Discovery of Microservice-based IT Landscapes at Runtime: Algorithms and Visualizations

Martin Kleehaus
TUM, Germany
martin.kleehaus@tum.de

Nicolas Corpancho Villasana
TUM, Germany
nicolas.corpancho@tum.de

Dominik Huth
TUM, Germany
dominik.huth@tum.de

Florian Matthes
TUM, Germany
matthes@in.tum.de

Abstract

The documentation of IT landscapes is a challenging task which is still performed mainly manually. Technology and software development trends like agile practices and microservice-based architectures exacerbate the endeavours to keep documentation up-to-date. Recent research efforts for automating this task have not addressed runtime data for gathering the architecture and remain unclear regarding proper algorithms and visualization support. In this paper, we want to close this research gap by presenting two algorithms that 1) discover the IT landscape based on historical data and 2) create continuously architecture snapshots based on new incoming runtime data. We especially consider scenarios in which runtime artifacts or communications paths were removed from the architecture as those cases are challenging to unveil from runtime data. We evaluate our prototype by analyzing the monitoring data from 79 days of a big automotive company. The algorithms provided promising results. The implemented prototype allows stakeholders to explore the snapshots in order to analyze the emerging behavior of the microservice-based IT landscape.

1. Introduction

The current Information Technology (IT) in organizations is evolving rapidly for fulfilling the fast-changing requirements. The reason is that companies operate in a dynamic and high competitive marketplace in which the capability to adapt to changing conditions has become fundamental for companies to survive and to successfully compete against their rivals.

Those conditions require more and more the collaboration between different stakeholders for improving the quality of software applications while being able to develop them more quickly and reliably. New software development methodologies such as agile practices [1], DevOps [2] and continuous delivery [3] of containerized applications emerged from this development. One architecture style that gained popularity in the last years is microservices [4]. Microservice architecture is a variant of the service-oriented architecture (SOA) style that structures an application as a collection of loosely coupled services that run in their processes. Many well-known companies promote microservices such as Amazon, Spotify, Linkedin, and Uber. By using this architectural style, these companies claim to have achieved high scalability, agility and reliability [5]. Microservice architectures support heavily the aforementioned collaboration aspect by releasing the rigid structure of monolithic systems towards independent deployments of single applications. Hence, microservices embed easily into the agile environment and continuous delivery approach.

Enterprise architecture (EA) management (EAM)[6] has been established as an important instrument for managing the complexity of the IT-landscape and enabling enterprise-wide transparency. EAM is typically conducted to document and analyze the status-quo of the current EA in order to define requirements and plans for transformations to an architecture that enables the business strategy, optimizes the business processes and therefore reduces costs. IT landscape modelling as a subarea of EAM aims to discover and visualize the artifacts and relationships of the company's IT landscape. It is typically conducted manually by people within one organization having different technical and nontechnical backgrounds. This activity is mandatory for analyzing transformation strategies.

Unfortunately, even though microservices have several advantages in contrast to monolithic systems, this architecture style introduces a high level of complexity with regard to IT landscape modelling [7]. For instance, due to agile practices new microservices or communication paths between microservices can be introduced very quickly into the current infrastructure or removed when they are no longer needed. Hence, it

is crucial to keep track of the emerging IT landscape, which raises the documentation overhead.

The consequences are out–of–date EA models which lead to decisions made on wrong data or bad data quality [8]. Farwick et al. [9] and Hauder et al. [10] identified runtime data as a promising information source for delivering EA relevant information. Hence, a few research endeavours [11, 12, 13] leverage runtime data for automating IT landscape modelling. However, the presented solutions did not allow to capture the complete IT landscape: An important aspect which is missing in most of the solutions are the proper identification of communication dependencies from an end-to-end perspective, i.e. the communication paths between runtime artifacts and the used interfaces (API) for the information exchange. Although there exists a plethora of commercial and open-source monitoring vendors that provide powerful agent-based instrumentations to gain insights from an end-to-end perspective via tracing [14], those tools were primarily developed for i.a. monitoring application performance (APM) and not for documentation purposes. For instance, many monitoring solutions are event based [15], i.e. they provide runtime data as soon as a specific event occurs like a user request or a system failure. Those events are traced and kept in the database for a certain period of time. Within an event, the communication behavior between IT components is unveiled. Outside of events only the state (running, paused, down, etc.) of IT components is captured. As a consequence runtime data provide only the IT architecture within a specific point of time which does not mean it is complete at all. In order to capture the entire as-is IT landscape with all identified communication paths a request of the complete runtime history is required which is mostly not possible due to performance reasons and resource limitations. In addition, old components and communication paths would also be extracted that were already removed several sprints ago.

In this work, we present a solution that leverages runtime instrumentation for continuously reconstructing and documenting the as-is microservice-based IT landscape in any given point of time. The contribution of our work is threefold: 1) First, we describe an approach for continuously tracking architecture changes and applying those changes to our maintained architecture in order to ensure an up–to–date IT landscape documentation. 2) We store every change to the architecture as snapshots in our database and visualize this emerging behaviour based on a timeline. The visualization of the information exchange dependencies is built upon a graph-based scheme provided by GraphQL as the query language. 3) We allow users to manually refine the reconstructed architecture and save those refinements in our database. We evaluate our prototype in a big automotive company located in Germany.

The remainder of the paper is organized as follows: Section II presents related academic work that influenced our design decisions. In Section III, we describe the concept in more detail, whereas in Section IV, we dive deeper into the implementation aspects. Afterwards, we continue in Section V to discuss our evaluation results. We finish the paper with our limitations and a conclusion in Section VI and VII.

2. Related work

There exist a few concepts on how to integrate runtime data from existing data sources for IT landscape modelling. Holm et al. [13], as well as Alegria et al. [16] make use of network analysis tools in order to infer information on the IT infrastructure. Buschle et al. [12], on the other hand, interpret the configuration of an Enterprise Service Bus (ESB) to include knowledge on communicating information systems in the EA model. These approaches have in common that they are limited to a specific layer of the EA and the authors do not consider communication paths between runtime artifacts. In addition, they are not appropriate for microservice-based architectures.

O'Brien et al. [17] provide a state-of-the-art report on several architecture recovery techniques and tools. The presented approaches aim to reconstruct software components and their interrelations by analyzing source code and by applying data mining methods.

Cuadrado et al. [18] describe a case study of the evolution of an existing legacy system towards SOA. The proposed process comprises architecture recovery, evolution planning, and evolution execution activities. Similar to our approach, the system architecture is recovered by extracting static and dynamic information from system documentation, source code, and the profiling tool. This approach, however, does not analyze communication dependencies between services, which is an outstanding feature of our prototype.

Van Hoorn et al. [19][20] propose a framework *Kieker* for monitoring and analyzing the run-time behaviour of concurrent or distributed software systems. Although the framework focuses on application-level monitoring, the authors also present a way how Kieker could be used to recover microservice-based IT landscapes via analyzing the profiled traces. Unlike us, Kieker does not store and process architectural changes in runtime. In addition, it remains unclear how communication deletions are processed.

MicroART, an approach for recovering the architecture of microservice-based systems is presented in [21][22]. 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. However, the tool does also not track architectural changes and remain imprecise what happens when communication between microservices are deleted.

3. Architecture discovery concept

3.1. Monitoring techniques

Our algorithm for discovering microservice-based IT landscapes is based on the combination of three monitoring concepts:

A best practice pattern[1] for building microservice architectures is the usage of **service discovery** [23] that serves as a repository to find the network location of a specific microservice dynamically. Microservices frequently change their status and IP-address due to reasons like updates, autoscaling or failures. In order that the microservices are still able to find each other in the network, the service discovery serves as a gateway that always provides the current network locations. In case a change in the architecture (removed service, added service, updated service) is detected, this alteration is reflected in the repository of the service discovery. By retrieving this information, we are able to reveal the current status of each service instance.

The service discovery mechanism already provides useful data about the status of the microservice but lacks in reporting detailed infrastructure information and interrelationships. For that reason, an additional **monitoring agent** is required that delivers 1) infrastructure-related data, like host, container type and address, database type and address, operating system or information about the cloud provider. 2) Interrelationships unveil schematic connections between microservices and other infrastructure elements, like on which host the microservice is running, in which specific container or operating system the microservice is deployed, or with which database the microservice is communicating. The repository data is enhanced with this information.

Both concepts discover the status of running microservices in run-time and unveil infrastructure-related information. However, the real communication behaviour between the microservices still remains unknown. For that reason, it is necessary to instrument each microservice with a monitoring probe that tracks request flows through the system. This technique is called **distributed tracing** [14] and adapted by many commercial, or open-source monitoring solutions[2]. Distributed tracing tracks all executed HTTP requests in each service by injecting tracing information into the request headers. The main purpose of tracing is to analyze application performance (APM) and to troubleshoot latency problems. In addition, it also provides capabilities to add further information in the form of annotations to each request. These annotations contain additional infrastructure and software-related information like executed endpoint address, class and method name, requested port, etc. We leverage distributed tracing in order to unveil the communication behaviour between microservices.

Last but not least, huge microservice infrastructures are load balanced to avoid single points of failures. Tracing data or monitoring agents also provide service instance information. Each instance of a service has the same name but distinguish itself in IP address and port. Therefore, the uniqueness of a service instance is defined by the service description in combination with the used IP address and the service port. In order to discover all instances that belong to a specific service, we aggregate the information based on the service description.

It is mostly not required to instrument the IT-landscape with three different monitoring agents. This would increase the administration and resource overhead unnecessarily. Modern monitoring tools like Dynatrace, AppDynamics or Instana already integrate all mentioned monitoring techniques in one monitoring agent which is a huge benefit from a DevOps point of view.

3.2. Process description

Even though the combination of the aforementioned monitoring techniques discovers most architecture elements of an IT landscape, it only unveils a snapshot of the current architecture which created runtime data for a defined period of time. This is mostly enough for analyzing the general existence of an IT element. However, it cannot be ensured that the communication structure is uncovered completely, as it

[1] https://microservices.io

[2] https://openapm.io/landscape

would require all possible communications between the applications happened in the considered period of time. Consequently, we also have to analyze historical runtime data. Hereby, we face the following four challenges: 1) In order to reduce network overhead, most tracing techniques are based on sampling, i.e. only a percentage of requests is traced and forwarded to the monitoring server. In the worst case, specific communication paths are rarely seen. 2) Due to resource limitations, most monitoring tools can only provide a small timeframe of runtime data, e.g. last 6 hours, depending on the frequently incoming data volume. A request for runtime data for a longer duration would be too resource intensive and cannot be served. 3) Most monitoring tools store runtime data for only a specific period of time and archive or even delete older data in order to ensure free storage capacity is always available. 4) The history does also contain old components and communication paths that were already removed several sprints ago. This legacy data must be filtered in order to unveil the real architecture. This can be easily performed with the general existence of IT artifacts, as they frequently provide health data. If no health data is coming from a component anymore, it was certainly removed from the architecture. However, it is a different case with communication paths, as they only get visible by request events. No communication does only mean there have been no events reported.

Considering the listed challenges, we developed a concept that discovers the architecture of the current IT landscape based on a three-step process. First, we reconstruct the architecture by analyzing historical data. We developed the *backwardDiscovery* algorithm for this purpose. This algorithm runs recursive and retrieves in every iteration historical tracing data (*tD*) with a timeframe of $T = t_1 - t_0$. In case no further data is available the discovered architecture is returned for manual refinement. In the second step, we support the user with a visualization for adapting the architecture manually, i.e. the user is able to change the structure of the communication paths. Finally, the algorithm *forwardDiscovery* gets triggered on a fixed time interval and consumes new incoming runtime data for continuously adapting the final architecture.

3.3. Algorithms description

We execute two algorithms in a chronological order to unveil the complete IT landscape architecture. We assume the architecture $A(E, C)$ is a directed graph with runtime artifacts E and communication paths C, whereas $C \subseteq E \times E$ on a finite set E.

The algorithm *backwardDiscovery* analyzes the available historical runtime data and reconstructs the architecture $A\prime(E\prime, C\prime)$ from the last reported time t_{-1} until the present time t_0. It excludes communications that did not occur in the regarded history yet or includes communications between microservices that were already removed in last sprints. Both scenarios must be handled accordingly. Hence, we define $A\prime(E\prime, C\prime)$ as

$$E\prime = E :\Longleftrightarrow \forall e(e \in E\prime \leftrightarrow e \in E)$$
$$C\prime := \{c \mid (c \in E\prime) \wedge (c \in E\prime \cap E)\}$$

The algorithm is executed one time. Due to resource limitations, we need to provide a timeframe T that represents the maximum time period that is accepted by the monitoring tool to go back in history. First, we instantiate the architecture $A\prime(E\prime, C\prime)$ based on the repository data $rD(E)$ (line 3 and 4) most monitoring tools provide in order to identify running IT artifacts. We use the function REPOSITORYDATA() for this purpose. The communication paths C remain empty. Next, we retrieve the tracing data $tD(E, C)$ for the last considered timeframe via the call TRACEDATA(t_0, t_1) (line 7). If the tracing data tD is not empty (line 8), we iterate through all elements $e \in tD(E)$ and validate whether the elements e are also included in the repository data (line 9 and 10). If this is the case, we add all communication paths assigned to this runtime element to the architecture (line 11) and start over with the next timeframe (line 12). If no data is received from the monitoring server, the algorithm returns an incomplete architecture (line 14) which can be used as a basis for further refinements. Line 9 to 11 can also be described as an intersection between the elements e in $tD(E, C)$ and $rD(E)$, but for simplicity reasons we use the imperative representation.

The next algorithm *forwardDiscovery* is executed after *backwardDiscovery* and the manual refinement. It runs continuously based on a defined frequency and is eventually returning the complete architecture of the instrumented IT landscape. As an input, the *forwardDiscovery* function consumes 1) a timeframe T for retrieving the monitoring data, 2) the deletion threshold τ which defines how old a communication path is allowed to be, before it gets removed and 3) the incomplete architecture returned by the *backwardDiscovery* function or the manual refinement. First, the function fetches both the current content of the repository (line 3) and the trace data (Line 4) for a specific period of time. Based on the retrieved data the architecture $A\prime\prime$ is refined accordingly. For the runtime elements, we apply the intersection (line 5) and for the communication paths, we use the union (line 6) to return the complete architecture which is eventual

consistency in case the missing communication paths were available in the tracing data. However, we are still facing the issue that removed communications are not recognized without any manual input. Hence, we incorporate a threshold $\tau > 0$ that defines the maximum period of time how long communications are allowed to be invisible in the tracing data. In case the threshold is exceeded (line 8), the particular communication path is marked as deleted (line 9). Hereby, we use the last seen timestamp of each communication. We never remove communications from the current architecture as we never can make sure that the communication is not appearing in future traces again. Finally, we store the current snapshot of the discovered architecture (line 10). The algorithm itself is designed to be idempotent as long as no changes have occurred in the architecture, therefore running it multiple times has no further impact on the result.

Algorithm 1 Backward Discovery

Require: $T > 0$

1: **function** BACKWARDDISCOVERY(A, t_0, T)
2: **if** $A = \emptyset$ **then**
3: $rD(E) \leftarrow$ REPOSITORYDATA
4: $A\prime \leftarrow A(rD(E), C)$
5: $t_1 \leftarrow t_0$
6: $t_0 \leftarrow t_1 - T$
7: $tD(E, C) \leftarrow$ TRACEDATA(t_0,t_1)
8: ***if*** $tD \neq \emptyset$ ***then***
9: ***for all*** $e \in tD(E)$ ***do***
10: ***if*** $e \in rD(E)$ ***then***
11: $A\prime\prime \leftarrow A\prime(E, C \cup tD(C_e))$
12: BACKWARDDISCOVERY($A\prime\prime, t_0, T$)
13: ***else***
14: ***return*** $A\prime$

Algorithm 2 Forward Discovery

Require: $T > 0, \tau > 0$

1: **function** FORWARDDISCOVERY(A, τ, T)
2: $t_1 \leftarrow t_0 + T$
3: $rD(E) \leftarrow$ REPOSITORYDATA
4: $tD(E, C) \leftarrow$ TRACEDATA(t_0,t_1)
5: $A\prime \leftarrow A(E \cap rD(E), C)$
6: $A\prime\prime \leftarrow A(E\prime, C \cup tD(C))$
7: ***for all*** $c \in A\prime\prime(C\prime)$ ***do***
8: ***if*** $c(lastSeen)+\tau \leq t_0$ ***then***
9: $c(deleted) \leftarrow true$
10: $V(i+1) \leftarrow A\prime\prime$
11: ***return*** $A\prime\prime$

The *forwardDiscovery* algorithm can also be adjusted in a way it does not trigger the architecture modification based on defined time intervals but on specific events that occur in the organizations. When the trigger is aware of changes immediately when they occur, this could potentially give birth to the concept of "real-time" IT landscape architecture documentation. An overview of potential trigger events is shown in the following:

- Scheduled trigger: Default trigger running on a defined schedule
- Pipeline trigger: Triggers the algorithm as soon as changes are deployed to production via a continuous delivery pipeline
- Manual trigger: Allows further external tools or a user to trigger the *forwardDiscovery* algorithm manually

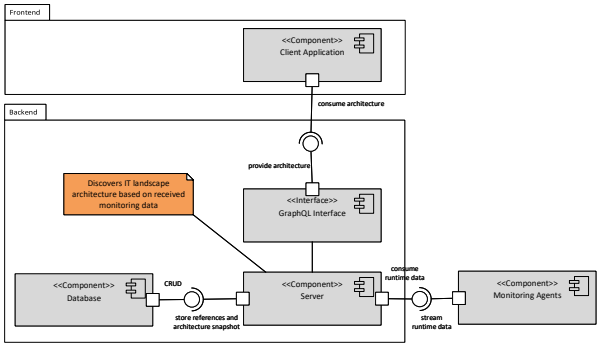

Figure 1: Tracing data of sample version figure

4. Implementation

The architecture of the prototype is built on four main components: 1) the server receives the runtime data from the monitoring agents and reconstructs the architecture, 2) the database stores the architecture snapshots and the references to the runtime artifact, 3) GraphQL interface communicates with the server and provides a query language to traverse through the discovered IT-landscape architecture, and finally 4) a client application for visualizing the architecture and enabling manual refinements. The corresponding component diagram is illustrated in Figure 1.

4.1. Data model

The data model of our prototype is depicted in Figure 2: The class "Snapshots" contains all snapshots made for the IT landscape architecture during the

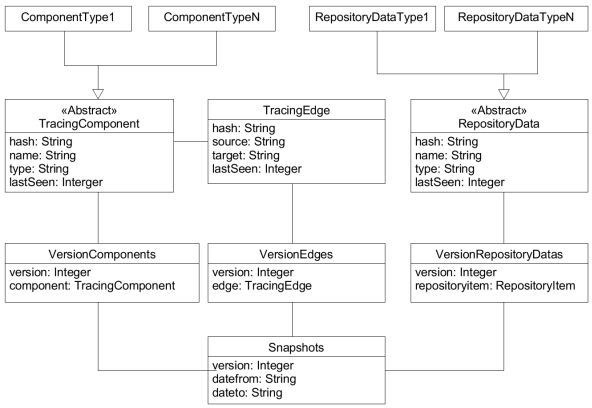

Figure 2: Architecture reconstruction data model figure

time. Every snapshot version represents a defined timeframe that is appropriate for the used monitoring tool starting at ”datefrom” to ”dateto”. A version number is an incrementing number meaning that the highest version number represents the latest stored IT landscape architecture version.

The tracing data are stored in the classes ”TracingComponent” and ”TracingEdge”. Tracing edges describes the communication paths between the tracing components. For every record in those classes, a hash is generated as the primary ID. The attribute ”lastSeen” indicates the snapshot version in which the tracing data have been seen the last time.

The repository data retrieved from the monitoring tool is stored in the class ”RepositoryData”. Unlike the tracing data, monitoring tools mostly does not provide history about the runtime artifacts, hence we version this information by our own through frequently pulling the data from the monitoring tool. The attribute ”lastSeen” contains the version as an integer in which the repository items last existed.

Due to performance and implementation reasons, a version class was integrated for every mentioned IT artifact (component, edges and repository items). These version classes (VersionComponents, VersionEdges and VersionRepositoryData) are related to the created particular snapshots.

4.2. Graph-based visualization

With the support of GraphQL, we are able to provide stakeholders with a query language that enables them to retrieve all information about the IT landscape and to traverse through the discovered IT landscape architecture. In order to allow data to be queried, resolvers have to be defined and implemented on the root

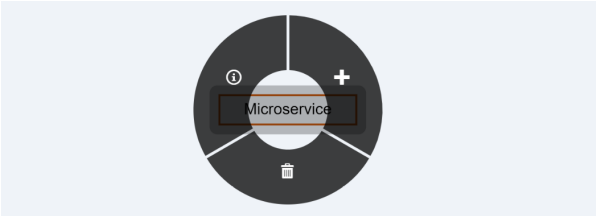

Figure 3: Refinement menu for manual adaptation figure

level. Resolvers such as ”database”, ”microservice”, ”host”, etc allow to query for an artifact or collections of artifacts and work in a similar manner to Remote Procedure Calls (RPC). The client application primarily calls the resolvers and retrieves a JSON-based response with all IT landscape elements. The IT landscape architecture itself is visualized as a directed graph with nodes and edges. Nodes represent the runtime artifacts. The node types are identified with different colors. Edges visualize the communication paths between runtime artifacts. The direction of the edges indicates request calls via TCP or HTTP. An example of this visualization is depicted in Figure 4. Red color indicates microservices. Blue color represents databases and green color describes file storages.

4.3. Architecture refinement support

Our frontend application does not only visualize the IT-landscape architecture based on stored snapshots but also provides features for a manual refinement of the architecture. The user can adapt the following elements:

- Add new and remove legacy communication paths between runtime artifacts
- Add and remove runtime artifacts in case they are not instrumented
- Add annotations to runtime artifacts and communication paths
- Provide a detailed view of a runtime artifact incorporating more information

Figure 3 presents a screenshot of the refinement menu. It appears when the user clicks on an element or a communication path.

4.4. Architecture comparison support

In order to analyze how the IT landscape emerged over time, we integrate a visual comparison between two architecture snapshots. Same runtime artifacts and

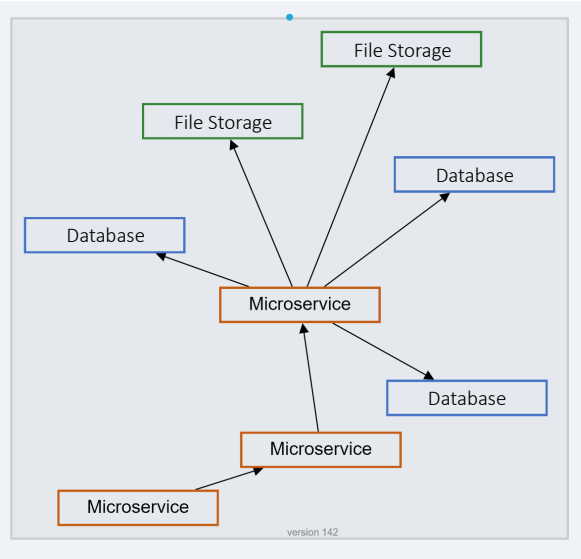

Figure 4: IT landscape visualization with different color codings.
figure

communication paths that occur in both snapshots are highlighted accordingly. The comparison of different snapshots enables a number of use cases. 1) It can be used to get instant feedback about architectural changes, like new or deleted runtime artifacts or communication paths. 2) A comparison also enables feedback regarding the fulfilment of architecture-related requirements and 3) it supports the analysis of the emerging behavior of the architecture. Hence, architects are able to intervene in a timely manner in order to prevent bad design decisions.

5. Evaluation

5.1. Environment description

The company from which we got access to their instrumented IT landscape for evaluating our concept is located in the automotive industry. The company hosted the microservice architecture on the cloud provider Amazon Web Services (AWS). The microservices use the NoSQL database DynamoDB for storing their transaction data. Data streams are realized via Kinesis streams. The streams are processed in Kinesis Firehose and forwarded to all subscribed microservices. The architecture itself provides services for other departments realizing and contributing to various business use cases. The Enterprise Architect keeps the main responsibility regarding overall architectural design decisions and documentation.

AWS provides monitoring data in three different forms. 1) All runtime artifacts are registered in the AWS repository and their health status is frequently reported. In order to remove a specific service, the artifact must be unregistered and deleted accordingly. 2) A further monitoring probe (CloudWatch) creates infrastructure and application logs that record failure events or hardware related data like CPU, memory or network utilization. 3) The tool X-Ray enables tracing for analyzing requests from an end-to-end perspective. For automating the documentation of the microservice architecture, we combine the output of the monitors from the AWS repository and X-Ray via the unique ID.

Due to performance issues and configuration settings, the monitoring tools restricts the access to runtime data to a timeframe of 6 hours and only the last 30 days can be retrieved. In addition, the repository does only store the set of artifacts that are currently running. No history is kept in the database. The accessed microservice-based IT landscape contains 279 runtime artifacts and 34 communication paths. The artifacts consist of 50 microservices, 46 dynamoDB tables, 8 kinesis streams and 175 S3-Buckets that represent simple data storage. The image on the right in Figure 8 shows the final architecture with all runtime artifacts. Due to space limitations, we only show those runtime artifacts that correspond to one product. We ignored the rest.

5.2. Accuracy calculation

After implementing our prototype, we created 316 snapshots representing the last 79 days. The *forwardDiscovery* algorithm was executed at snapshot version 203. After the 79 days, our architecture discovery result was validated by our evaluation partner. Modifications have to performed on the Kinesis streams in order to achieve a complete and accurate model, which we call *base model*. The evaluation itself is done in an iterative process. For each iteration, we compare the reconstructed architecture model against the base model and calculate the accuracy acc.

$$acc = \frac{(TPE_i + TPN_i) - (FPE_i + FPN_i)}{(TPE_i + TPN_i + TNE_i + TNN_i)} \quad (1)$$

TPE = Edges are found both models
TPN = Nodes are found in both models
FPE = Edges are found in the reconstructed model but are not available in base model
FPN = Nodes are found in the reconstructed model but are not available in base model
TNE = Edges are not found in the reconstructed model

but are available in base model
TNN = Nodes are not found in the reconstructed model but are available in the base model

1. Iteration: After analyzing the complete history the *backwardDiscovery* algorithm finally discovered in total 279 correct runtime artifacts. 36 communication paths were unveiled out of 5 are reconstructed that are not present in the base model anymore. 3 communications are missing. Hence, the accuracy is $acc = 82,4\%$

2. Iteration: After receiving the result of the *backwardDiscovery*, we execute the *forwardDiscovery* algorithm with a deletion threshold of $\tau = 168$ hours (28 snapshots) representing the sprint length of 1 week. Hence, the algorithm ran in total 113 snapshots that equals approximately 28 days. Overall, the second algorithm improves the accuracy to $acc = 91.2\%$. After 28 days, 279 correct runtime artifacts were discovered in total. 33 correct communication paths were unveiled. 1 communication was still not found that is available in the base model and 2 communications are marked as deleted although they are still available. Further 2 communications were correctly marked as removed.

We recognized after the second iteration, that an overall threshold for all communication paths cannot be applied as the communication behavior between runtime artifacts differ significantly. Hence, we modified the *forwardDiscovery* to ensure every communication path gets an individual threshold that is recalculated from snapshot to snapshot. We define the threshold as the maximum time in which the communication was not visible regarding the considered timeframe. As an example, Figure 6 illustrates the profile of two different communication paths. Whereas the communication 1 is marked as deleted after 11 snapshots ($\tau_1 = N[247; 258]$), communication 2 can be removed already after 7 snapshots ($\tau_1 = N[259; 266]$). The adapted algorithm could not be executed again due to resource restrictions of the evaluation partner. Hence, we were content with the last 28 days. Unfortunately, the present data pool did not allow an improvement of the accuracy.

5.3. Deletion threshold discussion

The selection of an appropriate deletion threshold strategy is fundamental to keep a high architecture discovery accuracy. In the following, we discuss different approaches on how to define the threshold for deleting potentially removed communication paths:

Manual definition: The period of time of how long the algorithm has to wait until specific communication paths should be highlighted as removed could be based on simple manual input. That means the user defines the number of days as the threshold based on experience. The advantage of this option is the simplicity of this approach. However, it is rather inflexible and does probably not conform to development behavior.

Machine learning based: The drawbacks of the manual method could be neglected by a machine learning approach. Hereby, we create a model that learns the behavior of the developers and predicts future communication removals accordingly. However, the creation of the prediction model is challenging to perform as it depends on the availability of labelled data, i.e. each communication removal must be recorded.

Event based: Specific events that describe a situation in which selected communication paths must be deleted can be leveraged for defining an event based threshold. However, the threshold is not a period of time anymore but represents rather a boolean value that triggers the deletion workflow. An advantage of this option is a resource optimization and near real-time documentation. On the contrary, the definition of possible events is challenging.

Tool support: In the last option, no threshold calculation is performed at all. The removal of obsolete communications is achieved by tool support. Based on an application that visualizes the IT landscape architecture the developers can decide which communication path is obsolete and must be removed. That means, the decision is outsourced to a manual task, which realizes a high accuracy if developers maintain the communications via the tool. As a disadvantage, no automation mechanism is achieved.

5.4. Snapshot comparison

Figure 8 illustrates the microservice-based IT landscape from two different snapshots. Version 145 was created during the execution of the *backwardDiscovery* algorithm and version 316 was the last created during *forwardDiscovery*. Again, we only visualize the runtime artifacts that correspond to one product. Both images unveil how architecture has changed after 42 days. In total, 37 new runtime artifacts were added to the landscape.

6. Limitations

In the course of the development of this paper a few assumptions have been made that lead necessarily to the following limitations: First, every runtime artifact must be instrumented. Otherwise, the IT landscape cannot be discovered completely. To this end, some capabilities from the monitoring tool has been seen as given. Especially, the report of distributed traces and the propagation of APIs for reading runtime data.

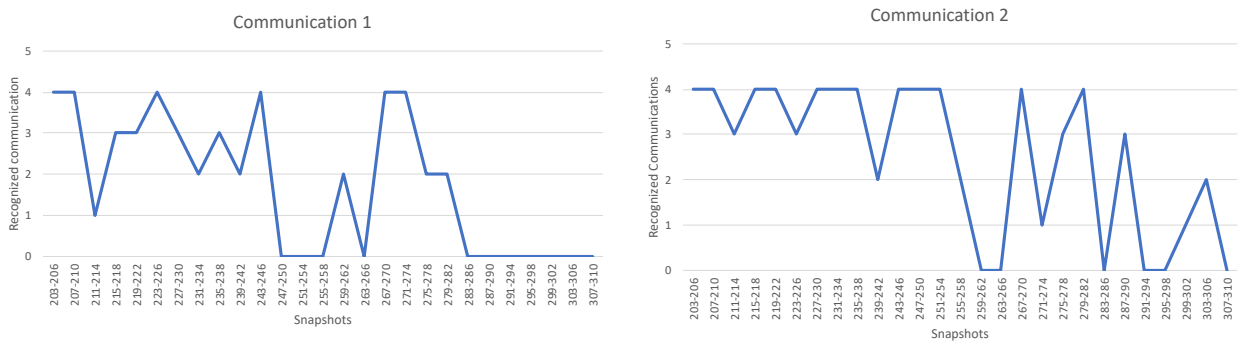

Figure 6: Communication behavior between two microservices. The y-axis represent the number of recognized communications within one day, i.e. 4 snapshots. A snapshot is created every 6 hours.
figure

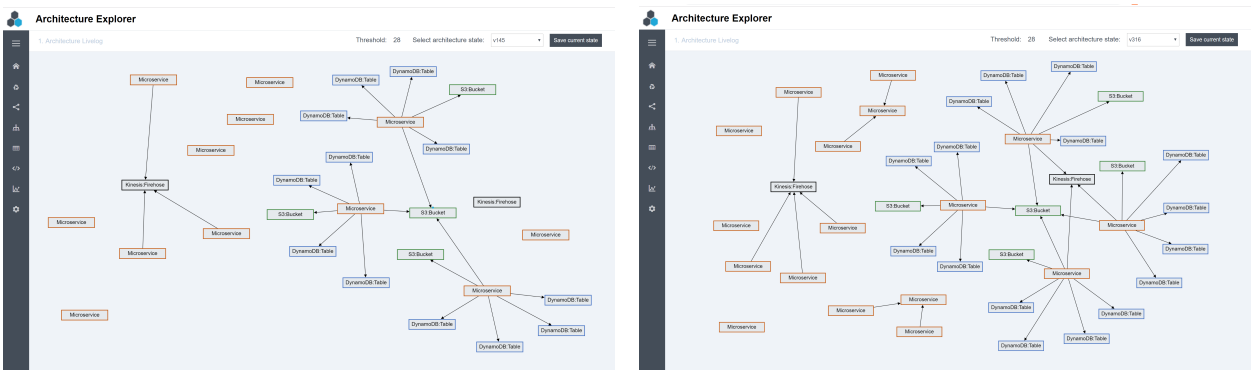

Figure 8: Comparison of two different snapshots of the same microservice-based IT landscape. **Left**: Discovered architecture after snapshot 145. **Right**: Final architecture discovered after snapshot 316 with manual refinement.
figure

Second, we mainly focus on runtime artifacts. If the related process is not running and the monitoring agent is not providing data anymore it is interpreted as deleted, which might be wrong.

Third, the evaluation was conducted within a timeframe of 79 days. To this end, some of the concepts as explained in Section 3 should be tested for a longer time especially when it comes to the proper inclusion in the workflows of teams and architects as well as the calculation of the correct deletion threshold.

Fourth, we evaluate the developed algorithms by calculating the related accuracies. However, we did not incorporate the stakeholders to evaluate the provided visualizations based on structured or semi-structured interviews. Hence, the visualizations were not adapted accordingly. This is part of our future work.

Last but not least, we use CloudWatch and X-Ray as the monitoring providers, as those tools are natively integrated in AWS. However, our proposed algorithms were not evaluated on different environment configurations. Hence, in this current research phase, we cannot confirm a global applicability of our concept. Nevertheless, we are convinced that our approach is also applicable in other technological environments. For instance, further APM vendors like Dynatrace, AppDynamics, NewRelic, or Instana just to name a few also provide powerful instrumentations to gain insights from an end-to-end perspective. Those tools expose several APIs for extracting the application repository and communication behaviour between microservices, and yet show the same issues regarding to IT landscape documentation described in Section 1. Hence, the usage of those tools in other technical environments should not reduce the applicability of the presented algorithms and visualizations.

7. Conclusion

The trend of developing larger applications in the form of microservices as well as the accompanying agile practices expose new challenges to the practices of IT landscape documentation. In order to support

this process, we developed two algorithms that discover continuously the IT landscape by analyzing runtime data. The results are stored as snapshots in our database and visualized via a graph library. Each different runtime artifact is coloured accordingly. We evaluated our prototype in the automotive industry. In total, we created 316 snapshots within 79 days. Our algorithms were capable to discover the IT landscape architecture on the accuracy of $acc = 91.2\%$. One of the biggest challenges we faced was the accurate reconstruction of communication behaviors between microservices.

The proposed approach works well if one important prerequisite is fulfilled: Each runtime artifact has to be instrumented by a monitoring solution that supports distributed tracing and service discovery. In case one of those tools is not installed, the prototype will not become fully operational, which presents our most significant limitation.

References

[1] T. Dingsyr, S. Nerur, V. Balijepally, and N. B. Moe, "A decade of agile methodologies: Towards explaining agile software development," *Journal of Systems and Software*, vol. 85, no. 6, pp. 1213 – 1221, 2012. Special Issue: Agile Development.

[2] J. Smeds, K. Nybom, and I. Porres, "Devops: A definition and perceived adoption impediments," in *Proceedings of the International Conference on Agile Processes in Software Engineering and Extreme Programming (XP 2015)*, pp. 166–177, Springer International Publishing, 2015.

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

[4] M. Fowler and J. Lewis, "Microservices," tech. rep., ThoughtWorks, 2014.

[5] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 243–246, April 2017.

[6] I. Hanschke, *Enterprise Architecture Management einfach und effektiv: Ein praktischer Leitfaden fr die Einfhrung von EAM*. Carl Hanser Verlag GmbH Co. KG, 2016.

[7] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 44–51, IEEE, 2016.

[8] S. Roth, M. Hauder, M. Farwick, R. Breu, and F. Matthes, "Enterprise architecture documentation: Current practices and future directions," in *Wirtschaftsinformatik*, 2013.

[9] M. Farwick, R. Breu, M. Hauder, S. Roth, and F. Matthes, "Enterprise architecture documentation: Empirical analysis of information sources for automation," in *46th Hawaii International Conference on System Sciences*, pp. 3868–3877, Jan 2013.

[10] M. Hauder, F. Matthes, and S. Roth, "Challenges for automated enterprise architecture documentation," in *Trends in Enterprise Architecture Research and Practice-Driven Research on Enterprise Transformation*, pp. 21–39, Springer Berlin Heidelberg, 2012.

[11] M. Farwick, B. Agreiter, R. Breu, M. Haering, K. Voges, and I. Hanschke, "Towards living landscape models: Automated integration of infrastructure cloud in enterprise architecture management," *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 35–42, 2010.

[12] M. Buschle, M. Ekstedt, S. Grunow, M. Hauder, F. Matthes, and S. Roth, "Automating enterprise architecture documentation using an enterprise service bus," in *Americas Conference on Information Systems (AMCIS)*, 2012.

[13] H. Holm, M. Buschle, R. Lagerström, and M. Ekstedt, "Automatic data collection for enterprise architecture models," *Software & Systems Modeling*, vol. 13, pp. 825–841, May 2014.

[14] 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," tech. rep., Google, Inc., 2010.

[15] D. J. Lilja, *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University, 2005.

[16] A. Alegria and A. Vasconcelos, "It architecture automatic verification: A network evidence-based approach," in *2010 Fourth International Conference on Research Challenges in Information Science (RCIS)*, pp. 1–12, May 2010.

[17] L. O'Brien, C. Stoermer, and C. Verhoef, "Software architecture reconstruction: Practice needs and current approaches," Tech. Rep. CMU/SEI-2002-TR-024, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2002.

[18] F. Cuadrado, B. García, J. C. Dueñas, and H. A. Parada, "A case study on software evolution towards service-oriented architecture," in *Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on*, pp. 1399–1404, IEEE, 2008.

[19] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous monitoring of software services: Design and application of the kieker framework," 2009.

[20] A. van Hoorn, J. Waller, and W. Hasselbring, "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), pp. 247–248, ACM, 2012.

[21] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Microart: A software architecture recovery tool for maintaining microservice-based systems," in *IEEE International Conference on Software Architecture (ICSA)*, 2017.

[22] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Towards recovering the software architecture of microservice-based systems," in *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, pp. 46–53, IEEE, 2017.

[23] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 1st ed., 2015.