2018 IEEE Symposium on Service-Oriented System Engineering

An Approach to Extract the Architecture of Microservice-Based Software Systems

Benjamin Mayer
Johannes Kepler University
Linz, Austria
benjamin.mayer@jku.at

Rainer Weinreich
Johannes Kepler University
Linz, Austria
rainer.weinreich@jku.at

***Abstract*—Microservices decouple network-accessible system components to support independent development, deployment, and scalability. The architecture of microservice-based software systems is typically not defined upfront but emerges by dynamically assembling services to systems. This makes it hard to extract component relations from static sources since component relationships may only become evident at runtime. Existing systems focus either on the static structure of service relations, neglecting runtime properties, or on (short-term) monitoring of runtime properties to detect errors. We present an approach to extract and analyze the architecture of a microservice-based software system based on a combination of static service information with infrastructure-related and aggregated runtime information.**

***Index Terms*—Microservices, architecture extraction, microservice monitoring, microservice management**

I. Introduction

Software architecture helps to analyze the qualities of a software system. It is essential to understand how a system is split into components and how those components are interacting. System componentization is very important, because it directly influences the organization of software development teams [1]. Since individual components are typically developed by a specific team, dependencies between components also lead to dependencies between teams, and thus to an effort that requires greater coordination.

A software system's architecture is usually described as part of the system documentation. Since system documentation is often incomplete, out of date, or entirely unavailable [2], [3], our previous work has focused on the automatic extraction, visualization, and analysis of software systems architectures, especially those of service-oriented software systems [4].

Microservices are a specific style of service-oriented architectures, lately established in many organizations [5]–[7]. Componentization in a microservice-based architecture uses loosely coupled services running in their own processes and communicating with lightweight communication protocols [8], [9]. This style promises independence in development, scaling, and deployment [10] and more dynamic software architectures [11]. Usually, teams developing microservices are organized around business capabilities, not technical capabilities [8]. Each new requirement should be addressed by only one microservice to retain independent development [10]. Requirements that affect more than one service (and thus more than one team) indicate an unsuitable design, revealing the need for architectural changes. Changing the architecture of a microservice-based software system may require moving functionality from one service to another. Since each microservice is a distinct deployment unit, and since different services may use different technologies [8], [10], moving code between services can be challenging.

Since, in a microservice architecture, systems are assembled from network-accessible services, their relations and thus the overall system structure may only become evident at runtime [11]. Therefore, an important part of the design of microservice architectures is to analyze the communication relationships between services, which requires a monitoring infrastructure. Other architectural information, like service and API descriptions, can be extracted through static analysis. To retrieve an overall view of the system, the static and dynamic information must be combined. Since services may also use different implementation technologies, extracting the architecture is harder than in homogeneous systems, because each different technology must be supported by an extraction mechanism. Finally, the agility of the system, which results from the independent development and integration of new services, leads to continuous architectural changes [10], which must be continuously extracted from the microservice-based system.

Extracting the architecture of deployment monoliths and homogenous systems often involves source-code analysis and, optionally, the static analysis of additional meta-information and documents [4], [10], [12]. Static analysis is challenging in heterogeneous systems, since it essentially requires a suite of parsers (as demonstrated in [4]) to extract the architecture of the whole system. Existing systems for analyzing and validating distributed software systems usually either focus on static information and neglect runtime properties or focus on short-term monitoring of runtime properties to detect errors [4], [13]–[15]. In addition, existing architecture-extraction approaches often provide snapshots of the current architecture, without considering the evolution of microservices over time [12], [16].

A recently published mapping study [17] observed increasing scientific interest in architecting microservices, but it also noted that only little investigation has been performed to extract the architecture of microservice-based software systems. In this paper, we present an approach for continuously extracting the architecture of REST-based microservice software systems based on a combination of static service information, like service and API descriptions and communication relationships captured at runtime, along with infrastructure-related information. The approach comprises a generic way to retrieve the necessary static and dynamic data from different distributed microservices, with the collected information combined in a central location. Close attention has been paid to continuously extracting the architecture over a long period of time. To support long-term analysis, an aggregation process has been established to condense the collected runtime information. This allows an overview of the software architecture of the implemented system to be obtained (in terms of the services involved and their runtime relationships), which enables supervising this architecture over time. The approach can also be used to identify design weaknesses, manage service APIs, and manage scaling issues.

The remainder of this paper is structured as follows. Section II presents an overview of the different types of architectural information that are important for microservice-based software systems. Section III presents a data model to capture this architectural information for REST-based microservices. In Section IV we describe intended use cases. Section V contains a description of the architecture extraction process and the components involved. Section VI describes the steps we have taken to validate the approach, especially regarding the identification of relevant use cases and required data, along with discussing its viability for long-term data collection. Section VII discusses limitations and threats to validity, and Section VIII presents related work. Section IX concludes the paper.

II. Architectural Information

Architectural information is important at three different levels in a microservice-based software system: service, infrastructure, and interaction.

The first category includes static information about microservices. Aside from a service's general description, this includes its API, organizational information, and its domain model. The service API defines the functionality provided to other services and is required for analyzing what is offered by a service. Since organization and architecture are closely related in a microservice architecture, we also include organizational information in a service description. *Domain-Driven Design* [18] is frequently used to define domain models, which are an important element of microservice design. These domain models are valid within a specific context (bounded context). Both domain models and bounded context are also part of a service description.

The second category includes information on the structure of the underlying infrastructure. Microservice architectures impose specific requirements on the infrastructure of a system. Services must be able to be individually deployed and scaled, and running services should not influence each other in order to avoid cascading failures. Further, microservices may require different runtime environments because they may use different implementation technologies [9], [10]. These additional requirements are mainly addressed by using virtualization mechanisms and continuous delivery pipelines, which allows automatic deployment of services to isolated virtual machines [8]. Lightweight virtualization techniques, like Docker [19], have become very popular for such architectures [10]. These lightweight mechanisms allow each service to run in its own container, which in turn increases the resilience and scalability of the system [9]. Representing the infrastructure as part of the architecture is important in analyzing whether the infrastructure adequately addresses the requirements of the software system architecture.

The third category of architectural information includes the communication relationships among services. Service interactions can only be retrieved at runtime [10]. To analyze service interactions in detail, this information must be available at multiple levels [9]. For example, it might be necessary to know which interface of a service in a particular version has been requested. Also, for the service triggering the request, the requested interface and version can be relevant. It might also be helpful to obtain information at the infrastructure level regarding how services running on specific physical or virtual machines are communicating with each other. To allow such detailed analysis, this information must be captured at a low level and aggregated into a higher-level overview [9]. It also makes sense to include long-term and aggregated runtime information like response time, throughput, and error rate. This information can be used to determine the strength of specific relationships and the success of architectural changes.

III. Data Model

The data model for representing this architectural information is shown in Figure 1. To reflect the three categories of architectural information described in the previous section, the data model comprises the same three sections: service, infrastructure, and interaction.

For several reasons, the data model was optimized for use in a graph database that contains nodes which are connected with directed edges. First, in comparison to SQL databases, we expect to achieve a performance advantage when reading data, because expensive joins of tables [20] are unnecessary when using a graph database. Second, using a graph database, simple transitive queries can be written [20], for example, to load all services with which a specific service is communicating directly or indirectly.

The left part of the data model shown in Figure 1 contains general service information, collected through static analysis. The *Service* node, which represents a microservice, includes a title, a version, and a description of the service. Further, this node contains information about the technology stack used for its implementation and the service creation date (when the service was deployed for the first time). Each version of a microservice is represented by an individual node, which captures the evolution of services. *Services* are connected to a *Contact* node, which represents a person within the organization who is responsible for the service, and a *Domain* nodes. The latter represents the bounded context [18] of a

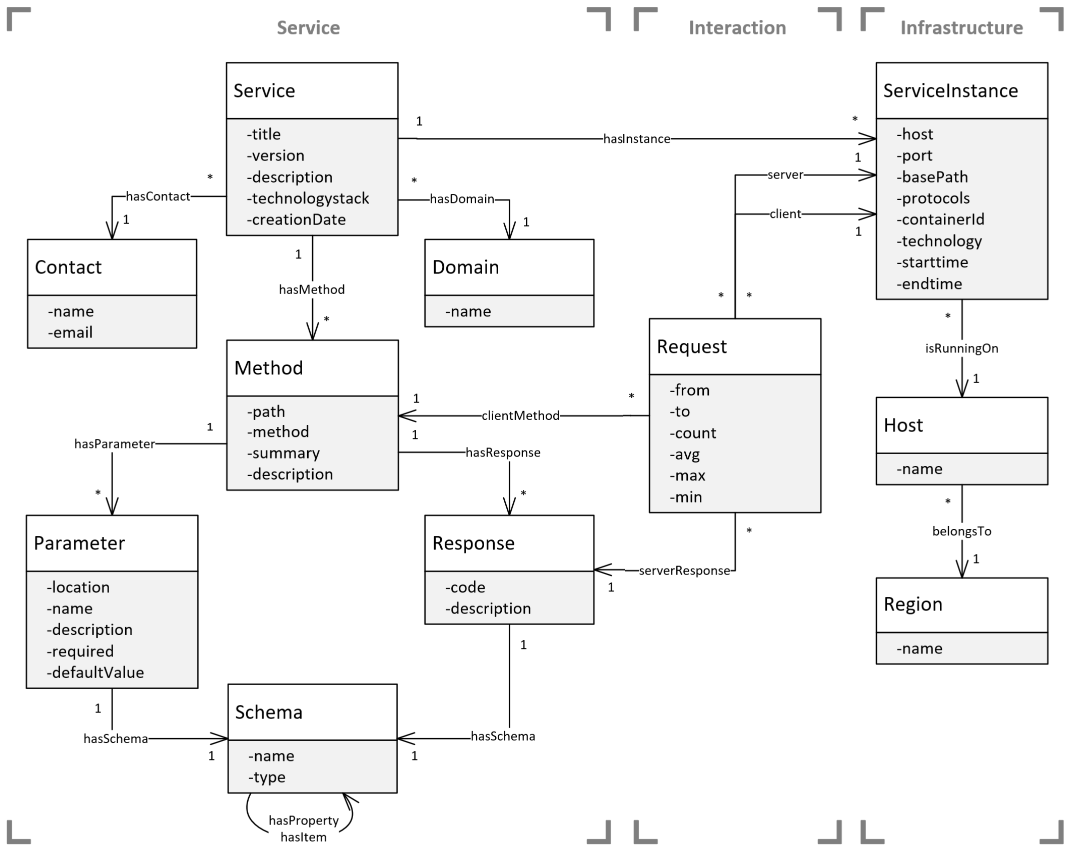

Fig. 1. Data Model

service and can also be used to group services logically. The other nodes of the *Service* section describe the interface of a service. Currently, the interface model is restricted to REST-based interfaces, since REST is often used in microservice-based software systems [10]. Each service may have multiple *Method* nodes that define the functionality a service offers. Each *Method* node describes the underlying function and briefly summarizes this description. The *Method* node also specifies the HTTP method and URI used to invoke the endpoint. Method parameters are represented as *Parameter* nodes. Each *Parameter* node specifies where the parameter should be provided (e.g., body, header) and whether it is mandatory or optional. In addition, a description is provided for documentation purposes. The possible responses to method invocations are modelled as *Response* nodes, which also contain a description of the response. *Response* and *Parameter* nodes are connected to a *Schema* node, which describes the types of invocation parameters and response data. A *Schema* node may comprise additional *Schema* nodes to describe complex objects and arrays.

The infrastructure-related information is shown on the right side of Figure 1. We assume that each service is running in its own container, which has become the accepted way to operate microservices [9]. Containers or instances of services are represented in the data model as *Service Instance* nodes. Each service instance, therefore, is associated with exactly one *Service* node, but a *Service* node may be connected to multiple *Service Instance* nodes. The *Service Instance* node contains information on the service's location and protocol used to interact with a service, along with information on the service's container and implementation technology. It also contains information on the period of time the service has been active. The physical infrastructure is represented as *Host* and *Region* nodes in the data model. A *Host* node represents the physical computational unit, while a *Region* node groups these units. For example, data centers can be modelled as regions.

The last section in the data model, the interaction section, is used to represent the communication relationships between microservices, which are captured at runtime. Each runtime interaction with a service is represented by a *Request* node. A *Request* node in our model relates to the server-side response, which can be used to identify the success of the request (using the response code), the requested method, and the service. The *Request* node also relates to the client-side method, which triggered the request during its execution. This information can be used to trace a service request through the system. We store the captured requests in an aggregated manner to allow long-term analysis. Additionally, we capture information on the infrastructure level by providing relationships between the *Request* node and the server- and client-side service instances, which allows us to identify the virtual and physical machines that are involved in a service interaction.

IV. Use Cases

Microservices are designed to decouple the main system components, but there are situations in which a global overview of the system is important. We identified several use cases that are supported by the described data model.

A. Creating Architecture Documentation

One of the main purposes of our architecture-extraction approach is the automatic documentation of a microservice architecture. By combining static and dynamic information with infrastructure-related information, our approach offers detailed information on individual services and on the overall system architecture. Since we capture data on a fine-grained level, we can provide information on the behavior and interaction of a specific service, and even for a specific version of a service. Using aggregation functions, we can provide a high-level overview of the system, which can be used to understand the system structure, to reason about its properties, and to check the conformance of the emerging runtime architecture with the originally planned system architecture.

The need for an automated approach to architecture documentation is especially evident due to the characteristics of microservice architectures:

- Individual services are frequently updated, making central architecture documentation expensive.
- Service development teams should act as independently from each other as possible [8], and central coordination in documentation tasks should therefore be avoided.
- Assembling systems from independently developed services reveals the system structure only at runtime.

B. Analyzing Architecture Evolution

In a microservice-based system, new service versions and instances are deployed and removed frequently and independently from other services. This means the system is continuously evolving. Our approach allows analysis of these changes over time on different levels. On the service level, we are managing different service versions, which allows identification of architectural changes triggered by new versions of a service. On the infrastructure level, we collect information about the status of containers or service instances (active or not) over time. Finally, service interactions are also associated with time, allowing identification of how they change over time.

One important use case when observing an architecture over time is to identify design problems. For example, strong communication relationships between services might indicate strong coupling between services [10], which prevents independent service development and scalability. Likewise, services that are always deployed together might hint at strong coupling between these services. Cyclic dependencies, which our approach can also identify, are also a problem for independent development [10]. Finally, the size of a microservice may be an indicator of an unsuitable architecture. A microservice which cannot be further implemented by a single team should be split into multiple services [10]. The number of interfaces managed in the defined data model can indicate whether a microservice fits its purpose or whether it needs to be split into several services.

C. Supporting Architectural Changes

Our approach provides additional information for performing necessary changes to a microservice-based software system architecture.

For example, if one microservice heavily uses a specific function from another microservice, this indicates that this function should be separated from its current service to improve service independence. This raises the question in turn of whether the identified functionality should be moved to another service or implemented instead as a new service. It may be difficult to move functions between services implemented with different technologies. Therefore, we provide information on the technology stack each service uses. Moving functionality between services also impacts other services using that functionality. Identifying services using a particular functionality helps to analyze the impact of such changes and also identify any necessary adaptations. The captured runtime metrics (response time, error rate) help to determine the success of architectural changes and identify potential side effects.

D. Managing the Infrastructure

Infrastructure-related information comprises one important part of our data model. This information could be used to check if a service is available at least twice, if a service is available in all data centers, or if specific services are running on the same physical machine. Combining this information with the communication relationships captured at runtime can be used to determine whether or not service communication crosses data center boundaries, if service interaction occurs on the same physical machine, wherever applicable, if the number of requests is balanced between the running service instances, or if some service instances are never called. Moreover, infrastructure data allows predicting the impact of failing hosts to improve the distribution of microservices, and it allows analysis of where instances of specific services are running and how to interact with them, which can be important to know for development.

E. Scaling

The captured runtime and infrastructure-related information also provides support for scaling. Runtime metrics like workload or response time help identify heavily or less heavily used services, which need to be scaled up or down. Through long-term workload analysis, it might also be possible to predict future workloads, which is useful for automatic scaling. Information about the strength of service interactions

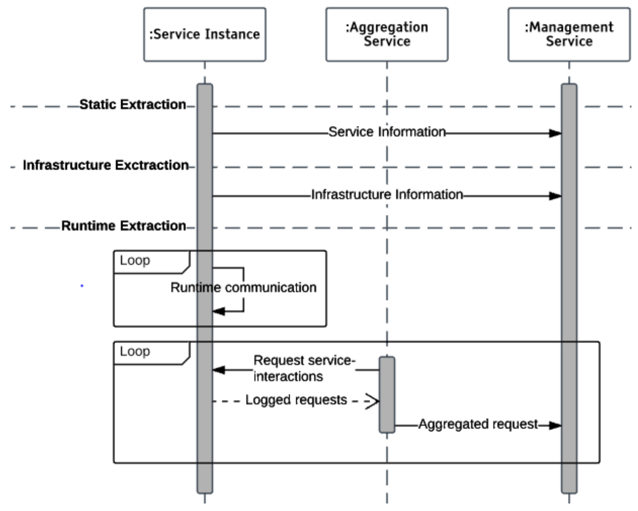

Fig. 2. The Architecture Extraction Process

is essential to identify those services that must be scaled together. Information about physical infrastructure is important for defining onto which machine a new service instance should be deployed. The provided runtime metrics can also be used to determine if a —potentially automatic—scaling process works as expected.

F. API Management

The captured API descriptions support several use cases. First, they provide an overview of a service's functionality. Second, the continuous extraction of API information allows tracking changes to APIs over time. In combination with runtime information, this allows developers to make statements about the success and impact of new funtionality. Third, it is also possible to identify which of a service's interfaces are hardly or not used at all; such interfaces and their implementations could be removed to decrease the size of the microservices and keep them maintainable. Finally, the defined data model allows determining which microservices and their functions will be affected by changes to a specific API so their functions can be changed accordingly, if necessary.

V. Architecture Extraction

Our approach to architecture extraction consists of three main phases—data collection, data aggregation, and data combination—each modelled as three main components in our system:

- The *Data Collection Library* is used to instrument individual microservices, to automatically collect runtime data, and to provide, using REST APIs, both static and runtime data about a service to the other components of the architecture extraction approach.
- The *Aggregation Service* is used to aggregate the information collected at runtime, supporting long-term analysis.
- The *Management Service* is the central part of the system, combining the data from the different microservices and storing the information based on the described data model.

Instrumenting services is a prerequisite to collect the necessary static and runtime information for each service. For instrumenting services, we use the *Data Collection Library*. For each service, this library collects and provides service-, interaction-, and infrastructure-related information. It must be implemented for all technology stacks used in a microservice-based software system. Currently, we only support the Spring Framework [21]. The effort required to support additional technology stacks depends on a specific stack's support for implementing interceptors for service calls. If interceptors can be realized with similar effort as in Spring and if the respective technology stack is based on Java, most of the existing code could be reused to implement the *Data Collection Library*.

By integrating the *Data Collection Library* into the service's implementation, the instances of each service are automatically providing static and runtime service information through additional REST APIs. We use Swagger [22] to generate the API descriptions of services, provided as JSON data. Swagger is available for a broad spectrum of technologies, and the provided API information is based on the *OpenAPI* specification [23]. Other service- and infrastructure-related information is obtained from infrastructure-specific information providers (e.g., by reading infrastructure-related information from a configuration file) that must be configured by the user. To capture the communication relationships at runtime, each request of each service instance is stored in service-specific log files. Access to these log files is provided by the service via REST APIs, as described below.

Depending on the type of architectural information extracted from a system, we distinguish three different kinds of architecture extraction:

- Static information extraction, which extracts information like the interface (API), developer, and domain from a service.
- Infrastructure information extraction, which processes the infrastructure-related information of each service instance.
- Runtime information extraction, which handles the runtime information of each service instance and the communication relationships between services.

Figure 2 shows how these different types of extraction are supported by the main components of our approach. First, static service information and infrastructure information are extracted and sent to the central *Management Service*. Next, runtime information is extracted continuously for each service and stored to service-specific log files. Finally, the logged requests are fetched from the *Aggregation Service*, which aggregates and forwards them to the *Management Service*. The *Management Service* combines the aggregated information with the other service information. The different kinds of service extraction and the components involved are described in more detail below.

A. Static Information Extraction

The static extraction process starts after a new service is deployed (which means that a service instance is created). During the service's startup process, the Swagger framework first generates static service-interface information (essentially a JSON object). This information is combined with domain information and sent to the *Management Service*. Based on the information received, the *Management Service* determines if the deployed service instance is an instance of an already existing microservice or if it is new, since a service can be identified uniquely based on service name and version.

Therefore, if the *Management Service* cannot find a service in the database with the provided name and version, a new entry describing this service is created. Similarly, the required contact or domain nodes for the service are created, if these do not already exist. If, on the other hand, the *Management Service* does find a service with the provided name and version in the database, the received service information is checked against the existing information for the service. If service name and version are identical, method and parameter names, response, domain, contact, and type descriptions must also be identical. Otherwise, there is an inconsistency within the system, and the new services cannot be deployed. The static extraction phase ends by storing all static service information in the central *Management Service* information database.

B. Infrastructure Information Extraction

After the static extraction is complete, infrastructure-related information is extracted. This information, also generated by the newly deployed service instance based on the provided configuration, is exchanged with the central *Management Service* during the startup process. Based on the received information, the *Management Service* creates a new service instance node in its database, connecting it with the associated service and the physical infrastructure information. If the corresponding physical infrastructure nodes (host, region) do not exist, they are created accordingly. If a new service instance is deployed to the same location as an already existing instance, the *Management Service* checks if both service instances are the same, or if the existing instance is outdated and should be set inactive. After the completion of the infrastructure-extraction phase, all infrastructure-related and service information is stored in the database, according to the defined data model.

C. Runtime Information Extraction

During the lifetime of a microservice instance, all outgoing and incoming requests of the service are recorded to a local log file for the service instance. Each log entry represents a specific request, comprising a timestamp, response time, response code, a unique ID of the source service instance, the URL of the target service instance, and the requested method. These log files can be accessed via REST interfaces and are mainly consumed by the *Aggregation Service*. The responsibility of this service is to periodically fetch, aggregate, and preprocess the logged requests.

The *Aggregation Service* periodically performs the aggregation process in a configurable interval. The first step of the aggregation process is to fetch all service-request log entries stored since the last aggregation cycle (local log files are deleted after they are fetched). Requests within the defined interval are combined into an aggregated request entry if they refer to the same client- and server-side service instances, the same client method, and the same server response. To calculate the aggregated request entry, the fetched requests from the service instances are stored in a local aggregation database and then aggregated. Each aggregated request contains the time interval, the number of requests, and the average, maximum, and minimum response time. Depending on the configured interval, the aggregation process significantly reduces the amount of data in the aggregation database (see Section VI). At the end of the aggregation process, the aggregated service requests are forwarded to the central *Management Service*, which stores them in its management database. If the *Management Service* receives no further runtime information from a specific service instance, it marks that instance as inactive.

The *Aggregation Service* can be configured individually. By increasing the interval between aggregation processes, more requests are processed, thereby more strongly aggregating the runtime information and reducing the size of the graph database but the runtime data can then not be analyzed over a short period of time. The *Aggregation Service* can be set up to deploy one global instance for all microservice instances in a system or instead to deploy one instance per microservice instance. The global variant has the advantages that the *Aggregation Service* exists only once and the individual service containers are not affected. The per-service variant has the advantage that the aggregation process is more lightweight, since it need only process local runtime information. Another advantage of local *Aggregation Service* instances is that they can also be used alongside firewalls, because they are colocated with their respective services.

VI. VALIDATION

To validate the defined data model, we conducted a combined survey and interview study (see [24]) to identify important information and use cases for managing microservices. We received feedback from 15 architects, developers, and operations experts. The study participants identified information about service APIs, service interactions and dependencies, service version, the number and distribution of service instances, and system metrics as most important.

Based on these results, we created a microservice dashboard supporting the identified use cases. The realized dashboard comprises visualization components, defined based on the information identified in the survey.

Figure 3 shows one particular view of the dashboard, offering an impression of the provided visualization components. The dashboard view presented in the figure overviews a microservice-based software system. It contains a graph showing the allocation of services and service versions to system nodes (top left) and charts showing the failure rate and

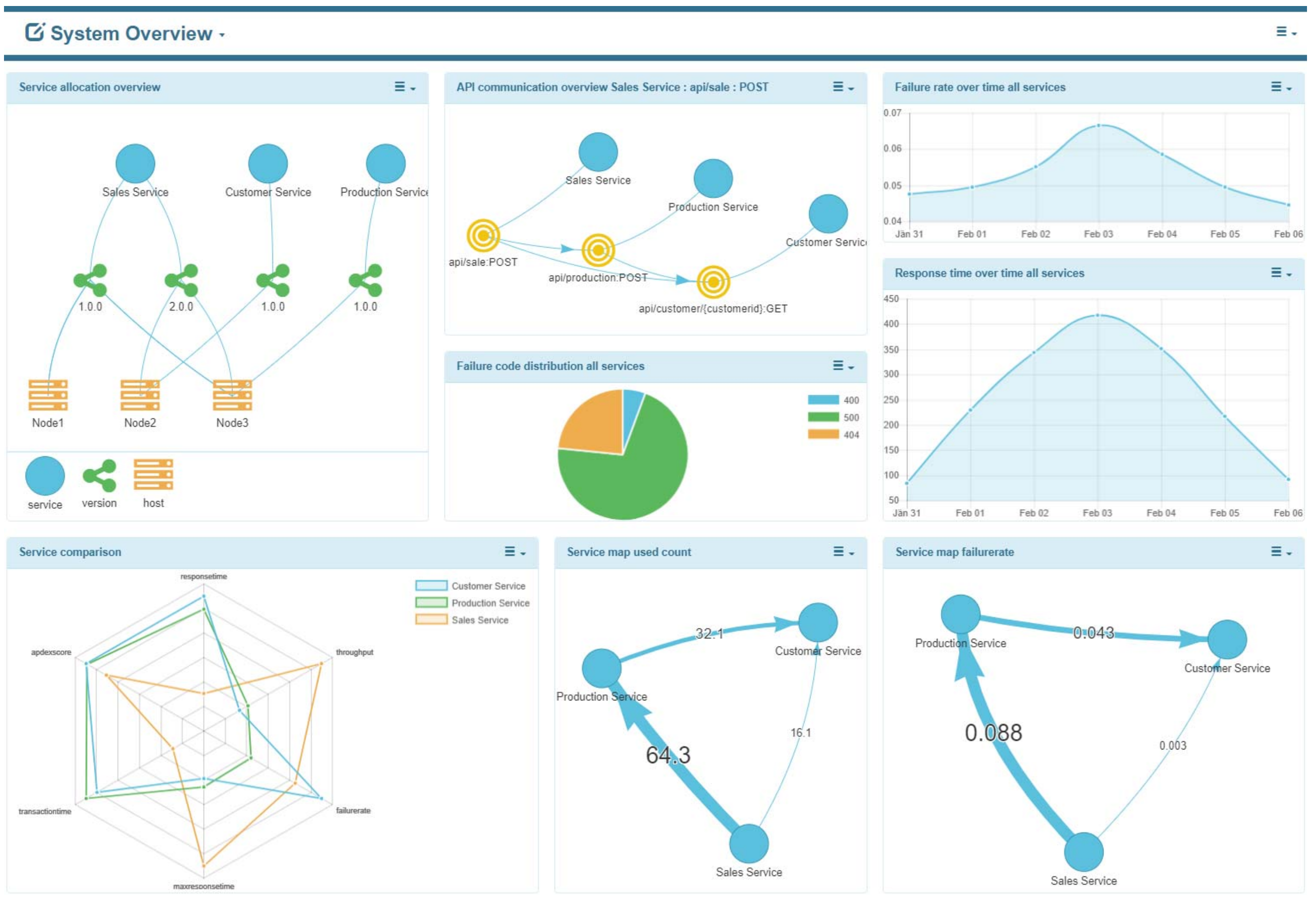

Fig. 3. Dashboard: System Overview

response time of the entire system (top right). The example dashboard also shows graphs of the system architecture based on how services are interacting with each other. For example, the *service map used count* shows that the *Sales Service* uses the *Aggregation Service* quite often, meaning that it has a high communication frequency on average per interval compared to other service relationships. This strong communication relationship is shown through thick lines between the services on the graph. The *service map failure rate* shows the same relationship from the perspective of the failure rate of synchronous calls. It shows that the failure rate between the sales and *Aggregation Service*s is about 9% on average, meaning that about 9% of requests sent from the *Sales Service* to the *Aggregation Service* are failing.

Service relationships can also be viewed in even more detail. For example, the graph in the top center shows the communication paths between different services upon the call of a particular method of a service. In this case, if a POST request is performed on *api/sale* of the *Sales Service*, it will issue a POST request to the *Aggregation Service*, which will in turn issue a GET request to the *Aggregation Service*. While not showing particular traces, this does show particular communication paths between services alongside the concrete communication endpoints and HTTP methods used.

Other dashboard views of the realized system provide runtime information about specific services, detailed information about service interactions, the services' APIs, and comparisons of different services or service versions. The dashboard shown in the figure and the described dashboard views are also available online (see [25]). By using our architecture extraction system as a backend for the realized dashboard, we showed that information participants in the survey identified as important could be offered based on the data model presented in Section III.

To test our approach in terms of long-term data collection, we set up a small test scenario comprising three microservices: a *Sales Service* (in two versions), a *Aggregation Service*, and a *Aggregation Service* (see Figure 3). We used a centrally configured *Aggregation Service* in this scenario, meaning that a single *Aggregation Service* periodically fetches monitoring data from all services in the system at a specified time interval. We used parts of the Netflix technology stack for service discovery and client-side load balancing. To generate interesting runtime information, we instrumented the services to randomly produce failures and response times. The results of this experimental system can be explored in [25].

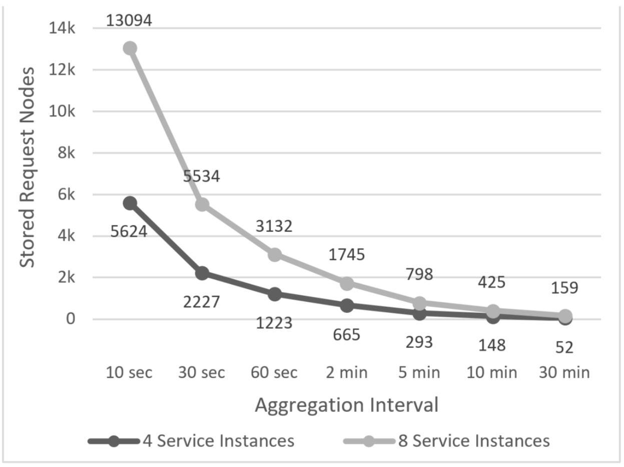

Fig. 4. Number of service request nodes with differently configured intervals

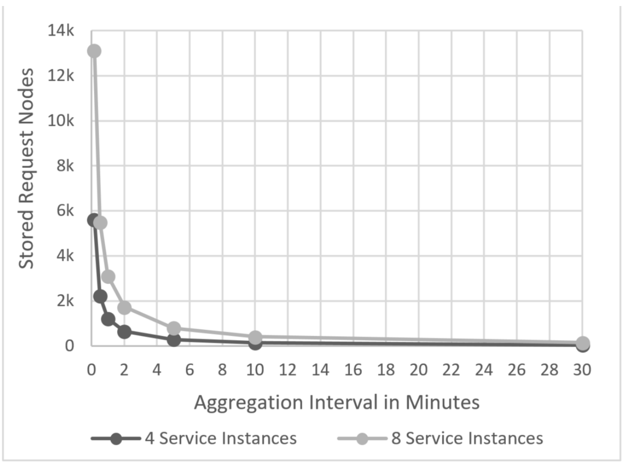

Fig. 5. Number of service request nodes per aggregation interval

The effect of the aggregation functions used for long-term data collection is illustrated in Figures 4 and 5, which show that the aggregation function leads to a significant reduction in request nodes (and thus data) that must be stored in the central service management database. Both figures show reductions with both four and eight service instances. For example, with eight service instances, if the aggregation interval is set to 10 seconds, 13094 request nodes must be stored in the central management database, whereas only 159 request nodes must be stored if the interval is set to 30 minutes.

For both the four-instance scenario and the eight-instance scenario, the reduction essentially follows an exponential decay function, as is even more obvious in Figure 5. This means that the approach reduces the amount of collected data, as required, since an interval of 30 minutes or more is sufficient for architecture extraction. Comparing the number of service request nodes between the four-instance and eight-instance scenarios shows a linear relationship between data values at the specified intervals.

VII. Discussion and Limitations

Our approach currently primarily supports microservices following a REST-based style of architecture, meaning that communication between services is synchronous and based on HTTP.

Asynchronous communication is supported to some degree. A RESTful pattern for asynchronous (or deferred synchronous) communication frequently used in practice is to transfer operations into resources (see [26]). Using this pattern, the server-side service immediately answers a request with a response code of 202 (i.e., Accepted) and a link to the started operation. At the location of this link, the server provides the current information about the operation. Once the operation has finished, the server responds with a 303 response code (i.e., See Other) with a link to the result [26]. Another approach to asynchronous communication using RESTful mechanisms is to use user-defined callback methods, often called Web Hooks [27]. When requesting a resource or initiating an operation, the client specifies a link that the server should call with the results of the request. On the server-side service, the request is again answered immediately, and when the operation is finished, the specified client link is called with the results [27]. Our approach partially supports such patterns, capturing the involved request/response interactions. By analysing the HTTP response codes, we can distinguish simulated asynchronous requests from synchronous requests. This means that such asynchronous requests also contribute to the extracted architecture, showing which services are communicating and how strong their communication relationships are. Our approach currently cannot link a (deferred) response to such asynchronous requests, meaning that we cannot calculate the overall response time to such requests.

Since our approach and the associated data model were designed to support REST-based services, we do not currently support microservice architectures based on asynchronous message communication. In general, we can even capture point-to-point communication, using our aggregation approach for such systems. In terms of asynchronous or deferred synchronous communication, the same restrictions would apply as described above for REST-based communication. This means that we could capture service-to-service communication and the frequency (strength) of particular interactions, deriving an architecture diagram in terms of components and (communication) relations. However, to calculate the overall response time for deferred synchronous requests, we would need to introduce an additional correlation ID to associate requests with responses. Indirect communication using message queues or some other kind of message-based middleware is currently not supported by our approach.

We tested the completeness of the data model by implementing use cases in the microservice dashboard previously identified through a survey and interviews with domain experts. We also set up an experimental environment to test the whole extraction process and the utility of the aggregation function. The main threat to validity is that our approach has so far not been used in an actual production environment. For example, the storage savings derived from aggregation depend on the number of communication relationships in a microservice system. Fewer and stronger communication relationships may lead to larger savings than would a higher number of relationships between different services. The actual data storage required thus depends on system size and characteristics. However, we think that, for architecture extraction, the storage requirements can be mitigated by increasing the aggregation interval, as shown in our experimental setup.

VIII. Related Work

MicroArt [12], [16] is an architecture extraction approach for microservices that also combines static and dynamic information. The approach extracts static information, mainly from source code repositories. Runtime information is extracted from container log files. The approach is semi-automatic; results of an automatic recovery process must be manually reviewed and refined. The approach focuses on providing a snapshot of the current architecture. Currently, MicroArt provides no description of service APIs and does not include runtime metrics to determine the strength of relationships between services. Also, MicroArt does not support long-term architecture analysis and runtime data aggregation, which are supported by our approach.

Dynatrace [13] and New Relic [14] are monitoring solutions that can be used as a *Software as a Service* solution. They are not primarily built for use in microservice-based systems but deliver much of the runtime- and infrastructure-related information that is important for microservice architectures. Both systems provide full-stack monitoring, capturing infrastructure and service information. These systems focus on identifying problems in short time intervals. Long-term analyses are not possible; older runtime data are cyclically deleted. Compared to our approach, these approaches do not support API descriptions and version-related service information.

For analyzing the communication relationships at runtime, distributed tracing systems like Zipkin [15], Pivot Tracing [28], OpenTracing [29], and Apache HTrace [30] are available. By manually or dynamically defining tracing points, these systems try to identify those services, processes, or tasks that are mainly responsible for errors and performance issues. Therefore, these systems are mainly used to identify problems and determine the root causes of those problems on short time intervals. Zipkin [15], for example, designed based on Google's Dapper [31], captures distributed timing information based on spans. A span is a subprocess handling a specific request. Spans are connected with each other by a parent-child relationship and are combined to a trace. In Zipkin, the main spans are requests to other services. Tracing systems allow exact analysis of the flow of a specific request through a system, therefore allowing a more detailed analysis of individual requests than does our system. We currently aggregate such request information to determine a system's runtime architecture. Unlike our system, tracing systems provide no static or infrastructure-related information, and the supervision of traces over a long time period is also costly in terms of storage. While some tracing systems, like Zipkin, provide a sampling rate, with only a certain percentage of requests analyzed, this is not a viable approach in our case, since it would distort the calculated communication frequencies between services.

Prometheus [32] is an open-source monitoring and alerting system. Prometheus generates and stores data as time series, which can be individually defined. The resulting metrics are associated with multiple dimensions (i.e., the http-method is a dimension of http-requests). To retrieve the desired information, a flexible query language is available. Prometheus can be used to analyze the runtime relationships of a microservice architecture, focusing on short-term analyses to identify problems. By defining recording rules and aggregated metrics, long-term analyses are also possible. Since Prometheus stores data as a time series, the system provides no static information and is therefore less suitable for extracting architectural structures.

Other software architecture recovery and reverse-engineering systems typically use static architecture extraction (often based on source code artifact relationships). A comparison of software architecture reverse-engineering methods can be found in [33]. A comparative analysis of existing software architecture recovery techniques is presented in [34]. Such approaches are limited in the case of microservice architectures, in which the system architecture is usually determined at runtime. They may, however, be beneficial for migrating monolithic software architectures to a microservice architecture, since component dependencies can usually be determined from source code in software monoliths.

In our previous work regarding the LISA approach, we used static architecture extraction for service-oriented software systems [4], [35]. The system has evolved over several years [36] and is still in use by our industrial partners [2]. LISA uses different code-level and component-model parsers to extract the architecture of a heterogeneous software system, mapping the system to a uniform architecture model. Component-model parsers use component-model-specific code annotations and coding conventions to identify components, services, their properties, and their relationships. By simulating the behavior of component-model-specific dependency injection, the approach can also identify runtime structures between loosely coupled services. Contrary to the approach presented in this paper, LISA neither collects nor uses runtime information and infrastructure-related information and thus provides no information on the actual strength of communication relationships or on the underlying infrastructure.

IX. Conclusion

We presented an approach to continuously extract the architecture of a REST-based microservice software system. The main characteristics of the approach are the collection of static and dynamic information from services, the aggregation of collected runtime information, and the combination of static and runtime information. A configurable aggregation function condenses the captured runtime information to a single dimension, enabling analysis of the evolution of the architecture over a longer period of time. By combining static service information, like service and API descriptions, with infrastructure-related and runtime information, we can support the use cases we identified in an interview study with domain experts.

We tested our system, setting up an experimental environment comprising several communicating services. The results show the viability of the data collection and aggregation process. We used the combined information generated by the presented system in our dashboard (available at [25]) to create several visualization components that offer detailed information on individual services and on the overall system architecture, which is useful for documentation purposes. The defined visualizations are also useful for identifying design drawbacks, such as strong coupling between services, or other potential areas of architectural improvement.

In our future work, we plan to improve our approach by extending the process to other applications and infrastructure technologies. We also plan to improve the aggregation process by further condensing older runtime information, which would allow analysis of both short-term and long-term aspects. Finally, we intend to use the captured information to automatically detect design problems, and we are working on better support for asynchronous and deferred synchronous communication based on RESTful patterns.

REFERENCES

[1] M. E. Conway, "How do committees invent," *Datamation Journal*, vol. 14, no. 4, pp. 28–31, 1968.

[2] T. Kriechbaum, G. Buchgeher, and R. Weinreich, "Service Development and Architecture Management for an Enterprise SOA," in *Software Architecture*, P. Avgeriou and U. Zdun, Eds. Cham: Springer International Publishing, 2014, pp. 186–201.

[3] D. Rost, M. Naab, C. Lima, and C. von Flach Garcia Chavez, "Software Architecture Documentation for Developers: A Survey," in *Software Architecture: 7th European Conference, ECSA 2013, Montpellier, France, July 1-5, 2013. Proceedings*. Springer Berlin Heidelberg, 2013, pp. 72–88.

[4] R. Weinreich, C. Miesbauer, G. Buchgeher, and T. Kriechbaum, "Extracting and Facilitating Architecture in Service-Oriented Software Systems," in *6th European Conference on Software Architecture*. IEEE, Aug. 2012, pp. 81–90.

[5] G. Buchgeher, M. Winterer, R. Weinreich, J. Luger, R. Wingelhofer, and M. Aistleitner, "Microservices in a Small Development Organization," in *Software Architecture*. Cham: Springer International Publishing, 2017, vol. 10475, pp. 208–215.

[6] J. Gray, "A Conversation with Werner Vogels," in *ACM Queue 4(4)*, 2006, pp. 14–22.

[7] A. Singleton, "The Economics of Microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 16–20, Sep. 2016.

[8] J. Lewis and M. Fowler, "Microservices," Mar. 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[9] S. Newman, *Building Microservices*, 1st ed. Sebastopol, CA: O'Reilly Media, Inc., 2015.

[10] E. Wolff, *Microservices: Flexible Software Architecture*, 1st ed. Boston: Addison-Wesley Professional, Oct. 2016.

[11] E. Woods, "Software architecture in a changing world," *IEEE Software*, vol. 33, no. 6, pp. 94–97, Nov 2016.

[12] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle, "Towards Recovering the Software Architecture of Microservice-Based Systems," in *IEEE International Conference on Software Architecture Workshops*. IEEE, Apr. 2017, pp. 46–53.

[13] "Digital Performance & Application Performance Monitoring | Dynatrace." [Online]. Available: https://www.dynatrace.com/

[14] "New Relic: Application Performance Management and Monitoring." [Online]. Available: https://newrelic.com/

[15] "OpenZipkin - A distributed tracing system." [Online]. Available: http://zipkin.io

[16] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle, "MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems," in *IEEE International Conference on Software Architecture Workshops*. IEEE, Apr. 2017, pp. 298–302.

[17] P. D. Francesco, I. Malavolta, and P. Lago, "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption," in *IEEE International Conference on Software Architecture*. IEEE, Apr. 2017, pp. 21–30.

[18] E. Evans, *Domain-driven Design: Tackling Complexity in the Heart of Software*. Boston, USA: Addison-Wesley, 2004.

[19] "Docker." [Online]. Available: https://www.docker.com/

[20] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases - New opportunities for connected data*, 2nd ed. O'Reilly Media, Inc., 2015.

[21] "Spring Framework." [Online]. Available: https://spring.io/

[22] "Swagger The World's Most Popular Framework for APIs." [Online]. Available: https://swagger.io/

[23] "Open API Initiative." [Online]. Available: https://www.openapis.org/

[24] B. Mayer and R. Weinreich, "A Dashboard for Microservice Monitoring and Management," in *IEEE International Conference on Software Architecture Workshops*. IEEE, Apr. 2017, pp. 66–69.

[25] "Microservice Dashboard | WIN-SE." [Online]. Available: https://www.se.jku.at/microservice-dashboard/

[26] C. Pautasso, "RESTful Web Services: Principles, Patterns, Emerging Technologies," in *Web Services Foundations*, A. Bouguettaya, Q. Z. Sheng, and F. Daniel, Eds. New York, NY: Springer New York, 2014, pp. 31–51.

[27] L. Jeff, "Web hooks to revolutionize the web," May 2007. [Online]. Available: http://progrium.com/blog/2007/05/03/web-hooks-to-revolutionize-the-web/

[28] J. Mace, R. Roelke, and R. Fonseca, "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. ACM Press, 2015, pp. 378–393.

[29] "opentracing.io." [Online]. Available: http://opentracing.io/

[30] "Apache HTrace." [Online]. Available: http://htrace.incubator.apache.org/

[31] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google, Inc., Tech. Rep., 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[32] "Prometheus - Documentation." [Online]. Available: https://prometheus.io/docs/

[33] C. Stringfellow, C. Amory, D. Potnuri, A. Andrews, and M. Georg, "Comparison of software architecture reverse engineering methods," *Information and Software Technology*, vol. 48, no. 7, pp. 484–497, Jul. 2006.

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

[35] R. Weinreich and G. Buchgeher, "Towards supporting the software architecture life cycle," *Journal of Systems and Software*, vol. 85, no. 3, pp. 546–561, Mar. 2012.

[36] G. Buchgeher, R. Weinreich, and T. Kriechbaum, "Making the Case for Centralized Software Architecture Management," in *Software Quality. The Future of Systems- and Software Development*. Cham: Springer International Publishing, 2016, vol. 238, pp. 109–121.