From Monolith to Microservices: A not yet defined Approach

Bachelor's Thesis of

Niko Benkler

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. B
Advisor:	Dr. Robert Heinrich

xx. Month 20XX – xx. Month 20XX

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . . . .

(Niko Benkler)

Abstract

Powered by the rise of cloud computing, agile development, DevOps and continuous deployment strategies, the microservice architectural pattern emerged as an alternative to monolithic software design. Microservices, as a suite of independent, highly cohesive but loosely coupled services, overcome the shortcoming of centralized monolithic architectures. Therefore, prominent companies recently migrated their monolithic legacy applications successfully to microservice-based architecture. They key challenge is to find an appropriate partition of legacy applications - namely *microservice identification*. So far, the identification process is done intuitively based on the experience of system architects and software engineers - mainly by virtue of missing formal approaches and a lack of automated tool support.
However, when application grow in size and become progressively complex, it is quite demanding to decompose the system in appropriate microservices. This thesis provides a formal identification model to tackle this challenge. The identification process is based on... // Hier jetzt noch kurz beschreiben was ich eigentlich machen will We use

Zusammenfassung

Deutsche Zusammenfassung

Contents

List of Figures

List of Tables

1. Introduction

The monolithic software architecture is the traditional pattern to design software, where functionality is bundled in one single, large application [5]. Although monoliths have their strength, like fast development and simple deployment, they become an obstacle when they grow in size and become more complex [21]. Incomprehensible code structure makes it difficult to add functionality, fix bugs and enable new software engineering approaches like Continuous Delivery and Continuous Deployment. Besides, the rise of cloud computing demands a new architecture that can fully exploit the rich set of features given by the cloud infrastructure [16].
Microservice Architecture is about to become a promising alternative to overcome the shortcomings of centralized, monolithic architectures. Inspired by service-oriented computing, microservices gain popularity in both, academia and industry [2]. Benefits like the increase of agility, resilience or scalability [22], the ability to use different technology stacks and independent deployment [3], and the efficient resource utilization in cloud environments [16] explain, why big companies like Google, Netflix, Amazon, eBay [5] and Uber [22] migrated their monolithic architectures to microservice-based applications.
This thesis describes the current state of the art regarding microservices extraction and provides a systematic approach to decompose a (legacy) application into microservices.

1.1. Motivation

Monolithic software applications develop over time and become more and more complex. The software structure is highly coupled and hard to maintain [9]. To tackle this issues, software engineers started to decompose their system into modules and provide the functionality over the network as Web Services [11]. The so-called *Service-oriented Architecture* (SOA) provides logical boundaries between the different software modules to address the design challenge of distributed systems. Nevertheless, Baresi et. al state that the boundaries between modules in SOA are too flexible and the application results in "a big ball of mud" [3]. Microservices make these boundaries physical as each service runs in its own process and only communicates with other services through well-defined lightweight mechanisms like REST [22]. Chen et al. consider the microservice architecture as a particular approach for SOA [5]. Others look at it as an evolution of SOA with differences in service reuse [3] or consider it to be the "contemporary incarnation of SOA" combined with modern software engineering practices like continuous deployment [11]. There is no consensus about the relationship between microservices and SOA, but clearly, SOA paved the way for the rise of the microservice pattern.
The microservice architecture has many advantages over the monolithic style. Sec.2.2 elaborates the main aspects of microservices, including several benefits. Netflix, for in-

stance, is able to cope with one billion calls a day to its video streaming API, by migrating their monolithic system to a high flexible, maintainable and scalable microservice architecture [5]. Consequently, moving existing applications to a microservice landscape is a hot topic in academia and industry [2].
Nevertheless, decomposing a system in loosely coupled, fine-grained and independent microservices is a time consuming task that requires tedious manual effort [11] and is technically cumbersome [6]. So far, it is done mainly intuitively and relies on the experience of software architects and system designers. Hence, a formal approach to identify microservices is required. This thesis intends to describe an approach to systematically decompose a monolithic system into loosely coupled, but high cohesive fine-grained microservices.

1.2. Problem Statement

The microservice architecture is a fast rising approach to structure a system in high cohesive but loosely coupled and independent services. Many companies like Amazon, migrated their monolithic legacy software to microservice in order to fully leverage the benefits of cloud computing and new software engineering approaches like Continuous Deployment [16]. Large applications are decomposed into small, independent microservices where each service can be independently scaled and deployed.
However, one of the biggest problem in designing a microservice architecture is to decompose a monolithic application into a suite of small services while keeping them loosely coupled and high cohesive. This challenging task is also known as *microservice identification* [2].

Baresi et al. state that "proper" microservice identification defines how systems will be able to evolve and scale [3]. Others claim, that finding the optimal microservice boundaries and service granularity is the key design decision to fully leverage the benefits of microservices [10] [12].
So far, the partition is performed mainly intuitively based on the experience and know-how of experts that perform the extraction. Hassan et al. criticises a lack of systematic approaches to reduce the complexity of the extraction process [12]. Extracting microservices from monoliths therefore requires tedious manual effort and can be very costly [22] [17]. This thesis aims to reduce the complexity by providing an approach to systematically decompose a monolithic application into microservice. In the following, the challenges of microservice identification are presented.

1.3. Challenges

*highly interconnected, challenging task to decompose system intoa ppropriare ms / objet-aware

* preserve coherent features (minimize cross talk) / heurisstics *restructuring technically cumbersome, tedious search, identifixcation of suitable parts, program code rewrites /heuristics * costly, error prone, millions LOCm multitude of functional dependecies across

modules and software packages /heur *achieving high scala, availa, low system latencies through the cloud while atrtaining hc, lc between sw components /heur *abstent of sound evaluation method aggravate challenges /heur

* big challenge: Find appropriate partition of sytem into microservices /robert * ms architecture can significantly affect performance * microservice granularity/number of ms influence QoS /robert

*Finding adequate granularity/cohesiveness of microservices /interface * tradeoff between size and numbe rof ms /interface

*distribution not trivial since lack of desin and architecture decision, absent rasoning on original design decision, udermining irigial desin decision a many additions alterations have been made

1.4. Thesis Outline

1.5. Stichpunkte

1.5.1. Approaches

* [2]: Object aware,identification from business processes using structural dependency and data object dependency
* [11]: Service Cutter, service decomposition based on 16 coupling criteria extracted from literature and industry experience [3]: interface Analysis, "automated process for identifying candidate ms my means of lightweight, domain.aghnostic semantic analysis of the concepts in input sepecification with regard ti a reference vocabulary"
[22] : Functional decomp, systematic approach to identify ms in design phase by identifying relationships between required sytem operations and state variable that ops read/write, visualized in Graph and the clustering
[5]: Dataflow-driven, Top down dataflow-driven decomposition algorithm, purified dataflow-driven mechanism
[20]: Domain Engineering, methodology that uses DDD patterns to partition ms by respecting ms architecture design and characteristics
[17]: Extractio of Microservices: Graph-based clustering approach with three different coupling strategies that rely on (meta-) information from monolithic code bases (VCS)

[6]: Heuristics, find consumer-oriented parts of enterprise systems that could be reengineered as ms

1.5.2. General Annotations

* Several companies recentyl migrated to MS like Netflix, Amaozn, Uber * not many approaches and identified approaches are not mature
* many use clustering
* based on code inspection, models... various approaches * use chapter "Supporting Definitions"
* Fowler (widely adopted definition): an approach for developing a single application

as a suite of small services, each running in its own process and communicationg with lightweight mechanisms, often a RESTful API.
* [3]: nmot fully migrate application to ms, preserve monolithic structure and only replicate certain components...
Decomposition Approaches by (according to [22]):
*Usecases(service that is responsible for particular action) *verbs (service that is responsible for particular action) *nouns/resources (service responsible for all operations on entities/resources of a given type) * business capability: Something that a business does in order to generate valu * domain-driven design sub-domain (business splitted in different sub-domains) –> Seems to be most common approach

1.5.3. Object-aware Identification

[2]
*Microservices:
*MS architectural style inspired by service-oriented computing
*cohesive and loosely coupled components, implement business capabilities (each service one capability)
* gain popularity in academia and inudtrsy
*overcomeshortcomings of centralized, monolithic architectures
* single app as suite off small services
* communication via leigthweight mechanisms
* 3 principles:
* i)Bounded contexts(focused on business capabilities, related functionality implemented in one capability and implemented in one service)
* ii) Size(if servize too large, split up, maintaining focus on providing one business capability only in one service)
* iii) Independence(loose coupling and high cohesion)
* –> encourages lossely... as each ms operates mostly independent of others
* independet collection of highly inter-related activities

Challenges:
* decompose system in cohesive lossely couped, fine-grained MS
* done intuitively by experience of system archjitects/Domain experts
* functionalities highly interconnected
* Problem well known

1.5.4. Service Cutter

[11]
Microservice:
*important to split distributed systems in lc/hc units
*autonomous, network-accessible services.
* functional decomposition not new (1972 D.L. Parnas " Ont the Criteria to be used in

Decomposing Systems into Modules)
* Allow developers to chose most appropriate technology for each particular business problem
* MS is "contemporary incarnation of SOA" combined with modern sw engineering practices like continous/idependetnt deployment
* decomposition og monolith in services not fully understood ("very much of an art")

Motivation:
* SW became more complex, sw engineers started to distribute modules/functionality over network (Web Services)
* SOA tackled design challenge of distributed systems
* requirements-driven, repeatable and scalable service decomposition method (supported and partially automated by tools) does not exist
* use DDD: services derived from bounded contexts are aligned to domain model boundaries and team organization structures
* service accessesd remotely through invocation interface

Challenges:
* loosley coupled and highly cohesive services crucial for maintainability/scalability of software
* Not only DDD but also stakeholder requirements: Architecturally significant requirements (sw quality attributes)
* Non-functional requirements of sw is key aspect of analysis and design

1.5.5. Microservice Identification Interface Analysis

[3]
General:
* MS evolution of SOA but differences: service reuse less; instead of reusing existing ms for new tasks they should be small enough to rapidly implement new one that can coexists, evolve, replace the previous one

Microservice:
* ms style: suite of small, autonomous, conversational services
* contrary to monoliths: independent deployability(scalability, using different technology stacks)
* boundaries between software module in tradiditonal services (SOA) often too flexible –> "big balls of mud" –> MS make boundaries physical
* partition ease system maintenance and defines hgow system is able to evolve and scale

Motivation:
*overcome shortcomings of centralized monolithic architecture (deployed as a big chunk)
* definition of granularity level and trade-off between soze and number of MS is still

blurred

Challenges:
* finding right granularity/cohesiveness (when starting or transforming project)
* identification extraordinary affects how system will be able to evolve –> define proper services is key challenge

1.5.6. Functional Decompositions

[22]

General:

Microservices:
* rising fast, many companies use it to structure their systems
* usually defined intuitively based on experience of designers
*advantages: scalability (thus resillience), enhanced performance
* inteact via messages using standard data formats and protocols, publish interfaces using well defined lightweight mechanisms such as REST
* each microservice has own domain model (data, logic, behaviour), related functionality combined in one business capability (= bounded context), one MS implements capability (more if too big)
*tackles complexity of large application by decomposition in small pieces (each in own bounded context)
* architecture enable traceability between requirements and system structure –> Only 1 ms needs to be
changed/redeployed (overcome shortcomings of monoliths: Deployment easy but everything deployed to a server making changes impact whole application –> redeploy whole app)
* two main advantages: functional decomposition, decentralised governance

Challenges:
* find appropppriate partition of system –> architecture can significatnly affects performance of system (intra-service calls?)
* lack of systematic approaches
* hardly any guidelines on what is "a good size" of ms –> Differs from system to system according to research
* two concepts for DB
* i) Share nothing: Each ms has own db –> Higher speed, horizontal scalability, but price of data consistency (only evenutal consistency)
*ii)"microservice is not an island" –> share DB but price of less Independence

1.5.7. Dataflow-Driven Approach

[5]

General:
* Monolith Strength: simple to develop, test, deply, scale. But: Grow in size –> monstrous monolithic architecture, complex/incomprehensible code structure, hold back bug fixes, slow down development, obtsacle for CD
* particular approach for SOA but no consensus on relationship of ms and SOA
* Advantages accepted in academia and industry
* Advantages: Netflix can deal with billion calls every day by streaming API in ms structure
* decomposition process represented by Y-Axix of scaling cube
* decomposition by business capability and by domain-.driven design sub domain (abstract patterns, require human involvement); by verb/use cases or nouns/resources (easier to realize automation as lonmg as criteria have been predefined)
* Service cutter lack objectivity: Scoring the edge of the graph
*SOA: Services coarse-grained (vs. fine grained), decomposition aims at selecting the optimal composed service from all possible service combinations regarding quality requirements –> Bottom Up (vs. MS decomposition is top-down partition, then bottom-up integration)

Microservices:
* multiple small-scale and independently deployable ms
* design time features: Coupling between ms
*runtime feature: scalability against changing workload
* three design time principles:
* i) Fine grain and focus: small, autonomous as main characteristics, each ms does "one thing well"
* ii) High cohesion/loose coupling: ms implement relatively independent piece of business logic (hc), microservice should barely depend on other (lc)
*iii) Neutral devel. technology, services should be deployable as individual applications with own delivery pipeline

Challenges:
* ms not panacea, main issue is effectively decompose a monolithic application
* commonly manual decomposition process prevent practice for achieving those benefits

1.5.8. Domain Engineering approach

[20]

General:
* No guidelines for designing right microservice regaqrdin scope/size

Microservice:
* ability to make rapid functional changes (compared to monolith)
* Issues: integration between communications of applications, complexities arising from creating distributed system (testing, deployment, increased memory consumption)
* no common size for all services –> different sizes possible

Challenges:
*determining th right size of business capability
* too fine-grained services can cause inefficiency (high amount of interactions to fulfil one request)

1.5.9. Extraction from Monolithic software architecture

[17]

General:
* mobile computing, cloud computing infrastructure, DevOps, elastic computer –> MS driven by there developments
* Service Cutter: no means of mining/constructing necessary structure information of monolith –> relies on user to provide sw-artifacts in specific model
*Single Responsibility Principle: sw should only have one reason to change –> ms has to follow this principle

Microservice:
* half the team size
* more flexible horizontal scaling in IaaS environment
* dedicated to relatively narrow and concise task

Challenges:
*key challenge: Extracting ms from existing codebases (migration of monolith)
* tedious manual effort for analysis of many dimensions of software
* tool support and automation is not satisfactory
* good part of efforts in ms research is only conceptual

1.5.10. Function Splitting Heurisitcs

[6]

General:
*SOA: services include all logical related operations *MS: fine-grained components with individual operations

Microservices:
* availability, processing efficiency,
* distributed system allows scalability and availability as deployed in different containers (dynamic de-allocation of ressources on demand) –> Load balancing
* circuit breaker: redirection to another ms instance if no answer after certain time
*

Challenges:
*restructuring enterprise systems (large compley, contain complex business processes encoded in application logic) is technically cumbersome
* requires tedious search, identification of suitable parts to restructure, program code rewrites –> Costly and error prone (because of millions of LOC)
* studies reveal that success rate of re-modularisation techniques especially for large system remain low

*————————Approaches finished here————————-

1.5.11. Migrating web Applications to Clouds with Microservice Architectures

[16]

General:
*Cloud Computing advanced, rich set of features in cloud applications, low cost, low threshold
* Before: CLient server pattern for old web applications

Microservices:
* organized around business capability with borad-stack modules for achieving the business (UI, Business Logic, persistent storage)
*encapsulated with firm modle boundary –> Independan
t * decentralizes transaction and datastorage (own managing)
* collaborating with other services to accomplish desired business capability

Challenges:
*need extensive support for infrastructure automation techniques –> Suitable on well infrastructure supported cloud environment

1.5.12. Entice approach

[13]

General:
*Cloud Computing enabled elastic and on-deman service –> achieve more efficient resource utilisation/quicker response to varying application loads
* Large scale service oriented applications (monolith) mostly inelastic
*user only need some of the offered functionality: Still ned to host complete monolithic service –> Rest not needed and therefore large amount of VM ressources left unused

Microservices:
*Decomposing large services open towards elasticity
* single,well defined functionality offered by a particular VM –> Image can be optimised to host this functionality

1.5.13. A systematic Mapping Study in Microservice Architecture

[1]

General:
*(2016) "microservices architectural style research is still in its infancy" !!!!!!

Microservices:
*"the minimal independent process that acts via messaging"
* increase in agility, developer productivity, resiliencem scalability....

Challenges:
*Communication/Integration: finding right communication strategy is vital (protocol, response time expectations, timeouts)
* Service Discovery: Services need to discover each other, needs standard/consistens process to announce themselves, specify how API gateways are configured
*Performance: ms adds more communication between different services, one single business functional requirement could need several service calls –> End User experience endagerd –> Data sharing/synchrnisation necessary
* Fault tolerance: Partial failure recovery, cloud environment might crash (IaaS failure)
*Security: Communication created trust relationships –> user needs to be identified in all chains of a service communication (OAuth as solution)
*Tracing/Loggin: Distributed tracing necessary to track chain of service calls, logging vital for debugging purposes –> Needs careful design of central logging/Aggregation system
*Application Performance Monitoring: measuring individual ms
*Deployment operations: deployment/scaling fundamental infrastructure concerns –> selecting right platform/orchestration tools is fundamental

1.5.14. Migrating towards Microservice Architectures: Survey

[9]

General:
* legacy sw: highly coupled, hard to maintain
*industrial survey: roughly 28 months for migration –> Costly
* shows that most industrial develloper get inforamtion about the old system of source code and textual/architectural documents to better understand the pre-existing system and to be able to architect a new one
* it is significant to have a "crystal clear understanding of the domain of the system in order to identify bounded contexts properly"
*Main driver to mirgate: New functionality, long time to release new features, hard to maintain, side effects, low productivity of developers, hard to test, developers interfering with each other

Microservices:
*flexible/evolvable architecture

Challenges:
*Many technical challenges: Infrastructure automation distribute debuggin
*Organizational challenges: creation of cross functional teams

1.5.15. Infoq: Decomposing applications

[21]

General:
* Small services is not the main goal! Decompose system into services to solve problems of monolithic architecture is main goal –> Services can vary in size
* Call MS lightweight or fine-grained SOA
* solves problems that many organizations suffer from
* Application, several components with good mopdular design (domain model) but deployed as a monolith! –> simple to develop (IDEs are oriented arounf developing single applications), easy to test
* BUT: Becomes unwieldly for complex applications, difficult to adopt new technology without rewriting entire application –> Does not scale to support large long-lived applications
* SCALE CUBE:
* X-axis: mutltiple identical copies of app behind load balancer –> improve capacity/availability
* Z-Axis: Each server runs identical copy (similar to X), but each serve ronly responsible for subset of data –> Component in systems responsible for routing request to servers
* –> Z and X improve applications's capacity/availability but do not solve problems of increasing development and application complexitiy –> Y

* Y-Axis: functional decomposition, Z-Axis splits similar things, Y-Axis scaling splits different things
* API Gateway: No talking directly to service, API Gateway in between that provides coarse grained API for CLients
* Inter-Service Communication:
* i) synchronous HTTP-bases: simple, familiar, firewall friendly, does not support other patterns like publish/subscribe, servers must be simultaneous available, HTTP client need to know host/prt of server
*ii)asynchronous AMQP-based message broker: decouples message producers and consumers, broker handles conversation, broker another moving part

Monolith:
* Commonly used pattern for enterprise application, works good for small applications: testing developing deploying relatively simple
* For large/complex apps, monolithic architecture becomes obstacle –> Migrate to MS

Database:
* individual best-fitting database for each service, service needs ACID-transactions –> relational, social media –> graph base
* new Problem: Handle requests that access data owned by mutliple services (approaches are RPC–> increases response ime, reduces availability as other service must be available too; other approach is store copy –>consistency problems)

Microservices:
* small code base does not slow down IDE, makes developers more productive
* services typically start a lot faster than monolith
* independent deployment: change for local change do not need coordination with other developers
* each service scaled independently: X-axis clonin and z-axix partitioning
* service deployed on best suitable hardware
* easier to scale development: organize development around mutliple small two pizza teams
*improves fault isolation
* eliminates any long-term commitment to a techn ology stack

Challenges:
* additional complexitiy of creating a distributed system
* inter-process communication required
* IDE/other devel. tools currently still focused on building monoliths
* operational complexity, many more moving parts (inst6ances of services)
* features spanning mutliple services require careful coordination
* WHEN to use this architecture: first version of application does not have problems ob monolith –> MS slows down development –> Start upds need to deliver fast results
* Granularity on APIs too fin-grained –> results in many required service calls –> Latency

for clients (solution API Gateway)

1.5.16. From Monolith To Microservices Classification

[10]

General:
*high costs/effor for refactoring
* high effort to adapt newer and beter technologies, changing initial desgin choices later nearly impossible
* need to duplicate monolith in order to scale –> inefficient
* refactoring: extend lifetime of existing software product; code level vs architectural refactoring
*service Cutter as "general purpose" approach –> most mature tool support, most strucured and generally applicable way to decompose tool support

Microservices:
* more agile flow of development and operaions (referred as DevOps)

Challenges:
*finding right service granularity to fully leverage advantages
* ms from scratch OR refactorin is vey expensive and time consuming

1.5.17. Toward understanding and evolution

[7]

General:
*monolith does many things and has many responsibilities
* ms is fine-graines SOA
* SRP: each module, subsystem, class, method should not have more than one reason to change
*parts can be migrated directly to ms, other parts need to be refactored to make code more modular

Microservices:

Challenges:
* Distribute legacy artifacts into ms not trivial: lack of sesign and documentation,
* absent reasoning on oroginal design and architecture decisions

1.5.18. Workload-bases Clustering

[14]

General:

Microservices:
*size of MS directly defined by its feautures
* several metrics: LOC, able to rewrite it in 6 weeks, two pizza team per service –> no quality attributes!
*Moving features to other services directly inacts performance/scalability: Smllaer services -> more scalability of system;
* performanace decreases when splitting up –> significant communication overhead
* merging results in loss of scalabiity
* –> Optimize performance vs. scalability by modigying placement of features (= chunk of functionality that delivers busines value)

Challenges:

1.5.19. Architectural Meta-Modelling

[12]
General:
*First class object: Object passed to function or as return value or allocated to variable
*general lack of systematic approaches that model ms design decisions, including deciding the optimal microservice boundaries

Microservices:
*isolated fine-grained business functionalities that act through standardized interfaces
*rapidly increasing: Netflix, Amazon, Uber
* Isolating business functionalities aisma at enhancing the autonomy, replaceability of individual microservice
* enhances decentralised governance

Challenges:

2. Background

2.1. Monolithic Software Architecture

The monolithic software architecture is a well-known and the most widely used pattern for Enterprise Applications, which usually are built in three main parts (top to bottom): The client-side user interface (Tier 3), the server-side application (Tier 2) and the persistence layer (Tier 1). The server-side application - *the monolith* - is a single unit and deployed on one application server [21]. The software structure, if well defined, is composed of self-contained modules (i.e. software components), where each module consists of a set of functions [6]. The monolith implements a complex domain model, including all functions, many domain entities and their relationships. For small applications, this approach works relatively well. They are simple to develop, test and deploy [22]. Fast prototyping is supported by the current frameworks and development environments (IDE), which are still oriented around developing single applications [21].
But once they grow in size, they become exceedingly difficult to understand and hard to maintain without reasonable effort [22] [10]. A complex and large code base prevents a fast addition of new features and makes the application risky and expensive to evolve [15]. Alterations to the system, even though they might be small, result in a redeployment of the whole monolith application due to its nature being a single unit [22]. Moreover, it is difficult to adopt newer technologies without rewriting the whole application, as monolith are build on a specific technology stack [21] [17].
Scaling is only possible by duplicating the entire application - namely *horizontal scaling*. Consequently, large portions of the infrastructure remains unused, if only parts of the application need to be upscaled or even used [13] [9].
Chen et al. provide a short résumé:

"Successful applications are always growing in size and will eventually become a monstrous monolith after a few years. Once this happens, disadvantages of the monolithic architecture will outweigh its advantages." [5]

2.2. Microservices

"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." [8]

2.2.1. Definition

The above quotation is a widely adopted definition of the term *Microservice*, provided by M. Fowler and J. Lewis, the pioneers of the microservice architecture. However, the term is not formally defined. Amiri et al. describes microservices as a collection of cohesive and loosely coupled components, where each service implements a business capability. The author introduces three principles upon which the architecture is build: *Bounded Context, Size, Independence* [2].
The first principle is about related functionality, that is combined in a single business capability - the *bounded context* [22]. Each capability is implemented by one microservice. The *Size* of a microservice is defined by the number of features it provides (namely bundled functional capabilities) [14]. There is no consensus about the "proper" size of a microservice [20], but several guidelines exists: Services should focus on one business capability only [2]. Others state, that the size of a microservice should not exceed a level, where it cannot be rewritten within six weeks [14]. However, the sizes vary from system to system [22] and even different sizes for each microservice in a specific system are possible [20]. The bottom line of *Independence* is in Amiri's description of microservices as "a collection of high cohesive and loosely coupled components" [2]. High cohesive services implement a relatively independent piece of business logic (at the most one business capability). Further, microservices should hardly depend on each other, which is the idea of being loosely coupled [5].
Communication between microservices is achieved by lightweight message passing mechanisms such as *REST*. Each service exposes a well defined interface (*API*) with endpoints that provide information using standard data formats [22]. The design of microservices mainly follows the *Single Responsibility Principle (SRP)*: Each service should not have more than one reason to change [7]. The SRP mainly corresponds to the idea of not implementing more than one business capability. The following covers the benefits and challenges of the microservice architecture.

2.2.2. Benefits

Fast and Independent Deployment
As a matter of fact, each microservice is deployed independently [3]. Changes to the code do not result in a full redeployment of the entire application [22]. Consequently, software developers are able to react much quicker to changes in business requirements. This includes an acceleration in error correction. Per contra, any changes in a monolithic code base requires a time consuming build of a new version and the redeployment of the entire application [8].

Availability, Resilience and Fault Isolation
Microservices are designed to operate independently of each other and to tolerate failure of services [8]. Large parts of the application remain unaffected of partly failures and the availability of the system is, at least partly guaranteed. Monolithic application do not provide this type of fault isolation. If a failure occurs, the whole application remains unavailable as it is usually running in a single process [17].

Scalability and Resource Utilisation
Small and independent microservices allow more fine-granular horizontal scaling [14]. Single services can be duplicated to cope with changing workload during runtime [5]. Thus, dynamic (de-)allocation of resources on demand prevent infrastructure from being idle [6]. Scaling monoliths can only be attained by duplicating the entire application , leaving resources unused [10]. Further, each microservice is deployed on the best suitable infrastructure for its needs, allowing a more efficient system organization [21].

Improved Productivity
In traditional software development, teams are divided based on their expertise: Database architects, UI-developers and server-side engineers, resulting in a three-tiered application (cf. Sec.2.1). Additionally, software engineers are responsible for the development only. Deployment is part of the operations team. This team structure results in high communication overhead and slows down the productivity [18].
In contrast, microservices are organized around business capabilities requiring cross-functional and independent teams [2]. Each team has the full range of skills required for the end-to-end realization of a microservice, including UI-development, database architecture, back-end engineers and project management. This minimizes the communication and interaction between the teams and thus, speeds up the productivity. Ultimately, microservices enable a more agile flow of development and operation [10], also referred as *DevOps*.

Neutral development technology
Microservices are highly decoupled from each other, as they use standardized and lightweight communication mechanisms such as REST [22]. Microservices can be realized using different programming languages, technologies and even deployment environments [5]. Developers are consequently not longer limited to use a single technology for the whole application. They can choose the most appropriate technology for each particular business problem or try out some new technology without rewriting the whole application [11] [15].

2.2.3. Challenges

The previous section provides a vast amount of benefits that come with microservices. However, it is not the panacea of software engineering and has to face some challenges before being able to fully benefit from them. The challenges are further described in the following.

Expensive Communication
Microservice use network protocols such as *HTTP* to communicate with each other. Compared to standard, inter process communication (*IPC*) as used in monoliths, remote procedure calls a more expensive [1]. As a consequence, applications experience a decrease in performance as network communication is generally slower than IPC.

Technical Challenges
Microservices require a high degree of infrastructure automation [9]. The benefits of fast and independent deployment cannot be utilized, if it has to be done manually. Dynamic (de-)allocation of resources when scaling individual microservices need a well defined and structured cloud environment [16]. Besides, the distributed microservice landscape complicates the logging mechanisms and performance monitoring [1]. Traditional centralized logging, as it is used in monolithic applications, is not longer applicable. Instead, a careful aggregation system to gather logging and monitoring data from each service is required.

Organizational Challenges
The microservice approach needs the establishment of cross-functional teams [8]. Adopting *Continuous Practices*, such as *Continuous Deployment*, are essential for the success of a profitable microservice architecture. Therefore, closer collaboration between development teams, operational staff and management has to be established. In summary, a costly and time consuming restructuring process of the entire organization is required [4].

Data Consistency
Distributed systems need to share data. Heinrich et al. propose two concepts for the database architecture [22]: The first concept applies the basic idea of the microservice approach, as it splits the database into several parts. Each microservice has its own database which manages the entities that belong to the corresponding bounded context. Higher speed and horizontal scaling are facing data consistency issues. Data needs to be synchronized which leads to inconsistency, if services are unavailable. The second concept is about sharing a single database. This approach overcomes the issue of consistency, as data is stored centrally. But sharing results in a loss of independence. Scaling can only be achieved through replicating the whole database. Research revealed, that the first concept is preferred [22].

Decomposition
Decomposing a system into microservice is a very complex task that requires experienced system architects and domain experts [8]. Identifying the right granularity of microservice is one of the key issues. Too fine grained services cause inefficiency due to a high amount of expensive inter-service calls [20]. Developing the basic communication infrastructure adds additional complexity and slows down the initial developing process [21].

3. CoCoME

3.1. Introduction to CoCoME

[19]

4. State of the Art

4.1. Literature Review

Link	Titel	Author (Year)	Origin	Search String
[17]	Extraction of Microservices from Monolithic Software Architectures	G. Matzlami et. al. (2017)	Google Scholar	*microservice identification*
[2]	Object-Aware Identification of Microservice	M. J. Amiri (2018)	IEEE	*identification microservices*
[3]	Microservices Identification Through Interface Analysis	L. Baresi et. al. (2017)	google scholar	*microservice identification*
[22]	Identifying Microservices Using Functional Decomposition	S. Tyszberowicz et. al. (2018)	*provided*	*n/a*
[20]	Partitioning Microservices: A Domain Engineering Approach	I. J. Munezero et. al. (2018)	IEEE	*identify microservices*
[5]	From Monolith to Microservices: A Dataflow-Driven Approach	R.Chen et. al	IEEE	monolith to microservice
[6]	Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems	A. De Alwis et. al. (2018)	Google Scholar	identify microservices
[11]	Service Cutter: A Systematic Approach to Service Decomposition	M. Gysel et. al. (2016)	[3]	*n/a*

4.2. Comparison and applicability of the approaches

* bedeuted INFO
+ beudeted PRO
- bedeuted CONTRA

4.2.1. Extraction of Microservices from Monolithic Software Architectures

* informal migration patterns exists. Lack of Formal Models
* small and recent body of work on how to migrate monolith to MS
* clas based extraction model, construct graph, process by clustering algorithm
* references Service Cutter (Pros and Cons)
* 2 phases: Construction (monolith to graph), clustering (decompose graph to cluster)
* starts with code base/repo from VCS
* each class is a node, edges have weights according to coupling strategy (classes that are not coupled are discarded)
* Logical Coupling Strategy(LC): Single Responsibility principle (Software has only one reason to change), enforce strong module boundaries (concept of MS) –> developers only make changes to the module (found in Change History, Class Files changed together belong togehter) ==> Weight is : for each pair of class look how often they changed together
* Semantic Coupling Strategy(SC): each MS correspond to one bounded context (DDD) from domain, examine contents/semantics of source code, term-frequency invese-documents-frequency method (tf-idf), compute relation of two classes regarding domain concepts
* tf-idf: Compute scalar vector for each class and compute cosine similarity between pairwise distinct classes
* tf-idf: Tokenize class, set of words, filter stop words, compare two classes regarding their common words with tf-idf formula
* Main Concern: Well organized teams, cross-functional but also reduce communication overhead to external teams while maximize internal
* Contributor Coupling (CC): team/orga info used to recover relationship among sw artifacts ==> Ownership architecture read from VSC history by identifying how many developers worked on the same pair of classes (weight!)

* CLustering Algo: Invert weights to favor edges with heigh weight, Kruskal for MST (calculate remaining edges), sort edges, reverse list, delete first element (originally lowest weight; now first in the list), in each iteration step: DFS(Depth-First) on edgesMST returns number of partitions, iterate and delete while n<nGiven
*Reduce Cluster: after n cluster were formed, method reduceClusters splits up unusually large clusters (due to language and framework, special classes may have extraordinary high coupling) –> Delete those classes until size of node (numbe rof classes seems to be appropriate)
————————————

+ algorithmic recommendation of ms candidates implemented in web-based prototype
+ unites traditional decomposition techniques and microservice extraction approaches/design principles
+ algorithm uses 3 different coupling strategies: Can be combined for better results
+ performance of algortihm was satisfying according to author
+ shows significant team size reduction (less than half), given that alls contributirs for given cluster work on a MS

+ no work in advance!!!!

————————————————

- rely on (meta-)data extracted from codebase
- needs VCS (proper change history)
- needs ORM model that models data entities as ordinary classes - 2 (independent) changes in one commit destroy SRP
- SW must have gone through evolution process for LC
- Naming of class, methods, attributes needs to reflect domain language to make tf-idf possible
- Contributor Coupling: requires more developers
- Reduce Clusters: Useful to delete those classes? wat if this is core of application that belongs together
- nPart is given by user: What is the right granularity?

————————————————
*Conclusion: Beta implementation existing, paper is precise in usage, master-thesis exists (LINK!!!)with detailed information, easy algorithms, good performance,
* No Information about right n, not domain oriented, ORM necessary, BUT: Not applicable to CoCoME as VCS history, software evolution, several contributor necessary

4.2.2. Partitioning Microservices:A Domain ENgineering Approach

* One BusinessCapability determines one MS, based on domain engineering
* Business Capability: Something hat a business does;combined as functionality that have something in common (instead of usin collections of data entities and CRUD operations in them)
* Paper states: Developers struggle to define granulaity of business capability (too small => Communication overhead, to large: heavyweight SOA)
* Appropriate MS size determined by component boundaries
* DDD: Choosing appropriate boundary. MAIN GOAL: systematically group requirements in domain model and implement code of that
* Design Domain model with sub-domains (each sub-domain has one to more bounded contexts)
(* Sub-Domain is problem space, bounded.context is solution space and maps software artifacts to sub-domains ==> Info aus Internet)
* DDD Patterns: Context Map/Counded Contexts (makes explicit boundaries between domains,), Aggregates (logical boundaries for cluster of domain objects that change during one transaction, ensure consistency among parallel operations that are considered to be one unit in reagrd to change), Ubiquitous Language (ensures that impementation uses the same terms as business), Separation of Entities and Value types (Entities are Objects of DDD, Not all Objects need to be entitites, for value objects identity is not necessary))
* MS should be autonomous (changes not affecting others): One bounded context is one

MS (ore more if fine grained needed, BUT never on C in different MS)
* Approach: Defined domain and ubiquitous language ar prerequisites, split into sb domains if domain is too wide, find boundary of each responsibility, make it as business capability (focus on relationship among different services), each capability is a MS, find out relationships between services with domain model and reconfigure if necessary (too many dependencies)

————————————————

+uses DDD. Many papers and experience reports state that this is the way to gto

————————————————

- draw the boundaries is the ky task and requires domain experts
- requires well defined domain, pre-existing ubiquitous language
- "Find the boundary of each responsibility and make it as a business capability" ==> But how? Requires DOmain experts
- Procedure is not concise at all and only explains approach on a top level (half a page)
- Does not define how procedure analyses "loose coupling possible"
- how break down in subdomains?
- uses brainstorming and interaction wit domain experts to identify and define parts of the microservice

————————————————

* Summary:
*Hot topic according to other papers (interviews, reports) but this paper lacks on accuracy.
* Simply said: Get some domain-experts and let them cut your domain in different bounded contexts
* This is what was already done when migrating CoCoME (probably not following the patterns but having domain experts)

4.2.3. From Monolith to Microservices: A Dataflow-Driven Approah

*top-down dataflow driven decomposition algorithm
* 3 step approach
* purified DFD (focus on data's semanteme and operations only, excludes side information), more data focused than traditional
* purified DFD (PFD) is a directed Graph with nodes (processing operations nodes and data nodes) and Approach:
* construct traditional DFD (according to business logic extracted from users' natural-language descriptions)
* Manual construction of purified dataflow by two rules (more like guidelines)
*algorithmic construction (condensing of PDS) of the decomposable Dataflow by apply-

ing a set of rules
* combines same operations with same output data
* each operation at the end is a MS

————————————————————

+systematic Algorithm
+ purified DFD represents real information flow of corresponding business logic
+ reduce human mistakes when drawing a composable DFD
+ the 5 rules are easy to understand

————————————————————
- Semi Automated
- traditional DFD constructed on users descriptions and code (detailed data flow) ==> exact? Expertise necessary
- or: DFD needs to be existant
- transforming traditional to purified is a non trivial task
- identifying same data operation requires expertise: Combination of sam operations based on operation names (what if names are different through the project)
- no implementation
- not applied to larger projects - based on user's natural language: semantic verbs correspond activities (later a MS)
-really really fine-grained MS, based on single operation (most fine graines as possible)
- no domain information included
- what about get/create.... hier weiß ich nicht, wie ich beschreiben soll dass es nicht geht!
- zweites beispiel zigt das deutlicher mit der operation QUERY

————————————————————

Conclusion

TODO: Why not applicable to CoCoME

4.2.4. Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems

* function splitting based on i) Object subtypes (lowest granularity of sw based on behavioural properties) and ii) common execution fragments across software (lowest ... based on structural properties)
* authors say: MS not adopted for enterprise systems (ERP/CRM Sw which is large and has complex business process)
* lot of BO's (business objects) with many-to-many relationship, functional dependencies
* key strumbling block: "limited insight from syntactic structures of code for profiling software dependencies and identifying the semantics available through the BO relation-

ships. * Process:
* ES system with modules that have function which execute basic CRUD ops on BOs (centralized DB) vs. own DB in MS (structural difference) * Behavioural property: Based on invocation of properties in well-defined processing sequences (different execution sequences reflect a set of SESE regions single-entry single-exit)
*Enterprise System as finite automaton with operations as labels: Vertices defined by states and relation defined by transistion functions
* SESE-Fragments: Induces a function (SubGraph of ES), Function is set of operations
* behaviour (of both MS and ES) based on invocation of operations –> Analyse those processng sequences
*Heuristics:
* given: several similar Execution paths with changed BOs, only Attributes they use are different (structural splitting of objects at BO level) * i) Subtype Relation in SESE(S): Subtype between callgraphs exists if ops, input/output of child are present in parent AND 80percent of parent states appear in child
* given: execution pattern occurs in several patterns (depend on functional relationship): A always before B ==> "as a relationship" property
* ii) Commonality: two call graphs have common sub-graph ==> Small Subgraphs (small MS), large Subgrapgh (large MS)
*Process:
* two components: Business Object Analyser (BOA), System Dynamic Analyser(SDA)
* BOA: Two models, one for evaluating the SQL queries to identify relationships between DB tables and one for identifying BOs based on the relationships and data similarities ==> Input for SDA
*SDA: BOs and callgraphs as Input, identifies Frequent Execution Patterns(FEP) in provided SESEs(S) by using clustering algorithms
*FEP is evaluated against heuristics and classified in categories
* categorized patterns evaluated by BO Relationship Analysis and SESE derivation model
* Microservice Recommendation Interface (MRI) provides MS recommendations
* SDA+MRI: two algos i) set of subgraphs in given set of callgraphs ii) analyse subgraphs for SESE code fragments (functions) that are related to same BO

————————————————————
+ prototype implementation exists
+ already conducted two experiments

————————————————————
- incremental process: most prominent components extracted and reimplemented as MS
- really complex
- BOA not given in paper
- uses a lot of algo that are not explained
- First experiment: log data analysed with DISCO to generate call graphs

————————————————————
Coclusion:

* Approach not fully implemented, implementation not available
* Complex algorithms that use other algorithms from other papers
* log data to get call graphs
* We do not want to incrementally extract MS
* Experiments focused on speed/ressources/time per requests ==> Did not focus on finding the "right" MS according to MS principles
* Not applicable to CoCoME

4.2.5. Identifying Microservices using functional decomposition

* based on UC specifications of sw requirements and functional decomposition of those requirements
create model of the system: finite set of system operations (public methods as response to external triggers) and system state space (system variables written/read by operations)
*system decomposition: each MS has own state space and operations (Goal are disjoint sets, read/write of others state space only via API)) −>syntactical partition of the state space
*system requirements: give by natural language/formal models/existing implementation
*functional requirements: use cases describe how users interact with system
* system operations/ system state variable extracted of Use cases −> operation/relation table
* operation/relation table: relationship between each op and state variable this op reads/writes
* extracting was done using text analysis tools
* Source COde Clustering is possible but only to help to understand structure of the system (if design is bad, the resulting clusters might not be appropriate MSs)

*Approach:
* first approximation: verbs in UC are operations, nouns are state variables (list might be updated, see page 5)
* generate operation/relation table
* Visualize op/rel table as graph, create cluster (good ms candidate low coupling (small amount of information sharing with others) and strong cohesion criteria(internal relationships are dense)!)
* Graph: vertices are state variable AND operations, edge between them if read/update, weights (1 for read, 2 for write)
* Use Graph Analyse Tools to determine clusters and read the MS (cluster = MS)
* Create API (all public ops in the cluster) and own DB for each MS
*Add Getter to API if other cluster needs information of this cluster
*

————————————————————

+ approach is universally applicable once ops and state variable are identified (do not need to be extracted from use cases)
+ 3 ways to use the visualization: separation in low coupled partitions, partitioning in MS

by non-functional constraints, visualize changes (re-engineering sw architecture)
+ identify MS based on business capability is main goal, not reducing MS size
+ 3 evaluation by three independent sw implementations: really similar to this approach
+ manual approach was matter of days, this approach only matter of hours

————————————————————
- synonymous/irrelevant nouns need to be identified via brainstorming –> User might be biased
- no security concerns
- no differences between different reads/writes (always 1/2)
- difference between read or write if other service? same value as REST time dominates

————————————————————
Conclusion:
* already done to CoCoME with good results.
* Hard to find substantial improvements?!

4.2.6. Microservice Identification Through Interface Analysis

* based on semantic similarity of available functionality described through OpenApi Specification and reference vocabulary –> collocation and similar words (DISCO)
*Idea: Same reference concepts are highly cohesive (Level in Hierachy of Vocabulary determines Granularity)
* match terms used in OpenAPI specification (as input) against reference vocabulary
* this is done iteratively on concepts by means of fitness function based on semantic similarity (DISCO)
* co-occurence matrix that contains all mappings of possible pairs of terms and concepts

* Process:
* for each api specification: Map each operation to concept that describes the operation most accurately
* Disco-based semantic assessment: Each operation along with resources (parameters, return values, complex types) is analysed and mapped to reference vocabulary concept using a score (Best Mapping for of term list and all concept available)
* Uses term separator (robust according to author), filters stop words, split word in input terms
* score based on fitness function: all collocation scores (between different words in term list and concept) stored in Matrix
* Mapping is non-trivial, use Hungarian algorithm to determine most adequate mapping
* Concept with highest mapping score is elected as reference concept

————————————————————
+ automated apart from defining OpenApi
+ according to authors: shared vocabulary helps to identify certain concepts with differ-

ent meanings across system
+ APIMatic Tool can generate OpenApi sepcifications from existing interface
+Level parameter to define granularity of groupings (= MS)
+ implementation available

————————————————————

- Needs OpenApi Specification, re-engineering of available interfaces or new interface definition based on available artifacts
- based on reference vocabulary: They use Schema.org (too wide) –> Any other shared vocabulary ir own ontology needs to be constructed/used)
- Mapping to operation highly depends on used vocabulary
- Api needs well defined (provide proper naming) –> Might happen that two different operations (from two different sub-domains accoding to DDD have same naming).
- Reference vocabulary need to be tree
- Schema.org seems to be way to coarse-grained for CoCoME
- define own ontology probably ends in more work than manually defining bounded contexts
- lots of Intangibles (Unbestimmbar in Schema.org)
- Vocabulary might be misleading. From DDD point of view, similar words according to Disco belong not automatically to same business capability

————————————————————

Conclusion:
*Not applicable to CoCoME mainly because of Schema.org (for example: Sell Action and OrderAction in same concept)
*Concerns about well defined API
* No idea how to adapt it

4.2.7. Service Cutter

* service decomposition based on 16 coupling criteria (coming from industry and literature)
* service requires resources: Data (should only managed by one service), operations(service encapsulates business rules), artifacts(snapshot of data or operation results transformed into specific format) –> Geneeralize under term nanoentity
*Approach: Identifying set of services and assign all nanoentities (each) to one service
*Coupling criteria: architecturally significant requirements and arguments why two nanoentities should or should not be owned/exposed by same service
*Software System Artifacts(SSAs): design/analysis atifacts that contain information about coupling criteria
* service cut: output of single execution of service decomposition process
* coupling criteria catalogue: <some examples>..., 4 categories: cohesiveness, compatibility, constraints, communication

* structures using coupling criterion cards in specific format: ubiquitous language to structure, identify decisions

*Process:
* Input in form of SSAs (Use Cases, DDD entities/aggregation, Entity relationship models)
. Service Cutter extracts coupling criteria information of them
* various additional SSAs
* Service Cutter creates nanoentities(coupling criteria based on SSAs)
* User priorizes these criteria (start calulation)
* SC creates unidirected weighted graph with nodes that represent nanoentities and weighted edges that represent cohesiveness/coupling of two nanoentities
* Clustering Algorithm (Can be changed!)
* Weights: Sum of all scores (5 different types of scores) multiplied by priority

————————————————————

+ cited by many papers
+ catalogue consists of DDD pat7terns but also software quality attributes for architecture
+ Clustering Algorithm can be changed by user if necessary
+ user can prioritize coupling criteria (needs user experience, otherwise might result in wrong clustering)
according to whats important for users use case (security yes or no...)
+ implementation available that was used by other papers to compare their approach
+ universally applicable to all software systems
+ wiki exists
+ can be used in forward and backward engineering

————————————————————

- not fully automated –> Only support
- user-prioritized coupling criteria (well can also be positive)
- lots of user input, needs to be in specific format (work intense!)
- results and other papers show that SC does not "cut" the best solution

————————————————————
Conclusion:
* First attempt to automatize service extraction
* To much effort for input (for bigger systems)
* Already implementation and good approach (Further improvement possible?)
* Good to compare with own approach
* Nice tool

4.2.8. Object-aware Identification of Microservices

*again based on clustering, structural and data object dependecy
* three principles:
* i)Bounded contexts(focused on business capabilities, related functionality implemented in one capability and implemented in one service)
* ii) Size(if servize too large, split up, maintaining focus on providing one business capability only in one service)
* iii) Independence(loose coupling and high cohesion)
* paper focuses on identifying from business process point of view
* business process: set of activities to achieve business goal (activity in process = operation in MS)
* goal: decompose business process in fine-grained... MS
* partition by: structural relation (direct edge), data relation (read/write on same data objects)
* Technically: BPMN with data objects read/writes (see paper for notation)

*Process:
* 2 matrices (combination of each activity pair)
* First(structural): 1 if direct edge (or only gateways in between), o else
* Second(object read write): Sum of shared object read/writes (values are 0, 0.25. 0,5, 1 for different combinations of read/writes)
* uses genetic algortihm with Turbo-Mq Fitness Function for clustering (picks random clusters and try to iteratively improve fitness of identified cluster till convergence)
* If more processes: Just recalculate matrix

————————————————————
+ combination of structural and data object dependency seems to be successful
+ easy to understand
+ clustering algorithm can be changed
+ several process can be easily combined
+ values can be changed (adapt parameters)
+ experimental evaluation seemed to be successful (validated against identification by domain experts)
+ Method can be easily replenished by other aspects (requirements, security...)

————————————————————
- no parallelism (gateway)
- needs business process models
- Clustering algo not defined (information in other paper)
- short paper (no detailed information)
- no info about evaluation process, data set...

————————————————————
Conclusion:

* Seems to be applicable to CoCoME
* BPMN already done by Kiana (at least some parts)
* Further improvements?!

5. Solution Overview

6. Evaluation Planning

6.1. Applicability to CoCoME

6.2. Comparison to Functional Decomposition Approach

7. Timetable

7.1. Milestones

Bibliography

[1] N. Alshuqayran, N. Ali, and R. Evans. "A Systematic Mapping Study in Microservice Architecture". In: (Nov. 2016), pp. 44–51.

[2] M. J. Amiri. "Object-Aware Identification of Microservices". In: (July 2018), pp. 253–256. ISSN: 2474-2473. DOI: 10.1109/SCC.2018.00042.

[3] Luciano Baresi, Martin Garriga, and Alan De Renzis. "Microservices Identification Through Interface Analysis". In: (2017). Ed. by Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen, pp. 19–33.

[4] Niko Benkler. *From Traditional Development to Continuous Deployment: Strategies and Practices in CI/CD Pipelines*. Accessed on 20.01.2019. URL: https://github.com/Benkler/Proseminar/blob/master/Niko_Benkler_Proseminar.pdf.

[5] R. Chen, S. Li, and Z. Li. "From Monolith to Microservices: A Dataflow-Driven Approach". In: (Dec. 2017), pp. 466–475. DOI: 10.1109/APSEC.2017.53.

[6] Adambarage Anuruddha Chathuranga De Alwis et al. "Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems". In: (2018). Ed. by Claus Pahl et al., pp. 37–53.

[7] D. Escobar et al. "Towards the understanding and evolution of monolithic applications as microservices". In: (Oct. 2016), pp. 1–11.

[8] Lewis Fowler. *Microservices*. Accessed on 17.01.2019. URL: https://martinfowler.com/articles/microservices.html.

[9] P. Di Francesco, P. Lago, and I. Malavolta. "Migrating Towards Microservice Architectures: An Industrial Survey". In: (Apr. 2018), pp. 29–2909.

[10] Jonas Fritzsch et al. "From Monolith to Microservices: A Classification of Refactoring Approaches". In: *CoRR* abs/1807.10059 (2018). arXiv: 1807.10059. URL: http://arxiv.org/abs/1807.10059.

[11] Michael Gysel et al. "Service Cutter: A Systematic Approach to Service Decomposition". In: (2016). Ed. by Marco Aiello et al., pp. 185–200.

[12] S. Hassan, N. Ali, and R. Bahsoon. "Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity". In: (Apr. 2017), pp. 1–10.

[13] G. Kecskemeti, A. C. Marosi, and A. Kertesz. "The ENTICE approach to decompose monolithic services into microservices". In: (July 2016), pp. 591–596.

[14] S. Klock et al. "Workload-Based Clustering of Coherent Feature Sets in Microservice Architectures". In: (Apr. 2017), pp. 11–20.

[15] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. "Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems". In: *CoRR* abs/1605.03175 (2016). arXiv: 1605.03175. URL: http://arxiv.org/abs/1605.03175.

[16] J. Lin, L. C. Lin, and S. Huang. "Migrating web applications to clouds with microservice architectures". In: (May 2016), pp. 1–4.

[17] G. Mazlami, J. Cito, and P. Leitner. "Extraction of Microservices from Monolithic Software Architectures". In: (June 2017), pp. 524–531.

[18] Genc Mazlami. *Algorithmic Extraction of Microservices from Monolithic Code Bases*. Accessed on 20.01.2019. URL: https://www.merlin.uzh.ch/contributionDocument/download/10978.

[19] Frank Mittelbach. "How to influence the position of float environments like figure and table in LaTeX?" In: *TUGboat* 35 (2014), pp. 248–254. URL: https://www.latex-project.org/publications/tb111mitt-float.pdf.

[20] I. J. Munezero et al. "Partitioning Microservices: A Domain Engineering Approach". In: (May 2018), pp. 43–49.

[21] Chris Richardson. *Microservices: Decomposing Applications for Deployability and Scalability*. Accessed on 08.01.2019. May 2014. URL: https://www.infoq.com/articles/microservices-intro.

[22] Shmuel Tyszberowicz et al. "Identifying Microservices Using Functional Decomposition". In: (2018). Ed. by Xinyu Feng, Markus Müller-Olm, and Zijiang Yang, pp. 50–65.

A. Appendix

A.1. First Appendix Section

Figure A.1.: A figure

...