Towards a Continuous Feedback Loop for Service-oriented Environments

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

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

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

***Abstract*—Agile software engineering practices aim at unifying software development (Dev) and software operation (Ops) in order to quickly release new software and gather feedback on new features in the latest version. The feedback loop gets closed mostly after new releases are deployed to production. During development, engineers do not receive feedback whether their changes are still aligned with the requirements and the formalized concept. This increases the risk of removing implemented code and decreases productivity. In this paper, we propose a tool for closing the feedback loop after each continuous deployment stage i.e. development, test and production. The continuous feedback is provided via a dependency model that represents the current software architecture on early stages. Hereby, each deployment phase and final release are compared against each other in order to uncover inconsistencies in regard to the predefined requirements.**

***Index Terms*—devops, agile, feedback loop, continuous delivery, continuous delivery pipeline, continuous deployment, microservice, monitoring, architecture discovery, service discovery, distributed tracing, application performance monitoring**

I. Introduction

Continuous delivery [1] is considered as an emerging paradigm that aims to significantly shorten software release cycles by bridging existing gaps between developers, operators, and other stakeholders involved in the delivery process. A prerequisite to establish continuous delivery, is a high degree of technical automation. This is typically achieved by implementing an automated continuous delivery pipeline (also known as deployment pipeline) [2], covering all required steps such as retrieving code from a repository, building packaged binaries, running tests, and deployment to production. Such an automated and integrated delivery pipeline improves software quality, e.g., by avoiding the deployment of changes that did not pass all tests.

In most concepts the continuous feedback loop is used to learn and respond to customer needs. It is mostly triggered after a new release is deployed to production. However, observations [2] indicate that engineers performing frequent deployments require more often and faster feedback on every continuous deployment stage, like after the changes are tested in the test or preproduction environment and deployed into production. This would provide engineers valuable feedback regarding their development is still aligned with the requirements and the architecture concept.

For instance, a big microservice based environment involves many developers that change frequently the code of each service. It must be ensured that after each deployment from development to production environment no additional differences emerge into architecture except the expected ones. This risk is valid as not everything can be deployed in an automated manner. Many steps are performed manually which might introduce failures. In addition, the product owners (PO) are also interested in getting notifications about significant differences between the several environments as they are responsible to approve the next release. Differences in version of services, communication channels, hardware utilization, performance KPIs amongst others might indicate, for instance, a broken delivery pipeline.

Furthermore, developers might not understand the requirements collected by the product owner or the product vision described by the customer. Hence, they deploy a wrong solution which is only recognized after the first product review, or during the planning of the next sprint. This has to be identified very soon by the PO.

In this work, we introduce an automated architecture discovery tool called *MICROLYZE*, which provides continuous feedback on each deployment stage by exposing the differences between deployed architectures. Hereby, it can be quickly reviewed if a new release was build correctly against the requirements or not. The concept is illustrated in figure 1. *MICROLYZE* consumes monitoring data from every deployment stage and recreates the runtime architecture model of the whole infrastructure in each environment. The recovered runtime models are compared against each other in order to uncover inconsistencies and validated against the planned architecture model. This comparisons unveil 1) differences that might exist between the deployment environments and 2) provides information about the right services with the right changes are deployed, 3) the developers understood the technical concept and 4) the new release lead to the expected improvements. Furthermore, the architecture is continuously documented and made available for software engineers, architects and operators. In particular, the operator is supported in the investigation of problems by the help of the documentation of the dynamic microservice infrastructure. For instance, the operator detects missing or broken communications between related services, which might indicate the root cause of a failure. The following

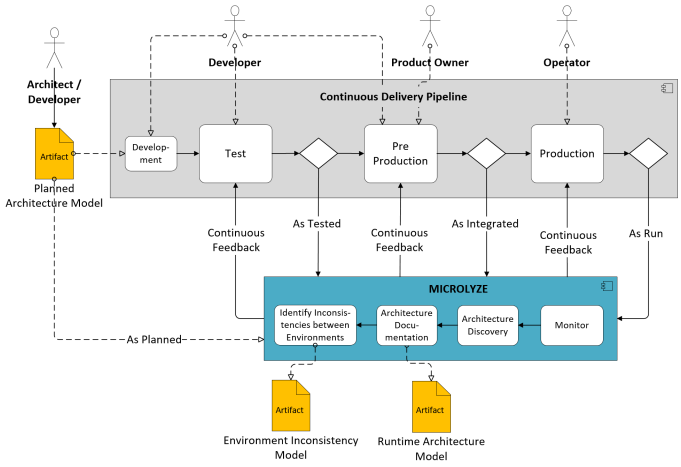

Fig. 1: Process of the continuous feedback pipeline. After each deployment stage the architecture is reconstructed based on monitoring data and compared against the *environments*.

requirements were defined for the prototype:

- R1: The prototype must be able to recover all architecture components which are relevant from a software architecture perspective. This includes clients (R1-1), services (R1-2), databases (R1-3) and the hardware (R1-4)
- R2: The information which services are load balanced
- R3: The relationship between each component
- R4: How does the services communicate with each other, i.e. synchronously or asynchronously.
- R5: The classes and methods which are called via the exposed APIs.
- R6: The libraries which are included in each service
- R7: The recovered architecture can be exported in a standard format
- R8: A comparison of the recovered architecture between each deployment environment in order to uncover inconsistencies
- R9: The capability to create KPIs in order to measure the performance in each environment

The remainder of the paper is organized as follows. Section II recaps shortly the characteristics of modern microservice infrastructures. Section III describes the layers and components that are involved in the architecture recovery. Section IV presents the technical details of the architecture recovery approach. Section V describes how a comparison between the runtime models can be achieved. Section VI and VII evaluate the prototype and Section VII and IX close the paper with related work, the limitations and future research.

II. Microservices

The popularity of microservice-based architectures is increasing in many organizations, as this new software architecture style has many desirable properties from a DevOps point of view. It introduces high agility, resilience, scalability, maintainability, separation of concerns, and ease of deployment and operation [3]. An important goal of DevOps is to place new features in the hands of users faster. This is a consequence of the greater amount of modularization that microservices provide [4]. The benefits emphasize the reason why over the last decade, leading software development and consultancy companies (see [5] [6] [7]) have found this software architecture to be an appealing approach that leads to more productive teams in general and to more successful software products.

Even though microservices release the rigid structure of monolithic systems due to the independent deployment, this style also introduces a high level of complexity. It is more and more demanded to deploy new code quickly, while guaranteeing reliability and effectiveness. As microservices have more integration points between systems, they suffer from a higher possibility of failure than monolithic systems. It is possible to partially mitigate this problem by making an effort to monitor the services and take appropriate action to malfunctions. Hence, many different monitoring approaches [8] [9] [10] and service discovery mechanisms [11] [12] are developed 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 microservice infrastructure [13]. In addition, monitoring systems are mostly used in the production environment in order to react quickly to production failures. It is hardly considered to apply monitoring already in the development and testing phase for documentation and in particular architecture comparison in order to receive valuable feedback through the delivery pipeline.

III. Architecture Model

MICROLYZE reconstructs the microservice infrastructure based on a multi-layer architecture model. This model divides the infrastructure into three layers as proposed by many Enterprise Architecture frameworks that emerged in the last decades, like ArchiMate [14], Zachman [15], TOGAF [16], amongst others. 1) the technology layer encompasses all technological-related aspects and the environment in which the services are running, like hardware. 2) the application layer defines the particular software components that are deployed into production. Services, services instances and databases are assigned to the application layer. 3) the client layer operates on top of the aforementioned layers and defines the user interface and all related activities that a user performs and are processed by the microservices.

The meta-model of the concept is illustrated in figure 2. It is composed of 13 metaclasses whereas *Environment* is the root node of the system being designed. The *Environment* determines the specific phase (test, preproduction, production) in the continuous delivery pipeline from which the microservice architecture is reconstructed. *Component* represents the parent class from which each component subclass inherits. In the following we describe the several components in more detail.

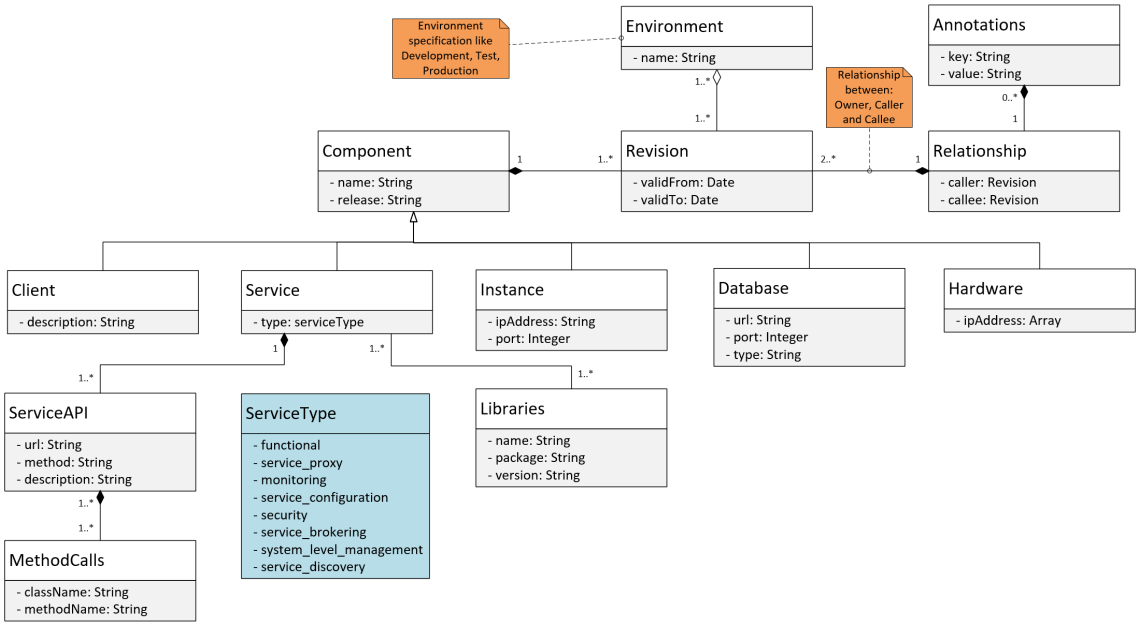

Fig. 2: Class diagram of the architecture discovery concept

A. Client

A client represents a specific front-end application. A client consists of several activities that define user events which are always processed by a sequence of related microservices in the backend.

B. Service

A service is a logical unit that represents a particular microservice application. According to the service type classification discussed by Richards [4], services can be classified in either functional or infrastructural services. Infrastructure services do not handle business logic and perform infrastructure related tasks like gateway, load balancing, service discovery, API management etc. Their interface are not exposed and are treated as private shared services only available internally to other services. Functional services are responsible for processing the business logic and their interface are exposed to external applications. They can forward the user requests to other functional services in order to receive business relevant information. The set of possible values of the ServiceType enumeration allows to define a service as functional by assigning to the service the value functional, or implicitly define the service as infrastructural by assigning to it one of the remaining possible infrastructural value, which were extracted according to the classification provided in [17].

We align the service layer to the reference architecture proposed by Yu et al. [18]. Microservices normally comprises three layers as a typical 3-tiered application, which consists of an interface layer, business logic layer and a data persistence layer:

- The interface layer contains all exposed API endpoints that can be accessed by client application or by other microservices and is represented in the metamodel as *ServiceAPI*. In order to provide all APIs of a specific service, a common interface is required to tell exactly what each service is supposed to do. For that reason, the interface are mostly documented via an API repository like Swagger[1] that is shared across the enterprise.
- The business logic implements the business functions, computations and service integration logic. Many monitoring agents are able to analyze transactions in an aspect-oriented way and extract the information which classes and methods are called for processing a specific request [19] [20]. We assign the *MethodCalls* to the exposed service APIs, as they are only executed as soon as the particular API endpoint is triggered.
- The persistence layer comprises all databases that are accessed by the services. We define *Database* as an own component as several services may share the same database. *Database* is defined with the attributes URL, port and database type like postgres, mysql, etc.

In addition to the 3-tiered architecture, we analyze the required libraries in every service and store the information in the *Libraries* class. The attributes describe the name of the used package, its version and its author.

C. 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 that 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. Instances are identified

[1] https://swagger.io/

by the used IP address and port but always contain the same endpoint service name.

D. Component Revision

In order to provide developers and operators the capability to compare different stages of the developed microservice architecture within the delivery pipeline and between different releases, we introduce the concept of component revisions. A *Revision* is always assigned to a *Component* in the architecture and describes it for a particular validity period as well as in which *Environment* it was discovered. If the system recognizes a change in the component like the service was removed, a new service was introduced, the list of exposed interfaces was extended, the used libraries were changed, the database was replaced, etc. the current revision gets invalid and a new revision is created for the specific component. Hence, there exists only one valid revision for each component through its lifecycle but may contain several invalid revisions.

In addition, we establish the relationship between *Components* through the *Revision* and not directly in the *Component* class. The reason is that the interaction or mapping between the architecture components could also be changed after a new deployment. For instance, the database type was replaced, or the data exchange between two services is removed. If a new revision for a specific component is created all related relationship revisions are becoming invalid. This is necessary as we cannot be sure whether a relationship to another component was changed with this release as well. As a consequence, we continuously rebuild the outgoing relationships from a component as soon as it experience a change.

E. Relationship between architecture components

The architecture model constitutes two relationship types between the components: The intra-relation defines connections within a specific abstraction layer. For instance, several services in the application layer contribute to serve a user request. The proxy service forwards the request to the responsible functional service that may communicate with other services in order to process this request. Microservices communicate directly via RPC or architecture patterns that propose asynchronous communication like message broker (see Apache Kafka[2] or MQTT[3]). For that reason, we introduce the *Annotations* class that describes the relationship in more detail. It features a key-value structure so that several annotations can be assigned to one relationship. One example is the definition of synchronous or asynchronous communication.

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, like the docker container in which particular services are deployed or which services are responsible to process requests from clients. Especially the last example is very important for bug fixing purposes. In case a specific button in the client application is not working anymore and triggers an error respond, the developers must understand which services, API endpoints and methods are called after clicking this button. Due to the inter-relationships, developers are able to quickly identify which user activities are affected by certain back-end failures.

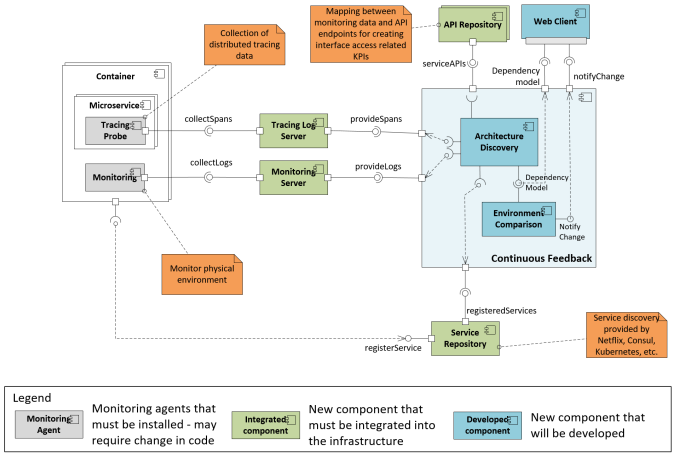

Fig. 3: Component diagram of the architecture discovery concept. The tool consists of two components that consumes data from five different sources.

IV. Architecture Discovery Process

Microservice architectures evolve over time [21]. That means, new services are added or removed, and newly implemented interfaces lead to a change in the information exchange and dependency structure. Hence, it is important to keep track of architectural changes, especially when new releases cause failures or performance anomalies.

Based on the aforementioned considerations, our architecture recovery concept consumes data from four different sources and consists of three specific components that were developed from scratch as illustrated in figure 3. The addressed data sources are existing in most modern microservice infrastructures as stated by Yu et al. [18]. The *Architecture Discovery* component consumes the monitoring data and recovers the architecture in the particular environment. The result of the discovery process is incorporated into a dependency model that is exposed to the *Architecture Comparison* component which recognizes the delta between each environments or releases. Both components provide interfaces for a *Web Client* that finally visualizes the architecture dependency model.

A. Discover services

Initially, *MICROLYZE* automatically rebuilds the current microservice infrastructure that is registered in a *service discovery* tool like Eureka[4], or Kubernetes[5]. Service repositories

[2]https://kafka.apache.org/
[3]http://mqtt.org/
[4]https://github.com/Netflix/eureka
[5]https://kubernetes.io/

are generally 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 (unregistered service, new service, updated service) is detected, *MICROLYZE* alters the dependency model by creating a new revision for this service, as it described in section III-D.

B. Discover libraries

Some build systems, such as Maven[6], provide configuration files (Maven POM) that contain descriptions of which build dependencies are needed. In case they are not already present on the build server they are fetched automatically by the build system. By accessing these configuration files *MICROLYZE* is capable to extract all libraries that are needed by the individual microservice. The information are stored in the *Library* class. Moreover, in the configuration files are also the release number defined which is extracted as well.

C. Discover service relationships and annotations

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. The communication behavior is in particular important in performance testing as every service contributes individually to the overall response time. For that reason, it is necessary to install on each microservice a monitoring probe, that monitors the response time of a server while the performance testing software generates synthetic requests to each REST endpoints. A popular technology for extracting performance measures is the distributed tracing technology proposed by Google [19]. The technology was adapted by many projects and companies like openTracing.io[7], zipkin[8] developed by Twitter, jaeger[9] created by Uber, or commercial products like Dynatrace, AppDynamics and Instana. 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. In addition, distributed tracing uncovers the dependencies between microservices by tracking the service calls via a correlation identifier.

MICROLYZE frequently polls the distributed tracing data and analyzes the dependency structure. If a new communication relationship is recognized the dependency model is updated accordingly. All information are assigned to the particular classes in the UML diagram (see figure 2): The communication between all components are specified in the *Relationship* class. The communication details are stored in the *Annotation* class. Database specific information like url, port and type are stored in the *Database* class.

D. Determine service classification

Furthermore, *MICROLYZE* classifies each service based on the distributed tracing data. *Functional services*, for instance, have mostly no parent services that forward requests to child nodes [4]. The parent node 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. *Service proxy*, 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, the incoming HTTP requests are continuously analyzed. If the very first accessed microservice is always the same in most requests *MICROLYZE* flags it as the *Service proxy*. All child nodes after the gateway are flagged as *functional services* accordingly. Another specialty are *configuration services* that address the cross-cutting concern how to provide configuration data that expose the information how to connect to external services. One characteristic of these services is that they are used by other services only once in the very beginning of their lifetime. Hence, these services are flagged as *configuration services* as soon as the aforementioned considerations can be applied. *Service discovery* services are automatically recognized as they already serve as data source. Further infrastructure services that are listed in [17] can be defined via manual input.

E. Discover service API endpoints

REST APIs [22] are mostly documented by tool support like swagger and others. API documentation is a technical content deliverable, containing instructions about how to effectively use and integrate with an API. *MICROLYZE* consumes the API documentation and stores it in the ServiceAPI class that is assigned to the particular microservice. However, which specific API is accessed by the client or other services is not always visible in the monitoring data, as the REST API specification parameterized the interface URLs. For instance, the interface *GET /object/{objectId}* provide object data filtered by id. The monitoring data stores the request in the format *GET /object/123*. Hence, it is necessary to establish a mapping between the API specification and the monitored runtime data in order to automatically assign the request to the specification. This problem can be solved by translating the documented API URL into a regular expression which is applied, in turn, on the monitored request. The regular expressions are stored in the database and serves as a mapping table between RPCs and API specification. The process is illustrated in figure 4.

F. Discover service instances

Huge microservice infrastructures are load balanced to avoid single points of failures. Which services are load balanced can be unveiled in the monitoring data: Instances of the same service always have the same application name but distinguish itself in IP address and port. Therefore, we aggregate the data based on service description, IP address and service port in order to unveil the number of instances of a particular service.

[6]https://maven.apache.org/
[7]http://opentracing.io/
[8]https://github.com/openzipkin/zipkin
[9]https://github.com/jaegertracing/jaeger

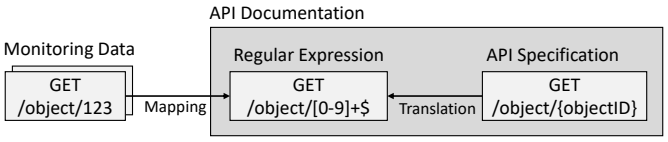

Fig. 4: Mapping monitoring data to documented API specification via regular expressions.

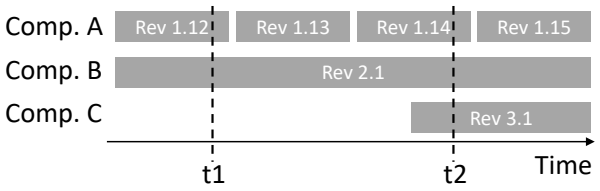

Fig. 5: Sequence of created revisions for component A, B and C and the retrieved revisions in t1 and t2

V. Revision Comparison

In order to track the emerging behavior of microservice architectures and to compare different environment against each other, we introduce a revision concept. The implementation of the revision concept is based on the assumption that every change in the infrastructure has an impact on the components in the dependency model. If a new component or a new relationship is identified a revision for this component is created simultaneously. The *validFrom* in the *Revision* class indicates the timestamp when the component was discovered, changed or removed. Hence, new revisions are always created after each deployment in the delivery pipeline and the timestamp determines when the deployment was triggered.

In order to retrieve the microservice architecture for a specific environment and time period one must simple select all revisions that were valid in a particular snapshot and the relationships are automatically assigned to the retrieved components. With this approach developers and operators are capable to compare the different architecture revisions in order to uncover unforeseen changes between releases and environments. In addition, it is possible to evaluate how the architecture emerged over time and what impact specific changes have on the performance.

Figure 5 presents the chronological sequence of the validity of a revision and how the architecture model can be retrieved for a selected time. Time t1 leads to the selection of revision 1.12 and 2.1. Component C does not yet exists at this time. T2 contains the revisions 1.14, 2.1 and 3.1 for the components A, B and C. That means, the architecture model has been significantly changed after t1, as component A was modified two times and component C was introduced into the architecture.

VI. Evaluation

The described architecture discovery concept has been prototyped and applied to a microservice-based product from a German company that is active in the production industry. The product is developed based on a continuous delivery strategy with the environments test, preproduction and production. Modified services are automatically deployed to the test environment as soon as the code is committed into the repository. The deployment from preproduction to production is triggered manually after it was approved by the Product Owner.

Each microservice runs in a docker container and is developed with Spring Boot. The architecture incorporates six functional services that expose the business logic and three infrastructural services which cover central configuration, service registry and a proxy service. All services are distributed on three virtual machines, each running on the same hardware.

For monitoring each infrastructure environment the application performance monitoring tool Zipkin is attached to each microservice. In the system under observation (SUP) we do not have access to hardware monitoring probes. The monitoring tools provide time series data about application performance, service communication and database communication. Zipkin complies with the openTracing standard, for that reason any other monitoring tool that follows the same standard like Instana can also be used with only little modifications. The monitoring agents are available for the languages, Go, JavaScript, Java, Python, Ruby, PHP, Objective-C, C++, C and corresponding frameworks that are provided by the community like Spring Boot, Django, Flask, etc. Zipkin provides time series data about application performance in form of ”spans”. Additional information about request processing is attached as ”annotations” to each span. Annotations contain the following attributes: service name, service IP and port, called class and requested method, HTTP related information like path, URL and status code as well as further information like asynchronous calls. All those information specify in more detail how the microservice process the request and which additional services are needed for data exchange. Requests to databases are stored as separated spans that contains SQL specific information. The time series data can be consumed via the following REST APIs: *GET /spans; GET /traces; GET /services; GET /dependencies*

MICROLYZE frequently accesses the *Eureka* service API in order to receive all registered services. In order to recover the relationships between the components, we produce traffic on the development and test environment by using JMeter[10], which simulates user transactions based on all 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 result for the test environment and for the current snaphshot, i.e most current revisions is illustrated in the adjecency matrix in figure 6. *MICROLYZE* correctly recognizes 9 services (S1 – S9), 9 instances (I1 - I9) and 3 hardware components (H1 – H3). Hence, in the test environment there is no load balancing in place. By hovering over a dependency field within the matrix, an information box is displayed and shows relation specific

[10]http://jmeter.apache.org/

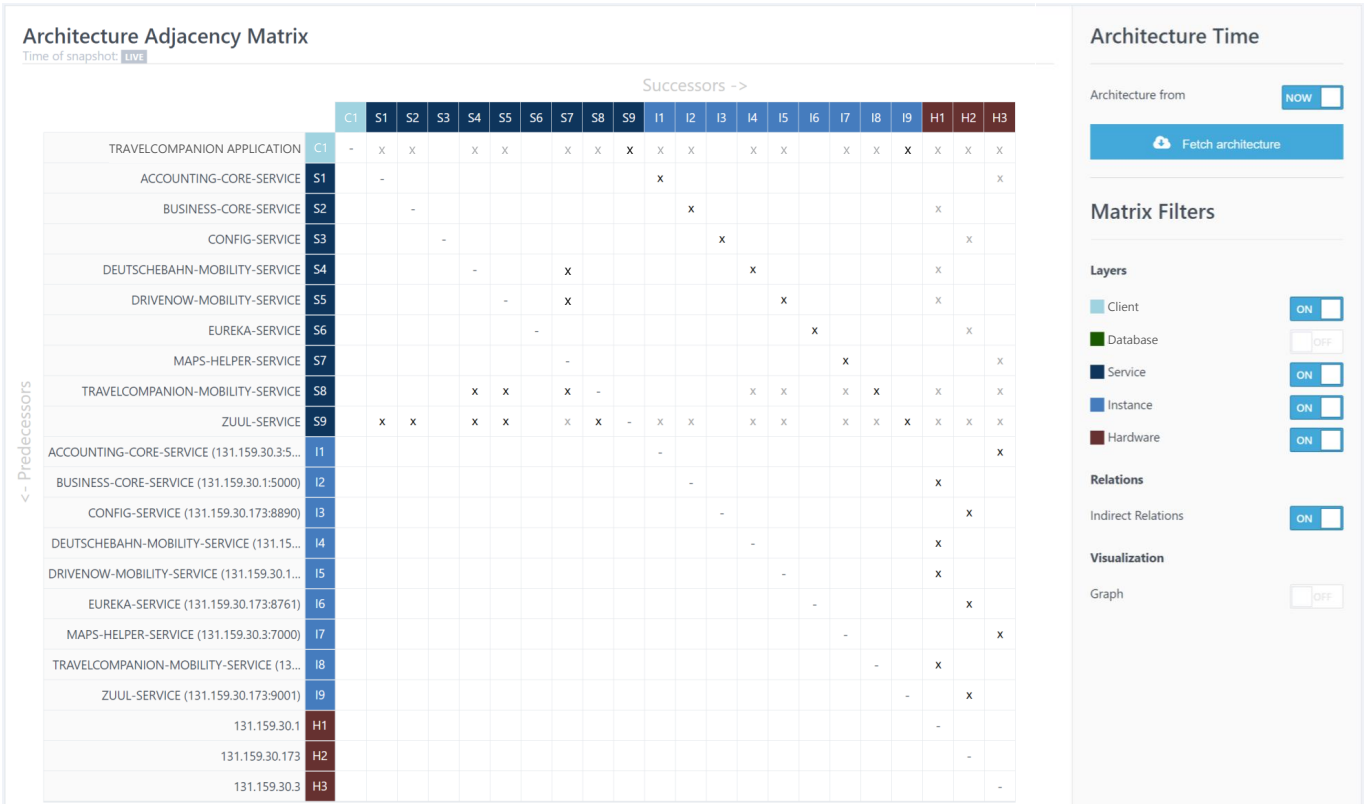

Fig. 6: Architecture discovery result visualized in a grouped adjacency matrix

information. For instance, the communication between S8 → S4 and S8 → S5 is asynchronously which is correctly reported. In addition, it is detected that service S9 consumes data from every functional service. Hence, S9 is successfully recognized as a proxy service.

The adjecency matrix visualizes the relationship types with a black $\boldsymbol{x}$ and a grey $\boldsymbol{x}$. The black one represents direct relationships between two components, like S8 communicates directly with S4 and is deployed on hardware H1. Indirect relationships unveils the whole communication path, e.g. C1 calls an API of S8 via the proxy service S9. We designed the adjecency matrix to be scalable for a large amount of microservices. On the right side of the application the user is able to disable specific architecture components. This reduces the amount of visible component relationships on the screen. However, as many companies, like Netflix, Spotify, etc. are already managing thousands of microservices which are all load balanced this visualization might still reach its limit fast. For that reason, we plan in future work to add an additional table view for listing all the components and its parent and child relationships. This table must be sortable and searchable.

We recognized one limitation regarding the recovery of the hardware layer. With the utilized monitoring solutions it is not possible to differentiate between physical and virtual hardware. We only receive the ip addresses but no further information about virtualization, containerization or operating system. The incorporation of further monitoring vendors like Dynatrace may unveil this information which could be part of future work.

MICROLYZE continuously creates new revision for every component as soon as modifications are detected. The whole recovered architecture can be exported in JSON format in order to compare different environments or deployments as described in section 5. Hereby, the user must select the required snapshot date which retrieves all components who fulfills the condition $validFrom < snapshot < validTo$. In order to receive the current architecture version all revision are selected whose *validTo* dates are null. An exception of the exported JSON file is illustrated in figure 7. We selected the current date for the preproduction and production environment. It is clearly visible that an old version of the *MAPS_HELPER_SERVICE* was accidentally deployed into the preproduction environment at this time. This service in version 1.12 still communicates synchronously with the service id 000712. Hence, the service release must be rolled back.

In conclusion, table I highlights which requirements stated in the introduction section was fulfilled by *MICROLYZE* and which limitations must be addressed in future work.

VII. Instrumentation Overhead

We investigate the extent of instrumentation overhead via a performance benchmark. We measure the time to complete selected requests for both the instrumented

Fig. 7: Excerpt of an export of the microservice architecture from preproduction and production environment. A conflict is recognized in the deployed service version.

TABLE I: Result of the requirement fulfillment

Req.	Result	Comment
R1-1	●	All clients that communicate with the SUP are discovered as long as they are monitored.
R1-2	●	All services are unveiled as long as they are registered in the service discovery application.
R1-3	◐	All databases are discovered. The type of the database is not recognized due to limited functionality of the provided monitoring tools. Missing information can be added via manual input.
R1-4	◔	Hardware number and IP address space is discovered. Other information like OS, virtualizations, containerization remain hidden. Access to further monitoring tools like Nagios, or Dynatrace are required.
R2	●	Load balancing is discovered and exposed via service instances.
R3	●	Relationships between components are discovered
R4	●	Communication types like synchronicity or asynchronicity are discovered
R5	●	Classes and methods that are executed by a specfic API call are unveiled
R6	●	Used libraries in each service are discovered by accessing configuration files (POM, gradle, etc.)
R7	●	The recovered architecture is exported in the JSON format
R8	●	A comparison between the recovered architectures is possible by comparing the JSON export
R9	●	As the architecture is extracted from runtime data, R9 is completely fullfiled

and unmodified version of the software. Each request executes a transaction that is initially processed by the gateway service (ZUUL-SERVICE) and forwarded to the responsible services like BUSINESS-CORE-SERVICE, TRAVELCOMPANION-MOBILITY-SERVICE, DRIVENOW-MOBILITY-SERVICE, DEUTSCHEBAHN-MOBILITY-SERVICE and ACCOUNTING-CORE-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 8. We use JMeter to perform each request 5000 times involving database querying. As figure 8 illustrates, the difference in performance is very small. On average, the requests take 1,03ms longer to respond. Based on the observations presented above, we conclude that the impact of the instrumentation is negligible.

VIII. Related Work

O'Brien et al. [23] provide a state-of-the-art report on several architecture recovery techniques and tools. The presented

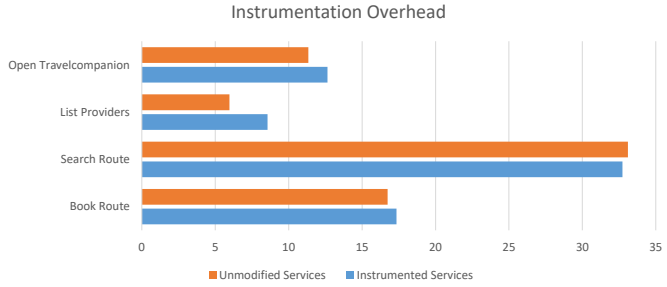

Fig. 8: Effect of instrumentation on the average time to complete – Average time to complete (in milliseconds) [95% confidence interval]

approaches aim to reconstruct software components and their interrelations by analyzing source code and by applying data mining methods.

O'Brien and Stoermer [24] 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 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. [25] 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 a main feature in our prototype.

Van Hoorn et al. [20] [26] 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 highlight architectural changes between two releases. Furthermore, it does not cover dependencies between the client and application layer.

MicroART, an approach for recovering the architecture of microservice-based systems is presented in [27] [28]. 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. In contrast to our concept, MicroART is not used to provide developers continuous feedback from each deployment stage.

IX. Limitations and Conclusion

In this paper, we introduce a tool for automated architecture discovery called *MICROLYZE*. It can be used to provide continuous feedback to developers, product owners and operators from each deployment stage in a continuous delivery pipeline. The tool recreates the dependency model of a microservice infrastructure based on a layered structure proposed by recommended EA frameworks. The tool consumes data from four different sources and discovers architecture relevant components. Based on the reconstructed dependency model the DevOps Teams are capable to recognize inconsistencies between the deployment stages and different releases. Hereby, the revision concept helps to historicize and emphasize changes made in the architecture. Finally, this information can be used to validate whether the developers are still on the right track and understand the technical concept.

The proposed approach works well if two implementations are presented in the regarded microservice architecture: First of all, for the whole infrastructure an appropriate monitoring architecture has to be present. Each microservice has to be instrumented by an monitoring probe that records intra-communications. Furthermore, a *service discovery* service has to be integrated. The monitoring architecture needs to be setup already in the development phase, or at least in the test environment. In case one of those prerequisites are not present, *MICROLYZE* will not become fully operational, which outlines our most significant limitation.

X. Acknowledgments

This work is part of TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bayerisches Staatsministerium für Wirtschaft und Medien, Energie und Technologie (StMWi).

References

[1] Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Signature Series (Fowler). Pearson Education (2010)

[2] Shahin, M., Babar, M.A., Zhu, L.: Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. IEEE Access **5** (2017) 3909–3943

[3] 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

[4] Newman, S.: Building Microservices. 1st edn. O'Reilly Media, Inc. (2015)

[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] 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)

[9] Josephsen, D.: Building a Monitoring Infrastructure with Nagios. Prentice Hall PTR, Upper Saddle River, NJ, USA (2007)

[10] 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

[11] Netflix: Eureka. https://github.com/Netflix/eureka Accessed: 2017-10-18.

[12] Montesi, F., Weber, J.: Circuit breakers, discovery, and API gateways in microservices. CoRR **abs/1609.05830** (2016)

[13] 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

[14] Group, T.O.: ArchiMate 3.0 Specification. Van Haren Publishing (2016)

[15] Zachman, J.A.: A framework for information systems architecture. IBM Systems Journal **26**(3) (1987) 276–292

[16] Haren, V.: TOGAF Version 9.1. 10th edn. Van Haren Publishing (2011)

[17] Francesco, P.D., Malavolta, I., Lago, P.: Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture (ICSA). (April 2017) 21–30

[18] Yu, Y., Silveira, H., Sundaram, M.: A microservice based reference architecture model in the context of enterprise architecture. In: 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC). (Oct 2016) 1856–1860

[19] 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)

[20] 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)

[21] 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)

[22] Fielding, R.T., Taylor, R.N.: Architectural styles and the design of network-based software architectures. Volume 7. University of California, Irvine Doctoral dissertation (2000)

[23] 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)

[24] 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)

[25] 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

[26] 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

[27] 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)

[28] 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