MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-based Environments

Martin Kleehaus, Ömer Uludağ, Patrick Schäfer and Florian Matthes

Chair for Informatics 19
Technische Universität München (TUM)
D-85748, Garching

Abstract. Microservices are an approach to distributed systems that promote the use of finely grained services with their own lifecycles. This architecture style encourages high decoupling, independent deployment, operation and maintenance. However, those benefits also leave a certain aftertaste, especially in continuous documentation of the overall architecture. It is fundamental to keep track of how microservices emerge over time. This knowledge is documented manually in Enterprise Architecture (EA) tools, which leads to an obsolete status. For that reason, we present a novel multi-layer microservice architecture recovery approach called *MICROLYZE* that recovers the infrastructure in realtime based on the EA model involving the business, application, hardware layer and the corresponding relationship between each other. It leverages existing monitoring tools and combines the run-time data with static built-time information. Hereby, *MICROLYZE* provide tool support for mapping the business activities with technical transactions in order to recover the correlation between the business and application layer.

1 Introduction

The popularity of microservice-based architectures [1] is increasing in many organizations, as this new software architecture style introduces high agility, resilience, scalability, maintainability, separation of concerns, and ease of deployment and operation [2]. Adrian Cockcroft at Netflix describes the architecture style as a "fine grained SOA" [3] that presents single applications as a suite of small services that run in their own processes and communicate with each other through lightweight HTTP-based mechanisms, like REST. Microservices are built around business capabilities and enclose specific business functions that are developed by independent teams [4]. The benefits emphasize the reason why over the last decade, leading software development and consultancy companies have found this software architecture to be an appealing approach that leads to more productive teams in general and to more successful software products. Companies such as Netflix [5], SoundCloud [6], Amazon [7] or LinkedIn [8] have adopted the microservices style in the cloud and pioneered the research in this area.

Even though microservices release the rigid structure of monolithic systems due to the independent deployment, this style also introduces a high level of complexity with regard to architecture monitoring, recovery and documentation [2]. In a recently published study on architecting microservices [9], it was investigated that monitoring solutions for microservice-based architectures have been addressed by many researchers and companies. However, it was also confirmed that limited research was conducted on topics of microservice recovery and documentation, although this is very important for understanding the emerging behavior of microservice-based architectures. For instance, microservices can dynamically change their status at run-time like the IP address or port for multiple reasons like autoscaling, failures, upgrades, or load balancing, among others [10]. Additional services can be introduced into the infrastructure or removed during upgrades or migration projects. In these scenarios, it is crucial to keep track on the current architecture orchestration and service dependencies.

In order to tackle this challenge, a lot of different monitoring approaches [11][12][13][14] and service discovery mechanisms [15][16] are applied that pull the status of services dynamically over the network. This simplifies many aspects of tracking the system, but lacks in connecting their obtained monitoring data in order to achieve an integrated and holistic view of the behavior and status of the whole Enterprise Architecture (EA) [17]. Moreover, the alignment of business needs to the IT infrastructure is acquiring increasing importance in microservice environments. Best practices propose the design of services based on their business domain in which they fulfill one specific business requirement [4]. Hence, we are convinced that the documentation of the relationship between the services and business domains including departments, teams and business processes has to be involved in an holistic architecture recovery, which is not yet covered in existing approaches.

According to the aforementioned considerations, we propose an architecture recovery approach called *MICROLYZE*, which combines static and runtime data in order to reconstruct IT infrastructures that are based on microservice architecture. The reconstruction includes all adjacent layers proposed by EA models, like hardware, application, business layer, and their relationships between each other. The tool recognizes changes in the infrastructure in real-time and uncovers all business activities that are performed by the users. The prototype obtains most of the required data via well-selected monitoring tools. The presented approach was evaluated in a microservice-based system called TUM Living Lab Connected Mobility (TUM LLCM)[1].

The rest of the paper is organized as follows: Section 2 describes the layers and components that are involved in the architecture recovery. Section 3 presents the technical details of the architecture recovery approach. In section 4 we evaluate our approach in a real case scenario, whereas section 5 provides a benchmark about the instrumentation overhead. Section 6 and 7 closes the paper with related work and our future efforts to improve the proposed approach.

[1] `http://tum-llcm.de/`

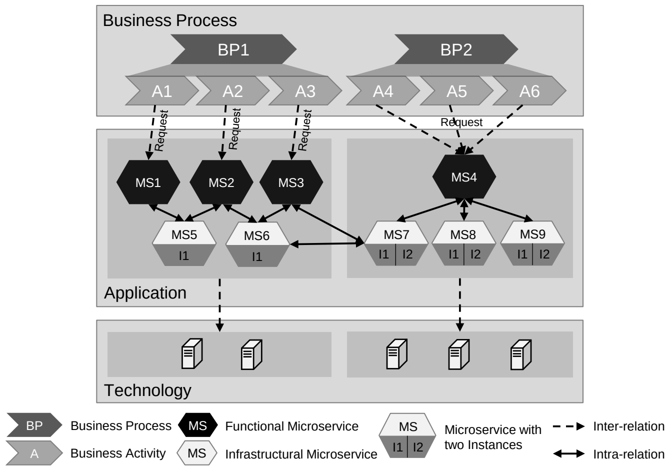


Fig. 1: Multi-layered microservice-based EA infrastructure

2 Architecture Model

MICROLYZE aligns the reconstruction model of microservice-based infrastructures to the EA model adopted by many of the EA frameworks that emerged in the last decades, like ArchiMate [18], Zachman [19], TOGAF [20], amongst others. These frameworks provide standards on how to model the EA and typically divide it into three abstraction layers: 1) the technology layer encompasses all technological-related aspects like hardware, network and other physical components and 2) the application layer defines software components running on the technology layer. We assign services and service instances to the application layer. 3) The business layer operates on top of the aforementioned layers and defines all business-related aspects like the business departments, business processes and business activities that are processed by the microservices. The first two layers are reconstructed completely automatically via analyzing monitoring data; the business layer requires additional domain knowledge and manual input in the first place that can only be provided by department staff members. The overall architecture and the relationship between each layer is depicted in figure 1 and described in more detail in the following.

2.1 Business Process

A business process is a collection of related activities that serve a particular goal for a user. In the context of distributed systems, each business transaction, or technically speaking a user request, contributes to the execution of a business process. Hence, several organizations and the according teams who develop the services could be involved in one business process. The reconstruction of a business process with the aid of event data is mostly associated with process discovery, which is the most used technique in process mining [21].

2.2 Business Activity

A business activity defines a business transaction and consists of a sequence of related events that together contribute to serve a user request. A request represents interactions with the system. Every business transaction is part of one or more business processes. Transactions can span multiple microservices that expose external interfaces to other services for intercommunication. Before process mining can be accomplished, each technical request that represents a business-related user activity must be named with a clear and understandable business description.

2.3 Service

A service is a logical unit that represents a particular microservice application. According to the service type classification discussed by Richards [22], services can be classified into functional and infrastructural services. Infrastructure services are not exposed to the outside world but are treated as private shared services only available internally to other services. Functional services are accessed externally and are generally not shared with any other service. They are responsible for processing business transactions but can forward the user request to other infrastructure services.

2.4 Service Instance

In contrast to services that form the logical unit of a microservice, the service instance represents the real object of this service. We introduce this separation as a logical service can *own* more service instances. This is often the case when load balancing is applied. However, a service is always represented by at least one instance. The instances are identified by the used IP address and port but always contain the same endpoint service name.

2.5 Hardware

The hardware layer covers all physical components of a microservice infrastructure. The service instances run on the hardware. We assume that hardware can be identified by its IP address.

2.6 Relationship between architecture components

The architecture model depicted in figure 1 constitutes two relationship types between the components: The intra-relation defines connections within a specific abstraction layer. For instance, as mentioned above, a business process is a sequence of business transactions and every transaction is defined by the organization that managed the service. In the application layer as an example, several services contribute to serve a user request. These services exchange data over their interface and, hence, feature an intra-relationship.

Besides relationships within abstraction layers, the inter-relation constitutes connections between two different layers. In order to obtain the holistic architecture of a microservice-based environment, inter-relationships uncover important information about the interaction between abstraction layers. All functional services are deployed to process specific business transactions that are defined by the executed user request. By tracing these requests, application-related metrics can be obtained, like the duration of a user request, the latency between two distributed services or the data volume that is transferred. Due to the inter-relationships, system administrators are able to quickly identify which business activities or business processes are affected by this failure. In addition, one can point out the team who is responsible for fixing the error.

3 Recovery Process

We regard architecture recovery as a never-ending process. It is a continuous monitoring of the service interaction within an IT infrastructure. Microservice architectures evolve over time [23]. That means, new services are added or removed, and newly implemented interfaces lead to a change in the information exchange and dependency structure. For that reason, in order to find a proper monitoring approach it is a prerequisite to receive data about the current health status of the application and insights about the communication behavior between microservices. It is important to keep track of architectural changes, especially when new releases cause failures or performance anomalies. In addition, the architecture recovery process should not only cover technology-related aspects but also business-related points, since each architecture refinement could add additional endpoints for interacting with the application. This leads, in turn, to an extension of the business process. Therefore, new business activities must be recognized automatically and added to the responsible business process.

Based on the aforementioned considerations, our architecture recovery process is composed of six phases as illustrated in figure 2. In the **first phase**, *MICROLYZE* automatically rebuilds the current microservice infrastructure that is registered in a *service discovery* tool like Eureka[2] or Consul[3]. These systems are integrated in microservice-based environments for storing the instance information of running microservices. Microservices frequently change their status due to reasons like updates, autoscaling or failures; the service discovery mechanisms are used to allow services to find each other in the network dynamically. By retrieving the information from the *service discovery* service, we are able to reveal the current status of each service instance. In case a change (unregis-

[2] `https://github.com/Netfix/eureka`
[3] `https://www.consul.io/`

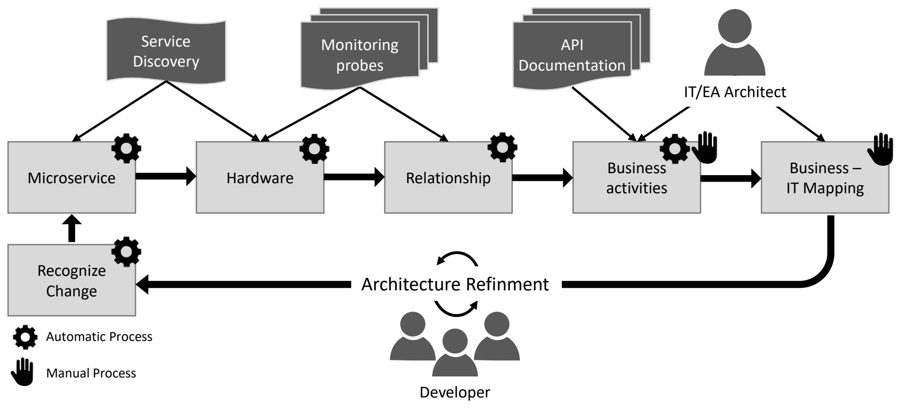


Fig. 2: Microservice architecture discovery process with all required phases (1-6) and included data sources

tered service, new service, updated service) is detected *MICROLYZE* alters the discovered architecture structure in order to indicate this particular change.

In the **second phase**, we also use the retrieved information for recovering hardware related aspects in order to establish the link between the microservices and the hardware on which the services are running. The *service discovery* service already provides useful data like, IP address and port but lacks in reporting detailed hardware information. For that reason, installing an additional monitoring agent on each hardware component that reveals hardware related information is required. The IP address is used to establish the link between the application and hardware layer.

Although service discovery mechanisms are often applied to discover the status of running services in run-time, they mask the real dependencies among microservices in the system. It remains unknown how the services communicate with each other as soon as user transactions come in. For that reason, it is necessary to install on each microservice a monitoring probe that supports the distributed tracing technology introduced by Google [24]. Distributed tracing tracks all executed HTTP requests in each service by injecting tracing information into the request headers. Hereby, it helps to gather timing data like process duration for each request in order to troubleshoot latency problems. Furthermore, additional infrastructure and software-specific data like endpoint name, class, method, HTTP request, etc. is collected and attached as annotations. Distributed tracing uncovers the dependencies between microservices by tracking the service calls via a correlation identifier. It is also capable of differentiating between concurrent or synchronous calls. As an implementation of the distributed tracing technology, we refer to the open-source monitoring solution zipkin[4] developed by Twitter. It supports the openTracing standard [5] and

[4] `https://github.com/openzipkin/zipkin`
[5] `http://opentracing.io/`

enjoys a huge community. Hence, by means of zipkin we are able to talk with any other APM tool as long as it also supports the openTracing standard. The zipkin probes stream the information via apache kafka to *MICROLYZE* and are stored in a cassandra database.

Moreover, we implemented an algorithm that determines how to classify each service on basis of the distributed tracing data. *Functional services*, for instance, have mostly no parent services that forward the request to their child nodes. The very first application is the client itself. Hence, the parent ID in the tracing data is mostly empty. However, there are situations in which this approach is not applicable. *Gateway services*, for example, provide a unified interface to the consumers of the system that proxies requests to multiple backing services. In order to recognize this type of service, we continuously analyze the incoming HTTP requests. If the very first accessed microservice is always the same in most requests *MICROLYZE* flags it as the *gateway service.* All child nodes after the gateway are flagged as *functional services* accordingly.

Last but not least, huge microservice infrastructures are load balanced to avoid single points of failures. Instances of a service always have the same name but distinguish itself in IP address and port. Therefore, the uniqueness of a service instance is defined by the service description in combination with the used IP address and the service port. In order to discover all instances that belong to a specific service, we aggregate them based on the service description.

During the **third phase**, all transactions executed by users are stored in a database. These requests provide information about the user behavior, which includes, first of all, what the user does, when and in which chronological order, but also uncovers the link between the business transactions and the microservices that are responsible for processing the request. However, most monitoring or CMDB solutions do not establish a mapping between the business activities and the technical transactions. It remains unclear which service is responsible for processing a specific business activity. For that reason, we extend *MICROLYZE* with a *business process modeller* that assists in creating such a mapping.

First of all, in the fourth phase each business activity that can be performed by the users and triggers a request in the backend are defined with a clear semantic description like *register, open shopping cart, purchase article*, etc. After this step is finished, we are able to create the mapping between the business activities and the technical requests extracted by zipkin. This covers **phase five**. In order to support this process, we enhanced the *business process modeller* with the regular expression language in order to describe technical requests. These expressions are mapped with the previously described business activities. All expressions are stored in the database and validate incoming requests. Hence, new incoming transactions that are not yet seen and might refer to a modelled business activity are already mapped by a regular expression.

The **sixth phase** is all about recognizing changes in the IT infrastructure. The user is notified as soon as the system recognizes changes in the architecture model that might occur after a component update. *MICROLYZE* frequently polls the *service discovery* service that indicates deleted or newly added services

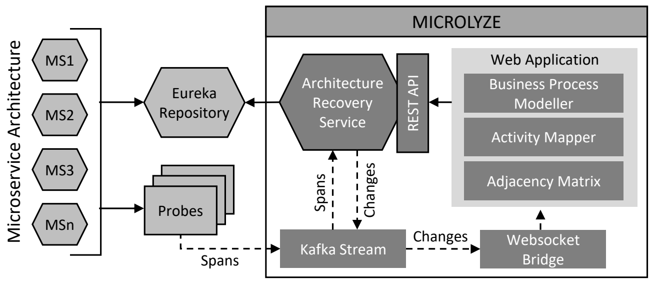


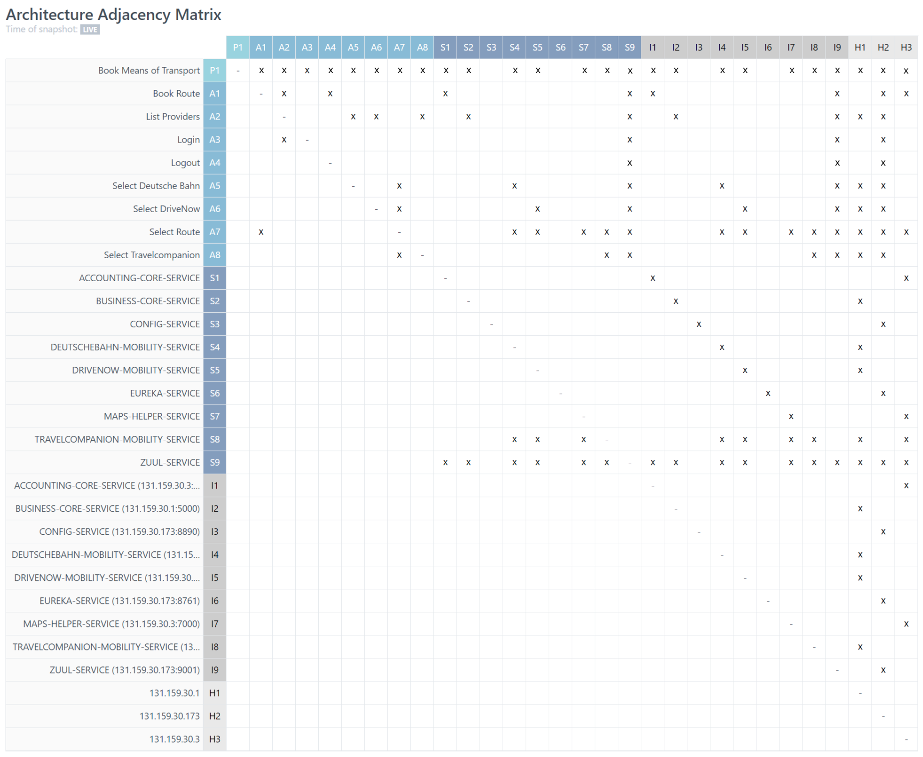
Fig. 3: Microlyze architecture components

that are not known yet. Unregistered services could indicate a service crash or a new update release. For that reason, they are not automatically removed from the database but only marked as such. This flag is removed as soon as the services are registered again. Unknown user requests that cannot be validated by a predefined regular expression are added to the list of unmapped URL endpoints. These endpoints have to be linked to a business activity afterwards. Changes in IP addresses and port might indicate an alteration in the underlying infrastructure, which leads to an automatic adaption of the architecture model.

The overall microservice architecture is depicted in figure 3. Apache kafka is used to stream all monitoring spans and architecture changes to the prototype in realtime. The web application includes the business process modeller, the activity to service mapper and the architecture visualizer that renders the obtained architecture model, making the discovered information, components, dependencies and mappings during the recovery process available for system administrators and enterprise or software architects. As microservice environments can grow to hundreds of services, we chose a representation that is scalable as well as capable of handling hundreds of dependencies and hiding specific aspects that are currently not needed to visualize. For that purpose, we applied the concept of the adjacency matrix and grouped the related columns and rows to the component dimensions that were described in section 2.

4 Evaluation

The described architecture discovery concept has been prototyped and applied to the TUM LLCM platform. This platform simplifies and accelerates the exchange regarding the development of digital mobility services. It allows service developers to build better mobility services by incorporating different partners who agree to share their mobility data. In this context, a service called *Travelcompanion* was developed that enables travelers to connect with a travel group who has the same destination. Hereby, travel costs can be shared between the group members. The *Travelcompanion* service consumes data from a BMW DriveNow and Deutsche Bahn (DB) service and provides a recommendation on how to plan the route with different transportation means used by other travel groups



Architecture Adjacency Matrix

Time of snapshot: LIVE

		P1	A1	A2	A3	A4	A5	A6	A7	A8	S1	S2	S3	S4	S5	S6	S7	S8	S9	I1	I2	I3	I4	I5	I6	I7	I8	I9	H1	H2	H3
Book Means of Transport	P1	-	x	x	x	x	x	x	x	x	x	x		x	x		x	x	x	x	x		x	x		x	x	x	x	x	x
Book Route	A1		-	x		x					x								x	x								x		x	x
List Providers	A2			-			x	x		x		x							x		x							x	x	x	
Login	A3			x	-														x									x		x	
Logout	A4					-													x									x		x	
Select Deutsche Bahn	A5						-		x					x					x				x					x	x	x	
Select DriveNow	A6							-	x						x				x					x				x	x	x	
Select Route	A7		x						-					x	x		x	x	x				x	x		x	x	x	x	x	x
Select Travelcompanion	A8								x	-								x	x								x	x	x	x	
ACCOUNTING-CORE-SERVICE	S1										-									x											x
BUSINESS-CORE-SERVICE	S2											-									x								x		
CONFIG-SERVICE	S3												-									x								x	
DEUTSCHEBAHN-MOBILITY-SERVICE	S4													-									x						x		
DRIVENOW-MOBILITY-SERVICE	S5														-									x					x		
EUREKA-SERVICE	S6															-									x					x	
MAPS-HELPER-SERVICE	S7																-									x					x
TRAVELCOMPANION-MOBILITY-SERVICE	S8													x	x		x	-					x	x		x	x		x		x
ZUUL-SERVICE	S9										x	x		x	x		x	x	-	x	x		x	x		x	x	x	x	x	x
ACCOUNTING-CORE-SERVICE (131.159.30.3:...	I1																			-											x
BUSINESS-CORE-SERVICE (131.159.30.1:5000)	I2																				-								x		
CONFIG-SERVICE (131.159.30.173:8890)	I3																					-								x	
DEUTSCHEBAHN-MOBILITY-SERVICE (131.15...	I4																						-						x		
DRIVENOW-MOBILITY-SERVICE (131.159.30....	I5																							-					x		
EUREKA-SERVICE (131.159.30.173:8761)	I6																								-					x	
MAPS-HELPER-SERVICE (131.159.30.3:7000)	I7																									-					x
TRAVELCOMPANION-MOBILITY-SERVICE (13...	I8																										-		x		
ZUUL-SERVICE (131.159.30.173:9001)	I9																											-		x	
131.159.30.1	H1																												-		
131.159.30.173	H2																													-	
131.159.30.3	H3																														-

Fig. 4: Architecture discovery result visualized in a grouped adjacency matrix

that still have free space left. By joining these groups, the travel costs can be minimized.

The platform is a microservice-based system implemented in Spring Boot. Each service runs in a docker container. The architecture incorporates infrastructural services like a configuration (config-service), the service discovery eureka (eureka-service) and a gateway service (zuul-service). Further services provide administration (business-core-service), geospatial (maps-helper-service) and accounting (accounting-core-service) functionality like booking and payment. DriveNow (drivenow-mobility-service), DB (deutschebahn-mobility-service) and the Travelcompanion (travelcompanion-mobility-service) service have their own data storage. Each service represents a eureka client and is registered in the eureka server. Each service except eureka is instrumented by zipkin probes. The microservice architecture is distributed on three virtual machines, each running on the same hardware.

After each service is started, the architecture discovery accesses eureka and consumes all registered services. As the matrix in figure 4 shows, *MICROLYZE* correctly recognizes 9 services (S1 – S9), 9 instances (I1 - I9) and 3 hardware components (H1 – H3). Each service is assigned to only one instance, which

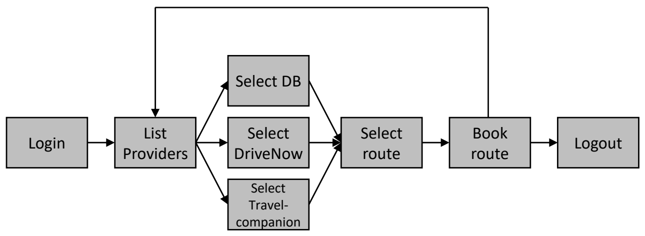


Fig. 5: Simulated user journey through the TUMLLCM platform. The business process describes the booking of a transportation mean

uncovers there is no load balancing in place. The services and service instances are correctly assigned to the hardware components.

In order to recover the relationships between the services, we produce traffic on the platform by using JMeter[6], which simulates user transactions based on the given REST API endpoints documented by Swagger. After each endpoint was called *MICROLYZE* is able to reconstruct the dependency structure among the microservice architecture. The adjacency matrix visualizes that service S8 (travelcompanion-mobility-service) consumes data from service S4, S5 and S7, which is intended. The communication between S8 $\rightarrow$ S4 and S8 $\rightarrow$ S5 is asynchronously which is also correct. In addition, it is detected that service S9 consumes data from every non-infrastructural service. Hence, *MICROLYZE* successfully recognizes S9 as the gateway service.

As soon as step three of the discovery process is finished and the system has collected enough transaction data, we start to enhance the technical transactions with a business semantic. The user click journey, which we want to simulate, is depicted in figure 5. Although this process is rather small it could grow to hundreds of user clicks that had to be modelled in *MICROLYZE*. In order to accelerate this step or even circumvent it, we could also apply a business process mining approach and import the XML model into *MICROLYZE*. In the scope of this work, we modeled the process manually. For simplicity, each business activity ranks among the same business process "Book Means of Transport". In total, we create one process containing eight activities (A1 – A8) as shown in the adjacency matrix. Afterwards, we proceed with the mapping of each business activity to a user transaction. Hereby, *MICROLYZE* supports this step by providing a framework for regular expressions that describe a particular transaction. For instance, the expression */[0-9]+/book$* is mapped to the activity "book route". In order to define the business instance for recovering the performed business process, we apply the user session as our business case.

The matrix in figure 4 visualizes the final result. The execution of business process P1 involves every service except the infrastructural services S3 and S6. Each activity is processed by service S9, which provides further proof that S9 represents a gateway service. Moreover, it is clearly visible that for performing

[6] `http://jmeter.apache.org/`

activity A7 also service S7 is executed besides the mobility services S4, S5 and S8. This is due to the communication dependency between service S8 and S7. Hence, service S7 is indirectly involved with performing A7.

By hovering over a dependency field within the matrix, an information box is activated and shows relation specific information. This information includes all relation annotations, the caller and callee name, and the owner of this request. The owner indicates from which business activity this request derives. This information is necessary in order to recognize which functional service is primarily responsible for the particular business activity. Otherwise, this information would get lost as soon as gateway services are applied. In case a service is associated with a specific pattern, *MICROLYZE* adds this information to the hover box as well.

5 Instrumentation Overhead

We investigate the extent of instrumentation overhead via a performance benchmark. We measure the time to complete each business activity for both the instrumented and unmodified version of the software. Each activity executes a transaction that is initially processed by the gateway service. The following <business activity>:<service>pairs are involved:

- List Providers: Business service
- Select Travelcompanion: Travelcompanion service
- Select Route: Map, DriveNow, DB service
- Book route: Accounting service

This time measurement is performed on the user side, and thus includes the communication overhead between the application and the client user. By measuring on the client side, we achieve an end-to-end processing benchmark. We repeated these measurements several times and calculated the average run-time and associated a 95% confidence interval. The results are presented in figure 6. We use JMeter to perform each request 5000 times involving database querying. As figure 6 illustrates, the difference in performance is very small. On average, the requests take 2ms longer to respond. Based on the observations presented above, we conclude that the impact of the instrumentation is negligible.

6 Related Work

O'Brien et al. [25] provide a state-of-the-art report on several architecture recovery techniques and tools. The presented approaches aim to reconstruct software components and their interrelations by analyzing source code and by applying data mining methods.

O'Brien and Stoermer [26] present the Architecture Reconstruction and MINing (ARMIN) tool for reconstructing deployment architectures from the source code and documentation. The proposed reconstruction process consists of two

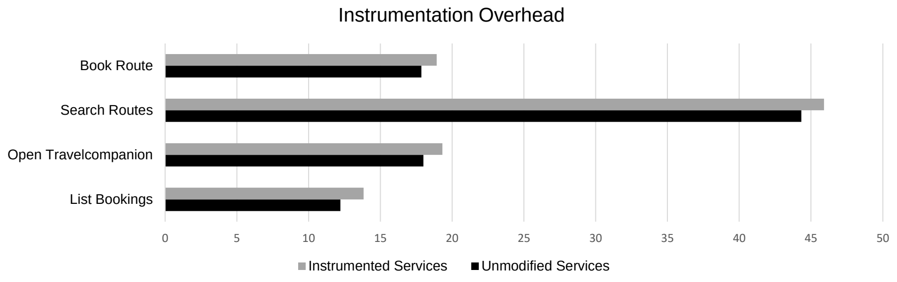


Fig. 6: Effect of instrumentation on the average time to complete – Average time to complete (in milliseconds) [95% confidence interval]

steps: extracting source information and architectural view composition. In the first step, a set of elements and relations is extracted from the system and loaded into ARMIN. In the second step, views of the system architecture are generated by abstracting the source information through aggregation and manipulation. ARMIN differs from our approach, as it only extracts static information of the system without considering dynamic information.

Cuadrado et al. [27] describe a case study of the evolution of an existing legacy system towards a SOA. The proposed process comprises architecture recovery, evolution planning, and evolution execution activities. Similar to our approach, the system architecture is recovered by extracting static and dynamic information from system documentation, source code, and the profiling tool. This approach, however, does not analyze communication dependencies between services, which is an outstanding feature of our prototype.

Van Hoorn et al. [28][29] propose the java-based and open-source Kieker framework for monitoring and analyzing the run-time behavior of concurrent or distributed software systems. Focusing on application-level monitoring, Kieker's application areas include performance evaluation, self-adaptation control, and software reverse engineering, to name a few. Similar to our approach, Kieker is also based on the distributed tracing for uncovering dependencies between microservices. Unlike us, Kieker does not process architectural changes in run-time. Furthermore, it does not cover dependencies between the business and application layer.

Haitzer and Zdun [30] present an approach for supporting semi-automated abstraction of architectural models supported through a domain-specific language. The proposed approach mainly focuses on architectural abstractions from the source code in a changing environment while still supporting traceability. Additionally, the approach allows software architects to compare different versions of the generated UML model with each other. It bridges the gap between the design and the implementation of a software system. In contrast, we propose an approach for recovering microservices in order to overcome the documentation and maintenance problem.

MicroART, an approach for recovering the architecture of microservice-based systems is presented in [10][31]. The approach is based on Model-Driven Engineering (MDE) principles and is composed of two main steps: recovering the deployment architecture of the system and semi-automatically refining the obtained system. The architecture recovery phase involves all activities necessary to extract an architecture model of the microservices, by finding static and dynamic information of microservices and their interrelations from the GitHub source code repository, Docker container engine, Vagrant platform, and TcpDump monitoring tool. The architecture refinement phase deals with semi-automatically refining the initial architecture model by the architect. MicroART considers the architecture model of the microservices without regarding the business layer, which is a main feature in our approach. Furthermore, MicroART does not differentiate types of microservices interrelations, like synchronous or asynchronous. Moreover, it does not rebuild the architecture model as soon as architectural changes emerge.

7 Conclusion

In this paper, we presented a novel approach to recovering the architecture from microservice-based systems. The discovery solution is based on a layered structure proposed by recommended EA frameworks. *MICROLYZE* is capable of recreating the dependencies between the business and application layer by providing a smart business-it mapper, and the hardware layer. The recovery process itself is subdivided into six phases. In the first three phases, we combine the monitoring data from a service discovery repository and a distributed tracing solution in order to reconstruct the microservice architecture, the dependencies between each service and the underlying hardware infrastructure. In the next phases, we describe each technical request as a business activity and map these activities to a business process. Hereby, *MICROLYZE* provides tool support via a business process modeller and a business activity mapper that integrates regular expression for describing the user requests recorded by the monitoring probes. The concept and the tool have been successfully applied to the TUM-LLCM microservice platform, which was able to recover the whole microservice infrastructure without any error.

The proposed approach works well if two implementations are presented in the regarded microservice architecture. First of all, each service has to be instrumented by an application performance monitoring solution that supports distributed tracing and complies with the open tracing standard. Furthermore, a *service discovery* service like Eureka or Consul has to be integrated. In case one of those tools is not installed, *MICROLYZE* will not become fully operational, which presents our most significant limitation. Furthermore, a system with high traffic volumes can produce large amounts of trace events. In most cases, it is sufficient to only collect some of the events to recover the architecture, which would improve the system performance.

In our future work, we plan on storing each architecture change in order to travel through the architecture evolution and on comparing the past and the current status. This feature will uncover important insights about the emerging behavior of microservice architectures, especially after new releases. It allows, for example, the analysis of the as-is and the intended to-be status. Furthermore, we want to develop a failure impact visualization based on the discovered architecture model, which covers the analysis of all EA abstraction layers.

References

1. Fowler, M., Lewis, J.: Microservices. Technical report, ThoughtWorks (2014)
2. Alshuqayran, N., Ali, N., Evans, R.: A systematic mapping study in microservice architecture. In: Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on, IEEE (2016) 44–51
3. Cockcroft, A.: Microservices workshop: Why, what, and how to get there (2016)
4. Evans: Domain-Driven Design: Tacking Complexity In the Heart of Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
5. Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., Edmonds, A.: An architecture for self-managing microservices. In: Proceedings of the 1st International Workshop on Automated Incident Management in Cloud. AIMC '15, New York, NY, USA, ACM (2015) 19–24
6. Calado, P.: Building products at soundcloudpart iii: Microservices in scala and finagle. Technical report, SoundCloud Limited (2014)
7. Kramer, S.: The biggest thing amazon got right: The platform. `https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/` (2011) Accessed: 2017-11-18.
8. Ihde, S.: From a monolith to microservices + rest: the evolution of linkedin's service architecture. `http://www.infoq.com/presentations/linkedin-microservices-urn` (2015) Accessed: 2017-11-18.
9. Francesco, P., Malavolta, I., Lago, P.: Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: International Conference on Software Architecture (ICSA). (2017)
10. Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., Di Salle, A.: Microart: A software architecture recovery tool for maintaining microservice-based systems. In: IEEE International Conference on Software Architecture (ICSA). (2017)
11. Aalst, W.M.P.v.d., Desel, J., Oberweis, A., eds.: Business Process Management, Models, Techniques, and Empirical Studies. Springer-Verlag, London, UK, UK (2000)
12. Rabl, T., Gómez-Villamor, S., Sadoghi, M., Muntés-Mulero, V., Jacobsen, H.A., Mankovskii, S.: Solving big data challenges for enterprise application performance management. CoRR **abs/1208.4167** (2012)
13. Josephsen, D.: Building a Monitoring Infrastructure with Nagios. Prentice Hall PTR, Upper Saddle River, NJ, USA (2007)
14. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. In: Proceedings of the 6th International Conference on Extending Database Technology: Advances in Database Technology. EDBT '98, London, UK, UK, Springer-Verlag (1998) 469–483

15. Netflix: Eureka. `https://github.com/Netflix/eureka` Accessed: 2017-10-18.
16. Montesi, F., Weber, J.: Circuit breakers, discovery, and API gateways in microservices. CoRR **abs/1609.05830** (2016)
17. Brückmann, T., Gruhn, V., Pfeiffer, M.: Towards real-time monitoring and controlling of enterprise architectures using business software control centers. In: Proceedings of the 5th European Conference on Software Architecture. ECSA'11, Berlin, Heidelberg, Springer-Verlag (2011) 287–294
18. Group, T.O.: ArchiMate 3.0 Specification. Van Haren Publishing (2016)
19. Zachman, J.A.: A framework for information systems architecture. IBM Systems Journal **26**(3) (1987) 276–292
20. Haren, V.: TOGAF Version 9.1. 10th edn. Van Haren Publishing (2011)
21. van der Aalst, W.M.P.: Extracting event data from databases to unleash process mining. In: BPM. Springer (2015) 105–128
22. Newman, S.: Building Microservices. 1st edn. O'Reilly Media, Inc. (2015)
23. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. CoRR **abs/1606.04036** (2016)
24. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc. (2010)
25. O'Brien, L., Stoermer, C., Verhoef, C.: Software architecture reconstruction: Practice needs and current approaches. Technical Report CMU/SEI-2002-TR-024, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2002)
26. O'Brien, L., Stoermer, C.: Architecture reconstruction case study. Technical Report CMU/SEI-2003-TN-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2003)
27. Cuadrado, F., García, B., Dueñas, J.C., Parada, H.A.: A case study on software evolution towards service-oriented architecture. In: Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on, IEEE (2008) 1399–1404
28. van Hoorn, A., Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Frey, S., Kieselhorst, D.: Continuous monitoring of software services: Design and application of the kieker framework. (2009)
29. van Hoorn, A., Waller, J., Hasselbring, W.: Kieker: A framework for application performance monitoring and dynamic software analysis. In: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering. ICPE '12, New York, NY, USA, ACM (2012) 247–248
30. Haitzer, T., Zdun, U.: Dsl-based support for semi-automated architectural component model abstraction throughout the software lifecycle. In: Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, ACM (2012) 61–70
31. Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., Di Salle, A.: Towards recovering the software architecture of microservice-based systems. In: Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on, IEEE (2017) 46–53