients \cdot operations\ in\ a\ service))$
	- *Total Service Interface Cohesion* [50]: average between SIDC and SIUC
Service Coupling	- *Coupling Between Microservices (CBM)* [43]. Extension of the CBO, ratio between the number of calls to other services and the number of classes of the microservice
	- *Absolute Importance of the Service (AIS)* [52][42]* number of clients that invoke at least one operation to the service.
	- *Absolute Dependence of the Service (ADS)* [52]* number of other services that a service depends on
	- *Absolute Criticality of the Service* [52]* defined as: ACS(S) = AIS(S) ADS(S)
	- *Services Interdependence in the System (SIY)* [52][42]*: Number of service pairs bidirectionally dependent on each other. If such dependencies between microservices exist, services could be merged.

*Metrics Adopted in SOA, that could be suitable for microservices [41]

5.1 Proposed matrices

After analyzing the dataset listed in table 4.4 we found the the answer of first two of our research questions which are to formulate dataset of microservice based systems and analyze the architectural pattern which we got from the visualization of graphml file. To answer the third research question we proposed a structural coupling metric [44]. Our metrics are meant to improve modularity, with the added benefit of assisting practitioners in understanding how decoupled their services are and, eventually, reasoning on decoupling solutions. More precisely, such measures can assist in swiftly identifying potential issues with constructions that are likely difficult to comprehend by humans and thus prone to creating errors. Metrics are important in both circumstances as a way to make quality related features measurable and quantifiable.

5.1.1 Structural coupling

Coupling is among the qualities that has the largest impact on maintenance since it has a direct impact on maintainability. In the field of software engineering research, one of the main design ideas is high cohesion and low coupling [53]. According to these principles, the coupling between modules in a software system should be as loose as feasible while maintaining strong relations between the software artifacts that make up the particular modules. One of the objectives of software designers in particular is to maintain the coupling in an OO system as low as feasible. Strongly connected microservices are more prone to be impacted by changes and faults in other services; as a result, these services have a higher architectural importance and must be detected. Coupling measures assist in such initiatives, and the majority of them are focused on some form of dependency analysis, which is based on available source code or design information.

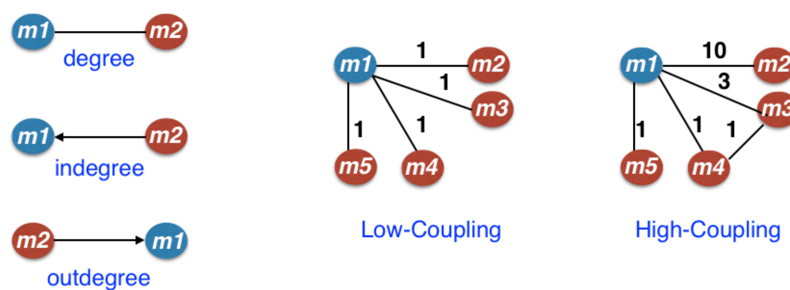

Figure 5.1. *Structural Coupling: graph representation*

If code/structural dependencies exist between two software modules or artifacts, they are structurally connected [54]. The bigger the amount of dependencies between these modules, the tighter the coupling. Any depiction of a software system must provide for the multiple and multifarious interactions among software components. Myers[55] in his work has shown how the interactions between software services can be depicted as collaboration graph where the nodes represents system components and the weighted edges indicate internal dependencies between the services.

In figure 5.1 different properties of graph for structural coupling is depicted. Each node in a directed network has an in-degree (the number of links pointing to it) and an out-degree (the number of links pointing away from it) (the number of links emanating from a node). The total-degree of a node is the sum of its in-degree and out-degree, and it is often referred to as total-degree to emphasize the network's directed character.

5.1.2 Microservice coupling measures

A service **A** is structurally connected with another service **B** if code/structural dependencies present between artifacts forming them, according to the conceptual framework of structural coupling (Savic et al., 2017). As a result, the bigger the number of dependencies between these services, the higher the degree of coupling. Low structural coupling is necessary to allow changes in a single service to be made without affecting other modules or artifacts in other services.

Because of the significant structural coupling, problems and modifications propagated throughout modules of various services, resulting in low maintainability and developer productivity e.g., developers have to coordinate their development work involving different services. Savic et al., in his work[54] has proposed the Afferent-Efficient Coupling Balance (C_k),is the average in-degree to total degree (CBO) ratio for modules with total degrees greater than or equal to k. Being motivated by the work of Savic et al., we calculated the structural coupling between a service **s1** and a service **s2**.

$$StructuralCoupling(s1, s2) = 1 - \frac{1}{(degree(s1, s2))} * LWF * GWF \quad (5.1)$$

Dependencies between two services **s1** and **s2** are weighted using both the Local Weighting Factor (LWF), which considers the degree and in-degree of s1 with s2, and the Global Weighting Factor (GWF), which considers the maximum degree across all services in the system, elements to consider while weighing:

$$LocalWeightFactor(s1, s2) = \frac{1 + outdegree(s1, s2)}{1 + degree(s1, s2)}$$

$$GlobalWeightFactor(s1, s2) = \frac{degree(s1, s2)}{max(degree(all_services))} \quad (5.2)$$

This re-weighted structural coupling assessment assures that the real coupling value between s1 and s2 is in the [0-1] range, and therefore these values are also influenced by the generic dependencies distributed to other services.

- **degree(s1,s2)** is the total number of structural relationships between s1 and s2.
- **outdegree(s1,s2)** is the real number of static dependencies directed from s1 to s2 among the total one.
- **in-degree(s1,s2)** is the actual number of static dependencies directed from s2 to s1 among the total one.
- **max(degree(all_services))** relates to the maximum number of dependencies (or degree) among all (possible pairs of) system services.

In order to compare the result of proposed structural coupling we also considered *Coupling Between Microservices*, CBM metric from the work of Taibi et al., [46]. The coupling measure CBM was inspired by Chidamber and Kemerer's well-known Coupling Between Objects (CBO) metric [39]. CBO counts the number of classes that are associated with a given class. Classes can be linked through a variety of mechanisms, including as method calls, field accesses, inheritance, parameters, return types, and exceptions.

$$CBM_{MSj} = \frac{Number of external links}{Number of classes in the microservices}$$

The number of call pathways to external services is represented by the formula "Number Of External Links." External services that are invoked several times, even by distinct classes of the microservice, are therefore counted just once. Other microservices or services outside of the system might be considered external services. This metric is an abstraction of the microservice's size, and it enables the developer to identify services that are either too large or too tiny in comparison to other microservices. Larger microservices should be avoided since they are more difficult to manage in general.

6 RESULTS

We selected the 17 projects developed in Java and using Docker from the dataset of 4.4. Before starting calculation for these 17 projects we will first calculate the matrices for an example system so that we can present all the calculations step by step.

We will now show how to determine the structural coupling using the system represented in Figure 6.1. The system used as an example is made up of five microservices that are linked together directly. Table 6.1 shows an example of metrics provided in the literature and derived for the example system represented in Figure 6.1.

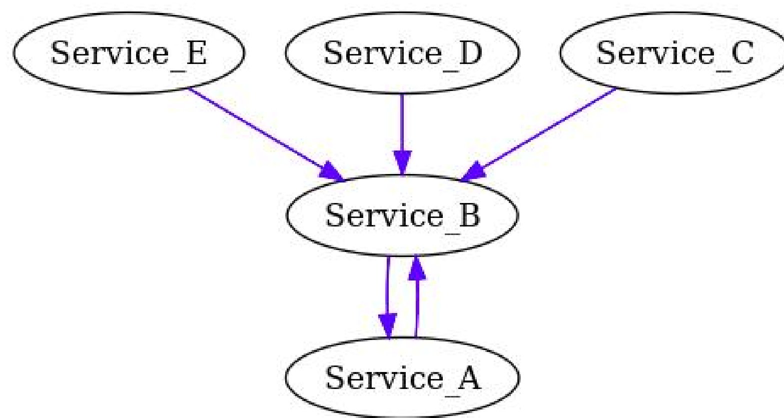

Figure 6.1. *An example of Microservices-based System*

We specifically describe the size of each microservice (number of classes), the in-degree of each microservice (number of incoming service calls), the out-degree (number of outgoing calls), and the degree of each microservice (sum of in-degree and out-degree). Table 6.2 illustrates how to calculate the Local Weight Factor (LWF), Global Weight Factor (GWF), and Structural Coupling (SC) for the same system.

Table 6.1. *Example of metrics for the system in Figure 6.1*

	in-degree	out-degree	degree	#classes
A	4	1	5	50
B	0	1	1	10
C	0	1	1	11
D	0	1	1	17
E	0	1	2	30

Table 6.2. *SIY, LWF, GWF and SC for the system in Figure 6.1*

	LWF					GWF					SC				
	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
A	0	0.66	1	1	1	0	2	0	0	0					
B	0	0	0.5	0.5	0.5	0	0	1	1	1	0.33			0.5	0.5
C	1	0	0	1	1	0	0	0	0	0		0.5			
D		1	0	1	0	1	0	0	0	0	0				

As we mentioned earlier we selected 17 projects from 4.4 where we already have information on the number of microservices in each system (MS), the size of each system in lines of code(KLOC), the number of commits and the number of dependencies (Dep). Table 6 shows the list of projects selected for matrices calculation.

Table 6.3. *Generated dataset of the projects*

Project Name	#Ms.	KLOC	#Commits	#Dep.
CQRS microservice application	7	1.632	86	3
E-Commerce App	7	0.967	20	4
EnterprisePlanner	5	4.264	49	2
eShopOnContainers	25	69.874	3246	18
FTGO - Restaurant Management	13	9.366	172	9
Lakeside Mutual Insurance	8	19.363	12	7
Microservice blog post	9	1.536	90	7
Microservices book	6	2.417	127	5
Open-loyalty	5	16.641	71	2
Pitstop - Garage Management System	13	34.625	198	9
Robot Shop	12	2.523	208	8
Share bike (Chinese)	9	3.02	62	6
Spinnaker	10	33.822	1669	6
Spring Cloud Microservice Example	10	2.333	35	9
Spring PetClinic	8	2.475	658	7
Spring-cloud-netflix-example	9	0.419	61	6
Vehicle tracking	8	5.462	116	5

To compare our metric to other measures presented in the literature, we measured the number of classes per microservice for each project. Then, using the Structural Coupling presented in Section 5.1 and the coupling metrics proposed in the literature, we estimated the coupling metrics proposed in the literature. We created a csv file with metrics such as in-degree, out-degree, degree, classes (number of classes), and LOC (Lines Of Code). We created a matrix for each metric LWF, GWF, SC and CBM to determine the dependence between all pairs of microservices.

We create a chart for each analyzed project that shows the SC for each pair of MS. This graphical depiction might help you figure out which MS in your system have the maximum coupling. We generate descriptive statistics of SC for each microservice in order to compare the different measurements. The whole process of computing the matrices,

Table 6.4. *Results of the Coupling Metrics Applied to the 17 projects.*

Project Name	Degree				SC					CBM				
	Max	Avg	Med.	Stdev	Tot	Max	Avg	Med.	Stdev	Tot	Max	Avg	Med.	Stdev
CQRS microservice appl.	1	1	1	0.0	7.25	0.88	0.80	0.75	0.06	2.5	1.0	0.35	1.0	0.23
E-Commerce App	2	1.14	1.0	0.34	8.87	0.88	0.80	0.75	0.06	3.27	1.0	0.46	1.0	0.42
EnterprisePlanner	3	1.00	1.00	0.00	3.50	0.83	0.70	0.67	0.07					
eShopOnContainers	8	1.16	1.00	0.46	71.06	0.94	0.91	0.94	0.03					
FTGO - Restaurant Man.	2	1.13	1.0	0.33	24.00	0.9	0.86	0.9	0.05	0.39	0.12	0.03	0.04	0.34
Lakeside Mutual Ins.	3	1.67	1.00	1.05	6.67	0.83	0.74	0.67	0.08	1.12	1.00	0.12	0.03	0.39
Microservice Blog post	4	1.10	1.00	0.30	13.75	0.88	0.81	0.75	0.06	3.61	1.00	0.36	0.50	0.25
Microservices book	1	1.83	1.00	1.86	1.00	0.50	0.20	0.00	0.24	2.75	1.00	0.46	0.30	0.41
Open-loyalty	3	1.20	1.00	0.40	2.83	0.83	0.71	0.67	0.07					
Pitstop - Garage Manag.	3	1.15	1.00	0.53	14.50	0.83	0.76	0.83	0.08	1.14	0.33	0.09	0.08	0.11
Robot Shop	4	1.50	1.00	0.76	9.38	0.88	0.78	0.75	0.05					
Share bike (Chinese)	3	1.10	1.00	0.30	11.33	0.83	0.76	0.83	0.08	2.38	1.00	0.24	0.13	0.43
Spinnaker	7	1.20	1.00	0.60	18.79	0.93	0.89	0.93	0.04					
Spring Cloud Micros.	7	1.10	1.00	0.30	23.21	0.93	0.89	0.89	0.04	5.75	1.00	0.57	1.00	0.41
Spring PetClinic	2	1.09	1.00	0.29	7.75	0.75	0.60	0.50	0.12	3.01	1.00	0.27	0.35	0.29
Spring-cloud-netflix	7	1.11	1.00	0.31	23.29	0.93	0.90	0.93	0.04	5.75	1.00	0.64	1.00	0.29
Vehicle tracking	4	1.00	1.00	0.00	13.88	0.88	0.82	0.88	0.06					

generating the csv files and also the graphs have been done automatically by MicroDep-Graph which we customised in a way to generate the matrices. The results of the analysis are also available in the github repository [1]

Table 6.5. *Results of the LWF and GWF of the 17 projects.*

Project Name	LWF					GWF				
	Tot	Max	Avg	Med.	Stdev	Tot	Max	Avg	Med.	Stdev
CQRS microservice application	7.0	1.0	0.78	1.0	0.25	2.25	0.25	0.25	0.25	0.0
E-Commerce App	8.5	1.0	0.77	1.0	0.25	2.75	0.25	0.25	0.25	0.0
EnterprisePlanner	4.5	1.0	0.9	1.0	0.2	1.67	0.33	0.33	0.33	0.0
eShopOnContainers	55.5	1.0	0.71	0.5	0.25	9.75	0.13	0.13	0.13	0.0
FTGO - Restaurant Management	20.0	1.0	0.71	0.5	0.25	5.6	0.2	0.2	0.2	0.0
Lakeside Mutual Insurance	7.0	1.0	0.78	1.0	0.25	3.0	0.33	0.33	0.33	0.0
Microservice Blog post	13.0	1.0	0.76	1.0	0.25	4.25	0.25	0.25	0.25	0.0
Microservices book	4.0	1.0	0.8	1.0	0.24	5.0	1.0	1.0	1.0	0.0
Open-loyalty	3.5	1.0	0.88	1.0	0.22	1.33	0.33	0.33	0.33	0.0
Pitstop - Garage Management	13.5	1.0	0.71	0.5	0.25	6.33	0.33	0.33	0.33	0.0
Robot Shop	10.5	1.0	0.88	1.0	0.22	3.0	0.25	0.25	0.25	0.0
Share bike (Chinese)	11.0	1.0	0.73	0.5	0.25	5.0	0.33	0.33	0.33	0.0
Spinnaker	15.5	1.0	0.74	0.5	0.25	3.0	0.14	0.14	0.14	0.0
Spring Cloud Microservice	19.5	1.0	0.75	0.75	0.25	3.71	0.14	0.14	0.14	0.0
Spring PetClinic	10.5	1.0	0.81	1.0	0.24	6.5	0.5	0.5	0.5	0.0
Spring-cloud-netflix	19.0	1.0	0.73	0.5	0.25	3.71	0.14	0.14	0.14	0.0
Vehicle tracking	12.5	1.0	0.74	0.5	0.25	4.25	0.25	0.25	0.25	0.0

[1] Result repository
https://github.com/clowee/Structural-Coupling-for-Microservices

7 DISCUSSION

We presented a carefully selected dataset for microservices-based systems in this study. We built a tool (MicroDepGraph) to discover the dependencies of services in microservices projects in order to analyze them. We looked at 20 open source microservice applications, including both demonstration and production operations. The projects include anywhere from 5 to 25 services. We looked at docker and internal API calls while analyzing dependencies. This tool can examine any microservice system that uses the Docker environment, independent of programming languages or frameworks, thanks to the Docker analysis.

Table 6.4 shows that the average values of structural coupling metrics computed across all services in each project tend to differ from CBM and the basic degree measure. This supports our hypothesis that such a statistic might give a distinct perspective on the services composition of microservice-based applications. We give a graph representation of some of the projects in Table 6 to help with the interpretation of such metrics and to corroborate our hypothesis. In particular, as stated in Section 5.1.1, the service composition of a project may be represented as a directed graph using the estimated coupling values among the services.

The results demonstrate that structural coupling may be quite useful for developers who want to see how their services are broken down. Furthermore, these metrics provide two distinct, but complementary, perspectives on the services decomposition. These kind of metrics, we think, may be utilized to guide refactoring operations or re-modularization at the microservice composition level. Additionally, when evolving/migrating systems to the cloud, they may be utilized to quickly diagnose any improper development collaboration techniques.

It's worth noting that CBM [46] can't be calculated in six of the 17 projects, while Structural Coupling can be used in all of them. It's also worth noting that our approach, when combined with the visual representation of structural coupling, makes it simple to spot microservices with a high out-degree and compare the Structural Coupling of each node graphically.

7.1 Threats to Validity

During this work we were aware of some limitations, such as under some circumstances, both SLOCcount and MicroDepGraph may inaccurately assess the projects. Furthermore, when it came to SLOCcount, we just looked at the Java lines of code. We recognize that certain projects may contain code written in a different language or that the tool may produce inaccurate results. Another significant risk is that the dataset will become too generalized. The list of projects was compiled using a variety of criteria (see Section 3.2). Furthermore, some projects are educational or experimental projects that cannot possibly reflect the entire open-source ecosystem. Furthermore, because the information excludes industrial initiatives, we are unable to speculate about closed-source projects.

Inter-service communication is critical in microservices architecture. There are several methods to accomplish this communication, but it is important to remember that endpoints are smart and pipes are dumb, as outlined by Martin Fowler in his paper [2]. There are two basic types of microservice communication: synchronous and asynchronous. It is necessary to know what form of communication is used in the provided microservice architecture in order to identify the architectural pattern. Despite the fact that it is presently limited to Docker compose files, the sctructutral coupling representation graph analysis might be expanded to include REST or synchronous calls between services in a microservice architecture.

However, from the perspective of the architecture, depending solely on REST or synchronous calls is not a good practice and has bad implications for the architecture's future evolution. Only relying on inner RESTful service calls, for example, provides tight coupling between services. When simply utilizing REST requests, there is also the problem of blocking to consider. When a REST service is invoked, it is blocked while waiting for a response. Because the thread may be executing other requests, this affects application performance.

8 CONCLUSION

Microservice-based systems are becoming more popular, but to the best of our knowledge, there are no proven metrics and dedicated dataset for evaluating service coupling and cohesiveness. Some scholars (Bogneret al., 2017a) proposed extending SOA coupling metrics to microservices, however these measures have never been confirmed or used.

This paper attempts to address this need by sharing a dataset and providing the first curated list of microservice-based applications. The collection is made up of 20 open-source applications that all use different microservice architectural patterns. Furthermore, the dataset contains information regarding inter-service calls and dependencies between the abovementioned projects. Throughout this work, we established the Structural Coupling, a measure based on structural interdependence between services, to aid professionals in precisely identifying coupling between services.

In this research, we established the Structural Coupling, a measure based on structural interdependence between services, to aid practitioners in precisely identifying coupling between services. We tested the structural coupling measure on 17 Open Source projects built using the microservice architectural pattern, and we provided a visualization to illustrate the metric graphically. The findings reveal that structural coupling clearly demonstrates the degree of coupling between current services, and that the visualization offered may be used to quickly identify coupling concerns in (micro)services. Unlike other microservice coupling measures, structural coupling appears to be always relevant, but CBM [46] is not always applicable because its denominator might be zero.

In the context of graph representation and evaluating dependencies in microservices architecture, checking asynchronous calls might be a useful feature, allowing any microservices architecture to be examined using this tool. There are various characteristics that can be used to do this. First, the tool should be able to determine which microservices are event generators. The services can then be mapped according to the producers. Then it may hunt for event listeners for a certain producer, building a dependency-like connection.

Because multiple technologies are utilized to achieve async communication, such as RabbitMQ or Kafka, analyzing both implementations in a project might be a difficulty in the future. However, if a general pattern can be found in both technologies, the problem can be overcome. Because the MicroDepGraph only analyzes the code statically, the ability to discover dependencies during runtime would have been a good addition. The findings of both static and dynamic analysis might be integrated in this way to improve the correctness of the dependency graph of the microservice architecture pattern.

The packaging of the script to compute the measurements and create the visuals into an Open Source utility is one of the next steps. Other types of links between services should be researched as far as the application of structural coupling to diverse systems is concerned. The creation and validation of metrics to evaluate the system decomposition, including cohesion measures, will be part of future work. Structural Coupling will also be used to other cloud-native technologies, such as serverless functions, in the future.

REFERENCES

[1] Jr, F. P. B. *The mythical man-month: essays on software engineering*. Reading (Mass.): Addison-Wesley, 1975.

[2] Lewis, J. and Fowler, M. *MicroServices*. www.martinfowler.com/articles/microservices.html, Accessed: December 2016. Mar. 2014.

[3] O'Hanlon, C. A Conversation with Werner Vogels. *Queue* 4.4 (May 2006), 14–22. ISSN: 1542-7730. DOI: 10.1145/1142055.1142065. URL: https://doi.org/10.1145/1142055.1142065.

[4] *Microservices Adoption in 2020*. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/.

[5] Taibi, D., Lenarduzzi, V. and Pahl, C. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing* 4.5 (2017), 22–32. DOI: 10.1109/MCC.2017.4250931. URL: http://tuni.summon.serialssolutions.com.libproxy.tuni.fi/2.0.0/link/0/eLvHCXMwjV1LaxsxEB6ang
RQiB1pH3r05pqYXEwvzVnoCaZ0beI19OdXo10vTsghp12hYZE0s5oZaeYbgKqc0emTPSEYGSU3jYjeGl
DL5jddjakwIIceihRm-5qt9v3F7PDm7SboJ0apO4EQy0.

[6] W, S. and Natis, Y. *'Service-Oriented' Architecture, Part 1, Gartner report SPA-401-068*. Apr. 1996.

[7] W, S. and Natis, Y. *'Service-Oriented' Architecture, Part 2, Gartner report SPA-401-069*. Apr. 1996.

[8] Papazoglou, M. P. Service -oriented computing: Concepts, characteristics and directions. *Proceedings - 4th International Conference on Web Information Systems Engineering, WISE 2003* (2003), 3–12. DOI: 10.1109/WISE.2003.1254461.

[9] Laskey, K. B. and Laskey, K. Service oriented architecture. *Wiley Interdisciplinary Reviews: Computational Statistics* 1.1 (2009), 101–105. ISSN: 19395108. DOI: 10.1002/wics.8.

[10] C. Matthew MacKenzie and Ken Laskey and Francis McCabe and Peter F. Brown and Rebekah Metz. Reference Model for Service Oriented Architecture 1.0, Committee Specification 1. August (2006), 1–31.

[11] Ramamoorthy, C. V. and Tsai, W. T. *Advances in software engineering*. Vol. 29. 10. 1996, 47–58. ISBN: 9783540897613. DOI: 10.1109/2.539720.

[12] Bianco, P. Evaluating a Service-Oriented Architecture. September (2007).

[13] Almeida, E. de, Alvaro, A., Lucredio, D., Garcia, V. and Lemos Meira, S. de. RiSE project: towards a robust framework for software reuse. eng. *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004*. IEEE, 2004, 48–53. ISBN: 0780388194.

[14] Wolff, E. *Microservices: Flexible Software Architecture*. eng. 1st ed. Addison-Wesley Professional, 2016. ISBN: 9780134602417.

[15] *Conway's Law*. URL: http://www.melconway.com/research/committees.html.

[16] Yahia, E. B. H., Réveillere, L., Bromberg, Y.-D., Chevalier, R. and Cadot, A. Medley: An event-driven lightweight platform for service composition. *International Conference on Web Engineering*. Springer. 2016, 3–20.

[17] Rudrabhatla, C. K. Comparison of event choreography and orchestration techniques in Microservice Architecture. *International Journal of Advanced Computer Science and Applications* 9.8 (2018), 18–22. ISSN: 21565570. DOI: 10.14569/ijacsa.2018.090804.

[18] Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J. and Tilkov, S. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35.3 (May 2018), 24–35. DOI: 10.1109/MS.2018.2141039. URL: https://ieeexplore.ieee.org/document/8354433.

[19] Esposito, C., Castiglione, A. and Choo, K.-K. R. Challenges in Delivering Software in the Cloud as Microservices. *IEEE Cloud Computing* 3.5 (2016), 10–14. DOI: 10.1109/MCC.2016.105.

[20] Ranjan, R. The cloud interoperability challenge. *IEEE Cloud Computing* 1.2 (2014), 20–24.

[21] Bernstein, D. Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Computing* 1.3 (2014), 81–84. ISSN: 23256095. DOI: 10.1109/MCC.2014.51.

[22] Ibrahim, M. H., Sayagh, M. and Hassan, A. E. A study of how Docker Compose is used to compose multi-component systems. *Empirical Software Engineering* 26.6 (2021), 1–27.

[23] Harrison, N. B. and Avgeriou, P. How do architecture patterns and tactics interact? A model and annotation. *Journal of Systems and Software* 83.10 (2010), 1735–1758.

[24] Bass, L., Clements, P. and Kazman, R. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[25] Shaw, M. and Clements, P. A field guide to boxology: Preliminary classification of architectural styles for software systems. *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*. IEEE. 1997, 6–13.

[26] Taibi, D., Lenarduzzi, V. and Pahl, C. Architectural patterns for microservices: A systematic mapping study. *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science* 2018-Janua.March (2018), 221–232. DOI: 10.5220/0006798302210232.

[27] Akbulut, A. and Perros, H. G. Performance Analysis of Microservice Design Patterns. *IEEE Internet Computing* 23.6 (2019), 19–27. DOI: 10.1109/MIC.2019.2951094.

[28] Montesi, F. and Weber, J. Circuit Breakers, Discovery, and API Gateways in Microservices. (2016). arXiv: 1609.05830. URL: http://arxiv.org/abs/1609.05830.

[29] Tighilt, R., Abdellatif, M., Moha, N., Mili, H., Boussaidi, G. E., Privat, J. and Guéhéneuc, Y. G. On the Study of Microservices Antipatterns: A Catalog Proposal. Associa-

tion for Computing Machinery, July 2020. ISBN: 9781450377690. DOI: 10.1145/3424771.3424812.

[30] Runeson, P. and Höst, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14 (2 Apr. 2009), 131–164. ISSN: 13823256. DOI: 10.1007/s10664-008-9102-8.

[31] Seaman, C. B. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering* 25.4 (1999), 557–572.

[32] Caldiera, V. R. B. G. and Rombach, H. D. The goal question metric approach. *Encyclopedia of software engineering* (1994), 528–532.

[33] Patton, M. Q. *Qualitative evaluation and research methods*. SAGE Publications, inc, 1990.

[34] *SLOCCount User's Guide*. URL: https://dwheeler.com/sloccount/sloccount.html.

[35] Basili, V., Caldiera, G. and Rombach, H. The Goal Question Metric Approach. Encyclopedia of Software Engineering-2 Volume Set. *Copyright by John Wiley & Sons, Inc* (1994), 528–532.

[36] Newman, S. *Building microservices*. " O'Reilly Media, Inc."

[37] Fenton, N. and Bieman, J. *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[38] Jabangwe, R., Börstler, J., Šmite, D. and Wohlin, C. Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. *Empirical Software Engineering* 20.3 (2015), 640–693.

[39] Kramer, S. and Kaindl, H. Coupling and cohesion metrics for knowledge-based systems using frames and rules. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 13.3 (2004), 332–358.

[40] Basili, V. R., Briand, L. C. and Melo, W. L. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering* 22.10 (1996), 751–761.

[41] Bogner, J., Wagner, S. and Zimmermann, A. Automatically measuring the maintainability of service- and microservice-based systems - a literature review. Vol. Part F131936. Association for Computing Machinery, Oct. 2017, 107–115. ISBN: 9781450348539. DOI: 10.1145/3143434.3143443.

[42] Bogner, J., Wagner, S. and Zimmermann, A. Towards a practical maintainability quality model for serviceand microservice-based systems. Vol. Part F130530. Association for Computing Machinery, Sept. 2017, 195–198. ISBN: 9781450352178. DOI: 10.1145/3129790.3129816.

[43] Taibi, D. and Systä, K. From monolithic systems to microservices: A decomposition framework based on process mining. SciTePress, 2019, 153–164. ISBN: 9789897583650. DOI: 10.5220/0007755901530164.

[44] Panichella, S., Rahman, M. I. and Taibi, D. *Structural Coupling for Microservices*. URL: https://orcid.org/0000-0003-4120-626X.

[45] Engel, T., Langermeier, M., Bauer, B. and Hofmann, A. Evaluation of Microservice Architectures: A Metric and Tool-Based Approach. *Information Systems in the Big Data Era*. Ed. by J. Mendling and H. Mouratidis. Cham: Springer International Publishing, 2018, 74–89. ISBN: 978-3-319-92901-9.

[46] Taibi, D. and Systä, K. A Decomposition and Metric-Based Evaluation Framework for Microservices. *Cloud Computing and Services Science*. 2020. ISBN: 978-3-030-49432-2.

[47] Hirzalla, M., Cleland-Huang, J. and Arsanjani, A. Service-Oriented Computing — ICSOC 2008 Workshops. *A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures*. 2009.

[48] Bouwers, E., Correia, J. P., Deursen, A. v. and Visser, J. Quantifying the Analyzability of Software Architectures. *Int. Conf. on Software Architecture*. June 2011. DOI: 10.1109/WICSA.2011.20.

[49] Shim, B., Choue, S., Kim, S. and Park, S. A Design Quality Model for Service-Oriented Architecture. *Asia-Pacific Software Engineering Conference*. Dec. 2008.

[50] Perepletchikov, M., Ryan, C., Frampton, K. and Tari, Z. Coupling Metrics for Predicting Maintainability in Service-Oriented Designs. *Australian Software Engineering Conference (ASWEC'07)*. Apr. 2007.

[51] Chidamber, S. R. and Kemerer, C. F. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.* 20.6 (June 1994), 476–493. ISSN: 0098-5589.

[52] Rud, D., Schmietendorf, A. and Dumke, R. R. Product Metrics for Service-Oriented Infrastructures Product Metrics for Service-Oriented Infrastructures. *Int. Works. on Software Metrics (IWSM)*. 2006.

[53] Yourdon, E. and Constantine, L. L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1979. ISBN: 0138544719.

[54] Savic, M., Ivanovic, M. and Radovanovic, M. Analysis of high structural class coupling in object-oriented software systems. *Computing* 99.11 (2017), 1055–1079. DOI: 10.1007/s00607-017-0549-6.

[55] Myers, C. R. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical review E* 68.4 (2003), 046116.