



TECHNISCHE UNIVERSITÄT MÜNCHEN

LEHRSTUHL FÜR SOFTWARE ENGINEERING BETRIEBLICHER
INFORMATIONSSYSTEME

Model Management along the IT Value Chain in Microservice-based IT Landscapes

Martin Kleehaus

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Georg Carle
Prüfer der Dissertation: 1. Prof. Dr. Florian Matthes
2. Prof. Dr. Martin Bichler

Die Dissertation wurde am 16.02.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.06.2021 angenommen.

I confirm that this doctorate in natural sciences (dr. rer. nat.) is my own work and I have documented all sources and material used.

Munich, 28.01.2021

Martin Kleehaus

Acknowledgments

After many years of intensive work, this journey finally comes to an end. In this period of my life, I had to climb many mountains. Only through the support of a few people, who supported me countless times, I have managed this intense time. First of all, I am especially grateful to my professors. As the first supervisor, Prof. Dr. Florian Matthes has always supported me with his suggestions and inspiring discussions. Without your valuable academic advice, this work would not have been possible. I am also indebted to Prof. Dr. Martin Bichler for his second report. Thank you for the valuable feedback.

In this time, I have realized that an individual is only as strong as the whole team. Hence, I would like to thank my colleagues, who have repeatedly guided me into new and fruitful thematic paths with their enriching tips and contributions to discussions. Special thanks go to Ömer Uludag, Jörg Landthaler and Dominik Huth for their contributions to my research through co-authorships. Thanks to Klym Shumaiev, Adrian Hernandez, Dr. Manoj Mahabaleshwar, Dr. Pouya Aleatrati, Fatih Yilmaz, Gloria Bondel and Ingo Glaser for your support in reviewing my papers and providing valuable feedback. Thanks to Ahmed Elnaggar, Ulrich Gellersdörfer, Dr. Felix Michel, Dr. Anne Faber, Dr. Bernhard Walzl and Patrick Holl for the constructive cooperation in teaching and researching.

I would also like to thank my students who allowed me to guide them through their research. These thanks go to Daniel Hoyos, Jochen Graeff, Lukas Steigerwald and Tien Vu Duc. Special thanks go to Patrick Schäfer, Nektarios Machner, Nicolas Corpancho, Christopher Janietz and Ludwig Achhammer who contributed a lot to the development of my concepts and prototypes. I really enjoyed working with all of you.

On several occasions throughout my research, I did not only work with colleagues and students but also with industry partners. My thanks go to Dr. Matheus Hauder from Allianz, Dr. Stefan Volkert and Jan Schäfer from BMW, Jeffrey Ahmad, Stefan Melles and Prof. Dr. Andreas Both from DATEV as well as to Dr. Michael Pönitsch from Siemens.

Finally, and most importantly, I express my sincere gratitude to my wife Dr. Leonie Kleehaus for her unconditional love, support, and patience during this challenging time. Thank you for reminding me regularly to take a step back from my work in order to refuel new energy. I am also utmost grateful to my parents Waltraud and Rolf Kleehaus, my parents-in-law Anne and Wilfried Mathemeier and my brother and sisters. Without your great support and encouragement during my whole life, this work would not have been possible.

Garching b. München, 17.01.2021

Martin Kleehaus

Abstract

Model management is used in IT management to maintain an abstract representation of business- and IT-related artifacts. In enterprise architecture management (EAM), those models help to understand complexity, reason about changes and to achieve a holistic view on the enterprise. According to the IT4IT framework, models are used in various phases within the IT value stream. For example, low-fidelity models reflect the current state of each enterprise architecture layer and their interconnections, whereas run-time models capture elements of running systems. However, efficient model management faces numerous challenges, e.g. model creation and maintenance is still conducted manually, which is error-prone, cost-intensive and time-consuming. In addition, microservice architectures introduce a high level of complexity with regard to model management, as our empirical findings confirm.

In order to tackle those challenges, research endeavours conducted in EAM as well as in model-driven engineering propose to automate the creation of models by collecting the required information from several information sources used along the IT value stream and to reassemble the scattered information into one central model management. However, as literature confirms several problems remain: the extraction of models leads to ambiguous documentation of the architecture and the merging of models lead to conflicts. Moreover, model management in EAM restricts modelling to current and future states of an enterprise. An approach for the automated maintenance and linking of all models required along the IT value chain in a central repository does not exist at this time.

In this research, we present an approach that allows reconstructing the models from the individual phases of the IT value chain automatically based on runtime information and configuration files that are assigned to each application in the observed IT landscape. We transform the reconstructed models into a linked knowledge graph, which represents our central model management. This graph can be accessed via a uniform visual interface and query language. For this purpose, we design and develop a tool called *MICROLYZE*.

We apply design-science as our research methodology to evaluate the developed concepts and the corresponding prototype. By conducting two case studies in two different companies, we assess the applicability of our approach and the prototype's practicability. 19 interviews with practitioners from 17 different companies provide feedback about the proposed software-, process- and visualization design of the prototype. The reported experiences showed that *MICROLYZE* is able to discover most of the models from each phase of the IT value stream and successfully connect them to a linked knowledge graph. The elaborated solution approach was positively received by the practitioners. However, the concept still has to be examined with respect to its scalability, whether the information base is truly fully recovered, as well as the ability to uncover the rationale behind architecture changes and certain runtime behaviour.

Kurzfassung

Modelle beschreiben eine abstrakte Darstellung von geschäfts- und IT-bezogenen Artefakten. Im Enterprise Architecture Management (EAM) helfen Modelle, Komplexität zu verstehen, sowie eine ganzheitliche Sicht auf die Unternehmensarchitektur zu schaffen. Gemäß dem IT4IT-Framework werden Modelle in verschiedenen Phasen innerhalb der IT-Wertschöpfungskette verwendet. Beispielsweise beschreiben Modelle mit einem geringen Detaillierungsgrad die Schichten einer Unternehmensarchitektur und ihre Verbindungen untereinander. Laufzeit-Modelle hingegen erfassen das Laufzeitverhalten von Anwendungen. Die Schwierigkeit ein effizientes Modellmanagement zu etablieren, liegt in der fehleranfälligen und zeitaufwändigen manuellen Pflege der Modelle. Zudem bringen serviceorientierte Architekturen wie Microservices eine hohe Komplexität in Bezug auf das Modellmanagement mit sich.

Um diesen Herausforderungen zu begegnen, schlagen verschiedene Forschungsarbeiten vor, die Erstellung von Modellen zu automatisieren, indem die erforderlichen Informationen aus mehreren Informationsquellen gesammelt und wieder zu einem zentralen Modell zusammengeführt werden. Allerdings führt die Extraktion von Modellen aus mehreren Anwendungssystemen zu einer mehrdeutigen Dokumentation der Architektur und die Zusammenführung zu Modellkonflikten. Das Modellmanagement im EAM beschränkt sich auf die Erfassung von aktuellen und zukünftigen Zuständen eines Unternehmens. Ein Ansatz zur automatisierten Pflege und Verknüpfung aller Modelle, die entlang der IT-Wertschöpfungskette benötigt werden, existiert in einem zentralen Repository zum gegenwärtigen Zeitpunkt nicht.

In dieser Forschungsarbeit stellen wir einen Lösungsansatz vor, der es ermöglicht, die Modelle aus den einzelnen Phasen der IT-Wertschöpfungskette auf Grundlage von Laufzeitinformation sowie Konfigurationsdateien automatisiert zu rekonstruieren. Die rekonstruierten Modelle transformieren wir zu einem verknüpften Wissensgraphen, der unser zentrales Modellmanagement darstellt. Auf diesen Graphen kann über eine einheitliche Abfragesprache zugegriffen werden. Zu diesem Zweck haben wir die Anwendung MICROLIZE entwickelt, die wir als unser Forschungsartefakt evaluieren.

Als Forschungsmethode wenden wir Design-Science an. Aus den Ergebnissen der Durchführung von zwei Fallstudien in zwei Unternehmen bewerten wir die Anwendbarkeit der von uns erarbeiteten Konzepte und die Praktikabilität des Prototyps. Wir analysieren die Ergebnisse von 19 Interviews mit Praktikern aus 17 verschiedenen Unternehmen um das Software-, Prozess- und Visualisierungsdesign des Prototyps zu evaluieren. Die Ergebnisse zeigen, dass MICROLIZE die meisten Modelle, die zum Verständnis der IT-Landschaft erforderlich sind, automatisiert wiederherstellen kann. Allerdings muss die Skalierbarkeit des Konzepts verbessert werden. Es bleibt zudem unklar, ob die Informationsbasis tatsächlich vollständig rekonstruiert wurde.

Contents

Acknowledgments	v
Abstract	vii
Kurzfassung	ix
1. Introduction	1
1.1. Problem Statement	2
1.2. Research Questions	4
1.3. Research Design	6
1.3.1. Design-Science Research	6
1.3.2. Qualitative Content Analysis	10
1.4. Contribution of this Thesis	14
1.5. Preliminary Work	17
1.6. Structure of this Thesis	20
2. Foundations	21
2.1. Model Driven Engineering	22
2.1.1. Modeling Languages	22
2.1.2. Model Transformations	23
2.1.3. Model-Driven Reverse Engineering	24
2.1.4. Models at Runtime	25
2.2. Enterprise Architecture Management	26
2.2.1. Building Blocks of Enterprise Architecture	26
2.2.2. Management of Enterprise Architectures	29
2.2.3. Modelling Enterprise Architectures	30
2.2.4. Archimate Notation Language	30
2.3. Microservice Architecture	33
2.3.1. Definition of Microservice Architecture	34
2.3.2. Building blocks of Microservice Architectures	37
2.3.3. State-of-the Art in Microservice Adoption	41
2.4. DevOps	42
2.4.1. Agile Practices	46
2.4.2. Software Release Automation	48
2.4.3. Monitoring distributed Systems	49

3. Related Work	55
3.1. Model Reverse Engineering	55
3.1.1. Static based Reverse Engineering	56
3.1.2. Dynamic based Reverse Engineering	56
3.1.3. Hybrid based Reverse Engineering	57
3.2. EA Model Maintenance	57
3.2.1. Federated EA model Maintenance	58
3.2.2. Runtime based EA Model Maintenance	59
3.2.3. Modern Approaches for EA Model Maintenance	60
3.2.4. Change Events that trigger EA model Maintenance	61
3.3. IT Landscape Representation	61
3.4. Demarcation	65
4. Requirement Analysis	67
4.1. A Conceptual Framework for Managing Models along the IT Value Chain .	67
4.2. Identification of Requirements	70
4.2.1. Architectural Requirements	71
4.2.2. Organizational Requirements	73
4.2.3. Functional Requirements	74
4.2.4. Visualization Requirements	76
5. Automated Model Recovery via Runtime Instrumentation	79
5.1. IT landscape topology	79
5.1.1. AppDynamics	80
5.1.2. New Relic	83
5.1.3. Dynatrace	84
5.1.4. AppMon	87
5.1.5. Meta-Model Transformation	87
5.2. System Design	94
5.2.1. Monitoring probes	96
5.2.2. Monitoring Server	104
5.2.3. MICROLYZE.Collect: Collecting Architecture Models	105
5.2.4. MICROLYZE.Analyze: Analyzing Architecture Models	106
5.2.5. MICROLYZE.Store: Storing Architecture Models	107
5.2.6. MICROLYZE.Expose: Exposing Architecture Models	109
5.3. Process Design	115
5.3.1. Reconstruction of Architecture Model Dependencies	117
5.3.2. Reconstruction of Communication Dependencies	119
5.3.3. Validation of Architecture Changes	120
5.3.4. Change Events	122
5.3.5. Revision Concept	124
5.3.6. Recovering REST Calls using Runtime Data	125
5.3.7. Elaboration of a Deletion Threshold	128

5.4.	Visualization Design	128
5.4.1.	Visualization Architecture	129
5.4.2.	Graph Styling	131
5.4.3.	Visualization Process	134
5.4.4.	Architecture Model Deployment	137
5.4.5.	Architecture Model Communication	139
5.4.6.	Architecture Model Interaction	141
5.4.7.	Architecture Model Comparison	143
5.4.8.	Architecture Model Sidebar	145
5.4.9.	GraphQL Client	145
6.	Recovery of Business-related Models	149
6.1.	System Design	150
6.1.1.	Configuration Files	151
6.1.2.	General Extension of the Configuration File Content	153
6.1.3.	References to Federated Information Systems	155
6.1.4.	Importing and Processing of Configuration Files	156
6.1.5.	Continuous Delivery Pipeline	159
6.1.6.	JSON Schema validation	162
6.1.7.	Distribution and Location of JSON Schema Files	163
6.2.	Organizational Design	164
6.2.1.	Roles and Responsibilities	164
6.2.2.	Adapted Agile Development Process	166
6.3.	Process Design	167
6.3.1.	Performed Sequences in CD Pipeline	167
6.3.2.	Processing the content of the configuration file	170
6.3.3.	Meta-model update based on decision tree	171
6.4.	Visualization Design	171
6.4.1.	Architecture Model Cluster	174
6.4.2.	Architecture Model Table View	177
6.4.3.	Aggregated Architecture Model Communication	179
7.	Evaluation	181
7.1.	Evaluation Design	183
7.2.	Case Study in the Automotive Sector	185
7.2.1.	Requirement Analysis and Status Quo	186
7.2.2.	Prototype Integration	188
7.2.3.	MICROLYZE Execution Result	189
7.2.4.	Feedback from Practitioners	193
7.2.5.	Critical Reflection of Results	201
7.3.	Case Study in the Insurance Sector	202
7.3.1.	Requirement Analysis and Status Quo	202
7.3.2.	Prototype Integration	205
7.3.3.	MICROLYZE Adaption	207

7.3.4. MICROLYZE Execution Result	208
7.3.5. Feedback from Practitioners	210
7.3.6. Critical Reflection of Results	216
7.4. Interview Series	217
7.4.1. Assessment of the Solution Architecture	219
7.4.2. Assessment of Model Visualizations	223
7.4.3. Technical and Organizational Integration	228
7.4.4. Supported Use Cases	231
7.4.5. Action Plan	232
8. Conclusion	235
8.1. Summary	235
8.2. Critical Reflection	239
8.2.1. Functional Limitations of the Prototype	239
8.2.2. Critical Reflection on the Validity	241
8.2.3. Critical Reflection on the Research Methodology	242
8.3. Future Work	243
8.3.1. Business Process Recovery via Process Mining	243
8.3.2. Assessment of Architecture Quality	245
8.3.3. Failure Root Cause and Failure Impact Analysis	245
A. Appendix	247
A.1. JSON Schema	247
List of Figures	253
List of Tables	257
Acronyms	259
Bibliography	263

1. Introduction

Business is in the middle of an unfolding era of disruption, driven by digital transformation, which challenges how IT is organized and managed today. The role of IT in the business is elevated from being a support function to an enabler to drive innovation, enhance competitive advantage, boost productivity, and reduce cost by applying innovative technology. However, new technologies can only provide value to the business if these can be properly implemented and managed. Hence, within the last years, large IT organizations experience a culture shift that encourages collaboration for improving the implementation of those new technologies while being able to develop them more quickly and reliably. This shift yields new software development methodologies such as agile practices (Dingsøyr et al., 2012), DevOps (Bang et al., 2013; Leite et al., 2020) and continuous deployment of containerized applications (Fitzgerald et al., 2015), which have significant influence on the further development of Enterprise Architectures (EA) (Ross et al., 2006). These include a re-prioritization of conflicting goals, such as product-oriented vs. process-oriented IT organizations, a continuous evolution of the application landscape vs. long-lived stable products, and small microservices vs. large monolithic applications.

EA management (EAM)(Hanschke, 2016) has been established as an important instrument for managing the complexity of the application landscape and enabling enterprise-wide transparency. EAM is typically conducted to manage and analyze the status-quo of the current EA in order to define requirements and plans for transformations to an architecture that optimally supports the business strategy (Hanschke, 2010; Ross et al., 2006). Hereby, it aims to visualize the relationships among regulations, business processes, software and the underlying infrastructure. In the course of EAM, over the last years, a multitude of management approaches have been developed both by scientists (Frank, 2002; Hafner et al., 2008; M. Lankhorst, 2017; Ross et al., 2006) and practitioners (Dern, 2009; Hanschke, 2016; Niemann, 2005) that propose guidelines how to design, plan, implement, and govern IT today. Well-known frameworks that are applied in many organizations are The Open Group Architecture Framework (TOGAF) (Haren, 2011), the IT Infrastructure Library (ITIL) (Office, 2011b), or the Zachman Framework (Zachman, 1987).

A new framework that was developed by the Open Group to support an end-to-end workflow with a value-chain-based IT operation is called IT4IT (The Open Group, 2019). The IT4IT position dissociates from the previous hard-line of separating development and operations activities which are represented by the aforementioned EA frameworks. More fundamentally, IT4IT is suggesting a position of representing IT from a value chain perspective covering all capabilities and data needed to manage the IT services. It regards the IT function both as an IT service provider to the business and as a consumer of IT services that support the IT function. The value chain itself is aligned to the steps of a traditional IT service development workflow i.e. plan, build, deliver, and run. Hereby, a

newly developed IT service leaves a digital twin, i.e. a *model* in each phase of the value chain that describes the service from that perspective and is backed by the particular collaboration tools.

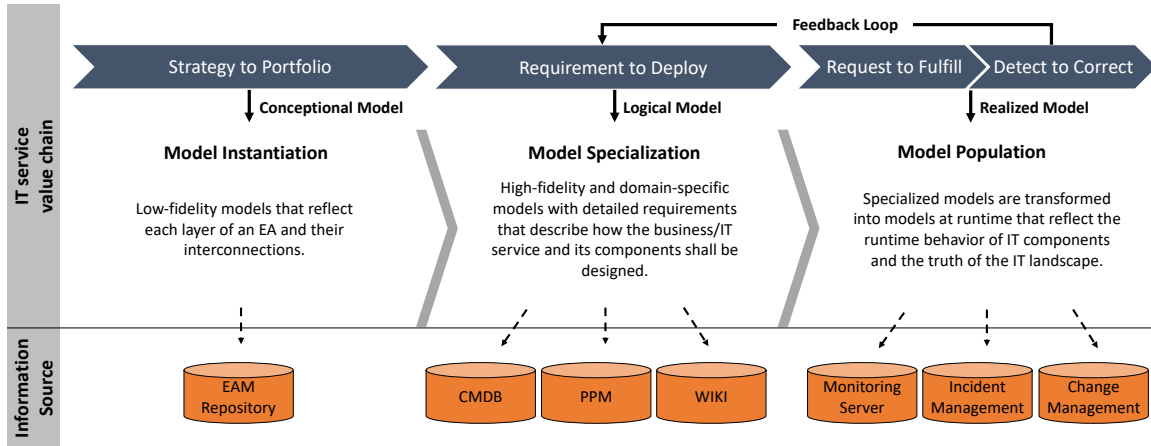


Figure 1.1.: Model representation along the IT value stream described by IT4IT (The Open Group, 2019). Each model undergoes various phases including the *Instantiation*, *Specialization* and *Execution* phase, whereas each phase in the value stream is supported by specific information systems.

The Figure 1.1 illustrates that each model representation of the IT service undergoes various phases within the value stream, the *Instantiation*, *Specialization* and *Execution* phase. Those phases describe the life-cycle of IT service models and emphasize the model interconnection. Unfortunately, those interconnections are not represented in the applied information systems, but only in the form of stakeholder collaboration. Hence, a strong collaboration is essential to maintain, as a lot of stakeholders contribute to the delivery of valuable IT services. This involves frequent communication, as well as knowledge and information sharing in order to establish an efficient continuous feedback loop in every model life-cycle phase.

1.1. Problem Statement

Unfortunately, this feedback loop is often not available. The reasons for this are manifold:

1. The level of specialization between the stakeholders have led to group-specific languages that thwart effective communication (Armour et al., 2003; Dreyfus, 2007; Espinosa et al., 2009).
2. "Stakeholders have different, sometimes even conflicting needs and perspectives" (Niemi, 2007). This often leads to distributed decision making (Dreyfus, 2007), in which decision-makers may make local design decisions without incorporating other stakeholders (Armour et al., 2005; Bubak, 2006; Dreyfus, 2007; Shah et al.,

2007). The impact of such changes are understood locally but often not recognized or understood globally, due to lack of information sharing (Armour et al., 2005). Dreyfus refers to this as *"local optimization with global ramifications"* (Dreyfus, 2007).

3. A further reason refers to information silos in organizations. The silo syndrome (Ensor, 1988) is caused by divergent goals of different organizational units and can lead to a decreased performance, as information is not adequately shared but remains sequestered within each organizational unit.

In line with the above observations, missing stakeholder collaboration also leads to weak EA knowledge management, which is, in general, a big issue in IT organizations as literature proves (Armour et al., 1999; Henttonen et al., 2009; Lam, 2004; Meilich, 2006; Shah et al., 2007; Templeton et al., 2006). Architecture rationale is often poorly documented, which makes it difficult to track *"what decisions were made and why"* (Armour et al., 1999). IT-landscape modelling (M. Lankhorst, 2017), as a sub-area of EAM, tries to discover and to document the EA of an organization by having EA-related models and their relationships as the formal representation of EA information. On the basis of EA models, future architectural plans can be established. This enables architecture decisions to be bundled at a higher level and architecture changes to be controlled with the support of IT governance.

However, many authors report that the EA model maintenance process poses a challenge and is often regarded both, error-prone and time-consuming (Armour et al., 2005; Farwick et al., 2011a,b). For that reason, further research endeavours (Buschle et al., 2012; Farwick et al., 2013, 2012a; Farwick et al., 2011b; Roth et al., 2013a) propose to automate the creation of EA models by collecting the required information from several information sources. Although practitioners widely agree that EAM uses information that is often already contained in existing tools (Farwick et al., 2013), the automated extraction and maintenance of EA models from those tools create new challenges (Hauder et al., 2012). Extracting models from several tools will just lead to ambiguous documentation of the architecture (Shah et al., 2007). Hence, it remains unclear how to reassemble the scattered information that resides in the various information sources used along the IT value chain to one common meta-model. That means the research community has not provided a solution yet for having a single repository that maintains models of all phases of the IT value chain.

In addition, we diagnose that EA model maintenance has been becoming a bigger challenge in the last years due to the rise of microservice-based architectures (Fowler et al., 2014). This new software architecture style supports heavily the continuous delivery approach by releasing the rigid structure of monolithic systems towards independent deployments of single applications. Even though microservices have several advantages in contrast to monolithic systems, this architecture style introduces a high level of complexity with regard to model management (Alshuqayran et al., 2016; Canfora et al., 2011; Kleehaus et al., 2019b). For instance, they aim at distributed transaction processing, foster communication dependencies and cross-domain data exchange. Due to agile practices (Schwaber et al., 2002) architecture decisions are shifted to developer teams. Hence, new microservices are introduced very quickly into the current infrastructure or removed when

they are no longer needed. From a model documentation perspective, microservice-based IT landscapes are very volatile which lead to a big effort in the maintenance of these models.

Due to little published studies about challenges current enterprises face in the model management of microservice-based IT landscapes, we investigated the status-quo in the usage of microservice-based architectures in the German market, what challenges organizations face in maintaining models and to what extent those challenges correlate with the usage of microservices. In (Kleehaus et al., 2019b), we present the result of our survey carried out among 58 EA practitioners. Hereby, two important key observations were made that are worth to mention in this thesis:

First, most of the survey participants confirm that keeping models up-to-date is perceived more challenging with microservice-based architectures, resulting in incomplete, inaccurate and out-dated models. Second, the reasons for bad model quality have a strong focus on content-related challenges. The increasing number of small components, the high velocity of changes and the complex communication behaviour between microservices are the most stated concerns which lead to wrong and out-of-date models.

Based on the stated challenges, in this thesis, we investigate on how to establish a central model management that delivers up-to-date information about each phase of the IT value chain by extracting the required information out of runtime data. With this novel approach the models do not only constitute static information about the IT landscape, but also runtime information derived from monitoring systems which represent the execution phase of the particular model. We apply our central model management solution in a distributed microservice-based IT landscape in order to address the model maintenance challenges identified in our conducted survey (Kleehaus et al., 2019b). For this purpose, we designed and developed a tool called *MICROLYZE* that represents our research artefact. In order to build this tool, we make use of the findings unveiled from the model-driven engineering, EA model maintenance and IT landscape visualization related work. In the next section, we present research questions to guide this investigation.

1.2. Research Questions

After outlining the problem investigated in the present thesis, we deduce research questions that guide the aforementioned investigations.

Our approach for a central model management is by extracting related information from runtime data in an automated manner. Even though many monitoring solutions (Agrawal et al., 1998; Hoorn et al., 2012; Josephsen, 2007; Ly et al., 2015), have been developed to account for a layered architecture and provide metrics for every EA layer, the primarily task of those systems is the measurement of IT performance and not model extraction. For that reason, it is necessary to develop a system design that is able to extract models from runtime data and aggregate them in a human understandable way. This leads to the following research question:

Research Question 1 (RQ1): *How can a system and a process design look like that automatically reverse engineers models from runtime data?*

The extraction of models from runtime data represent primarily the technical IT landscape including the application- and technology layer. However, an holistic model management of the whole EA also encompasses business-related models. Those models are created manually and remain static most of the time. This consideration leads to the following research question:

Research Question 2 (RQ2): *How to recover business-related models and how to establish a correct assignment of those models to technical layers?*

As stated above, several different information systems are used to build the IT service along the value chain. Each of those information systems describes the IT service from a certain perspective and provides important architecture-relevant information (Farwick et al., 2013). In order to consolidate this scattered information into a central model management, we elaborate a knowledge graph (Binz et al., 2013) that exposes all models and their relationships. This graph must be backed by an efficient meta-model that describes the whole EA in a uniform manner. This raises the following question:

Research Question 3 (RQ3): *How can a meta-model of the EA knowledge graph look like that represents the models from all EA layers and what relationship types need to be defined?*

The EA knowledge graph (Binz et al., 2013) is established by extracting information from several information sources that are used along the IT value chain. At some point the content of the knowledge graph (nodes and relationships) must be created and maintained. In which phase this should be performed is investigated in the following research question:

Research Question 4 (RQ4): *How and where to integrate the concept in the software development process and which stakeholders must be involved?*

As several stakeholders are involved into the creation of IT services, they require different information from the EA knowledge graph. For this reason, it becomes important to identify how information for different perspectives of the IT landscape can be exposed and visually supported. This challenge can be summarized with the following research questions:

Research Question 5 (RQ5): *How can stakeholders be supported in understanding and exploring the EA knowledge graph?*

According to Hevner et al. (Hevner et al., 2004), our concepts must be evaluated based on practitioner feedback to prove the utility of the solution in an industrial context. In order to evaluate our approach, its practical application is subject of the final research question:

Research Question 6 (RQ6): *What are the benefits and shortcomings of the proposed solution? What additional use cases can be addressed?*

1.3. Research Design

Information systems (IS) is an "applied" research discipline, which frequently leverages theory from other disciplines, such as economics, computer science, and the social sciences, to solve problems at the intersection of IT and organizations (Peppers et al., 2006). The IS research community established a wide range of research methods to ensure rigor. They can be distinguished by their degree of formalisation and by the underlying research paradigm (behavioural-science or design-science) (Frank, 2006; Schreiner et al., 2015).

For this research endeavor the research goals are motivated from practice and will therefore be approached together with industry representatives if appropriate. The goal of this thesis is to develop artifacts which implies the application of the design-science paradigm (Hevner et al., 2004). Design-science research is based on two important assumptions, according to Frank (Frank, 2006). First, the design of artifacts can be a sophisticated task that contributes to the development of new knowledge on a scientific level. Second, the scientific design of artifacts is supposed to require a specific research method. Hence, the goal of the design-science paradigm is utility (Hevner et al., 2004) which is achieved by developing innovative artifacts to solve relevant problems by applying rigorous research methods. Therefore, it can be differentiated from the behavioral-science paradigm whose goal is truth (Hevner et al., 2004), achieved by developing and validating theories.

In order to improve the research quality and significance of research results, mixed or multi-method designs are suitable including prototyping, reference modelling, field experiments and case studies (Mingers, 2001; Venkatesh et al., 2013).

In the following sections, we detail how we adapted the design-science research framework to our research process described in this thesis (see Figure 1.2). Furthermore, we want to point out a specific technique, called *qualitative content analysis* (Mayring, 2010) that is mostly applied for analyzing and interpreting textual data. We used this method to analyze the result of our interview series described in Section 7.4.

1.3.1. Design-Science Research

In general, the design-science paradigm is concerned with the creation and evaluation of innovative IT artifacts that solve identified organizational problems. In this scope, the design process itself is an iterative sequence of activities that produces several outputs (product or artifact). The subsequent evaluation of the output provides important feedback information that can be used to improve both the quality of the output and the design process. This loop is iterated a number of times before the final design artifact is generated (Markus et al., 2002).

According to Figure 1.2, the environment represents the problem space in which an artifact is applied and assessed. It refers to people, organizations, and technologies which drive the design of an IT artifact. We identified, people face the challenge to collect the required information to document and to manage the current IT landscape in different level of details. This challenge together with deficient stakeholder collaboration leads to weak EA knowledge management. This, however, is especially required for organizations

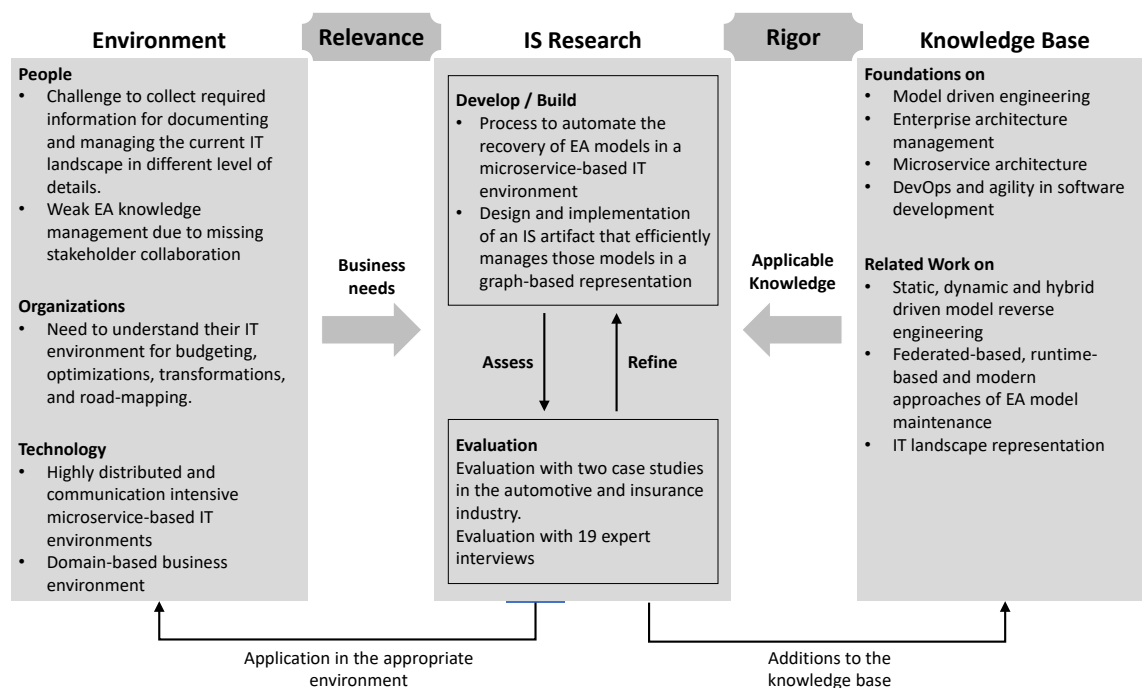


Figure 1.2.: Design science research framework by (Hevner et al., 2004) adapted to the present thesis' contribution

to understand their IT environment for optimizations, transformations, and road-mapping purposes. After analyzing the results of our online survey (Kleehaus et al., 2019b), we realized that this challenge is even more present in microservice-based IT landscapes. The derived requirements detailed in Chapter 4 represent the business needs and thus ensure relevance of the designed IT artifact.

The other driving force of design science is the knowledge base which ensures rigor of the IT artifact's design. It encompasses foundations which is addressed in Chapter 2 and related work summarized in Chapter 3. In this sense, we identified model driven engineering, EA management, microservice architecture, DevOps and agility in software development as the foundation in this research. In addition, we have included related work on model-driven reverse engineering, automated EA model maintenance and different approaches for representing IT landscapes to identify a clear research gap which we address with this thesis.

The core component of the design-science framework is the creation and evaluation of the designed artifact that meets the identified business needs with support of the applied knowledge base. This thesis' core IT artifact is the design and prototypical implementation of an information system that automates the recovery of EA models in a microservice-based IT environment and efficiently manages those models in a graphed-based representation that delivers up-to-date information about each phases of the IT value chain (cf. Chapter 5 and Chapter 6). A fundamental property of the design-science framework is its iterative nature, i.e. the IT artifact should be designed, assessed and refined in multiple iterations

against the business needs in general, and concrete requirements in particular. Chapter 7 of this thesis elaborates on how our IT artifact was evaluated and which different evaluation methods, e.g., case studies and expert interviews were applied.

In this scope, Hevner et al. (Hevner et al., 2004) defines seven principles that support researchers to understand the requirements of effective design-science research. In the following, we outline these guidelines briefly and express how they are addressed in this thesis:

Design as an artifact Artifacts in research are constructs, models, methods, or instantiations that addresses important organizational problems. According to design science research the design and creation of such a purposeful artifact represents a central aspect and one of the primarily task in this research endeavour. Following this aspect, in this thesis we established a framework which includes the prototypical implementation of an automated recovery of EA models, as well as a collaborative process to extend the recovered IT landscape with business-related information. The aim of the tool *MICROLYZE* is to support the management of rapid-changing microservice-based IT landscapes by continuously analysing runtime information and to provide interactive and tailored visualizations as modeling outcome.

Problem relevance Hevner et al. states in his second guideline that information system research has to develop technology-based solutions that address important and relevant business problems. The recovery and management of EA models is an omnipresent problem that concerns the research community (Buckl et al., 2011; Farwick et al., 2012a; Hauder et al., 2012; Holm et al., 2014; Roth et al., 2013a). In (Kleehaus et al., 2019b), we identified that this problem has been becoming even worse with microservice-based IT landscapes and rapidly changing environments due to agile practices. In order to find a different approach how to master this problem, we analyzed related problems to model driven reverse engineering, model management as well as model visualization (cf. Section 3.1, 3.2, 3.3). After elaborating a solution (Kleehaus et al., 2020, 2019a, 2018b) that incorporates the analysis of runtime information for model recovery and model management, we found two case study partners and 19 interview partners that face the same problem (cf. Section 7.2, 7.3, 7.4). This again represents the great importance of the problem and the research gap in this area accordingly.

Design evaluation In the third defined guideline, Hevner et al. make clear that only via well-executed evaluation methods, the utility, quality, and efficacy of a design artifact can be demonstrated. In order to address this guideline entirely, we conducted multiple evaluation methods. *"Because design is inherently an iterative and incremental activity"* (Hevner et al., 2004), we first implemented a preliminary version of *MICROLYZE* that mainly aim at analyzing runtime information for extracting EA models. The experience we made with the prototype and the feedback that we collected from practitioners initiated our next design phase as discussed in Chapter 4. For evaluating the main design artifact, we apply observational methods in Chapter 7. In this scope, we carried out two case studies,

that allowed us to study the artifact in depth in two practical environments and with industrial data. Additionally, we carried out an interview series with 19 experts in order to get feedback for our concepts from a broader audience. We analyzed the feedback via applying qualitative content analysis techniques (Mayring, 2010).

Research contributions The following guideline aims at providing *"clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies"* (Hevner et al., 2004). The core contribution of this thesis is the design and development of *MICROLYZE* and the corresponding processes and algorithms for analyzing runtime data to extract EA models, as well as the collaborative approach for enhancing runtime data with business-related information to fully support the management of models along the IT value chain. Beside the technical concerns of *MICROLYZE* described in Chapter 5 and 6 regarding how the recovered models are maintained, queried and visualized, as well as how the concept is integrated into the IT environment, we also identified in Section 6.2 core user groups that need to be involved in the proposed concept. Technical feasibility and utility of the prototype is shown in an empirical assessment in the context of case studies and interview series. The table in Section 1.4 summarizes the contributions of this thesis.

Research rigor The research rigor is an important guideline, as it highlights that *"design-science research relies upon the application of rigorous methods in both the construction and evaluation of the designed artifact"* (Hevner et al., 2004). The foundations of model reverse engineering, EA management, microservice-based IT landscapes and related fields for monitoring those environments were extensively studied. We report on the state-of-the-art in model recovery, model management and model visualization. Based on this, we derive a conceptual framework for model management of microservice-based IT landscapes along the IT value chain and subsequently derived concrete requirements for a respective approach. Afterwards, we precisely describe the used meta-modeling, model transformation, and model visualization techniques, applied software engineering patterns and software engineering tools. Finally, we discuss how we performed the different evaluation methods, and provide a detailed description on the respective results and findings.

Design as a search process Hevner et al. consider the design of an artifact as a continuous search process, i.e. *"The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment"* (Hevner et al., 2004). This research is an iterative process that can be roughly divided into three phases. In the first phase, we obtained an understanding of current practices and point out problems in the domain of model recovery, model management and model visualization in the context of EA and microservice-based IT landscapes. In the second phase, we searched for alternative ways to recover and manage models in an automated manner. In this phase, we experiment with analyzing log files and other runtime information exposed by log collectors and APM tools, as it was investigated by Farwick et al. (Farwick et al., 2013) that monitoring data could be a promising information source for automating model maintenance. We identified which models of the related EA layers can be extracted from

runtime data and which models still remain hidden. During the second phase, we created initial prototypes (Janietz, 2018; Kleehaus et al., 2018b; Schäfer, 2017) and use cases (Kleehaus et al., 2018a) with support of the industry. After it turned out that the concepts and prototypes elaborated in phase two seemed to be promising solution artifacts, the third phase was initiated. Based on the industry feedback, algorithms, processes and components were improved and finally integrated in a coherent solution design as a single artifact named *MICROLYZE*. In this phase, we also elaborated a concept of how to collect further EA relevant information that cannot be extracted out of runtime data, as well as where to integrate the artifact in a common development process.

Communication of research Finally, Hevner et al. proposes to present research to both, "*technology-oriented as well as management-oriented audiences.*" (Hevner et al., 2004). We present our research and preliminary results on scientific conferences in order to reach technology-oriented audience and to receive qualitative feedback. We chose conferences with a focus on Information Systems (IS). According to management-oriented audience, we take part of the Software Campus and also received industry feedback during the case studies and interview series. A list of the main publications can be found in Section 1.5.

1.3.2. Qualitative Content Analysis

An important instrument we use for analyzing feedback collected from industry partners about the created research artifact is *qualitative content analysis* (Mayring, 2010). This techniques were developed in the 1980s in a research project to handle huge amounts of data and does not only support counting of textual components, but also provide interpretative features (Ulich, 1985). The basic idea of the qualitative content analysis technique is to conceptualize the process of assigning categories to text passages as a qualitative-interpretive act, following content-analytical rules. In this scope, also quantitative analysis approaches are applied that count the frequencies of categories. Hence, the qualitative content analysis can be defined as a mixed methods approach (Mayring, 2012). The basic principles of the methodology is detailed in the following Sections.

Basic Principles and Definitions

The basic approach of qualitative content analysis is to retain the strengths of quantitative content analysis and to develop techniques of systematic, qualitatively oriented text analysis. This will be explained more closely in the following.

- **Systematic and rule-bound procedure:** Content analysis follows a defined procedure that is based on rules laid down in advance. However, as Mayring stated (Mayring, 2010), content analysis is not a standardized instrument that always remains the same. It must be adapted to suit the particular material at hand. This is defined in a procedural model in advance, which determines the individual steps of analysis. In addition, every analytical step and decision in the evaluation process is based on a systematic rule. The definition of content-analytical units (recording units, context units, coding units) entails an approach how the material should be analyzed

and in what sequence, as well as what conditions must be obtained in order for an encoding to be carried out. It is important that such units are theoretically well founded, in order to allow other analysts to comprehend the logic and method of the analysis. In general, the system should be described in such a way that another interpreter may carry out the analysis in a similar way.

- **Content-analytical units:** A central aspect of content-analytical procedures is that the material is not interpreted as a whole but divided into segments. The categories are assigned to segments of text. This segmentation has to be defined in advance. The segmentation rules, which are also called content-analytical units can be differentiated into the following units (Krippendorff, 2004): 1) The *Coding Unit* determines the smallest component of material which can be assessed and which may fall into one category. 2) The *Context Unit* determines the largest text component, which can fall into one category. 3) The *Recording Unit* determines which text portion should be analyzed, like interviews, articles, etc.
- **Definition of Categories:** The category system constitutes the central instrument of analysis. It aims to structure the coding- and context units into categories and concretize the objectives of the analysis. Even though creating categories is not an easy task (Krippendorff, 2004), the qualitative content analysis provides methods that guide the synthetic construction of categories. Overall, working with a category system is an important contribution to the comparability of findings and the evaluation of analysis reliability.
- **Forms of Interpretation:** A number of concrete qualitative content analysis techniques are differentiated which are based on the basic processes of summary, explication and structuring. The analysis technique *Summary* aims to create a comprehensive overview of the base material via abstraction and reduction the material in such a way that the essential contents remain. Regarding *Explication*, the objective is to provide additional material on individual incomprehensible text components in order to increase the understanding, explaining, and interpreting the particular passage of text. Last but not least, by means of the analysis technique *Structuring* particular aspects of the material is filtered to assess it according to certain criteria.
- **Integration of Quantative Steps:** As already mentioned before, efforts are made to combine qualitative and quantitative methods (Mayring, 2012). Quantitative steps of analysis gain importance when generalization of the results is required. For instance, counting how often a category occurs may give added weight to its meaning and importance. A precisely based qualitative assignment of categories to a certain material can also be supplemented by more complex statistical evaluation techniques, as far as these are appropriate to the purpose of analysis.
- **Quality Criteria:** The assessment of results according to quality criteria such as objectivity, reliability and validity is important in qualitative content analysis. In general, the validity denotes the trustworthiness of the results, to what extent the results are true and not biased by the researchers' subjective point of view (Runeson

et al., 2008). There exist different aspects of validity classification and threats to validity in the literature. We chose a classification scheme which is also used by Yin (Yin et al., 2003) and Runeson (Runeson et al., 2008) and similar to what is usually used in controlled experiments in software engineering (Wohlin et al., 2012). This scheme can be summarized as follows: 1) The aspect of *Construct Validity* reflects to what extent the operational measures that are studied really represent what the researcher have in mind and what is investigated according to the research questions. For instance, if the constructs discussed in the interview questions are not interpreted in the same way by the researcher and the interviewed persons, there is a threat to the construct validity. 2) *Internal Validity* is of concern when causal relations are examined. It is the approximate truth about inferences regarding cause-effect or causal relationships. For instance, if one factor affects an investigated factor there is a risk that the investigated factor is also affected by a third factor. If the researcher is not aware of the third factor there is a threat to the internal validity. 3) The *External Validity* validates to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people or are of relevance for other cases. In contrast, internal validity is the validity of conclusions drawn within the context of a particular study. 4) Regarding the aspect of *Reliability*, it is concerned to what extent the data are dependent on and the analysis results are biased by the specific researchers. In general, another researcher who conduct the same study should come to the same result. If it is not clear how to code collected data or if interview questions are unclear, this would threaten the aspect of reliability.

The above listed aspects of quantitative content analysis is regarded to be the foundation for a qualitative oriented procedure of text interpretation. In this scope, Mayring (Mayring, 2010) developed a number of procedures of qualitative content analysis amongst which two approaches are central: inductive category development and deductive category application.

Deductive Category Application

The content-analytical method *Deductive Category Creation* aims to extract certain structure from the material. This structure is applied on the material in the form of a category system. All text components addressed by the categories are then extracted from the material systematically. The procedure is deductive because the category system is established before coding the text. In general, the structuring procedure can be described as follows: 1) The particular categories are defined and it is precisely determined which text components belong in a given category. 2) Anchor samples, which are concrete passages belonging to the specific categories help to characterize those categories. 3) If there are problems of delineation between categories, rules must be determined for the purpose of unambiguous assignment to a particular category.

Subsequently, the following steps are carried out. Initially, text extracts are taken from the material to check whether the categories are applicable and whether the definitions, anchor samples and encoding rules make categorical assignment possible. This process

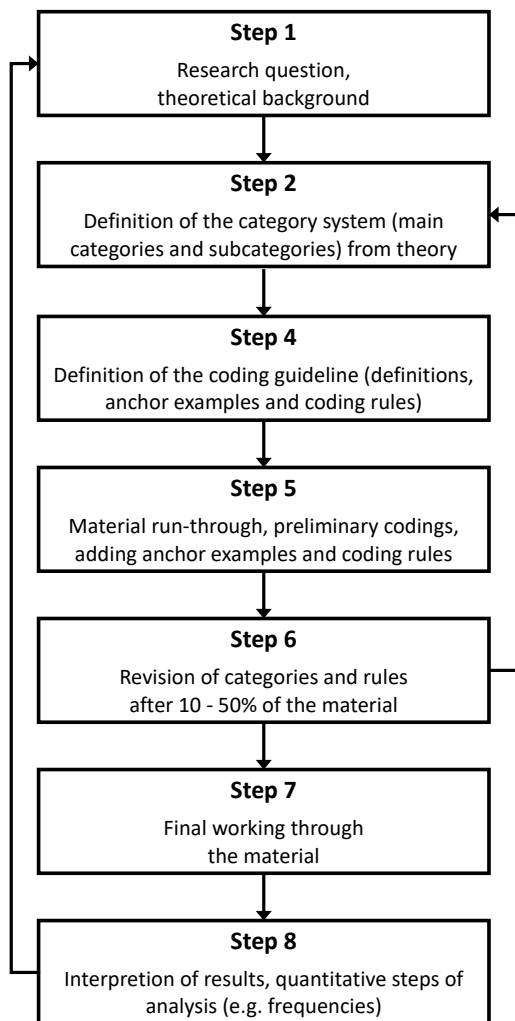


Figure 1.3.: Steps of deductive category assignment (Mayring, 2010)

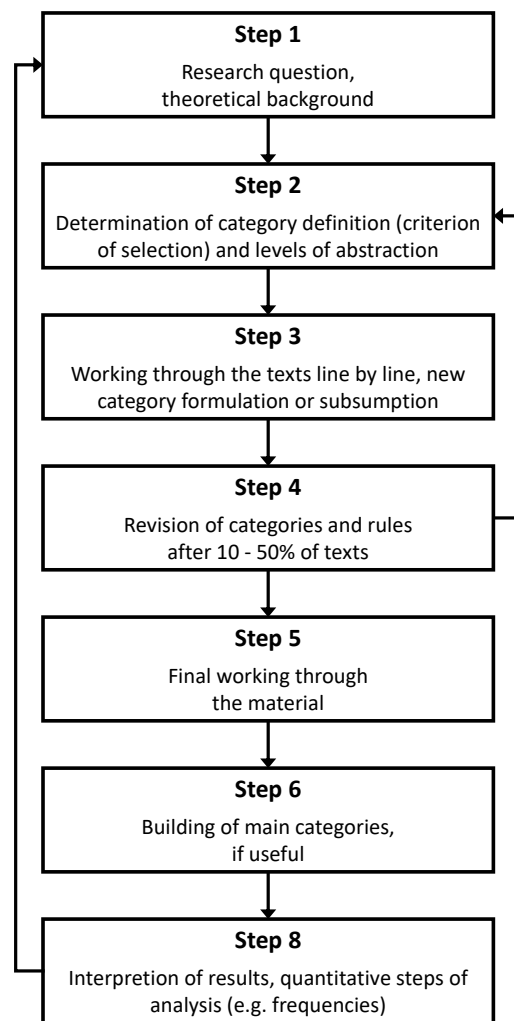


Figure 1.4.: Steps of inductive category development (Mayring, 2010)

results in a revision and partial reformulation of the category system and its definitions. After the revision process is finalized, the main material run-through can be performed. The result must then be summarized and analyzed according to category frequencies and contingencies interpretation. Figure 1.3, illustrates the process of structuring content analysis in detail.

Inductive Category Formation

Inductive category formation is a central process within the approach of Grounded Theory (Strauss, 1987), which in this context is called "open coding". It is used to identify categories and subject areas within the material at hand. The process model shown in Figure 1.4 describes the particular steps of inductive category development. We will now explain the

process in more detail:

Before the analysis can begin, the themes of categories to be developed must be defined previously. There has to be a criterion for the selection process in category formation. This is a deductive element and is established within theoretical considerations about the subject matter and the aims of analysis. After this is decided the inductive category formation process can be performed.

First of all, the material is worked through line by line. In case the text passage is fitting the category definition, a specific category has to be constructed. A term or short sentence, which characterizes the material as near as possible serves as category label. This is an iterative process, i.e. the next time a passage fitting the category definition is found it is either subsumed under the known categories, or a new category has to be formulated.

After a large part of the material has been processed (ca. 10 - 50 %) and no new categories have been found, the whole category system has to be revised. Hence, it must be checked, if the logic of categories is clear, no overlaps occurred and if the level of abstraction is adequate to the subject matter and aims of analysis. The category definition has to be changed eventually. If too many categories had been formulated which results to an unclear object area, the level of abstraction should be defined more general. As stated by Mayring (Mayring, 2010), a rule of thumbs is a set of ten to thirty categories in order to achieve a good overview.

Furthermore, sometime it is helpful and expedient to bring the set of categories into an order by formulating main categories. This step could be processed more inductively by only enhancing the level of abstraction in the sense of summarizing. On the other hand, this step could also be processed more deductively by introducing theoretical considerations in formulation main categories.

After the analysis a set of categories to a specific topic exist. The further analysis can go different ways: 1) The whole system of categories can be interpreted in terms of aims of analysis and used theories. 2) The links between categories and passages in the material can be analyzed quantitatively, like counting those categories that occur most frequently in the material.

1.4. Contribution of this Thesis

In the following Section, we summarize the contributions of this thesis and relates them to the research questions raised in Section 1.2. Figure 1.5 illustrates this correlation. In addition, we detail which concrete (peer-reviewed) publications and student theses were created in the context of this thesis that refers to the main contributions.

The first contribution of this thesis is a comprehensive introduction and description of concepts related to model-driven reverse engineering, enterprise architecture management, microservice-based IT landscape as well as an overview of DevOps processes (cf. Chapter 2). Furthermore, we conduct a state-of-the-art analysis of current approaches for model management in three research areas, namely model-driven engineering, EA model maintenance and IT landscape representation (cf. Chapter 3). We elaborate on how they are currently applied as information systems for the management of models along

the IT value chain, and from which shortcomings they suffer in this context. This leads to the identification of a research gap which was already briefly introduced in Section 1.1. Based on this as well as on related research, we define a conceptual framework for a central model management in microservice-based IT landscapes. This framework forms the foundation for the identification of requirements (cf. Chapter 4).

The second contribution is the conceptual and algorithmic design, as well as the prototypical implementation of *MICROLYZE*. The conceptual design phase includes the development of a conceptual meta-model capturing all concepts required for representing the IT landscape in the scope of EAM (cf. Chapter 5).

Research Questions	RQ1		RQ3	RQ5	RQ2	RQ3	RQ4	RQ6
	Chapter 2	Chapter 3	Chapter 4	Chapter 5	Chapter 6	Chapter 7	Chapter 8	
Chapter	Chapter 2	Chapter 3	Chapter 4	Chapter 5	Chapter 6	Chapter 7	Chapter 8	
Research Result and Artifacts	Concepts and terminology	State of the art analysis	Conceptual framework	Meta-model analysis of APM solutions	CD pipeline integration	Case Study Report I	Critical reflection	
		Identified research gaps	Requirements analysis	Meta-model and model transformation	Configuration file design	Case Study Report II	Limitations	
				Model recovery algorithms	JSON schema validation	Interview Series	Future research	
				Graph-based representation of models	References to different data sources			
				Model communication visualizations	Model dependency visualizations			
Publications		[Kleehaus et al. 2016] [Kleehaus et al. 2019b]		[Schäfer 2017] [Janietz 2018] [Machner 2019]	[Achhammer 2019] [Corpancho 2019]	[Achhammer 2019] [Machner 2019]		

■ Peer-reviewed research paper ■ Master thesis

Figure 1.5.: The main contributions of this thesis.

Furthermore, we describe how we translate the meta-model into a graph-based representation in order to empower end-users to query the EA models in an efficient way. The definition of several visualizations addresses different stakeholder concerns. Moreover, we present the design of configuration files that establish the relationship between business- and technical-related EA layers and how we validate the content of the configuration file during deployment phases (cf. Chapter 6).

The third contribution arises from the evaluation of our approach and prototype in case studies and interview series as described in Chapter 7. The thesis reports on specific findings from integrating the prototype into an industrial environment covering the

insurance and automotive sector. In addition, we demonstrate the tool to practitioners that had the roles of Enterprise Architects, Solution Architects and DevOps Team members. Those findings particularly include open issues and shortcomings of the current prototype. This thesis concludes with lessons learned from the implementation of our approach from which we derive limitations and future work (cf. Chapter 8).

1.5. Preliminary Work

This thesis describes the research project in a comprehensive way for scientists and practitioners. Various interim results were presented and discussed at conferences and workshops. In the following, we briefly describe and list our publications. Afterwards, the related student theses and their contributions to this thesis are briefly presented.

Publications:

- **(Kleehaus et al., 2016): State of the Art Report: Multi-Layer Monitoring and Visualization. Technical report. Munich, Germany**
This publication represents the results of a systematic literature review carried out on the topic of multi-layer monitoring approaches that cover the instrumentation of each EA layer. Additionally, we provide a survey of academic and commercial monitoring tools as well as an overview of different visualization types utilized in monitoring applications.
- **(Kleehaus et al., 2018a): Towards a Continuous Feedback Loop for Service-Oriented Environments. QUATIC. Coimbra, Portugal**
In this paper, we leverage *MICROLYZE* and extend the design to support the continuous delivery of software applications by providing metrics and structural information after each deployment stage i.e. development, test and production. The continuous feedback is provided via a dependency model that represents the current software architecture on early stages. Hereby, each deployment phase and final release are compared against each other in order to uncover inconsistencies in regard to the predefined requirements. The concept was evaluated through quantitative methods in a laboratory experimental setup.
- **(Kleehaus et al., 2018b): MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-based Environments. CAISE Forum. Tallin, Estonia**
In this publication, we introduce the concepts of *MICROLYZE* by presenting a multi-layer microservice architecture recovery approach that reconstructs EA models based on runtime data. The recovery process comprises models from each EA layer as well as the corresponding relationship between those models. In addition, we present a tool support for mapping business activities with request transactions in order to recover the correlation between the business and application layer models. This developed tool also visualizes the model dependencies based on a adjacency matrix. The tool was evaluated within an laboratory experimental setup. In this scope, we investigated technical aspects of our solution in a controlled environment.
- **(Kleehaus et al., 2019a): IT Landscape recovery via Runtime Instrumentation for Automating Enterprise Architecture Model Maintenance. AMCIS. Cancun, Mexico**
In this research, we present a concept design, and related processes for recovering EA models by combining runtime data with architecture information that reside in

federated information sources. It represents a further extension of *MICROLYZE* to empower stakeholders to explore EA information from different perspectives, which supports new use cases and analysis capabilities. We evaluated the prototype during a case study in a big German insurance company.

- **(Kleehaus et al., 2019b): Challenges in Documenting Microservice-based IT Landscape: A Survey from an Enterprise Architecture Management Perspective.** EDOC. Paris, France

In order to analyze the status quo in the adaption of microservices and what challenges organizations face while documenting microservice-based IT landscapes from an EAM perspective, we conducted a survey among 58 IT practitioners in German market. We identified eleven challenges and synthesized them into four categories namely content-, assignment-, tooling-, and business-related challenges.

- **(Kleehaus et al., 2020): Recovery of Microservice-based IT Landscapes at Runtime: Algorithms and Visualizations.** HICSS. Hawaii

In this publication, we introduce the design of two algorithms that run in *MICROLYZE* that 1) recover the architecture of microservice-based IT landscapes 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 uncover from runtime data. We evaluated our prototype by analyzing the monitoring data of a big automotive company.

- **(Kleehaus et al., 2021): Automated Enterprise Architecture Model Maintenance via Runtime IT discovery. Architecting the digital transformation.** Munich, Germany

In this work, we detail the linked enterprise topology graph that represent the persistence layer of *MICROLYZE*. The graph exposes all recovered models which empowers users to query the microservice-based IT landscape via an uniform language and allows them to explore information from a static and dynamic perspective. We evaluated our prototype by implementing it in a big German retailer and conducting interviews with 17 experts from two different companies.

Supervised master theses:

- **(Schäfer, 2017): Eine prototypische Implementierung zur Erkennung von Architekturänderungen eines verteilten Systems basierend auf unterschiedlichen Monitoring Datenquellen.** Technical University of Munich. 2017

Schäfer investigates in this thesis different approaches of architecture recovery and developed a prototype for reverse engineering microservice-based IT landscapes. In addition, Schäfer elaborates a method to link business process activities with technical requests to recover which microservices are responsible to process business transactions.

- **(Graeff, 2017): Enhancing Business Process Mining with Distributed Tracing Data in a Microservice Architecture.** Technical University of Munich. 2017

Graeff picks up the concept of linking business process activities with technical requests and elaborates how to enhance process mining with performance indicators obtained from application monitoring tools. The developed prototype empowers users to detect correlations between user behaviour from the business layer and system performance occurring in the application layer.

- **(Hoyos, 2017): Interactive Visualizations for supporting the analysis of distributed services utilization. Technical University of Munich. 2017**

Hoyos elaborates different visualization approaches for the obtained IT landscape recovery result. Different views providing information for various stakeholders are implemented in a prototype visualization tool. Evaluation of the implementation is performed in a lab environment.

- **(Janietz, 2018): Enhancing enterprise architecture models using application performance monitoring data. Technical University of Munich. 2018**

The thesis of Janietz is a first attempt to establish a linked graph for efficiently managing recovered EA models. Furthermore, Janietz elaborates a process for synchronizing recovered EA models with EA models stored and managed in EA tools. The goal is to automate the maintenance of EA models.

- **(Corpancho, 2019): Automated documentation of Business Domain assignments and cloud application information from an application development pipeline. Technical University of Munich. 2019**

The thesis of Corpancho focuses on the automated documentation of cloud applications by integrating the IT landscape recovery concept into a continuous delivery pipeline. In this scope, he developed a configuration file that contains a reference to the business layer in order to empower Enterprise Architects to maintain the allocation of applications in the business context.

- **(Achhammer, 2019): Assessing the Cost and Benefit of a Microservice Landscape recovery Method. Technical University of Munich. 2019**

Corpancho was not able to evaluate his concept in an industrial setting. For that reason, Achhammer elaborates prerequisites and requirements for integrating the concept in an industrial environment. As a result, the updated and enhanced prototype is able to recover and manage EA models that are enriched with business-related information. During a case study in a big German insurance company in combination with a series of interviews the feasibility of the prototype was finally evaluated.

- **(Machner, 2019): Assessing the Cost and Benefit of a Microservice Landscape recovery Method in the Automotive Industry. Technical University of Munich. 2019**

Machner investigates techniques and methods to extend the linked graph for managing an arbitrary amount of different recovered EA models. The prototype was implemented in a real world environment situated in the automotive industry. The tool was tested and evaluated through expert interviews with EA practitioners.

1.6. Structure of this Thesis

This thesis is divided into eight chapters. Figure 1.5 illustrates the structure of the thesis, the addressed research questions, created artifacts as outlined above as well as related core publications. Each chapter is outlined in the following.

Chapter 1: Introduction motivates the present thesis, details the problem, and derives research questions which guide the present thesis. Further, we describe the taken research design and outline the core contributions.

Chapter 2: Foundations serves the reader with foundations to understand the thesis and refers to literature relevant for the topics investigated.

Chapter 3: Related Work presents a summary of the state-of-the-art in model-driven engineering, automated EA model maintenance and IT landscape representation referring to work of others that influenced our design decisions. Hereby, we outline how our approach differs from the related work representing our research contribution.

Chapter 4: Requirement Analysis characterizes an efficient framework for managing models along the IT value chain, and derives requirements for an information system that recovers models automatically and represents them visually in different perspectives.

Chapter 5: Automated Model Recovery via Runtime Instrumentation embraces methods, techniques, processes and the conceptual design of *MICROLYZE* that realizes the automated recovery of microservice-based IT landscapes by extracting related models and their relationships from runtime data. We detail, how we expose the recovered models in a graph-based representation. In addition, we present a framework for visualizing the models in several perspectives to address important stakeholder concerns.

Chapter 6: Recovery of Business-related Models reveals further methods, processes and conceptual designs for linking business-related models to technical-related models recovered in Chapter 5 and to the model representation from all used information systems. In addition, the Chapter presents concepts of how to integrate *MICROLYZE* into the software development process in order to become part of the continuous deployment strategy.

Chapter 7: Evaluation reports on the setup and results of two case studies in an industrial setting. In addition, we further present qualitative insights from a broader audience and feedback as an outcome of an interview series.

Chapter 8: Conclusion summarizes the thesis' contributions, critically reflects on the contributions and the results, and finally informs about further research.

2. Foundations

This chapter provides important concepts relevant for the present thesis. As shown in Figure 2.1, our solution is developed by bridging the gap between four complimentary fields of research, namely 1) model-driven engineering, 2) enterprise architecture management, 3) microservice-based IT architecture and 4) DevOps processes.

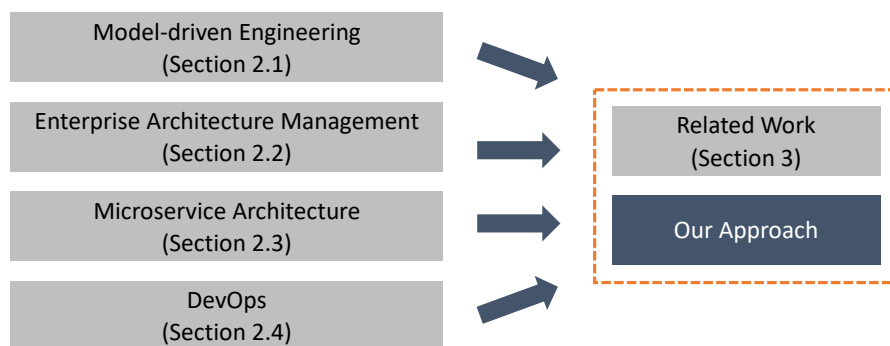


Figure 2.1.: Core topics that represent the foundation of related work and the thesis' approach.

Model-driven engineering is an engineering paradigm that handles models and their transformations as primary artifacts to develop, analyze, and evolve software systems. As the management of models is a core concept we use in the course of this thesis, we will give attention on model-driven engineering.

Enterprise architecture management represents our problem domain in which we observe insufficiencies concerning the management of models in microservice-based IT landscapes. In order to provide a general understanding of the domain, we describe fundamental concepts and principles of enterprise architecture management. In particular, we detail the IT-landscape modelling, as a sub-area of enterprise architecture management, that aims to recover and to maintain the IT-landscape models of an organization.

The microservice architecture build the frame of our research. This architectural style has received much attention in recent years. Even though microservices have several advantages in contrast to monolithic systems, this architecture style also introduces a high level of complexity with regard to model management (Kleehaus et al., 2019a).

In order to automate the recovery and maintenance of models, we leverage the concepts of agile practices, continuous delivery and monitoring. All those mentioned concepts are core elements of **DevOps** and therefore requires our attention.

2.1. Model Driven Engineering

Since decades, models play a core role in various disciplines, including software engineering (Ludewig, 2004). A commonly used reference for the notion and semantics of a model is Stachowiak (Stachowiak, 1973) who states that "*a model represents a relevant subset of a real-world object's properties for a specific purpose.*" Models are typically used to reduce complexity by abstraction and by omitting irrelevant details. The general notion of a model is not limited to graphs or diagrams. The same model may serve for descriptive and prescriptive purposes. This chapter introduces core concepts and technologies for model-driven engineering (MDE) (Brambilla et al., 2012; Völter et al., 2013), which is an engineering paradigm that manages models and their transformations as primary artifacts to develop, analyze, and evolve software systems. Core concepts of MDE are modeling languages, model transformations, model reverse engineering, and models at runtime which are introduced in the Sections 2.1.1, 2.1.2, 2.1.3 and Section 2.1.4 respectively.

2.1.1. Modeling Languages

A model is a simplified representation of reality. The simplification can be applied on sensual, especially optically perceptible objects or in theories. According to Stachowiak (Stachowiak, 1973) it is characterized by at least three features¹:

- Mapping feature: Models are mappings or representations that serve as surrogates of objects in the physical world or of artificial or mental originals.
- Reduction feature: Models capture not all attributes of the original represented by them, but rather limit the scope relevant to their respective model creators and/or other stakeholders. The art of reducing the model to such a purposeful scope is called abstraction.
- Pragmatism feature: Models are not uniquely assigned to their originals in the real-world. They fulfill a replacement function for 1) a particular subject (human or artificial receiver), 2) within particular time intervals, and 3) restricted to particular mental or actual operations, i.e. models serve a special purpose and are a means to an end.

A model can be conceptualized by defining properties of specific concepts in the real-world that are of interest for the modeler. From an object-oriented perspective, each physical object is an object that conforms to an entity. Such an entity has attributes. A special kind of attribute that refers to other objects is called relationship.

A modeling language is a formalism to express models. It comprises definitions of *abstract syntax*, *concrete syntax*, and *semantics* (Brambilla et al., 2012). The abstract syntax describes the structure of the modeling language and specifies the set of modeling primitives along with rules on how to combine them, independent of any representation.

¹We refer the interested reader to Thomas (Thomas, 2005). He presents a comprehensive review of different notions of models and an extensive discussion on the topic.

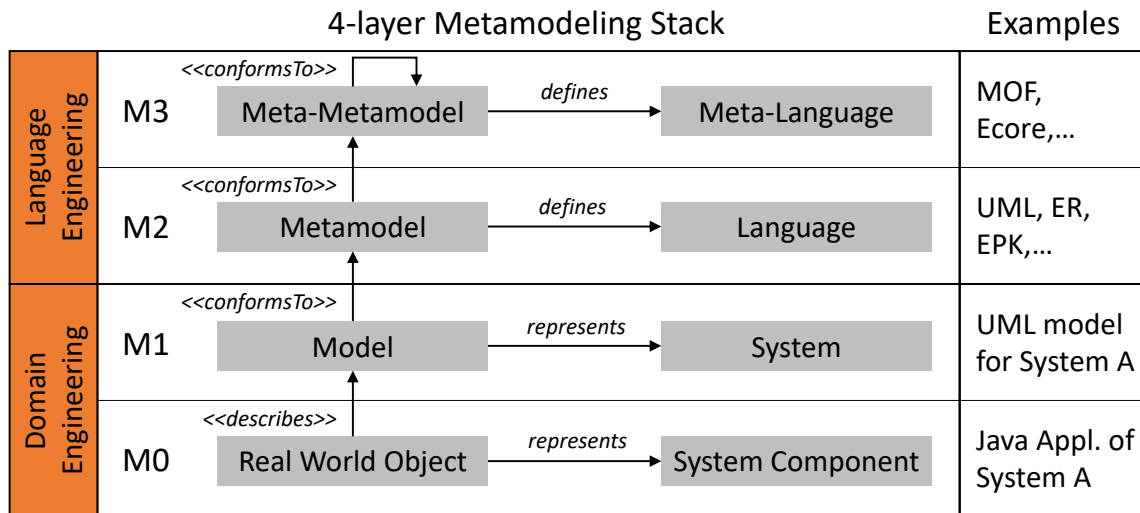


Figure 2.2.: Four-layered meta-modeling stack (Brambilla et al., 2012)

The concrete syntax defines specific representations and can be either textual or graphical. The meaning of the elements defined in the modeling language is provided by the semantics.

The abstract syntax of a modeling language is defined by a so-called meta-model, which describes relevant concepts of the problem domain in terms of its entities and their relationships. The formalisms used to express entities and relationships in a meta-model are meta-meta-model. Similar to concepts found in UML class diagrams (Group, 2017), a meta-meta-model typically provides concepts like (abstract and concrete) classes with typed attributes, class hierarchies through generalization, as well as (un)directed associations among classes.

Current MDE tools mostly employ a four-layer-based modeling stack, which is shown in Figure 2.2. The domain engineering layers M0 and M1 are concerned with building models for a specific domain. The layers M2 and M3 are concerned with language engineering, i.e. building models for defining modeling languages. In general, meta-meta-models like MOF or Ecore provide the meta-language to express modeling languages defined by a meta-model. These meta-models are used to express models of real-world objects. A common notion is that a model on meta-modeling layer M_i conforms to its meta-model on layer M_{i+1} .

2.1.2. Model Transformations

As Brambilla et al. (Brambilla et al., 2012) defines, a model transformation is a process that consumes a set of source models as input and maps these to a set of target models produced as output. Each source and target model conforms to a meta-model. As part of this transformation process, models are either merged (to homogenize different versions

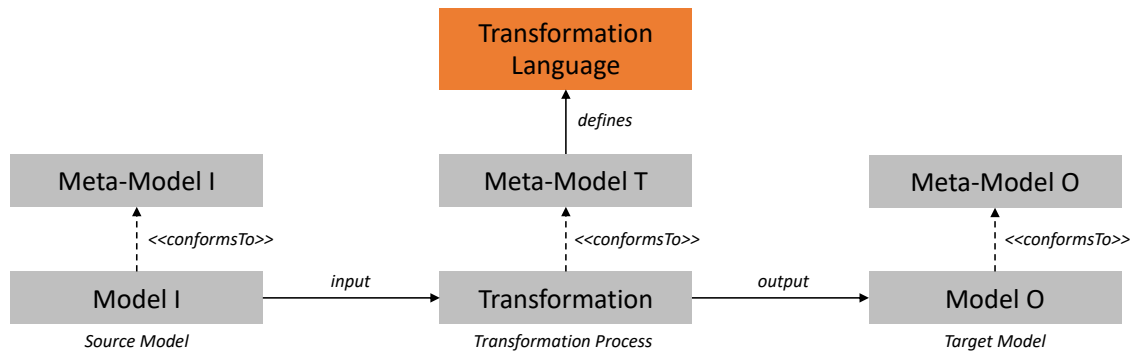


Figure 2.3.: Model transformation schema (based on (Czarnecki et al., 2006))

of a system), aligned (to create a global representation of the system from different views), refactored (to improve their internal structure without changing their observable behavior), refined (to detail high-level models), or translated (to other languages/representations). Figure 2.3 depicts the general schema of model transformations. All these operations on models are implemented either as Model-to-Model (M2M), Model-to-Text (M2T), or Text-to-Model (T2M) transformations. In M2M, the input and output parameters of the transformation are models, while in M2T, the output is a text string. Analogously, T2M transformations have a text string as input and a model as output. Such transformations are typically applied in model-driven reverse engineering (see Section 2.1.3). Transformations are performed as one-to-one, having one input model and one output model (e.g. transformation of a class diagram into a relational model), or one-to-many, many-to-one, or even many-to-many, which is mostly the case in model merge scenarios where the goal is to unify several class diagrams into one integrated view.

Transformations are expressed via transformation languages (Czarnecki et al., 2006). The four-layered meta-model stack described in Section 2.1.1 represents the basis for transformations (M1) and transformation languages (M2). As Figure illustrates a transformation is a model that conforms to a meta-model defining the transformation language. This allows transformations to be used as input and/or output for transformations, which are then called higher order transformations (HOTs) (Brambilla et al., 2012). HOT are used to facilitate the manipulation of transformations. Similar to a normal model, that can be created, modified, and augmented through a transformation, a transformation model can itself be created and modified.

2.1.3. Model-Driven Reverse Engineering

As described above a text-to-model transformation is called model-driven reverse engineering (MDRE). Chikofsky and Cross (Chikofsky et al., 1990) defines MDRE as *"the process of analyzing a subject system to identify the system's components and their inter-relationships and create representations of the system in another form or at a higher level of abstraction."* The term *subject system* represents the end product of a software development process. The main

objective of such model-based representations is to manage the complexity of IT systems and obtain a better understanding of the current state, for instance to correct it, update it, or even completely re-engineer it (Bruneliere et al., 2014). Typically, the process of MDRE start from a system model with a low abstraction level and try to build views at higher abstraction levels. This activity is typically performed using static or dynamic analysis, or a combination of both (hybrid). In general, static analysis techniques extract models from the source code or binary code of a software system without executing them. Dynamic techniques analyse the runtime behavior of software systems during execution. Hereby, the models generation and models transformations is performed automatically based on a transformation language. Figure 2.4 shows this process in detail. It comprise three phases (Brambilla et al., 2012):

1. **Model Discovery:** In the initial phase of MDRE, the low level abstraction of the system represented in static and/or dynamic data is consumed and translated into initial models, without losing any of the information required for the process. The main objective is to quickly switch from the heterogeneous real world to the homogeneous world of models. These initial models are sufficiently accurate to be used as a starting point, but do not represent any real increase of the abstraction or detail levels.
2. **Model Understanding:** During the second phase, the initial discovered models serve as input and are translated into the higher level output models via transformation chains. Thus, the second phase employ manipulation techniques to query and transform the initial models into more manageable representations.
3. **Model Generation:** From this point on, further transformation task can be performed on the abstract models in order to generate low level model representations. For instance, to migrate from a particular procedural technology (e.g. COBOL) to a object-oriented technology (e.g. Java). However, this step is not necessarily required. The MDRE process could only cover the model discovery and the model understanding phase (Bruneliere et al., 2014).

2.1.4. Models at Runtime

The research on models at runtime (also called *models@run.time*) (Bencomo et al., 2019) seeks to extend the applicability of models produced in model-driven engineering approaches to the runtime environment. Those, so called *runtime models* can be defined as abstract representations of a system, including its structure and behaviour, which exist in tandem with the given system during the actual execution time of that system.

Furthermore, those models can be used by the system itself, other systems or humans to support reasoning and decision making based on knowledge that may only emerge at runtime and was not foreseen before execution. Users can also use runtime models to support dynamic state monitoring and control of systems during execution, or to dynamically observe the runtime behavior to understand a behavioral phenomenon (Blair et al., 2009).

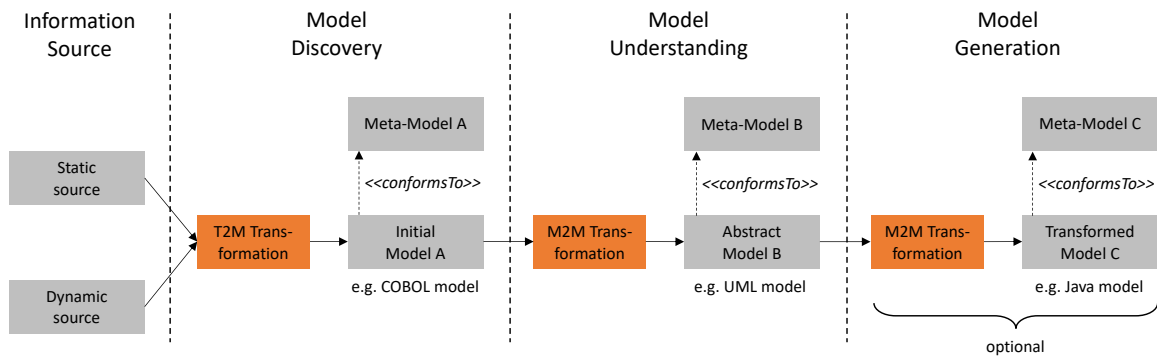


Figure 2.4.: Model-driven reverse engineering process (based on (Brambilla et al., 2012))

Runtime models represent a reflection layer (Maes, 1987), which is causally connected with the underlying system so that every change in the runtime model leads to a change in the reflected system (Blair et al., 2009). Significant advances have been made in recent years in applying this concept, most notably in self adaptive systems (Bencomo et al., 2019; Bennaceur et al., 2014). Those systems are able to autonomously modify its behavior at run-time in response to changes in the environment (Cheng et al., 2009; Lemos et al., 2013).

2.2. Enterprise Architecture Management

The IT architecture is an essential determinant for the future viability of companies. It highly contributes to the efficiency of the business model and business processes. Companies need a well thought-out plan to be prepared for future requirements. This is taken over by an EA. In the following sections, we introduce concepts which are central for EAM endeavors and illustrate the relationships between these concepts. After introducing our perspective of an EA as a whole, we revisit the ArchiMate notation language that supports the description and illustration of specific views on the EA. This language serves as an important tool for IT-landscape modelling that plays a significant role in the discourse of this thesis.

2.2.1. Building Blocks of Enterprise Architecture

An enterprise architecture creates an holistic view of the entire company by embracing all the major business and IT structures, as well as the associations that exist between them. This includes e.g. buildings, machines, telecommunication networks, information systems, but also organisational procedures and related information flows. It serves as basis for describing both the business and IT, as well as the connections between them. Thereby, dependencies and effects of changes in business and IT become transparent (Hanschke, 2010). A well established EA supports the management to evaluate the possibilities of the company and to develop it further in a targeted manner.

Architectures refer to logical constructs used in representing and interpreting things in the real life and their behavior. Architecture is concerned with understanding and defining the relationship between the users of the system and the system being designed itself. Based on a thorough understanding of this relationship, the architect defines and refines the essence of the system, i.e. its structure, behavior, and other properties (M. Lankhorst, 2017). A general definition of architecture is given in IEEE 1471-2000/ISO/IEC 42010:2007 (Society, 2000) as *"the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principle guiding its design and evolution."*

An EA is divided in different layers and cross-functional aspects which exert influence on all layers as many EA researchers (Buschle et al., 2012; Doucet et al., 2009; Matthes et al., 2020; R. Winter et al., 2007; Wittenburg, 2007) and practitioners (Hanschke, 2010) agree. Those layers can be clustered into three main areas that comprises business-related, application-related and infrastructure-related aspects of an IT landscape. Those aspects are depicted in Figure 2.5 and detailed in the following:

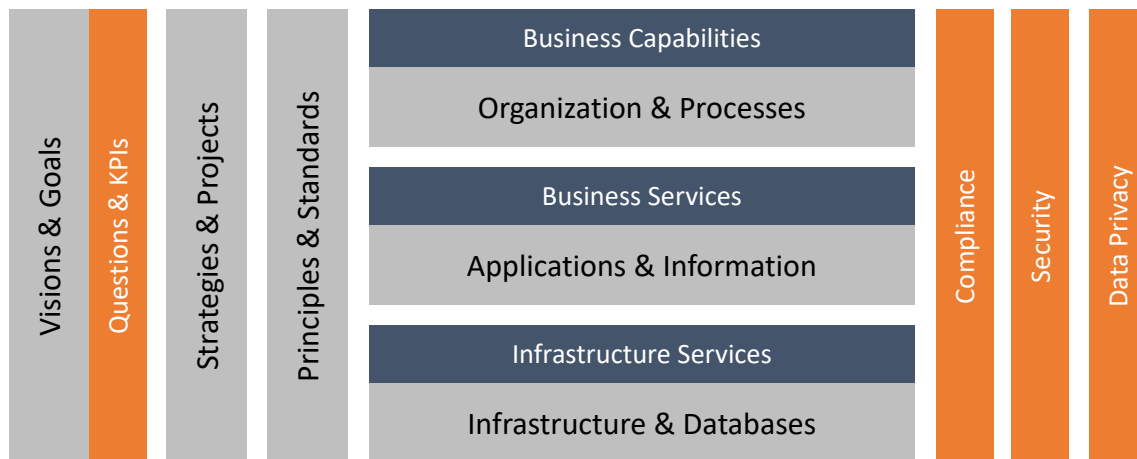


Figure 2.5.: Fundamental layers of an enterprise architecture

- **Business Capabilities** describe core competencies enterprises offer. They represent functional building blocks of the business architecture to support the business model and the business strategy. Each business capability is independent from other business capabilities and realized by combining different elements of the EA (Mannmeusel, 2012)
- **Organization and Processes** are used to realize and implement business capabilities. The organizational structure tends to be static while processes describe behavioral aspects for value creation (Weske, 2007). In general, business processes are collections of structured activities which in a specific sequence produce a service or product for particular customers. At this level, coarse-grained information is modeled such as business objects, e.g. customer, product, or contract.

- **Business Services** describe the provision of tangible products or services to accomplish certain processes executed by the organization.
- **Applications and Information** provides the required data for describing the enterprise's application landscape. This comprises applications, their data and interfaces or information flows. The application layer serves as the bridge linking the business architecture with the infrastructure architecture. The links into the business architecture shed light on the IT support and required information to accomplish a business service. This embraces not only in-house IT but also applications hosted by a third party, e.g. through Software as a Service (SaaS). The technical implementation of applications and interfaces is documented by assigning elements from the infrastructure architecture to each application.
- **Infrastructure Services** are technical services used to provide applications and information, e.g. access to the Internet, the provision of an application server, etc. Similar to the applications, procurement of third-party services has become a commodity through Infrastructure as a Service (IaaS) and Platform as a Service (PaaS).
- **Infrastructure and Databases** is the technical backbone of an organization and includes for instance network elements like routers, servers and clusters. Databases on the other hand are the physical data storage for the most valuable asset today's knowledge-intensive organizations possess—information.

These layers are influenced by cross-functional aspects which are driving forces to any of these layers. These aspects are:

- **Visions and Goals** are derived from the enterprise strategy. Visions are desired states of the reality in the far future. Visions are operationalized to goals.
- **Questions and KPIs** help to measure the accomplishment of goals in a quantitative manner. Metrics help to formalize questions and key performance indicator (KPI) values can provide a means to control efforts.
- **Strategies and Projects** are instruments to implement change and to create innovation. Especially, projects transform the EA whereas strategies provide courses of action to achieve goals (Buckl et al., 2010).
- **Principles and Standards** define guidelines and borders for the EA. Thus, they can be viewed as constraints for the solution space (Buckl et al., 2010).
- **Security** aspects are relevant to prevent e.g. industrial espionage, data loss, and contempt of private and personal data.
- **Compliance** relates to influencing factors like regulatory changes or audit trails.
- **Information privacy** is the privacy of personal information and usually relates to personal data stored on information systems. The need to maintain information

privacy is applicable to collected personal information, such as medical records, financial data, criminal records, political records, business related information or website data (Smith et al., 2011).

2.2.2. Management of Enterprise Architectures

The EA is administered by the EAM function. Gartner (Gartner, 2008) defines EAM as *"the process of translating business vision and strategy into effective enterprise change by creating, communicating and improving key principles and models that describe the enterprise's future state and enable its evolution."*

In general, EAM helps to master the complexity of the IT landscape and to develop the IT landscape strategically and business-oriented (Hanschke, 2010). It embraces all processes for documentation, analysis, quality assurance, the creation of current, planned, and target IT landscapes to align, plan, and control the evolution of the application landscape. The creation and maintenance of these views is part of the landscape management (Matthes et al., 2020).

Through the systematic and clear presentation of the business architecture and the IT landscape in its interaction, connections and dependencies become visible and understandable. This supports the access of knowledge, recognition of trends, as well as identification of optimization potential. Gaining transparency of the IT landscape is a prerequisite for mastering the EA complexity. Thereby, it is not sufficient just to collect the information ad-hoc and store it in the EA database for later use. Information may be outdated shortly after collecting, which lead to analysis on outdated information. The main problem – ensuring the availability of up-to-date information – arises due to the different information suppliers needed and the high effort for a consistent set of information objects. As EAM concerns different functions from both IT and business areas, Business Process Modelers, Product Owners, IT Architects, Project Managers, Developers etc. must supply the EAM process with the information needed for addressing the concerns of different stakeholders.

Hence, EAM is an iterative, incremental, and continuous process as stated by researchers like Ahlemann et al. (Ahlemann et al., 2012) and Buckl (Buckl, 2011), as well as practitioners, e.g. Hanschke (Hanschke, 2010), Keller (Keller, 2017), and Niemann (Niemann, 2005). Initially, EAM starts by motivating an EA endeavor by convincing all required stakeholders of the meaningfulness of EAM and long-term benefits for the entire organization. In particular, top management support is considered essential for successful EAM endeavors (Young et al., 2013). Next, the developed models and concepts represent a communication baseline that conveys made decisions, long-term benefits and further course of EA transformation projects. In this phase, the EAM team should show the turnover for each individual stakeholder. In the third phase, the EAM team reflects their practices, outcomes and behavior. It is essential that the delivered artifacts are challenged by constant feedback. This especially emphasizes the human and social aspect of EAM. A successful EAM initiative relies on continuous collaboration between the EAM team, their stakeholders as well as top management support (Roth, 2014).

2.2.3. Modelling Enterprise Architectures

IT-landscape modelling, as a sub-area of EAM, tries to discover and document the IT-landscape of an organization. It aims to generate and maintain a virtual representation – a digital twin – of the whole organization (Tao et al., 2018). This digital twin represents the baseline for the Enterprise Architects to address different concerns of the stakeholders and to assess the as-is EA. In addition, the Enterprise Architects use the models to create the to-be landscape and the practical interventions to usher the present landscape towards the to-be status. It also comprises all the supporting measures to help direct the evolution of the IT landscape. This is important for decision support especially when it comes to IT transformation and modernization. A comparison of the as-is and to-be landscape allows to disclose the progress of the IT transformation endeavour (Hanschke, 2010).

In general, the description of an EA is an explicit artifact formalized in models and views. These formalisms are provided by architecture description languages (ADLs). An ADL definition is provided by (Clements, 2003) *“A language (graphical, textual, or both) for describing a software system in terms of its architectural elements and the relationships among them.”* Typical types of architectural elements to be supported by an ADL are components, connectors, and configurations (Medvidovic et al., 2000).

Several frameworks (CIO Council, 1999; Haren, 2011; Zachman, 1987) and modeling languages (Group, 2017; Object Management Group, 2011; The Open Group, 2016) were introduced to provide a uniform representation for diagrams that describe the EA². The EAM team uses those views to address concerns of stakeholders and each of them provides the opportunity to present the current status and progress of one or multiple concerns. A comprehensive list of typical concerns in EAM presented in seven categories can be found in (Aleatrati Khosroshahi et al., 2020).

In the context of this thesis, we assume that an ADL is a modeling language defined by a meta-model on the M2 layer of the four-layered meta-modeling stack introduced in Section 2.1.1. However, an explicit concrete syntax is not required. This allows to use architecture descriptions in tool-supported contexts, e. g. for model analysis and transformations. This strict assumption on ADLs is not always required. For instance, Taylor et al. (Taylor et al., 2009) have a rather broad view on an ADL's requirements. They define *“ADLs can be textual or graphical, informal (such as PowerPoint diagrams), semi-formal, or formal, domain-specific or general-purpose, proprietary or standardized, and so on.”*

2.2.4. Archimate Notation Language

According to Lankhorst et al. (M. Lankhorst, 2017) *“Important for an architecture description language is that the properties of the system can be represented in their bare essence without forcing the architect to include irrelevant detail. This means that the description language must be defined at the appropriate abstraction level. The language and methods are the basis for unambiguous mutual understanding and successful collaboration between the stakeholders of the architecture.”*

This has required more coarse-grained modelling concepts than the finer grained concepts that can typically be found in modelling languages used at the level of specific

²An extensive literature review on EAM frameworks can be found in (Buckl et al., 2012).

development projects, such as e.g. UML (Group, 2017) and BPMN (Object Management Group, 2011). In these cases, it is unclear how concepts in one view are related to concepts in another view and whether views are compatible with each other. Therefore a new language was needed, leading to the development of ArchiMate (M. M. Lankhorst et al., 2010).

The ArchiMate Enterprise Architecture modeling language (The Open Group, 2016) provides a uniform representation for diagrams that describe Enterprise Architectures. It offers an integrated architectural approach that describes and visualizes different architecture domains, layers and their underlying relations and dependencies. ArchiMate distinguishes between the model elements and their notation, to allow for varied, stakeholder-oriented depictions of architecture information. The language is supported by a plethora of vendors and service providers. Many organizations are using it already as their company standard for describing enterprise architecture, and its value has been proven in practice (M. Lankhorst, 2017).

The core language of ArchiMate defines a structure of elements and their relationships, which can be specialized in different layers. Those layers are depicted in Figure 2.6 and are described as follows:

1. The **Business Layer** defines business services offered to customers, which are realized in the organization by business processes performed by business actors.
2. The **Application Layer** depicts application services that support the business, and the applications that realize them.
3. The **Technology Layer** comprises technology services such as processing, storage, and communication services needed to run the applications, and the computer and communication hardware and system software that realize those services. Physical elements are added for modeling physical equipment, materials, and distribution networks to this layer. The technology layer is equivalent to the EA infrastructure layer detailed in Section 2.2.1

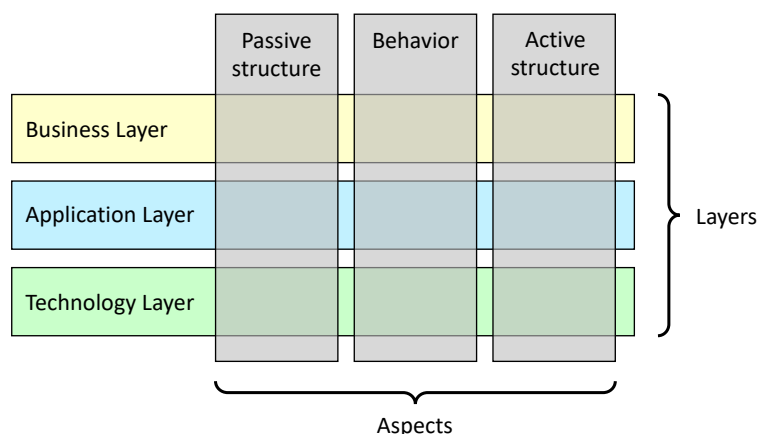


Figure 2.6.: ArchiMate core framework (The Open Group, 2016)

The framework allows to model the EA from different viewpoints, whereas the position within the layers highlights the concerns of the stakeholder. A stakeholder typically can have concerns that cover multiple aspects. Those aspects are defined as follows:

- The **Active Structure** aspect represents structural elements like the business actors, application components, interfaces or devices that perform actual behavior.
- The **Behavior** aspect represents the behavior performed by the actors like, processes, functions, events, or services. As ArchiMate describes, active structural elements are assigned to behavioral elements, to show who or what performs specific behavior.
- The **Passive Structure** aspect defines the objects on which the actual behavior is performed. These are usually information objects in the Business Layer and data objects in the Application Layer.

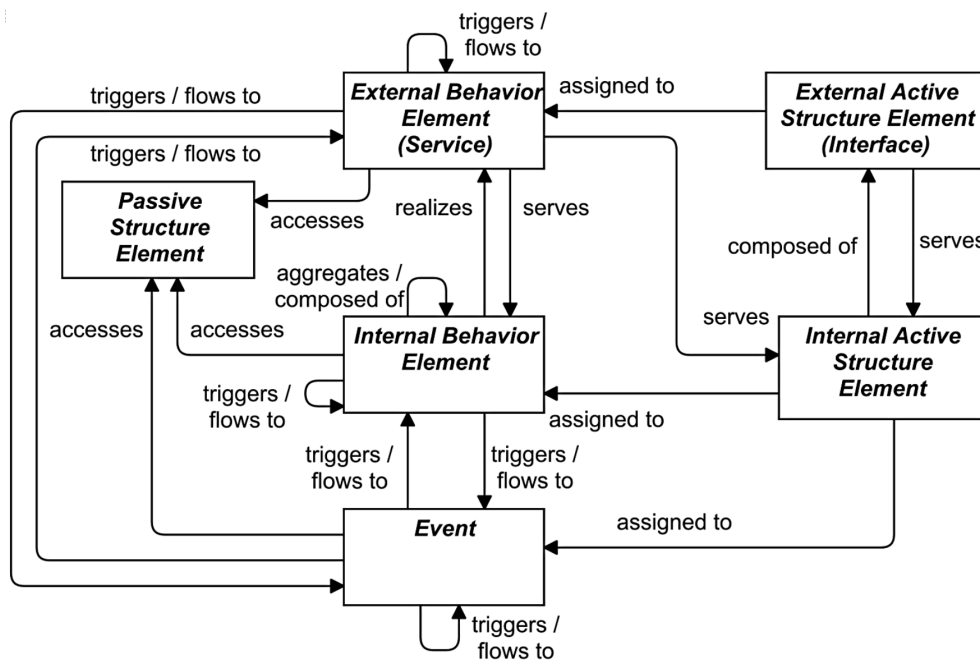


Figure 2.7.: ArchiMate behavior and structure elements meta-model (The Open Group, 2016)

The according meta-model of the structural and behavioral elements of the ArchiMate language is illustrated in Figure 2.7. It specifies the main relationships between the behavior and structure elements. Structure elements can be subdivided into active structure elements and passive structure elements. Active structure elements can be further categorized in external active structure elements and internal active structure elements. Behavior elements can be subdivided into internal behavior elements, external behavior elements, and events. These three aspects have been inspired by natural language, where a sentence has a subject (active structure), a verb (behavior), and an object (passive structure).

In addition to the aspects outlined above, the ArchiMate language defines a set of generic relationships, that define a connection between two aspects, like an interface is assigned to an application component. The relationships finally describe the overall structure of a stakeholder concern. The relationships are classified as follows:

- **Structural** relationships model the static construction or composition of concepts of the same or different types
- **Dependency** relationships model how elements are used to support other elements
- **Dynamic** relationships are used to model behavioral dependencies between elements
- **Other** relationships, which do not fall into one of the above categories

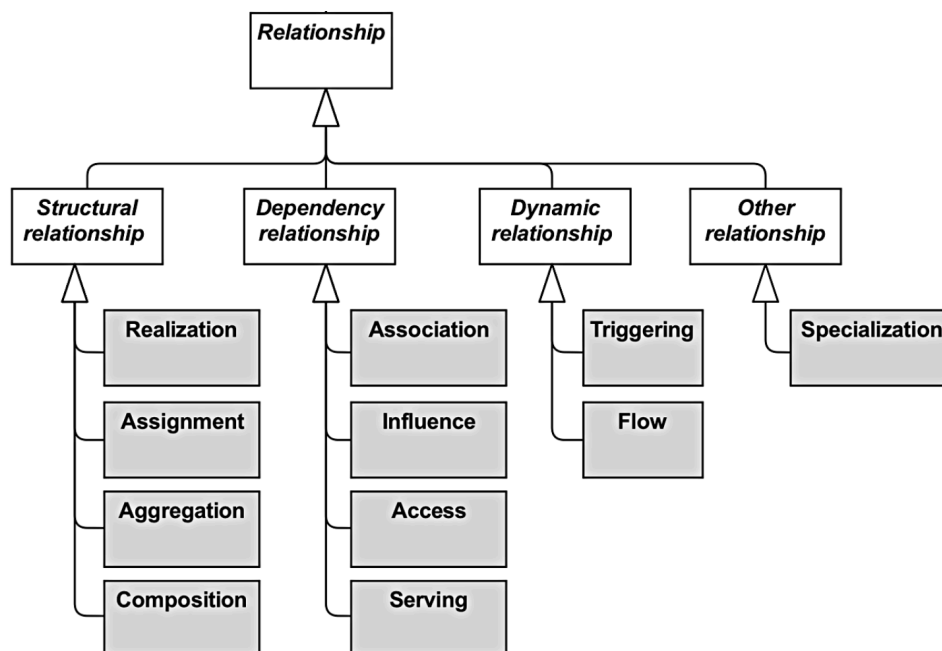


Figure 2.8.: ArchiMate relationship meta-model (The Open Group, 2016)

The different relationship types are depicted in Figure 2.8. A detailed explanation of the relationship types can be found in (The Open Group, 2016). The set of different aspects that are connected via relationship models addresses a particular stakeholder concern. A concern describes a specific view on the Enterprise Architecture.

2.3. Microservice Architecture

As the previous chapters indicate, IT organizations continuously search for better ways to build IT systems by adopting new technologies and observing technology trends that might have the potential to either optimize the organization's internal processes or open up new

markets. Domain-Driven Design (DDD) (Evans, 2003) fosters organizations to represent the real world in their code and suggests better ways to model their IT systems. Agile practices (Dingsøyr et al., 2012) and continuous delivery (Humble et al., 2010) accelerate the deployment of software into production. Cloud platforms (Mell et al., 2011) and virtualization helps to provision and resize machines at will. DevOps (Riungu-Kalliosaari et al., 2016; Smeds et al., 2015) and infrastructure automation (Humble et al., 2010) provide ways to handle these machines at scale. The development of microservice architectures (Newman, 2015) emerged from those new technologies. They represent an innovation derived from service-oriented architectures (SOA) (Bieberstein et al., 2008) that were in fact already applied years before but were not able to meet the new requirements of the present time. In the following sections, we detail key characteristics of microservices, how this architecture style differs from SOA, as well as present a conducted study about the state-of-the-art in microservice adoption in industry.

2.3.1. Definition of Microservice Architecture

The microservice architecture style was introduced 2012 by Martin Fowler and James Lewis (Fowler et al., 2014) during a workshop of software architecture. The participants of the workshop agreed on the term "microservices" as the most appropriate name for describing a common architectural style that many of them had been recently explored in software development projects. There is no commonly accepted definition about microservices. This architecture is mostly described by their key characteristics that are detailed in (Fowler et al., 2014). The National Institute of Standards and Technology (NIST) (Karmel et al., 2018) describes Microservices as:

"A basic element that results from the architectural decomposition of an application's components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communications protocol and a set of well-defined APIs, independent of any vendor, product or technology."

Adrian Cockcroft at Netflix (Cockcroft, 2016) summarized the key characteristics of microservices as *"fine grained SOA"* that presents single applications as a suite of small IT services that run in their own processes and communicate with each other through lightweight HTTP-based mechanisms, like REST (representational state transfer) (Fielding et al., 2000). As DDD suggests, these IT services are built around business capabilities, are modularized in a way they can be deployed independently without affecting other services and use their own data storage. As standard communication protocols are used, the services can be written in different programming languages which provides a high degree of flexibility.

Figure 2.9 illustrates how microservices differs from traditional monolithic applications. A monolithic application consists mostly of three different parts: 1) A frontend that represents the client-side user interface (UI), 2) a server-side application that contains the requires components (C) for executing the business logic and processing client-side requests and 3) a database that retrieves transactions from the server and stores all necessary data. This traditional way of building applications has been dominated the IT architecture many years for most of the IT organizations. The development team was

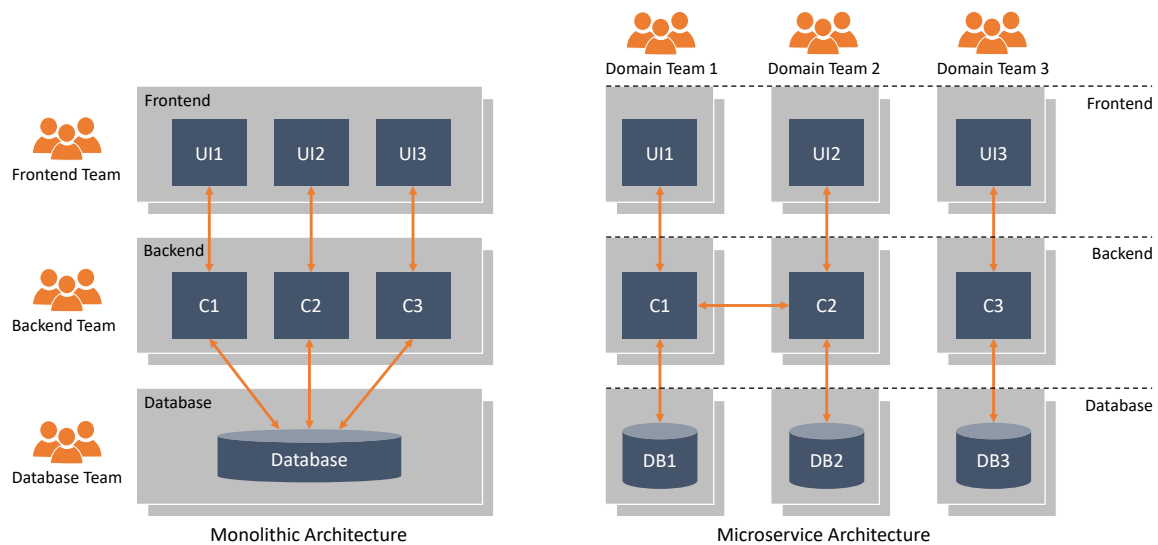


Figure 2.9.: Monolithic Architecture vs. Microservice Architecture

structured towards the three layers which is beneficial, as all experts are assigned to their well-known field and are capable to support each other.

However, it also leads to major drawbacks. A new feature or the adaption of an existing one requires a change on each architectural layer. The UI implements the feature in order to be accessible by the users, the backend must implement the business logic and the table schema in the database must be adapted for storing the corresponding data. Ultimately, this approach leads to a large amount of dependencies, as well as communication and coordination overhead:

- The implementation of new features requires the communication and coordination between three different teams.
- The result of the work of the teams are dependent from each other and must be aligned accordingly. For instance, the backend team can hardly work without the submission of the database team - and the UI need the deliveries from the backend.
- In case agile practices are applied for the development teams, the dependencies lead to temporal delays. In the first sprint, the database team creates the necessary table structure. In the second sprint, the backend team implements the logic and in the third sprint the UI team starts with their implementation. In total, three sprints are required for developing one feature.
- The database and the backend of monolithic applications must run stable and new features tested intensively, as failures would affect too many client-side applications otherwise. Whereas, frontends undergo many changes and customer feature requests. This implies a fast-speed, customer-centric frontend running alongside a slow-speed, transaction-focused legacy backend, which leads to a two-speed architecture (Bossert,

2016). These two different software-release cycles cause further delays in software deployment.

In fact, microservice architectures follow a different approach, which lies buried in the fundamental concept of Conway's Law (Conway, 1968): *"Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure."* That means, development teams should be formed cross-functional according to domains and all software development within the domains are performed by the individual teams, as Figure 2.9 illustrates on the right-hand side. They work at the respective IT services, which are technically divided into UI, backend and database. The division into the technical artifacts and the interface between the artifacts can now be clarified within the team. In the simplest case, one developer is a full-stack developer that covers all technical layers. In cross-functional teams only one developer needs to talk to the developer sitting next to him. Hence, the communication and coordination overhead between teams can be reduced to a minimum, as cross-domain communications hardly affect software development (Newman, 2015).

In the following, we describe the characteristics of microservices summarized by Cockroft in more detail. A complete analysis of microservice concepts was provided by Martin Garriga (Garriga, 2018) in the form of a taxonomy, encompassing the whole microservice lifecycle, as well as organizational aspects:

Modularization: Microservices represent an isolated unit of software that is independently deployable, replaceable and upgradeable. It contains all required software libraries to run autonomously. This allows to deploy new features faster. The change of a component in a monolithic application requires the whole application to be deployed in order to release the change. This could have a large impact on down-time and customer satisfaction. Every microservice is built to serve one specific business functionality. This opens up opportunities for reuse of functionality, i.e. microservices can be consumed in different ways for different purposes, making software release cycles faster.

Resilience: Microservice architectures are designed to be tolerant of service failures. If a problem occurs, it can be isolated quickly to an individual microservice, which makes troubleshooting easy to achieve and does not impact the whole application. If monolithic applications fail, everything stops working. However, as the communication frequency and volume are higher than in monolithic applications, any microservice request call could fail due to unavailability of the supplier. The client microservice has to respond to this as gracefully as possible. This is a disadvantage compared to a monolithic design as it introduces additional complexity to handle it. For that reason, many architectural patterns (Richardson, 2018) like Circuit Breaker, Bulkhead and Timeout has been introduced by the resilient engineering community.

Product orientation: Microservices foster organizations to align their software development efforts towards product orientation instead of projects. The purpose is that a development team should own a product over its full lifetime. A team is not only responsible for developing products, but also for their operation, which is a key characteristic in DevOps (Bass et al., 2015). Amazon's notion of *"you build it, you run it"* (O'Hanlon, 2006) is a common inspiration for this approach. This brings developers into day-to-day contact

with how their software behaves in production and increases contact with their users, as they have to take on at least some of the support burden.

Distributed communication: The only way to communicate with microservices is via well-defined exposed interfaces. Hence, all communication between microservices are via network calls, to enforce separation between the services and avoid the perils of tight coupling and low cohesion (F. Beck et al., 2011). The microservice community defines this characteristic with smart endpoints and dumb pipes (Fowler et al., 2014). Every microservice receives a request mostly via simple RESTful-API, process the request by applying the intended business logic and produces a response. Hereby, it is irrelevant where on earth the requested microservice runs. The communication can be fully distributed and executed asynchronously via lightweight messaging services like Apache Kafka³, RabbitMQ⁴, or MQTT⁵.

Technology Heterogeneity: With a system composed of multiple modularized and collaborating services via standardized interfaces, the use of technology within each service can be outsourced to the preferences of the development team. This allows to select the appropriate tool for each task, rather than having to select a more standardized, one-size-fits-all approach. For instance, a system that needs to improve its performance should better use C++ as programming language, whereas a system that provides simple reports can go with Node.js as server framework.

The technology heterogeneity can also be applied for the choose of the right database system. While monolithic applications prefer a single logical database for persistent data, microservice architectures prefer letting each service manage its own database and database technology. This approach is called Polyglot Persistence (Sadalage et al., 2013). For example, for a social network, a graph-oriented database that reflects the highly interconnected nature of a social graph is more appropriate than a relational database. However, the posts the users make in the social network could be stored in a document-oriented database, giving rise to a heterogeneous data storage architecture.

2.3.2. Building blocks of Microservice Architectures

There exist many reference architecture models of microservice architectures proposed by industry, as well as by academic researcher. A systematic mapping study on microservices was performed by Pahl et al. on a set of 21 primary studies from 2014 to 2015 (Pahl et al., 2016). It is a classification of the research directions in the field and highlights the relevant perspectives considered by researchers. In (Alshuqayran et al., 2016) Alshuqayran, Ali and Evans presented a mapping study with the focus on 1) the architectural challenges faced by microservice-based systems, 2) the architectural diagrams used for representing them, and 3) quality requirements that need to be considered for developing microservice architectures. Di Francesco et al. (Francesco et al., 2017) identified and evaluated the current state of the art on architecting microservices from the perspectives publication

³<https://kafka.apache.org/>, last accessed: 2020-10-28

⁴<https://www.rabbitmq.com/>, last accessed: 2020-10-28

⁵<http://mqtt.org/>, last accessed: 2020-10-28

trends, focus of research, and potential for industrial adoption. They produced an overview of the state of the art by synthesizing the obtained data from 71 different studies.

Based on the related work, specific key architectural components and building blocks for implementing and managing microservices in the context of enterprise architecture can be identified that occur very frequently. For that reason, several publications (Mayer et al., 2018; Munaf et al., 2019; Yu et al., 2016) exists that collect those patterns, and proposed building blocks and elaborate potential reference architectures. In the following, we illustrate a microservice reference architecture in Figure 2.10 derived from the literature and describe the key components in more detail.

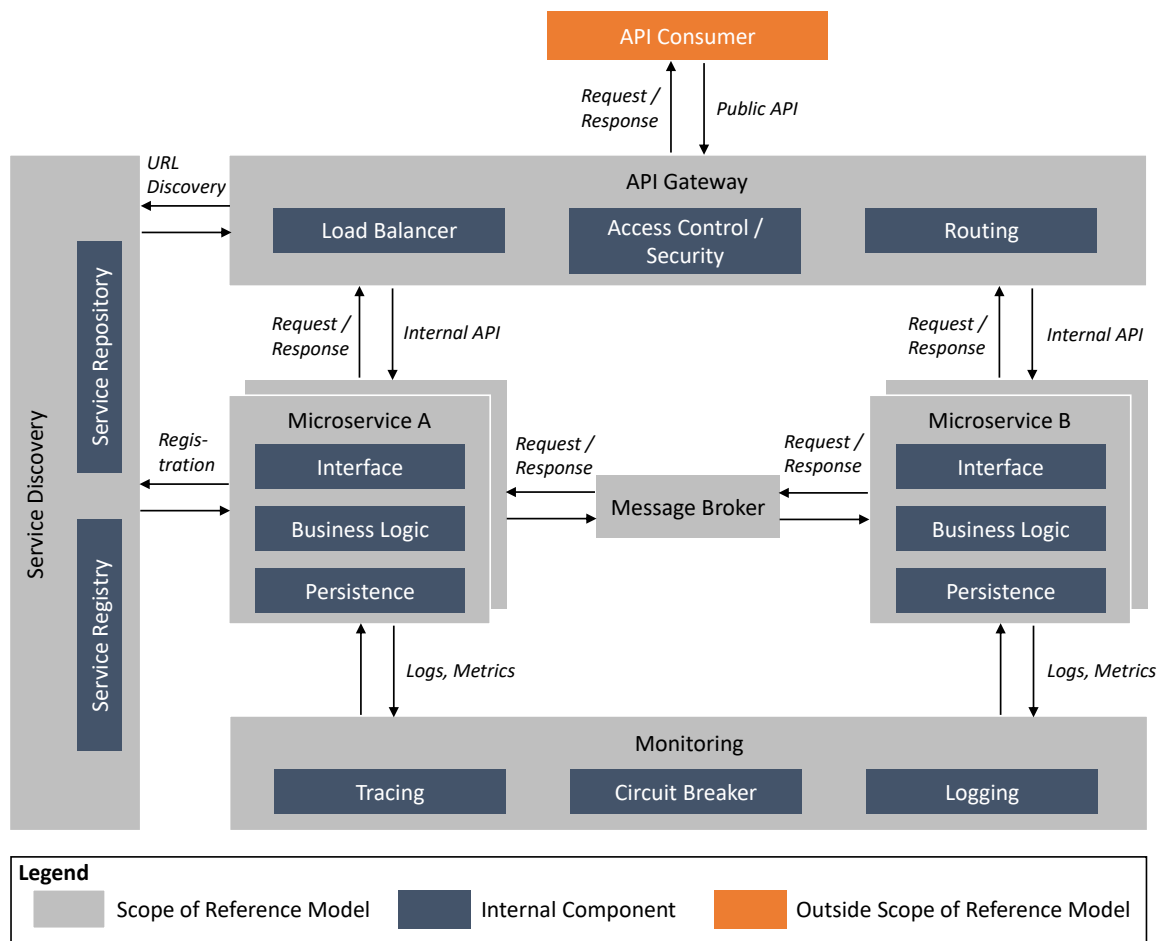


Figure 2.10.: Microservice Architecture Reference Model

Microservice: The microservices itself are lightweight, independently deployable applications that perform one assigned task. Microservices exist of three layers: 1) each service contains its own database which represents the persistence layer. A database that is not shared provide the benefit of better concurrency control, scaling and technology heterogeneity as various types of databases can be used as per requirements (e.g. NoSql). 2) the business logic implements some sort of required functionality to perform the business task.

3) the interface layer expose RESTful-APIs, that must be addressed in order to execute the particular business logic.

Service Discovery: With the reduction in size of microservices, in large and complex software systems the number of microservices will also increase. In order to manage all those microservices and to find the appropriate service (Alshuqayran et al., 2016) for specific task, service discovery components were developed to provide a service repository. It contains the location of all microservice instances along with their details and status of the instance. The service registry act as a registration system for microservices which can be consulted by API Gateways and microservices for services discovery. That means, as soon as a new microservice instance is introduced into the system, it must first register its metadata to the service discovery. This also supports API gateways and other microservices to find the requested service (Xiao et al., 2016). Service discovery also act as a monitoring system and identifies active microservice instances, temporarily inactive instances due to some issues, or failed instances.

API Gateway: API gateways (Newman, 2015) represent the single entry point for all clients and other API consumers. The API gateway takes all API calls and routes them to the appropriate microservice with request routing, composition, and protocol translation. Without the gateway, each client must know the exact endpoint and location of the microservice which is responsible for delivering the required information. However, this is rather challenging as microservices can change their location (domain and port) dynamically due to scaling or redeployment processes. For that reason, API gateway were developed to handle endpoints and optimize the response. As API gateway need the location information from the service repository, both components are often assembled into one service.

API gateways also take over the task to load balance requests. A load balancer (LB) distribute calls to one or more microservice instances based on some algorithm. With a LB the utilization of available resources can be optimized and, hence, the resilience of the overall system increases. In addition, this also reduces the impact of a single service failing. However, it must be ensured that the persistence layer of the microservices are also scaled when increasing the amount of microservice instances (Newman, 2015).

Access control and security mechanism are further important components in API gateways. Each request handled by any microservice needs to be verified and validated against the allowed permissions of clients in order to prevent unauthorized access to business operations. Since API gateways handle any incoming client requests before they arrive the microservices the access control is performed by those services.

Message Broker: Microservices mostly communicate via their RESTful interfaces. REST calls have a synchronous character, and synchronous calls are blocked due to the request/response model of the technology. That means, every incoming request is processed sequentially. Especially, by using the API gateway for direct communication between microservices, the request is handled synchronously, which results in tight runtime coupling. Hence, both the caller and caller service must be available for the duration of the request.

In order to apply an asynchronous communication between microservices that prevents thread blocking and tight runtime coupling, the usage of the message broker pattern

is quite usual (Wise et al., 1993). This pattern introduces a further service into the architecture that handles all message exchanges between microservices. It achieves loose runtime coupling since it decouples the message sender from the consumer. In addition, the availability is improved since the message broker buffers messages until the consumer is able to process them. That means, sent messages are never lost.

Monitoring: Breaking a monolithic system up into smaller, fine-grained microservices results in multiple benefits. However, it also adds complexity when it comes to proper surveillance of the system in production. Monitoring a microservice architecture is not achieved by only analysing the resource utilization. It also covers further tasks that go beyond of hardware metrics. This includes the monitoring of transaction status, performance issues, graceful handling of failed microservices during active operations, instance management, distributed tracing, distributed logging, anomaly detection or failure root cause analysis. A detail description of monitoring distributed systems like the microservice architecture is covered in Section 2.4.3.

The purpose of logging (Fu et al., 2014) is to track error reporting and related data in a centralized way. The term logging can refer both to the practice of event logging or to the actual log files that result. Log files can show any discrete event within an application, such as a failure, warning, information, debug messages, or a state transformation. When errors occur, such transformations in state help indicate which change actually caused an error. The challenge that need to overcome in microservice logging is the efficient collection of distributed logs and to aggregate them in a centralized logging application (T. B. Sousa et al., 2017).

Tracing allows the tracking of transactions through the microservice-based system and analyzes transaction states and performance issues. Especially distributed tracing (Fonseca et al., 2007; Sigelman et al., 2010; Zhou et al., 2014) was developed for microservices in order to understand the path of data as it propagates through the services. While Logs can record important checkpoints when servicing a request, a trace connects all these checkpoints into a complete route that explains how that request was handled across all services from start to finish.

Monitoring also implement the circuit breaker module (Nygard, 2007) which will be responsible for graceful handling of failed, not responding or lagging microservices during active operations. A circuit breaker is required especially in synchronous communications. After a certain number of requests to the called microservice have failed, the circuit breaker is activated. All further requests will fail immediately while the circuit breaker is in its active state. After a defined period of time, the client sends a few requests through to see if the downstream service has recovered, and if it gets enough healthy responses it resets the circuit breaker.

Service Mesh: The concept of service mesh is rather new (Li et al., 2019). It was first introduced by William Morgan (Morgan, 2017). Morgan defines a service mesh as *"a dedicated infrastructure layer for handling service-to-service communication by using a side-car proxy. It is responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application*

code, without the application needing to be aware.”

As all service-to-service communications will go through a side-car proxy, common microservice architecture components used for service discovery, routing, access control, observability, load balancing and security are already covered by service meshes (Li et al., 2019). This makes them obsolete and reduces the complexity of managing several components that are required in parallel to the microservices. Currently, four service mesh platforms get great attention. These are Istio⁶, Linkerd⁷, Amazon App Mesh⁸, and Airbnb Synapse⁹.

2.3.3. State-of-the Art in Microservice Adoption

There are currently only a few empirical papers with microservices adoption in industry. The work of Alshuqayran et al. (Alshuqayran et al., 2016) contains a list of significant challenges in microservice architectures, in particular regarding to operation, integration and performance. A similar survey have been conducted by Ghofrani and Lübke (Ghofrani et al., 2018), that reveals the current state of practices and challenges in introducing microservice architectures in an existing IT landscape. Schermann et al. (Schermann et al., 2015) present the results of a survey of 42 participants which targets primarily on implementation specifics such as used communication protocols, data formats, monitoring data and preferred programming languages. Knoche et al. (Knoche et al., 2017) conducted a survey with 71 participants and reveal drivers and obstacles in the introduction of microservices. Francesco et al. (Francesco et al., 2017) report about an empirical study on migration practices towards the adoption of microservices.

According to Google Trends (Balalaie et al., 2016), widespread interest in microservices has been shown in early 2014, and it has grown steadily ever since. Although, the term itself was coined in 2012 (Fowler et al., 2014), implementations of this architectural style were already made much longer. For instance, the video streaming provider Netflix, one of the best-known early adopters, began in 2008 with the introduction of a microservice architecture to leverage the benefits of cloud computing (Meshenberg, 2016). As of today, many well-known companies use microservices such as Amazon (Kramer, 2011), SoundCloud (Calçado, 2014), LinkedIn (Ihde, 2015), or Zalando (Schaefer, 2016), just to name a few.

Particularly noteworthy about microservices is the fact that many companies make their research and technologies publicly available. For example, Zalando¹⁰ and Netflix¹¹ have technology blogs for discussing current ideas and experiences with the community. In addition, many tools, libraries and infrastructure components required for the development and operation of microservices are published free of charge as open source software. They invite the community to participate in their research in order to benefit from it.

⁶<https://istio.io>, last accessed: 2020-10-28

⁷<https://linkerd.io>, last accessed: 2020-10-28

⁸<https://aws.amazon.com/de/app-mesh>, last accessed: 2020-10-28

⁹<https://airbnb.io/projects/synapse>, last accessed: 2020-10-28

¹⁰https://jobs.zalando.com/en/tech/blog/?gh_src=4n3gxxh1, last accessed: 2020-10-28

¹¹<https://netflixtechblog.com/tagged/microservices>, last accessed: 2020-10-28

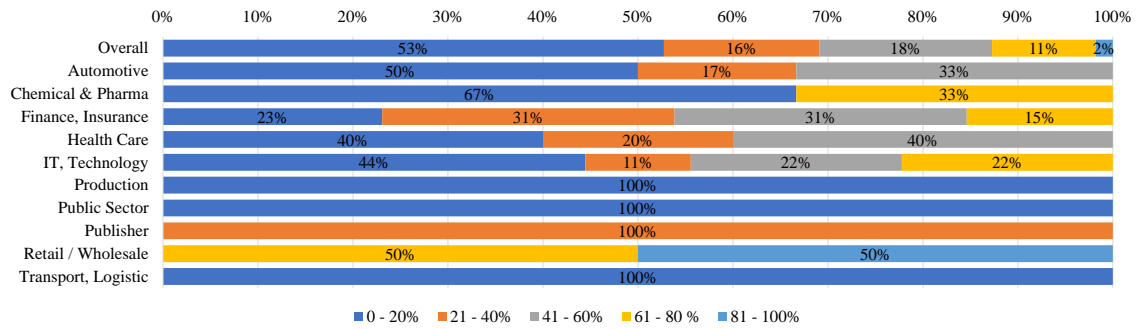


Figure 2.11.: Proportion of IT that is based on microservices grouped by industry sectors. N=58 (Kleehaus et al., 2019b)

Furthermore, cloud providers such as Amazon Web Services (AWS), Microsoft Azure, IBM Bluemix, or Google Cloud allow to procure the necessary resources quickly and with little effort. As a result, entry barriers for implementing microservices are relatively low.

We conducted a survey among 58 participants from industry in order to identify the current distribution of microservice architectures in the German market (Kleehaus et al., 2019b). Figure 2.11 reveals the result of the study.

54,5% (n=30) of the participants replied that the proportion of microservices in their IT is lower than 20% or even 0%. This little number indicates that the adoption of microservice-based IT landscape is still at an early stage in the German industry. 32,7% (n=18) of the participants stated they are using microservices on a mid-range extent i.e. 21% - 60% of the IT is based on this architecture style. Only a small group of 12,7% (n=7) represent experts in this area and already transformed 61% to 100% of their IT towards microservices.

A closer look at the sector-specific number depicted in the figure reveals differences. While results indicate that microservices seem to be used very intensively in the field of Retail/Wholesale, Finance and Insurance, Automotive and IT, Technology, they are rarely used in the Public sector, Production, and Transport, Logistic. However, based on the answers of 73,3% (n=22) of the participants which have a low amount of running microservices in their IT (0% - 20%) plan to increase this amount. This number indicate that companies recognize the potential and benefits of microservices and will intensify the usage of this architecture pattern. This result is in line with Knoche et al. (Knoche et al., 2017).

2.4. DevOps

Releasing a new application or a new version of an information system is one of the most sensitive steps in software development. Regardless whether the system or version is for consumers, or for internal use, releasing a new software opens the possibility of incompatibilities or failures, with subsequent customers dissatisfaction. Hence, organizations pay great attention to the process of defining a stable release plan.

According to ITIL (Office, 2011c), the following release planning steps are important to address. Traditionally, most of the steps are done manually (Bass et al., 2015):

- Define and agree on release and deployment plans with customers and stakeholders. This could be done at the team or organizational level.
- Ensure that each release package consists of a set of related assets and service components that are compatible with each other. Since, all assets including libraries, platforms, and dependent services changes over time, those changes may introduce incompatibilities. Hence, it is important to prevent incompatibilities from becoming apparent only after deployment.
- Ensure that the integrity of a release package and its constituent components is maintained throughout the transition activities and recorded accurately in the configuration management system. There are two parts to this step: First, it is important to make sure that old versions of a component are not inadvertently included in the release, and second a record must be kept of the components of this deployment.
- Ensure that all release and deployment packages can be tracked, installed, tested, verified, and/or uninstalled or rolled back, if appropriate.
- Ensure that organization and stakeholder change is managed during release and deployment activities.
- Ensure that skills and knowledge are transferred to service operation functions to enable them to effectively and efficiently deliver, support and maintain the service according to the agreed service level agreements. This includes to monitor, record and manage deviations, risks and issues related to the new or changed service and take necessary corrective actions.

If all of these activities above are accomplished primarily manually through human coordination then these steps are labor-intensive, time-consuming, and error-prone (Bass et al., 2015). For that reason, methods and tools were created to accomplish the listed activities with differing levels of automation (Humble et al., 2010).

Furthermore, help desk as well as operations personnel need to be trained in features of the new system and in troubleshooting any problems that might occur while the system is operating. The timing of the release may also be of significance because it should not coincide with the absence of any key member of the operations staff. In general terms, all of these activities requires coordination between the developers and the operations personnel. If there is no active exchange between both groups, the service operations will inevitably lead to customer dissatisfaction.

A more rapid release schedule which is striven by agile practices would emerge an even worse bottleneck in operation functions, when the software release is coordinated by the operation staff. As a consequence, this would lead to long delays in software releases. Hence, in order to solve this coordination issue, Debois (Debois et al., 2011) advocated a tighter integration between the development and operations functions which is finally termed as *DevOps*.

DevOps is a set of principles and practices for merging development and operations activities in order to continuously deploy stable versions of application systems (Hüttermann, 2012). DevOps therefore aims at a better integration of all activities in software development and operation of an application system life cycle. By means of DevOps integration, IT organizations are able to react more flexibly to changes in the business environment (Sharma et al., 2015). Humble and Molesky (Humble et al., 2011) summarize the DevOps approach into four principles:

- **Culture:** DevOps requires a cultural change of accepting joint responsibility for delivery of high quality software to the end-user (Brunnert et al., 2015). This can be achieved by setting up mixed DevOps teams that have end-to-end responsibility for the development, release and operations of software. At its most extreme, DevOps practices make developers responsible for monitoring the progress and errors that occur during deployment and execution. Another example would be to set integrated (agile) processes in place that force development and operations teams to work closer together.
- **Automation:** DevOps relies on full automation of the build, deployment and testing in order to achieve short lead times, and consequently rapid delivery and feedback from end-users. This can be achieved by applying continuous integration (CI) and continuous delivery (CD) pipelines (Humble et al., 2010).
- **Measurement:** Gaining an understanding of the current delivery capability and setting goals for improving it can only be done through measuring. This varies from monitoring business metrics (e.g. revenue) to test coverage and the time to deploy a new version of the software.
- **Sharing:** Sharing happens at different levels, from sharing knowledge (e.g. about new functionality in a release), sharing tools and infrastructure, as well as sharing in celebrating successful releases to bring development and operations teams closer together.

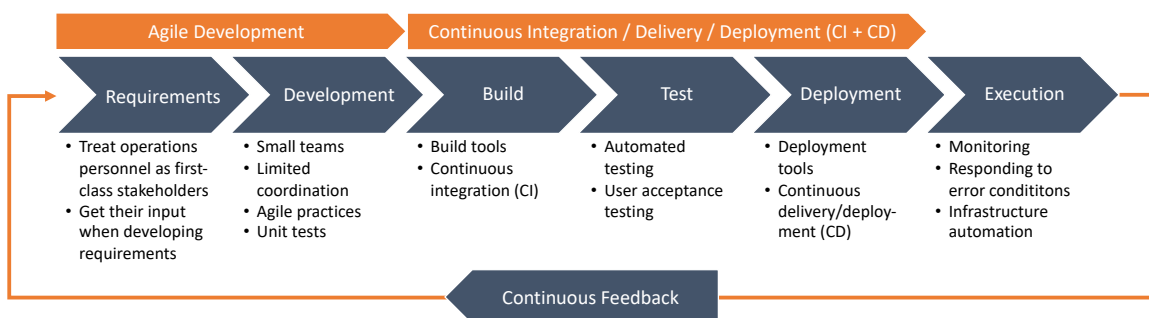


Figure 2.12.: DevOps life cycle phases (Bass et al., 2015)

Figure 2.12 gives an overview of DevOps processes. In the following, we detail the seven activities of DevOps practices (Bass et al., 2015):

1. **Requirements:** Operation staff must be treated as first-class stakeholders from the point of view of requirements. Operations have a set of requirements that pertain to logging and monitoring. Therefore, logging messages, for instance, must be understandable and usable by an operator. Monitoring agents must be integrated into the source code. Involving operations in the development of requirements will ensure that these types of requirements are considered.
2. **Development:** Development must be performed in small teams with less coordination. Most organizations move away from waterfall-based software development processes and apply agile-based practices. We further detail agile practices in Section 2.4.1.
3. **Build:** The build process of applications should be automated via continuous integration (CI) scripts. Practices that apply to the development of CI scripts are intended to ensure both high quality in the deployed applications and that deployments proceed as planned. Integration bugs are detected early and are easy to track down due to small change sets. This saves both time and money over the lifetime of a project.
4. **Test:** Test and staging are the previous environments of the deployment pipeline prior to deploying the system into production. The staging environment should mirror as much as possible the production environment. In both environments several test cases are performed including user acceptance tests (UATs), smoke tests and nonfunctional tests (Kaner et al., 2000).
5. **Deployment:** It is important, that the deployment process is used by both development and operations personnel in order to ensure a higher quality of deployments. This practice also refer to the time that it takes to diagnose and repair an error. The normal deployment process should make it easy to trace the history of a particular deployment artifact and understand the components that were included in that artifact. Furthermore, practices associated with continuous deployment (CD) are intended to shorten the time between a developer committing code to a repository and the code being deployed. Continuous deployment also emphasizes automated tests to increase the quality of code making its way into production. We detail the basic concepts in Section 2.4.2.
6. **Execution:** In the execution phase, the development team is more responsible for relevant incident handling by applying monitoring practices (see Section 2.4.3. These practices are intended to shorten the time between the observation of an error and the repair of that error. Organizations that utilize these practices typically have a period of time in which development staff has primary responsibility for a new deployment. Later on, the operations team takes over.
7. **Feedback:** Providing feedback about each phase in the software deployment process is not only the responsibility of customers but also the responsibility of operations staff. The detection of outliers and anomalies (Hawkins, 1980), or the collection of incidents must be forwarded to the development teams as soon as possible. In case the application is not working as expected, a continuous feedback often represents

the first trigger to start a new iteration of an application life-cycle (Kleehaus et al., 2018a).

In the following sections, we focus on those DevOps principles that are important in the course of this thesis.

2.4.1. Agile Practices

One of the characterizations of DevOps emphasizes the relationship of DevOps practices to agile practices. Agile software development projects have become widely accepted and adopted across the industry (Abrahamsson et al., 2017; Cohen et al., 2004; Dybå et al., 2008; Erickson et al., 2005). The traditional waterfall model is outdated and unable to keep up with the advantages offered by agile methodology (Highsmith et al., 2001). Conboy (Conboy, 2009) defines software development agility as the continued readiness “to rapidly or inherently create change, proactively or reactively embrace change, and learn from change while contributing to perceived customer value (economy, quality, and simplicity), through its collective components and relationships with its environment.” Agile methodology owes its success to its core principles such as (K. Beck et al., 2001):

- Individuals and interactions are valued over process and tools
- Working software is valued over comprehensive documentation
- Customer collaboration is valued over contract negotiation
- Change adoption agility is valued over project plan adherence

At the core of these principles is the idea of self-organizing teams whose members are not only collocated but also work at a pace that sustains their creativity and productivity. The principles encourage practices that accommodate change in requirements at any stage of the development process. That means, customers are actively involved in the development process and are allowed to expose feedback at any time. This minimizes overall risk and allows the product to adapt to changes quickly (Moran, 2014). The principles are not a formal definition of agility, but are rather guidelines for delivering high-quality software in an agile manner.

In general, the agile software development process is an incremental and iterative approach for developing software features or functionality. The development cycle is broken into cycles of two to four weeks to accomplish units of work. Small increments minimize the amount of up-front planning and design. Each iteration involves a cross-functional team working in all development tasks including planning, design, coding and testing. At the end of the iteration a working product is demonstrated to customers and surrogates. The structure and documentation are not important but a working prototype is considered valuable. In general, multiple iterations are required to release a final version of a software product.

There exists a plethora of different agile software development methods. The most important ones are extreme programming (K. Beck, 2000), Scrum (Schwaber et al., 2002),

crystal family of methodologies (Cockburn, 2002), feature driven development (Palmer et al., 2002), the rational unified process (Kruchten, 2004), dynamic systems development method (Stapleton, 1997), and adaptive software development (Highsmith, 2013).

Agile Method: Scrum

One of the most adapted agile software development method in industry is *Scrum* (Kurapati et al., 2012; Linkevics, 2014). The term "Scrum" originally derives from a strategy in the game of rugby where it denotes "getting an out-of play ball back into the game" with teamwork (Schwaber et al., 2002). Scrum is an empirical approach applying the ideas of industrial process control theory to systems development resulting in an approach that reintroduces the ideas of flexibility, adaptability and productivity (Schwaber et al., 2002). Scrum primarily concentrates on how the team members should function in order to produce high-quality software in a constantly changing environment. It does not define any specific software development techniques.

The most central artifact in Scrum is the so called *Sprint*. It defines a time-box of one month or less during which a potentially releasable product increment is created. All other Scrum events and artifacts are organized around *Sprint*. Each development iteration is divided into three phases (Schwaber et al., 2002), which we detail in the following:

1. **Planning:** In the planning phase the system that needs to be developed is defined. This includes the creation of a *Product Backlog* list that contains all the requirements for the software product. The requirements can originate from several stakeholders like customer, sales and marketing division, or software developers. The requirements are prioritized and the effort needed for their implementation is estimated. The requirements in the *Product Backlog* are constantly updated with new and more detailed items, as well as with more accurate estimations. Before each *Sprint* iteration, the updated *Product Backlog* is reviewed by the Scrum Team in order to gain their commitment for the next iteration. Before initializing a next *Sprint* the Scrum team selects and defines what *Product Backlog* items should be realized. This subset of items is called *Sprint Backlog*.
2. **Development:** The development phase is the agile part of the Scrum approach. In the development phase the system is developed in *Sprints*. Each *Sprint* includes the traditional phases of software development including analysis, design, test and delivery phases. The architecture and the design of the system evolve during the *Sprint* development. After each *Sprint* the developed increment is considered to be released if it fulfills the criteria of "Done". The *Definition of Done* (DoD) drives the quality of work and is used to assess when a *User Story* has been completed. The specification in the DoD ensures that the entire team is aligned on what "done" actually means. There may be, for example, three to eight *Sprints* in one systems development process before the system is ready for distribution. Also there may be more than one team building the increment.
3. **Review and Retrospective:** At the end of a *Sprint*, the team holds two events, the

Sprint Review and the *Sprint Retrospective*. At the *Sprint Review*, the team presents the completed increment to the stakeholders and reviews the work that was completed as well as the planned work that was not completed. During the *Sprint Retrospective* reflects on the past *Sprint* agrees on continuous process improvement actions.

A *Scrum Team* comprises the following roles:

- The **Scrum Master** (SM) is a new management role introduced by Scrum. The SM is responsible for team setup, conducting sprint meetings, and removing development obstacles. The SM is not a traditional team lead or project manager but acts as a buffer between the team and any distracting influences. The SM ensures that the scrum framework is followed, i.e. the project is carried out according to the practices, values and rules of Scrum.
- The **Product Owner** (PO) represents the product's stakeholders and the voice of the customer. The PO creates and prioritizes the product backlog, and is responsible for the delivery of the functionality at each sprint cycle. The main task is to maximize the value that the development team delivers. The PO should focus on the business side of product development and spend the majority of time liaising with stakeholders and the development team. The product owner should not dictate how the development team reaches a technical solution, but rather will seek consensus among the team members.
- The **Development Team** (DT) has the authority to decide on the necessary actions and to organize itself in order to achieve the goals of each sprint. The DT members perform the actual development tasks.

Agility in DevOps

DevOps extends agile in terms of the principles as DevOps can provide a pragmatic extension for the current agile activities. For example, as DevOps stresses more on the communication and collaboration between developers and operators rather than tools and processes, it can achieve agile goals to reduce team working latency and extend agile principles to entire software delivery pipeline (Farroha et al., 2014). The other way around, agile can support DevOps by encouraging collaboration between team members, automation of build, deployment and test, measurement and metrics of cost, value and processes, knowledge sharing and tools (Bang et al., 2013).

2.4.2. Software Release Automation

CI and CD processes are the inevitable items of medium and large scaled software projects that enable software release automation. They represent essential concepts in agile and DevOps cultures (Humble et al., 2010). The combination of CI and CD processes forms a *continuous delivery pipeline*. Many tools (Leite et al., 2020) were developed in the last years to

realize a CD pipeline. One of the most prominent example is Jenkins ¹². In the following Sections, we highlight the main concepts of these processes in more detail.

Continuous Integration

CI is an automated process that triggers inter-connected steps such as compiling code, running unit and acceptance tests, validating code coverage, checking coding standard compliance and building deployment packages. It ensures that the written code works by providing developers with rapid feedback on any problems that might be introduced with committed changes (Humble et al., 2010). In the CI process, every developer merges their code changes to a central version control system, each commit triggers automated build. So the latest versions are always available in the code repository and also built executable is from latest code. CI has increased in importance due to the benefits that have been associated with it (Ståhl et al., 2014). These benefits include improved release frequency and predictability, increased developer productivity, and improved communication. Without CI, developers must perform a lot of work before they are able to commit their own code in order to prevent merge conflicts.

Continuous Delivery

CD is the practice of continuously deploying software builds automatically to other environments like test or stage, but not necessarily to actual users (Humble et al., 2010). The CD process involves continuous integration and extends it with an automated deployment to pre-production environments. With CD the DevOps team is able to reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications. This fits perfect to agile development, as CD ensures that the most current and latest version of the software is readily available after each sprint.

Continuous Deployment

Continuous deployment (CDE) implies continuous delivery and is the practice of ensuring that the software is continuously ready for release and deployed to actual customers (Humble et al., 2010). This minimizes the lead time between the development of new code and its availability in production for usage. The rational of CDE is to accelerate the feedback loop with the customers and to reduce release pressure of the team. That means, there is no release day anymore.

2.4.3. Monitoring distributed Systems

The measurement and control of IT landscapes is based on a continuous process of monitoring, reporting, analyzing and performing subsequent actions. These steps are fundamental as they provide required enactment proposals to support and improve delivered IT services. Therefore, it is important to note that, although this monitoring process takes place

¹²<https://jenkins.io/>, last accessed: 2020-10-28

during service operation, it provides a basis for transformation strategies that improve the EA. ITIL (Office, 2011b) defines these steps as:

- **Monitoring** is the activity of observing a situation to detect changes that happen over time.
- **Reporting** refers to the analysis, production and distribution of the output of the monitoring activity.
- **Control** refers to the process of managing the utilization or behaviour of a device, system or service. It is important to note, that simply manipulating a device is not the same as controlling it. Control requires three conditions: 1) the action must ensure that behaviour conforms to a defined standard or norm, 2) the conditions prompting the action must be defined, understood and confirmed, and 3) the action must be defined, approved and appropriate for these conditions.

Monitoring can make sure that the system's Quality of Service (QoS) requirements are fulfilled as well as to detect, diagnose, and resolve QoS problems as early as possible. As many EA frameworks (Buckl et al., 2012; Haren, 2011; Zachman, 1987) delivers the holistic view on the EA on basis of a meta-model that describes the IT landscape in several layers, monitoring solutions have been placed on each layer of the EA stack, including physical server, network devices, and storage hardware, the operating system, virtualization and containerization, applications, as well as business processes and user activities. On each level, various QoS measures of interest exist. While these are typically hardware-agnostic for system level measurements, e.g. utilization of CPUs and storage resources, the set of measures becomes EA-specific, when it comes to application or business process level, e.g. involving measures like completed orders per hour.

In (Kleehaus et al., 2016), we described the basic concepts of IT monitoring classified in a taxonomy and presented the main research areas which are currently in focus by academia and industry. The following sections provide a brief introduction into selected aspects of IT monitoring with respect to instrumentation strategies, interrogation and data collection approaches (Section 2.4.3), instrumentation behavior and perturbation (Section 2.4.3), and performance measurement (Section 2.4.3). For a detailed presentation of foundations on performance measurement of software systems, we suggest to refer to Jain (Jain, 1991), Lilja (Lilja, 2005), as well as Menascé and Almeida (Menascé et al., 2002).

Monitors and Instrumentation

Monitoring can be classified based on the trigger mechanism, the result display ability, and the implementation level (Jain, 1991). Event-driven and sampling-based approaches are strategies for triggering the measurements of runtime data from a system (Lilja, 2005; Menascé et al., 2002). Event-driven techniques collect data whenever a relevant event in the system occurs. Example events are software exceptions, incoming user request, or the invocations of software operations. In the simplest case, an event-driven measurement routine updates an event counter. A technique for reducing monitoring overhead is defined

as *sampling*. With this approach only a subset of traffic within a specific time and in a specific node is collected. Sampling prevents the burden of collecting all traffic and still uncovers a good approximation of the runtime behavior (Duffield, 2004).

According to the implementation level, monitoring can be divided into hardware monitoring and software monitoring. Hardware monitoring focus on low level measurements of physical devices based on electrical signals and hardware registers. Important KPIs are CPU, Memory, hard drive, and network utilization. Software monitoring, as the opposite, are software routines integrated in the analyzed software system. The process of integrating software monitors into a system is called *instrumentation*.

Instrumentation can be divided into *agent-based* or *agent-less* instrumentation. The choose of the right method is often dependent on the business criticality of an application to a company's business revenue and processes (Pargaonkar et al., 2012). In an agent-based instrumentation, a software routine (agent – sometimes also called service or daemon) is installed in the application (source/object/byte) code or the underlying runtime environment in form of operating system, middleware, or application server with the primary purpose of collecting information and pushing it over the network to a central server. Agents intervene deeply into system and provide more features than agentless monitoring. That enables access to deeper levels of root-cause analysis, trouble shooting and performance analysis of applications, servers, and network devices. Various instrumentation techniques exist, e. g., direct code modification, indirect code modification using compiler modification or aspect-oriented programming (AOP), or middleware interception (Jain, 1991; Lilja, 2005; Menascé et al., 2002). Instrumentation is often already integrated into the runtime environment, like logging mechanisms that provides performance measurements based on event logs.

As opposed to this, in agent-less instrumentation, data is obtained from applications or network devices without installing any additional software. Instead, the monitoring solution use various protocols to gather the monitoring data such as SNMP, WMI, HTTP, POP, FTP, etc. or leverages the application programming interface (API) exposed by the applications. In particular, network traffic monitoring and analysis of log files can be performed without the installation of agents. In addition, it allows administrators to get monitoring up and running more quickly. However, agentless instrumentation has its own drawbacks. Agentless monitoring tools do not have the same depth of features and provide limited control over the IT system being monitored. Therefore, while the maintenance and deployment costs are negligible, the relatively high network traffic and the non-availability of real-time data are big disadvantages.

Instrumentation Behavior and Perturbation

According to ITIL (Office, 2011b), the way how applications can be instrumented can be abstracted into four categories. *Active instrumentation* refers to the ongoing "interrogation" of a system in order to analyze and determine its current status and to predict future behaviors. This approach is resource-intensive and is usually reserved to proactively monitoring the availability of critical systems or attempting to resolve an incident or diagnosing a problem. In contrary to this, *passive instrumentation* is addressing issues in

the system by analyzing historical log data. The main difference to active monitoring is that the passive approach shows how the system handles existing conditions but provides less insights into how the system will deal with future events. *Reactive instrumentation* reacts to certain type of events or failures and executes particular actions. For instance, server performance degradation may trigger a reboot, or a system failure will generate an incident. *Proactive instrumentation* is designed to detect patterns of events which indicate that a system or service is about to fail. This includes continuously analyzing of historical or streaming data in order to create patterns which determine on the one hand the normal condition of a system and on the other hand anomalies which have been detected previously. It has to be mentioned that reactive and proactive instrumentation could be active or passive. For instance, in a proactive - passive scenario event records are correlated over time to build trends for proactive problem management. Reactive - active scenarios are used to diagnose which system is causing the failure and under what conditions, e.g. "ping" a device to obtain its health status.

An important aspect to consider when instrumenting software systems is *perturbation*. Every software requires hardware resources like CPU, memory, and storage in order to fulfill its task. Hence, instrumentation agents compete for shared resources with the system under analysis. This may affect the system's runtime behavior, like the software control flow due to measurement routines. Perturbation that has an impact on a system's performance is often referred to as *overhead*. For that reason, it is critical that monitoring systems are required to introduce a rather low overhead – i.e., 1-2% on the system resources (Mappic, 2011) – to not degrade the monitored system's performance and to keep maintenance budgets low. In general, this overhead can be measured by analyzing response time and resource usage. However, instrumentation could also have an impact on other quality of service characteristics, e.g. reliability due to implementation errors. The degree of perturbation introduced by measurements depends on different aspects, e.g. the measurement strategy—event-driven vs. sampling based—, agent-based vs. agent-less monitoring and the granularity of instrumentation.

In order to find an acceptable degree of perturbation it is required to balance the interest between monitoring overhead and the gained detailed information on the runtime behavior. It is also important to consider the current stage of the application life-cycle. For instance, measurements with the purpose of debugging and profiling are usually performed at development time in development environments where a high degree of instrumentation perturbation is acceptable. On the other hand, logging and application performance monitoring are used primarily during operation in the production environment, which limits the accepted perturbation to a level that does not violate the system's SLAs.

Performance Measurement

One of the most important monitoring practices is *application performance management* (APM). As business success is directly influenced by the performance of the enterprise application systems that support and enables it, any performance issue that may arise during production use could impact revenue and customer satisfaction. Examples of such impacts are documented in various literature. Google reports a 20% traffic loss if their web

sites respond 500ms slower (Linden, 2006). Amazon loses 1% of revenue for every 100 ms in latency (Einav, 2019). A study conducted by Mozilla proved that users tend to leave web sites if the page is not loaded within one to few seconds (Cutler, 2010). Those examples show how important it is to monitor the response time of applications on regular basis.

APM is an agent-based practice of collecting, evaluating and interpreting performance of applications at runtime, as well as detecting, diagnosing, and resolving performance-related problems using runtime data. Data like endpoint response errors, request amount and method call instrumentation is utilized by APM to gain important insights about the behavior of the IT landscape. A state-of-the-art report in APM was developed by Heger et al. (Heger et al., 2017). Rabl et al. (Rabl et al., 2012) provides challenges current organization face when implementing APM solutions. A plethora of application performance monitoring tools have been developed since the past decade (De Silva et al., 2019). Best practices in monitoring the several layers of an EA and suggestions of well-suited tools is provided by Allspaw (Allspaw, 2008). The most mature and feature-rich APM tools are commercial products such as AppDynamics¹³, Broadcom APM¹⁴, Dynatrace¹⁵, New Relic¹⁶, Data Dog¹⁷ and Instana¹⁸, which are regularly reviewed by Gartner (De Silva et al., 2019). As an alternative to commercial solutions, also various open-source tools have been published for measuring application performance. The most known open-source projects are Nagios¹⁹, Zipkin²⁰ developed by Twitter (including the numerous forks for supporting other programming languages like Sleuth, Brave, HTrace, py_zipkin, just to name a few), Jaeger²¹ developed by Uber, PinPoint²² developed by Naver, Kieker²³ an academic project developed by the University of Kiel, or openTracing²⁴. An extensive landscape about open-source APM including monitoring agents, libraries, data transport techniques, collectors, data processing approaches as well as storage, visualization and alerting tools are provided by Novatec²⁵. A decision support guideline for choosing the best appropriate open-source solution for monitoring distributed architectures is created by Haselböck and Weinreich (Haselböck et al., 2017).

The core technique for performance measurement developed under the umbrella of APM is called *tracing* (Sigelman et al., 2010). Tracing is performed by following a request through a networked service from start to finish and collects variable level of processing details across application and network layers. This method is especially used in distributed

¹³<https://www.appdynamics.com>, last accessed: 10/28/2020

¹⁴<https://www.broadcom.com/products/software/aiops/application-performance-management>, last accessed: 10/28/2020

¹⁵<https://www.dynatrace.com>, last accessed: 10/28/2020

¹⁶<https://www.newrelic.com>, last accessed: 10/28/2020

¹⁷<https://www.datadoghq.com>, last accessed: 10/28/2020

¹⁸<https://www.instana.com>, last accessed: 10/28/2020

¹⁹<https://www.nagios.org>, last accessed: 10/28/2020

²⁰<https://zipkin.io>, last accessed: 10/28/2020

²¹<https://www.jaegertracing.io>, last accessed: 10/28/2020

²²<https://naver.github.io/pinpoint>, last accessed: 10/28/2020

²³<http://kieker-monitoring.net/apm>, last accessed: 10/28/2020

²⁴<https://opentracing.io>, last accessed: 10/28/2020

²⁵<https://openapm.io/landscape>, last accessed: 10/28/2020

environments (distributed tracing), where one request is not processed by one application but by a bulk of distributed servers in which the request is forwarded from server to server through the network in order to execute the particular business logic. A tracing server is collecting all information produced by the trace and joins each processing step for reconstructing the request call stack, i.e. the trace tree. This listing of the system calls can provide further hints on areas of an involved application that should be optimized to improve a request's response speed.

In detail, the tree nodes are basic units of work which we refer to as *spans*. A span could represent a function call in the same application, or a function call in another application that receives the request through the network. Each span is uniquely defined by a *spanID*. The request itself which is processed by the nodes is uniquely defined by a *traceID*. The identifier *traceID* and the previous *spanID* defined as *parentID* are forwarded from span to span through injecting the information into the HTTP header. Spans created without a *parentID* are known as *root spans*. The trace tree can be reconstructed based on the *traceID* and the chronological order of the *spanIDs*. The edges indicate a communication relationship between two spans. Independent of its place in the trace tree, a span is always a log of timestamped records which encode the span's start and end time, indicating the processing time and a list of annotations that describe the request in more detail, e.g. request path, port, class and method name, etc. An illustration of the trace tree is shown based on a waterfall diagram in Figure 2.13. The time between a request is sent by service A and received by service B is defined as the network latency. The time between a request is received by service B and responded to service A is defined as the request processing duration.

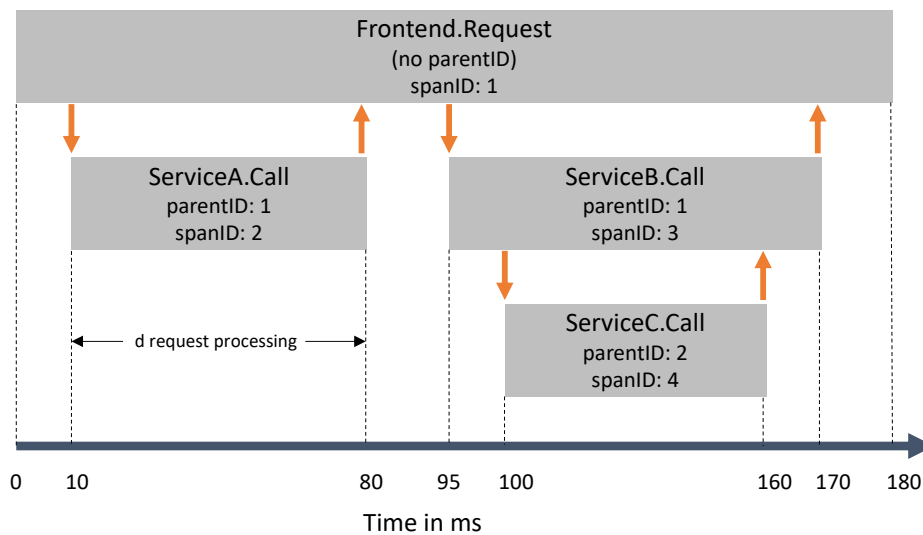


Figure 2.13.: The causal and temporal relationships between four spans in distributed tracing (Sigelman et al., 2010)

3. Related Work

The previous Section summarizes the foundations for our approach to obtain an overall understanding of the used concepts. In the following Section, we provide an overview over related approaches. We identified three different research areas which are related to our work: model-driven reverse engineering (cf. Section 3.1), EA model maintenance research area (cf. Section 3.2), and automated creation of IT landscape visualization (cf. Section 3.3). Finally, we detail in Section 3.4, to what extent our approach differs from the related work.

3.1. Model Reverse Engineering

As stated in Section 2.1.3, the automated extraction of models from IT landscapes can be regarded as (model-driven) reverse engineering and architecture recovery (Canfora et al., 2011; Chikofsky et al., 1990). Reverse engineering mainly focus on the recovery and documentation of the inner architecture of an application in order to understand the source code (Ducasse et al., 2009). The reconstruction of the outer architecture of IT landscapes for the purpose of understanding how network typologies interact with applications is called architecture recovery (AR) (Ali et al., 2017; Medvidovic et al., 2003)¹. It is concerned to provide an overview about runtime artifacts and their interactions. Especially for EAM practices, AR is fundamental to manage the complexity of emerging IT-landscapes. As we also use MDRE/AR techniques in our thesis, we present in this Section related methods and tools that consumes different static and dynamic information to recover architecture models.

A systematic literature review regarding to MDRE was conducted by Raibulet et al. (Raibulet et al., 2017). A study conducted at the university of Stuttgart (Bahle et al., 2013) evaluated 20 commercial and open-source products for IT landscape reverse engineering and related fields.

¹In several papers (Duffy, 2004; Hoorn, 2014; Ros et al., 2011), AR is also defined as architecture discovery (AD), although there exist some publications (Ali et al., 2017; Medvidovic et al., 2003) that clearly define the difference between AR and AD. AD is a top-down process that takes the requirements as input in order to create an architecture, whereas AR is a bottom-up process that extracts the architecture from an implemented system. In conclusion, the research community is currently divided over which term is the correct one.

3.1.1. Static based Reverse Engineering

Pivio² is an open-source project initiated by Wehrens (Wehrens, 2017). The objective of this framework is to provide a meta-model based language for describing a microservice-based architecture. Every microservice carries a JSON file at its root directory. This file contains several attributes that defines this particular service in detail. Example attributes comprise the name and the description of the service, the life-cycle state, dependencies to other services, as well as software dependencies. As soon as the microservice instance is started the JSON file is sent automatically to the central Pivio server, which stores the architecture models and its dependency information based on the provided description.

3.1.2. Dynamic based Reverse Engineering

Briand et al. (Briand et al., 2006) present an approach for reverse engineering UML sequence diagrams obtained from Java systems using AspectJ³. Similar to this thesis, the authors extract architectural models and communication relationships based on dynamic analysis. They primarily focus on distributed java-based IT environments.

Hrischuk et al. (Hrischuk et al., 1999), Israr et al. (Israr et al., 2007) as well as Fittkau et al. (Fittkau et al., 2015) present solutions that automatically extract architecture models based on trace data. The main purpose is to unveil the layered architecture and its corresponding interaction behavior in order to study concurrency, threading and performance issues. This also involves the interaction style, which is either synchronous, or asynchronous. Fittkau et al. extends the AR process with an interactive web-based IT landscape visualization. The authors provide different abstractions on each perspective and features two of them, namely the landscape and the application level perspective.

Besides communication extraction based on tracing data, Porter et al. (Porter et al., 2016) extends the recovery of architecture models of distributed systems with additional gossiping methods. Gossiping in distributed systems refers to the repeated probabilistic exchange of information between two nodes (Kermarrec et al., 2007). Through tracing, Porter et al. is able to identify synchronous and asynchronous communication patterns. Whereas, through the use of gossiping, the authors unveils the properties of scalability, global consistency among participating nodes, and resiliency.

Van Hoorn et al. (Hoorn et al., 2012) developed a framework called *Kieker* that also utilizes tracing data to 1) recover the information exchange behavior between microservices and 2) the internal data processing behavior within a microservice. The landscape topology is visualized via a graph-based representation. While the extracted traces can be used to reconstruct the IT landscape, the focus is primarily on analyzing transaction processing performance for capacity management.

Under the term application performance management (APM) (Menascé et al., 2002), various tools for continuously monitoring heterogeneous and distributed landscapes were

²<http://pivio.io>, last accessed: 10/28/2020

³AspectJ is an aspect-oriented programming (AOP) extension created for the Java programming language. AspectJ has become a widely used de facto standard for AOP by emphasizing simplicity and usability for end users.

developed. Gartner regularly analyzes the market of APM tools and publishes a report of the "Magic Quadrant for Application Performance Monitoring" (De Silva et al., 2019). As detailed by Gartner the following functional dimensions as a requirement for achieving APM objectives: 1) end-user experience monitoring, 2) application topology recovery and visualization, 3) user-defined transaction profiling, 4) application component deep-dive, and 5) IT operations analytics. Even though APM tools are primarily purchased for performance and capacity management, they also offer the ability to recover the IT landscape topology.

3.1.3. Hybrid based Reverse Engineering

Cuadrado et al. (Cuadrado et al., 2008) proposes QUE-es Architecture Recovery (QAR), which represents a semi-automated architecture recovery workflow that divides the extraction process into three activities, documentation analysis, static analysis and dynamic analysis. Documentation analysis covers the investigation of user manuals and design documents. Static analysis is performed by using reverse-engineering tools like Omondo UML Studio⁴. These tools automatically generate UML class diagrams by analyzing java source code. Dynamic information are extracted by profiling tools, such as JProfiler⁵ for java applications.

Bruneliere et al. (Bruneliere et al., 2014) propose MoDisco (Model Discovery), an open source project for model driven reverse engineering of IT systems. Its main objective is to provide support for activities dealing with legacy systems ranging from understanding and documentation to evolution, modernization, and quality assurance. Hereby, MoDisco consumes several static artifacts, like source code, database records, configuration files, or documentation in order to create the corresponding model representations.

Granchelli et al. (Granchelli et al., 2017a,b) presents a prototypical implementation of an architecture recovery tool dedicated to microservice-based systems. The recovery process applies different phases, that include the recovery of the model of the physical architecture and the transformation to a logical architecture model. Used data sources for extracting architecture model information are GitHub repository, the log files of the communication among microservices, and Docker runtime environment information. The main purpose of this work is to create a visual representation of the architecture for architectural reasoning, analysis, and documentation.

3.2. EA Model Maintenance

The model discovery detailed in Section 3.1 primarily focuses on reverse engineering the application layer of IT landscapes. This covers service identification, as well as unveiling the communication exchange behavior between services. However, the related work does not provide further information according to EA specific concerns like business layer

⁴<https://www.ejb3.org/>, last accessed: 2020-10-28

⁵<https://www.ej-technologies.com/products/jprofiler/overview.html>, last accessed: 2020-10-28

associations. For that reason, the following Section describes approaches that aim to discover the whole EA and how to automate the maintenance of EA models.

As stated in Section 2.2.3 EA models express an architectural description of the EA from several perspectives to address important stakeholder concerns. The maintenance of the corresponding models is still performed manually in most organizations (Lucke et al., 2010; K. Winter et al., 2010) with very little automation. This manual process is regarded as time consuming, cost intensive, and error-prone (Armour et al., 2005; Farwick et al., 2011a,b). In addition, the provided data quality often becomes challenging (Roth et al., 2013a), due to missing information, unstructured data, and wrong or outdated information. In the present time, most EA practitioners rely on a strong collaboration in order to keep the EA repository up-to-date, as Hacks et al. (Hacks et al., 2019a) identifies. They receive the required information from many stakeholders like Solution Architects, Domain Architects, Project Managers, etc. The information are further processed and transformed into the desired EA models. Initial thoughts on collaboration come from Fischer et al. (Fischer et al., 2007). The authors propose to involve EA stakeholders and data owners in the EA model maintenance process. Many authors followed this idea and elaborated solutions for different problems that arise in this scope (Buckl et al., 2011; Farwick et al., 2011a; Fiedler et al., 2013; Fuchs-Kittowski et al., 2008; Roth et al., 2013b).

Fiedler et al. (Fiedler et al., 2013) for instance, propose the integration of Enterprise Wikis into an EA repository and provide empirical grounds. This Wiki is maintained collaboratively and provide several benefits. For instance, file attachments could represent the basis for referencing EA documents. The management of these documents is supported by the integrated full-text search. Properly referenced EA documents help Architects to find the required information more efficiently. This also includes additional information that is not being captured in the model of the EA Tool.

Several research endeavours attempt to automate the creation of EA models in order to keep the EA documentation up-to-date. A literature review that summarizes those approaches was conducted by Silva et al. (Silva et al., 2020). In the following, we will provide an overview over related research in the field of automated EA model maintenance in general and how the related work addresses the issues outlined in Section 1.

3.2.1. Federated EA model Maintenance

In Section 1, we note that many EA related information is already existing within an enterprise and contained in several information systems. Following this remark, researchers as well as practitioners propose to import information to the EA repository in an automated manner to keep the EA models up-to-date. The resulting challenges are summarized by Hauder et al. (Hauder et al., 2012). The authors identified data-, transformation-, business- and organizational-, as well as tooling challenges, that need to be solved in order to achieve a practical solution. In this sense, Farwick et al. (Farwick et al., 2011b) conducted a literature review and an exploratory survey to derive requirements for an automated EA model maintenance solution. In addition, Farwick et al. (Farwick et al., 2013) provide a list of information systems that contain EA relevant information and may act as a source for automated data extraction. In the following, we list important solution examples.

Farwick et al. (Farwick et al., 2012a) elaborate a meta-model for automated EA model maintenance. An EAM tool that uses this model as its foundation, can use the contained information to drive automated and manual update processes, reconcile duplicate entries and automatically trigger updates based on expiry times. The authors divide the meta-model into three layers. The lowest layer *Instance Layer (M0)* defines concrete elements of the EA. M0 represents the instance of the second layer *Information Model Layer (M1)*. M1 defines the model classes in an object-oriented manner. The *Meta-Model Layer (M2)* finally exposes the interface of M1 and details the meta-information of the model classes.

How this meta-model is filled semi-automatically with information imported from federated information systems is detailed in (Farwick et al., 2011a). The authors present a data collection process, similar to Moser et al. (Moser et al., 2009) that consumes information from different data sources. However, in their process design, the EA repository manager and the data owners are supposed to collaboratively resolve arising conflicts. They argue that an automated conflict resolution is not possible due to the different abstraction levels of the imported EA models. In this scope, Roth et al. (Roth et al., 2013b) present a software supported process design that describes this collaboration during the resolution of conflicts in detail.

Silva et al. (Mira da Silva et al., 2016) specifies a set of migration rules for turning the manual task of migrating EA models into a faster, and automatic process. These migration rules aim to achieve two main objectives: 1) Automate the EA migration process, and 2) provide a complete set of steps throughout the migration of EA data. Nonetheless, Silva et al. identified also some limitations. First, only a tool that follows an object-oriented paradigm is capable of implementing their approach. Furthermore, they do not claim that the proposed approach covers all co-evolution scenarios, despite arguing that the most common cases are covered.

3.2.2. Runtime based EA Model Maintenance

As identified by Hauder et al. (Hauder et al., 2012) and Farwick et al. (Farwick et al., 2013) monitoring tools could be a promising information source for delivering EA relevant data and to achieve an automated maintenance of EA models. However, only a few research endeavors (Alegria et al., 2010; Buschle et al., 2012; Farwick et al., 2010; Holm et al., 2014; Välja et al., 2015) leverage runtime data for automating the EA model maintenance. The study conducted by Farwick et al. (Farwick et al., 2013) unveiled that incorporating runtime information for EA model maintenance is a challenging task as runtime data is mostly too fine granular and must be aggregated into a higher abstraction level.

The authors in (Farwick et al., 2010) presents a federated EA documentation approach that uses runtime information as one of the connected data sources. A central EA model controller retrieves application-related data from cloud platforms and a project portfolio management (PPM) tool and transforms it into an integrated model. All maintained infrastructure elements have a universally unique identifier (UUID) assigned to them, as soon as they are planned in the PPM tool. This way, once it is signaled to the central model that an instance, tagged with a planned UUID, is running, it can be inferred that a planned infrastructure element changed its status from planned to current.

Buschle et al. (Buschle et al., 2012) and Grunow et al. (Grunow et al., 2013) conduct a qualitative assessment of an Enterprise Service Bus (ESB) as the central component for extracting information flows between business applications. The authors reveal that ESBs are a profound source to discover EA models on the application layer and on the technology layer whereas there are weaknesses on the business layer. Based on the received feedback from the practitioners, ESBs are considered as a valid approach which provide correct data in most cases.

Alegria et al. (Alegria et al., 2010) uses network traffic monitoring to analyze the actual status of information systems and to verify the IT architecture model. Based on their developed network traffic analysis technique, the authors are able to automate the discovery of conceptual models of information systems. The extraction of relationship information between the discovered information systems is defined as a limitation and, hence, part of future work.

Holm et al. (Holm et al., 2015) takes up the research of Alegria et al. and assess network scanners as the main data source for an automated generation of ArchiMate based EA models. The obtained models encompass not only technical elements like infrastructure interfaces, application protocols, system software, application component, device, network elements but also business actor information. However, the authors also admit that the results do not contain many different EA model entities and often present the data too fine-grained for EA purposes.

Valja et al (Välja et al., 2015) present a generic, systematic process of how modeling automation can be achieved by using two different data sources that incorporate a network scanner and a network traffic analyzer. In order to maintain data quality, the authors elaborate certain steps that need to be taken. Finally, the data is used to automatically generate a security meta-model called P²CySeMoL (Holm et al., 2015). Reflected against typical enterprise architecture meta models, the approach is restricted to the technology layer.

3.2.3. Modern Approaches for EA Model Maintenance

Hacks et al. (Hacks et al., 2019b) took notice about the potential that resides in continuous deployment technologies for EA model maintenance purposes. They describe how to implement and partly automate an EA model evolution process from a given preexisting EA model towards a revised EA model. This process ensures the model quality by applying several quality gates and the calculation of KPIs. Only if all quality gates are passed successful in the CD pipeline, the new EA model version is released to production.

Landthaler et al. (Landthaler et al., 2018) presents a machine-learning based approach for detecting and identifying information systems in the IT landscape of an organization. The approach discovers and classifies binary strings of application executables on target machines. Hereby, the authors face the challenge that the binary strings of executables representing the same application differ depending on the devices. For further evaluation of the approach it is necessary to examine the work on a heterogeneous environment.

Johnson et al. (Johnson et al., 2016) also introduce a machine-learning technique to automate EA modeling. The solution considers Dynamic Bayesian Networks (DBNs) as a

suitable technique to apply to EA model maintenance since DBNs are probabilistic, hence capable of capturing the inherent and significant uncertainty of both as-is and to-be EAs. In addition, DBNs are dynamic, so they can represent the time-dependent nature of the architecture. Some parts of the architecture may be expected to remain stable for extended periods, while others change rapidly. The approach was tested using data produced by a network sniffer. However, the results were limited to two EA elements only, namely hosts and network messages. For all other EA elements, the paper remains theoretical.

3.2.4. Change Events that trigger EA model Maintenance

An important aspect that need to be considered for developing approaches for automated EA model maintenance is the right time when changes on EA models must be applied. This aspect centers around *change events* or *triggering events*. For the maintenance of an EA model, Fischer et al. (Fischer et al., 2007) discuss two different strategies to initiate a new maintenance cycle, these are:

periodic, which is initiated by the EA team and based on a maintenance schedule. In this regard, Fischer et al. introduces a contract that incorporates model definitions for relevant information sources and the maintenance schedule. In their periodic approach, the EA team triggers a data owner that provides the corresponding model as defined in the contract.

non-periodic can be triggered manually by the EA team as well as by data owners. A non-periodic cycle is initiated, e.g. if architecture models changed significantly. An example Fischer et al. provides is a change in the model due to project work. Upon project completion, the data owner informs the EA team about the changes which then decides whether or not to initiate a maintenance cycle for the respective information source.

Several publications (Ahlemann et al., 2012; Hanschke, 2010; P. Sousa et al., 2011) that discuss the notion of triggering events for EA model maintenance is listed and reviewed by Farwick et al. (Farwick et al., 2012b). Those publications can be assigned into one of the aforementioned categories. To some extent, the authors assume that these events are triggered by tools rather than persons. This includes for instance, project start/end, new application release, new IaaS instance, new technical services, new hardware acquired or the corresponding removal of those elements.

3.3. IT Landscape Representation

As the related work listed in Sections 3.1 and 3.2 highlight, the discovery and automated maintenance of models are important steps to manage the complexity of IT landscapes. However, a reconstruction also requires an appropriate visual representation in order to understand and to communicate the IT infrastructure complexity (Matthes et al., 2020).

In particular, the research community around Matthes underpin the importance of visual means for the analysis of EA models. An extensive research about current EA visualization tools was conducted by Roth et al. (Roth et al., 2020). The authors address in their survey both the industry perspective as well as the EA tool vendor perspective. Such tools are

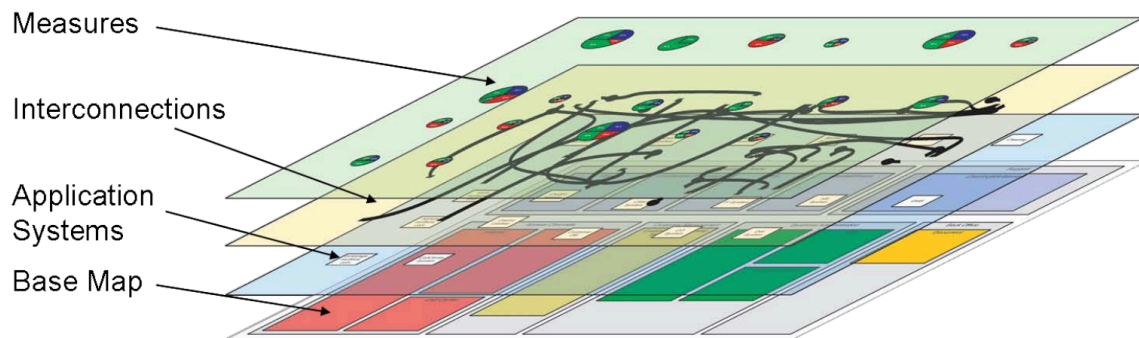


Figure 3.1.: Layered architecture of a software map (Lankes et al., 2005)

often based on a Configuration Management Database (CMDB) or similar databases and allow the management of arbitrary models, and provide different approaches for model visualizations.

A further survey in this area was conducted by Aletrati et al. (Aletrati Khosroshahi et al., 2020). The goal of the EAM pattern catalog project is to support systematically the situational adaptation and the gradual establishment of a company-specific EAM. Besides the definition of which concerns are addressed with a particular EAM pattern, as well as which methods and information are required, also the supporting representations are described, i.e. which representations support stakeholders to carry out their activities collectively.

Roth et al. (Roth, 2014; Roth et al., 2014) elaborate visualizations for highlighting differences in EAM models between different states of an EA. Similar to the proposed method of Binz et al. (Binz et al., 2013), the representation of EA models and their relationships follows a graph-based approach.

Buckl et al. (Buckl et al., 2007) discovers a large number of different visualizations for application landscapes, which they refer to as *software maps*. They categorized them into three different types: cluster map, cartesian map and graph layout map.

In order to support the visualization of different aspects on a software map, Lankes et al. (Lankes et al., 2005) proposes a layered architecture as illustrated in Figure 3.1.

The represented software map consists of a base map including organizational units and multiple layers, which are used to visualize relationships between different objects. The layers contain applications on the first layer, interconnections representing a technical aspect on the second layer as well as measures on the third layer, visualizing operational or economical aspects. Each layer has a reference layer to which the elements correspond.

Similar to Buckl et al. and build upon Lankes et al. Wittenburg (Wittenburg, 2007) elaborates a large number of different visualization approaches for EAM concerns in his PHD thesis. Wittenburg divides the individual visualizations into the IT project life-cycle. This include visualizations for requirements management, strategy and objectives management, project portfolio management, the synchronization management and IT architecture management.

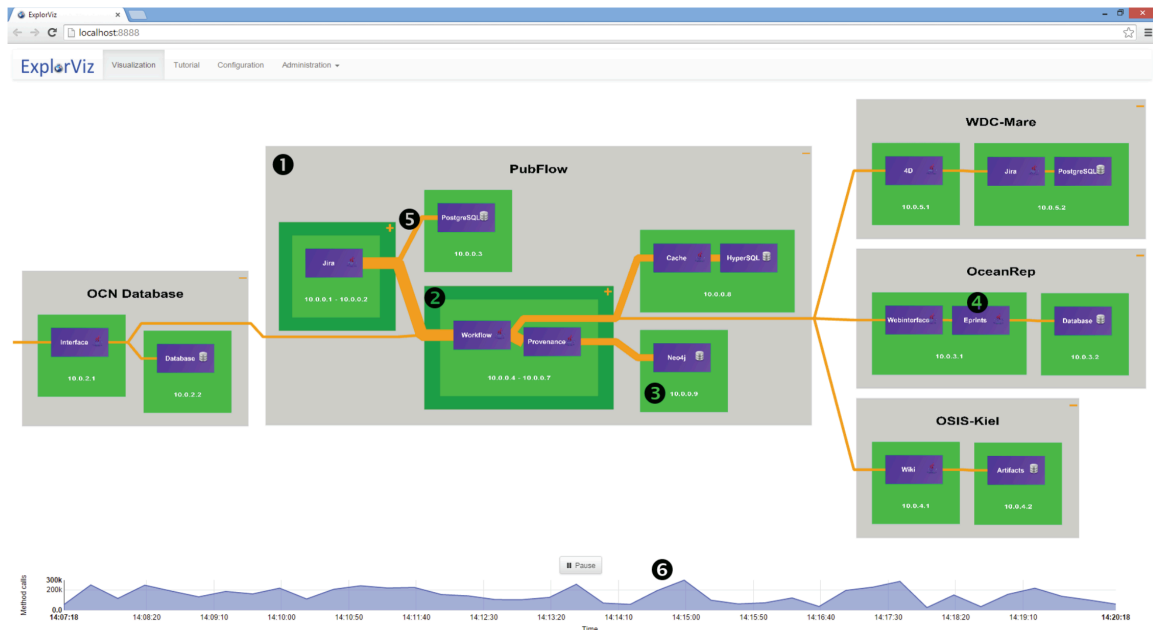


Figure 3.2.: IT landscape visualization with ExplorViz (Fittkau et al., 2015)

Fittkau et al. (Fittkau et al., 2014, 2013; Fittkau et al., 2015) develop *ExplorViz* that enables users to view into each running application while still providing the landscape overview. In general, the approach provides visualizations of IT landscapes on two abstraction levels. The landscape level provides an overview of application communications and is represented utilizing 2D elements. An example is depicted in Figure 3.2. The application level perspective is more detailed and leverages a 3D city metaphor (Knight et al., 2000) to visualize one particular application running in the landscape. The reconstruction of the EA models is performed automatically by utilizing application performance monitoring.

Frank et al. (Frank et al., 2009) presents an IT domain specific modeling language (ITML). This language focuses on modeling technical IT landscapes which are extended by a business perspective, detailed as process maps. The left side of Figure 3.3 illustrates an example. It shows various types of IT concepts in a layered perspective. For instance, IT services, software, diverse hardware, and locations amongst others. Furthermore, relationships are visualized in case the models are interrelated or dependent, e.g. software runs on hardware and enables IT services. With respect to the business perspective, the models allow for evaluating a resource's relevance for the business, e.g. by analyzing the business impact of a resource in case of its breakdown. In addition, the visualized models can be enriched with additional run-time information about the actual state of the model instances, like resource utilization, availability or transaction processing duration. This is shown by the right side of Figure 3.3.

Brown (Brown, 2018) introduces the C4 model for visualising software architectures, which is gaining popularity in many organizations. Essentially, the C4 model diagrams

3. Related Work

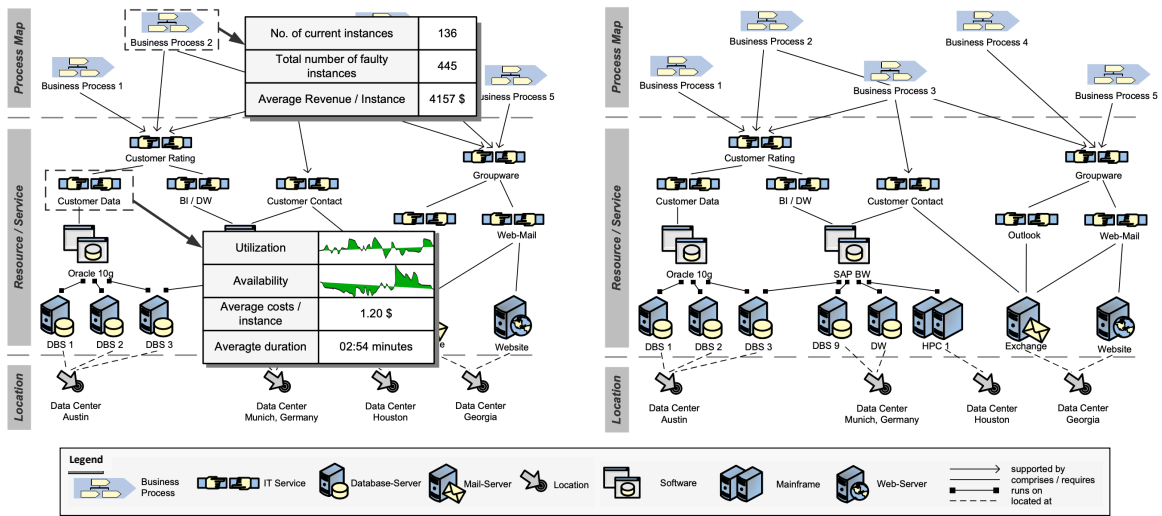


Figure 3.3.: IT domain specific modeling language (Frank et al., 2009)

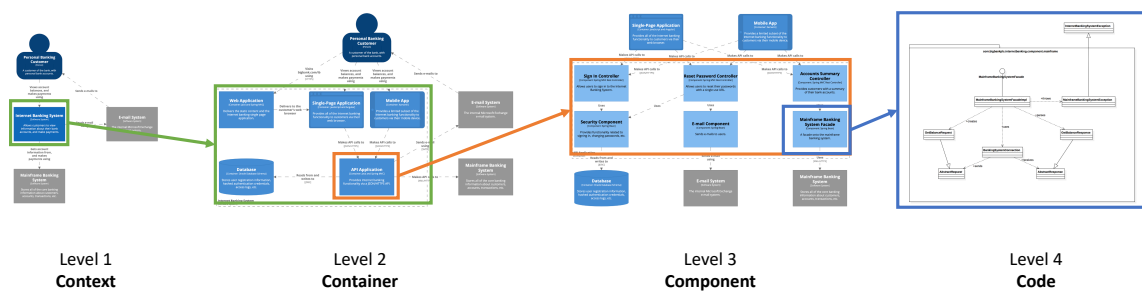


Figure 3.4.: Different levels of detail in the C4 model (Brown, 2018)

capture the complexity of software design in four levels. The primarily goal is to provide a way for software development teams to communicate their software architecture, at different levels of detail. The relationships of the different levels is shown in Figure 3.4 and include:

- *System context*, that provides a starting point and shows, how the software system in scope fits into the overall enterprise context.
- The *container* level zooms into the software system in scope and unveils the high-level technical building blocks.
- A *component* diagram identifies the major structural building blocks and their interactions within a single container.
- The *code* level details the implementation structure of a single component.

3.4. Demarcation

We reviewed literature on the automated recovery and maintenance of models and gave an overview of insights that influenced our design. In the following Section, we clarify how the solution approach detailed in this thesis differ in various aspects from the preexisting work.

Most of the related work that use dynamic reverse engineering approaches (Briand et al., 2006; Fittkau et al., 2015; Granchelli et al., 2017a; Hoorn et al., 2012; Hrischuk et al., 1999; Israr et al., 2007; Porter et al., 2016) focus on unveiling technical components and their communication relationships. However, they do mostly not incorporate infrastructure-related or business-related aspects. These systems proved to be effective on the application layer but lack to close the gap to EA related concerns. Another limitations is that the obtained models achieve only a specific abstraction level which is for some cases too fine-grained. A flexible aggregation covering different abstraction levels is not provided. The hybrid based reverse engineering approaches go a step further and try to enrich the discovered models with further information. Only the work of Granchelli et al. (Granchelli et al., 2017b) is comparable to the approach of this thesis. Their concept recovers static and dynamic information of microservices and their interrelations. However, they still ignore business-related information and do not provide a proper holistic visualization of the IT landscape. The project pivio.io initiated by Wehrens (Wehrens, 2017) proposes a similar way of instrumenting microservices with a configuration file. The content of the file is extracted as soon as the service is deployed. However, pivio.io aims to describe services from a merely technical point of view and neglect business- and infrastructure information. In addition, all information must be provided manually, which does not fully correspond to our intention.

Concerning the elaborated solutions for automating the EA model maintenance, we have found several issues. First, many researchers follow a federated approach to recover EA models (Farwick et al., 2012a; Farwick et al., 2011a; Mira da Silva et al., 2016; Roth et al., 2013b). As Farwick et al. (Farwick et al., 2013) investigated, there exists several information sources, that can be addressed for this purpose. The extracted models are transformed to the target models in order to achieve a general valid representation of the EA. The problem with these approaches is not the process description (Farwick et al., 2011a; Mira da Silva et al., 2016) or the resolution of model conflicts (Roth et al., 2013b), but the data sources themselves. The models are still inserted or generated manually in the federated information systems. That means, data completeness and actuality cannot be guaranteed. In addition, not all stakeholder concerns can be approached, as the data granularity and abstraction level differs too much between the information sources. Second, even though the incorporation of runtime data in order to extract EA models is the same approach we follow, most related work (Alegria et al., 2010; Buschle et al., 2012; Farwick et al., 2010; Grunow et al., 2013; Holm et al., 2015; Välja et al., 2015) in this scope only focus on a specific EA layer to prove their feasibility. In particular, the automated creation of business-related models are either incomplete or totally neglected, which is not the case in our solution. In addition, the feasibility of leveraging runtime information for EA model maintenance in a highly distributed, microservice-based IT landscape was not

conducted yet. Third, using CD pipelines for automating the maintenance of EA models as introduced by Hacks et al. (Hacks et al., 2019b) goes into the same direction as we do. However, the authors still rely on manual work for revising the EA models and do not leverage runtime data for this purpose. Last but not least, when considering the suitability of related meta-models (Binz et al., 2013; Braun et al., 2005; Farwick et al., 2012a; Fittkau et al., 2015; Frank et al., 2009) for EA model maintenance automation, we have found significant difficulties. A number of meta-models are not detailed enough to provide the information required for a microservice-based IT landscape management. Our intention is to address architecture-related concerns of several stakeholders, not only of Enterprise Architects. Especially, detailed information of data exchange relationships are not covered by most of the provided meta-models.

There are several conducted studies (Buckl et al., 2007; Frank et al., 2009; Roth et al., 2020; Wittenburg, 2007) about how to visualize IT landscapes and what different perspectives must be addressed. However, we only identified the work of Fittkau et al. (Fittkau et al., 2014, 2013; Fittkau et al., 2015), Brown (Brown, 2018) and Horn et al. (Hoorn et al., 2012) which developed a tool that creates a visual representation of IT landscapes based on a meta-model. A solution for the automated creation of Archimate models based on runtime data was also elaborated by Holm et al. (Holm et al., 2015). In comparison to the related work, beside the visual representation, we also provide the capability to query the information against the meta-model in order to offer the stakeholders maximum flexibility.

4. Requirement Analysis

In this chapter, we elaborate on how we envision the conceptual framework for reverse engineering EA models from a microservice-based IT environment and how to reassemble the scattered information that reside in the various information sources used along the IT value chain. Based on this conceptual framework and findings of related work, we systematically derive and describe requirements of five different categories for respective tool support.

4.1. A Conceptual Framework for Managing Models along the IT Value Chain

As stated in the previous chapters the management of models that describe any representation of an enterprise artifact provide the necessary transparency to uncover the as-is IT landscape and to support the engineering of future enterprises (Dietz, 2006). The research endeavours detailed in Chapter 3 mostly regard models in one specific aggregate state (instantiated, specialized, or populated) and restrict modelling to dateless states of an enterprise, as Aier confirmed (Aier et al., 2010). We claim that models can constitute the basis for engineering the EA by providing models in different aggregate states that are connected and managed in a linked knowledge graph. Referring to this statement, we follow the IT4IT framework (The Open Group, 2019) for conceptualizing our framework to manage EA models in a microservice-based IT landscape. The core aspect of IT4IT is to represent the IT from a value chain perspective in which each model representation undergoes various phases within the value stream. Those phases are a reflection of the aggregate state of models. The transition from one aggregate state to the next one is described visually in Figure 4.1. In the following, we explain this value stream in more detail.

Enterprise- and Domain Architects manually document and assess the IT-landscape of an organization in order to derive rough plans that optimally support the EA transformation strategy enabling IT alignment with business plans. For this purpose, low-fidelity models are *instantiated* that reflect each layer of an EA and their interconnections. This activity is carried out via IT-landscape modelling supported by EA modeling languages like Archimate (The Open Group, 2016). The output are conceptual blueprints that provide the business context of the IT landscape along with high-level architectural attributes. Finally, the *instantiated* EA models covering the blueprints are used to communicate the as-is IT landscape and future plans to other stakeholders. The created models are maintained in

4. Requirement Analysis

EA repositories, like Iteraplan¹, AdoIT² or LeanIX³ just to name a few.

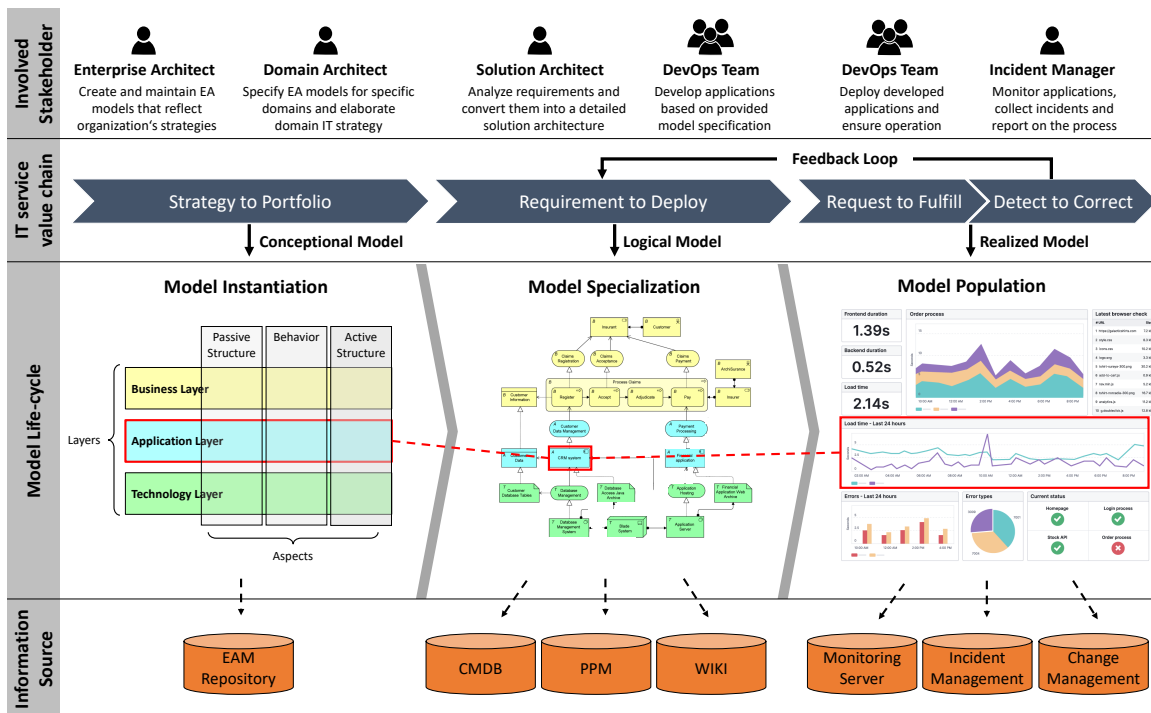


Figure 4.1.: Model interconnection along the IT value stream. Each phase in the value stream is supported by specific information systems. An holistic model management is essential for knowledge sharing as several stakeholders are involved in phase in the value stream

After the rough as-is IT landscape is modelled and future plans are created, the build phase is carried out by Solution Architects and DevOps Teams. They receive the conceptual blueprints and develops the logical models. In this phase, the instantiated models of each EA layer are translated into high-fidelity *specialized* and domain-specific models with more detailed requirements that describe how the newly requested business/IT service and its components shall be designed. The logical models are stored in Configuration Management Databases (CMDB), Project Management tools (PM) or UML charts.

Finally, the request to fulfill value stream receives the logical blueprint and is responsible for the tasks to transition the IT services into production. As soon as the service is deployed, the detect to correct value stream provides a framework for integrating monitoring, event- and incident management, or other aspects associated with service operation. In this phase DevOps teams, System Administrators, or Incident Managers keep the deployed IT service up and running, but are also responsible to observe their runtime behavior. This is achieved by having the specialized models in *executed* form, i.e. the specialized models

¹<https://www.iteraplan.de/>, last accessed: 2020-10-28

²<http://alfabet.softwareag.com/>, last accessed: 2020-10-28

³<https://www.leanix.net/>, last accessed: 2020-10-28

are transferred into models at runtime (Bencomo et al., 2019).

Our approach of how to capture and to manage the whole life-cycle of EA models along the IT value chain follows a bottom-up strategy in a three-step transformation process, which is illustrated in Figure 4.2.

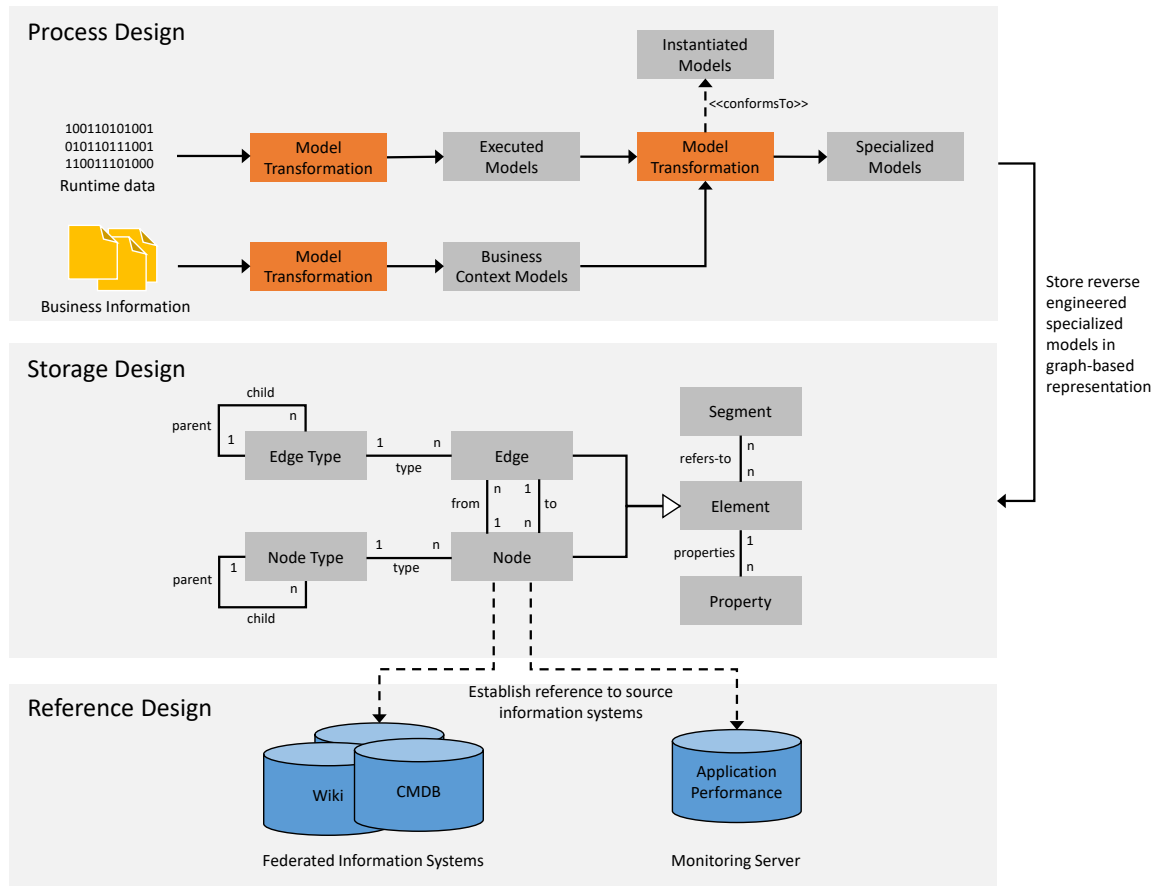


Figure 4.2.: Conceptual framework of EA model management in a microservice-based IT environment

In its core, the populated models are reverse engineered by analyzing runtime data retrieved from monitoring tools. As identified by Hauder et al. (Hauder et al., 2012) and Farwick et al. (Farwick et al., 2013) monitoring tools are a promising information source for delivering EA relevant data and to achieve an automated maintenance of EA models. Runtime data allows to recover all deployed applications and their communication relationships. In addition, they provide hierarchical information which can be used to describe deployment dependencies. They reflect models in its highest granularity level. However, runtime data primarily represent technical aspects from the application and infrastructure layer. In order to make the reverse engineered models more valuable from an EAM perspective, they are further enriched with static information that relate to the business layer, so that business-related statements can be made, e.g. which applications

are assigned to a product or domain, or which user is responsible for specific applications. As business information are not available in runtime information, they must be provided manually. Finally, the reverse engineered executed models and business models are combined and transformed into specialized models. In this process, we leverage the meta-model of the instantiated models as our transformation meta-model.

We store the reverse engineered specialized models and the model connections in a graphed-based representation. In this scope, we follow the design proposed by Binz et al. (Binz et al., 2013). That means, we store each specialized model as nodes and each model connection as edges. The edge and node types refer to the instantiated meta-model.

Furthermore, in order to reestablish the link between the models and its digital twins that are managed within the particular information systems that are used along the IT value chain, we attach a reference to each reverse engineered model. This reference is provided manually and describes an API that points at the particular model representative within the information system. With support of GraphQL as the query and manipulation language for APIs, we are able to query the model and model dependency structure either directly from the information system or from the database layer. This query language is also used to retrieve the required data for visualizing the model structures in several perspectives in order to address the concerns of all involved stakeholders.

4.2. Identification of Requirements

The conceptual framework described in the previous section defines the general architecture and which user roles need to be involved. Based on this understanding, we derive requirements of a tool-support for recovering and managing models in a central manner. The identified requirements originate from the following sources:

- the explicit requirements elaborated in the work of Farwick et al. (Farwick et al., 2011b) especially for automating EA model maintenance,
- the references and statements made by literature which have been studied during the literature research (cf. Chapter 2) and related work (cf. Chapter 3), and
- the study (Kleehaus et al., 2019b), we conducted for elaborating challenges current industries face in the documentation of microservice-based IT landscapes, as well as experience and knowledge we gained through the evaluation of initial prototypes with industry partners (Corpancho, 2019; Janietz, 2018; Schäfer, 2017).

In the following, we first present the requirements in general and then emphasize the relevance when managing and visualizing models. We categorize the requirements into *architectural requirements*, that stems from the heterogeneous environment that most enterprises are currently faced with. *Organizational requirements* derive from acknowledging that managing models and collecting required information without human intervention is unrealistic. *Integration requirements* relate to aspects that ensure a smooth integration of the concept into the IT landscape. *Functional requirements* describe specific functionality that define what the system is supposed to accomplish, and finally *Visualization requirements*

define concerns regarding the visual representation of the managed models and their dependencies.

4.2.1. Architectural Requirements

Requirement R1: Automated identification of models

The collection of model information and model dependencies must be performed automatically from every EA layer.

Microservice architectures are distributed, have a heterogeneous character, and experience frequent changes (Fowler et al., 2014). This makes it rather challenging to manage those architectures from an EAM perspective (Kleehaus et al., 2019b). Hence, in order to achieve an efficient management of microservice-based IT landscapes, the related models must be identified automatically. This includes the automated collection of model information from every EA layer, as well as the automated reconstruction of model relationships. A manual process for documenting models and keeping them up-to-date is not a successful approach, as several researchers identified (Farwick et al., 2011a; Kleehaus et al., 2019b; Roth et al., 2013a).

Requirement R2: Automated identification of structural dependencies

The collection of model dependencies between EA layers must be performed automatically.

Most EA frameworks (Office, 2011b; The Open Group, 2016; Zachman, 1987) that emerged in the last decades provide standards on how to model the EA and typically divide it into several abstraction layers (R. Winter et al., 2007). The models defined and managed within those layers are semantically related to each other. For instance, microservices are represented in the application layer and are deployed on hosts that are located in the infrastructure layer. This requirement demands a holistic management of models that covers the whole structure of the IT landscape including all model relationships.

Requirement R3: Automated identification of model communications

The identification of model communications with the application layer must be performed automatically.

Besides inter-relationships that define dependencies between EA layers, the intra-relationships detail dependencies within a specific layer. For instance, several microservices contribute to serve a user request. These services exchange data over their interface and, hence, feature an intra-relationship. This requirement extends R2 and demands to automate the identification of communication behavior within the application layer.

Requirement R4: Process support for maintaining relationships between business and technical EA layers

Runtime data only describes technical aspects of the IT landscape. Hence, a solution to extract business-related models and the according relationships must be elaborated.

The solution approach described in Section 4.1 uses runtime data to automate the recovery of EA models. However, due to the technical nature of runtime data, they describe primarily technical aspects of an IT landscape. However, the overall management of EA models does not only cover the application and infrastructure layer but also encompass the allocation of the models within the business area and the according relationship between business and technical EA layers. Unfortunately, this information remain hidden in runtime data. Consequently, this requirement demands to elaborate a solution to extract business-related information and map it with the runtime data in order to uncover a complete and holistic picture on the EA.

Requirement R5: Decentralized data collection process

Extraction of architecture-relevant information from different sources must follow a decentralized setup.

A decentralized collection of model information is inspired by Fischer et al. (Fischer et al., 2007). They argue that if data is collected at the side of the data owners, these owners can still use the modeling tools they are familiar with. Another researcher supporting this requirement is Breu (Breu, 2010). Nevertheless, the uniform automation mechanism that collects and integrates the model information into a central repository, as well as establishes the connection between each phase of the IT value chain must be deployed across all departments. Otherwise, complicated merging conflicts will occur (Roth, 2014). Hence, this requirement specifies a decentralized and uniform data collection process, and a central data analysis and storage mechanism.

Requirement R6: Model references to federated information systems used along the IT value chain.

The maintenance of model references to federated information systems used along the IT value chain and the according information extraction must be supported.

In order to establish a central model management that delivers up-to-date information about each model phase along the IT value chain it is required to maintain the references to the particular federated information systems that represent those models in different aggregate state. Those federated information systems provide further architecture-relevant data that completes the EA picture. Hence, it is required that the maintenance of those references and the according information extraction is supported by the prototype.

4.2.2. Organizational Requirements

Requirement R7: Organizational regulation of model management

An organizational process must be in place that regulates the data collection and management of models.

A key aspect of EA practice is the identification and modelling of the as-is IT landscape (Hanschke, 2009). This modelling endeavour is present in every phase of the IT value chain. Hence, several stakeholders contribute to the modelling process in order to achieve a certain perspective of the model life-cycle. However, the required knowledge is often distributed in the organization (Armour et al., 2005). For that reason, Enterprise Architects must get notified about changes from other stakeholders like Solution Architects, Product Owners, Developers etc. which is often not the case. Unfortunately, documentation in general is perceived as an unpleasant overhead and the responsible people lack of motivation to do it. Therefore, the following requirement put model management in its organizational context, by identifying stakeholders, owners as well as defining organizational regulations (Fischer et al., 2007; Kleehaus et al., 2019b; Moser et al., 2009). For instance, this requirement demands the assignment of ownership to a specific data source. This enables the recognition of the source of each model in the repository which is an important aspect of the quality assurance process.

Requirement R8: Technical support of the organizational maintenance process

The organizational maintenance process must be supported by a technical process.

The organizational process needs to be guided by a technical process that assists the involved participants to fulfill required tasks and perform maintenance activities in the desired intervals (Fischer et al., 2007; Moser et al., 2009). In this sense, the technical process is executed as a workflow engine that 1) specifies tasks that must be executed by the users and 2) lists quality gates that must be passed in the general application development process. Consequentially, this technical process also enforces the users to adhere to defined regulations in order to ensure data quality. Hereby, it is important to ensure that the technical process is adaptable in such a way that it fits into any existing organizational processes of an enterprise.

Requirement R9: Alignment of model management to superior EA concepts

The internal data structure and model visualization process of the system must be aligned to superior EA concepts.

Despite the variety of available EA frameworks (Shah et al., 2007), the description of how to maintain EA-related information is not considered detailed enough in current EA literature to assist organizations with this task (Haren, 2011; Kleehaus et al., 2019b). For that reason, sub-organizations often define their own documentation standard which is not aligned with superior EA concepts. For instance, some models like applications,

servers, products, etc. have different names in different sub-organizations. Hence, in order to overcome this issue, the system must propose an internal data structure that is 1) aligned to EAM concepts and 2) defines and persists the models based on an existing and well-known EA framework. As soon as this EA framework alignment is available and applied in every sub-organization, the communication about models is comprehensible for every stakeholder.

4.2.3. Functional Requirements

Requirement R10: Delivery of up-to-date information

The model management must deliver up-to-date information about each phase of the IT value chain

As described in Chapter 1 and inspired by the framework IT4IT (The Open Group, 2019), many different user roles contribute to the value chain of an IT service and require different information, and perspectives on the particular IT service models. Requirement R6 demands that in each phase of the value chain the users must work on a complete and up-to-date model repository. It has been confirmed in current studies (Kleehaus et al., 2019b) that especially in agile-based and microservice-based environments Enterprise Architects struggle to cope with rapid architectural changes and to document the IT landscape accordingly. Hence, Architects have to deal with high dynamics and constraints that are caused by shortened life-cycle phases of applications (Armour et al., 2005; Bubak, 2006; Dreyfus, 2007; Lam, 2004). In the specialization and execution phase of the IT value chain, it is undoubtedly necessary to have a complete and current repository of the managed models, due to the architectural and operational nature on this model perspective.

Requirement R11: Automated detection of changes

The system must be able to detect changes in the IT landscape automatically without human intervention.

The desired system must be able to detect changes in the IT landscape automatically and relate those changes to the models managed in the repository (Farwick et al., 2011a). This is an essential requirement which is also expressed in visions of future EA tools described in (Aldea et al., 2018) and (Doest et al., 2004). In this sense, it is important to evaluate in which organizational process or by which information sources this change could be triggered. Organizational process could be the development process. Information sources could potentially be anything from low-level infrastructure information, over information from release and change management tools, up to high-level governance tools. Changes need to cover not only new or modified EA models but also removed models and corresponding relationships.

Requirement R12: Automated change propagation

The system must provide mechanisms that allow for the automated propagation of changes.

In addition to Requirement R7, the detected changes in the IT landscape must be propagated to the system that updates the affected models and the corresponding relationships if required. In this scope, the system must also be able to distinguish between changes that has an impact on the models from an architectural perspective, and those changes that just represent stability and performance improvements. The importance here is that the information of changes can be abstracted into a granularity that is appropriate for all involved stakeholders among the IT value chain.

Requirement R13: Generic model transformation

The system must provide a mechanism to transform the information from incoming data to the internal model representation.

As soon as different information sources are used to achieve an holistic model management along the IT value chain, a model transformation (Brambilla et al., 2012) is required which translates the source information into the desired model representation (Moser et al., 2009). This model transformation must be as generic as possible to support many different information sources as possible. Furthermore, the data schema of the central system must not be too rigid in order to support extensions of the model-, and model relationship types. Otherwise, the system can only be used for specific IT environments which prevents the aspect of generality.

Requirement R14: Management of model evolution

The system must maintain models in different states for analyzing model evolution.

According to literature (Armour et al., 2005; Bubak, 2006; Dreyfus, 2007; Lam, 2004) current IT landscapes face high dynamics and agility that lead to shortened life-cycle phases of applications. This requirement demands that stakeholders are capable to compare different architecture revisions in order to uncover unforeseen changes and, where necessary, to intervene as early as possible. With this requirement it is also possible to evaluate how the architecture emerged over time and what impact specific changes have on the overall IT landscape performance. As a result, each model needs to have meta-data attached to it that indicates the creation time stamp and a form of expiration date in order to manage the volatility.

Requirement R15: Network-based management of models

The system must maintain the models and its relationships in a network oriented way.

Enterprise architectures in general and microservice-based IT landscapes in particular can be regarded as a network of intra-related or inter-related components (Foltête et al.,

2012; Naranjo et al., 2015; Santana et al., 2016). Those relationships of IT landscape elements can occur in different forms, like 1) elements, such as applications, are linked due to their data exchange behavior, 2) applications are deployed on hosts representing hierarchical relationships, or 3) several applications are grouped into products. Hence, we regard the graph-based management of EA models as the only efficient way to store the underlying data structures. With this approach, we empower users to examine the effects of inter-dependencies between individual elements, in contrast to aggregated patterns of occurrences as typically suggested by EA endeavours (Fürstenau et al., 2015).

Requirement R16: Automated detection of interfaces

The system must be able to detect interfaces between information systems.

As it was identified in a study conducted by Farwick et al. (Farwick et al., 2011b), the automated detection of interfaces between information systems have the highest value for architectural roles. Especially in distributed microservice environments, the number of interfaces and the amount of data exchange between applications increases a lot (Salah et al., 2016). Therefore, an automation method needs to be provided that can infer the interfaces between information systems on a high level of abstraction.

Requirement R17: Definition of KPIs

The system must allow for the definition of KPIs, that are calculated from runtime information.

According to the IT value chain, especially in the model execution phase the definition of Key Performance Indicators (KPI) are required to analyze and control the runtime behavior of model execution. Hence, this requirement demands a language with which the calculation algorithm for KPIs can be defined (Farwick et al., 2011a). The ability to gather runtime information also brings the positive effect that the actuality and fine granularity of the collected data allows for up-to-date calculation of KPIs. Hence, this requirement also demands that the KPIs an be calculated from runtime information.

4.2.4. Visualization Requirements

Requirement R18: Web-based client for model visualizations

The system must be web-based and accessible without the need to install additional browser plugins.

The striving success of the web-based and service-oriented applications is confirmed by researchers (McAfee, 2007; Zajicek, 2007) and practitioners (Bughin et al., 2009). McAfee (McAfee, 2007) transfers the underlying technology, collectively summarized as Web 2.0 to collaboration requirements within an enterprise coining the term Enterprise 2.0. A platform that has a low entrance barrier as well as an intuitive design can provide utility

for organizations pursuing model management.

Requirement R19: Visualization of structural and communication dependencies

The solution must support the visualization of structural dependencies and model communications in order to address different stakeholder concerns.

As Figure 4.1 illustrates several stakeholders are required to gather model information from every IT value chain. Hence, they must be involved in the collection process which leads to a dependency of the system on user commitment. As a recognition of their contribution, the system must service model management and model visualizations for all involved user roles, as the study in (Kleehaus et al., 2019b) identified. For instance, Enterprise Architects are interested in an aggregated overall landscape visualization, whereas Solution Architects or Product Owners require detail insight to a specific product. A recurring concern when designing model views is the need to develop representations that are understandable by both business and technical experts (Moody, 2010), or in general, to provide a medium for communication between people with different professional backgrounds (Frank, 2002). Hence, the solution must support the visualization of different perspectives of the IT landscape including structural dependencies and model communication behavior.

Requirement R20: Visualization of runtime information

The system must be able to display runtime information of each technical model in order to represent the specialized models in executed form.

As described in Section 4.1, in order to capture and to manage the whole life-cycle of EA models along the IT value chain, the models must be represented in different forms. The models at runtime (Bencomo et al., 2019) is the final phase that reveals the true runtime behavior. Consequently, the solution must be able to display the models in executed form by extending the model information and their relationships with runtime metrics.

Requirement R21: Query language for retrieving model information

The system must provide a language to query model information and to adjust the granularity of data.

The correct granularity of data is crucial to realize the benefit of an holistic model management. If the managed model information is too fine-grained, it is useless for user roles that aim at drawing strategic conclusions from it. If it is too coarse-grained, the information contained in it is too unspecific. However, as the requirement R13 demands the system to address several user roles, the queried data granularity must be adjustable to the particular user needs. Hence, the system must collect data as fine grained as possible and provide a query language that can be used to adjust the data granularity via aggregation and filtering methods.

5. Automated Model Recovery via Runtime Instrumentation

Based on the conceptual framework and requirements described in Chapter 4 as well as related work summarized in Chapter 3, in this Section, we outline a system design and the core activities for recovering models via runtime instrumentation. A detailed description of the concept is presented in (Janietz, 2018; Kleehaus et al., 2020, 2019a, 2018b; Machner, 2019; Schäfer, 2017).

We would like to point out, that the following design, and presented concepts and processes aim to reverse engineer IT landscape from a higher abstraction level. We do not refer to software architecture reconstruction (Ducasse et al., 2009), which mainly focus on reconstructing object-oriented software systems. This is mainly important for software developers to get a better understanding of the system, but is not in the field of interest of architect roles, like Enterprise Architects, Domain Architects or Solution Architects. Those roles need a more holistic view on the complete IT landscape that is in their responsibility.

First, we outline the architecture design for *MICROLYZE* that serves to build the foundation for the subsequent sections. Next, we present algorithms and workflows that enable the recovery of models and the synchronization with architecture changes. The materialization of the recovered models is necessary to keep the documentation up-to-date and to unveil emerging architectures. For this purpose, we detail our corresponding meta-model. Finally, we describe our concept to expose an interface for accessing the recovered models and how we visualize the several perspectives on the IT landscape.

5.1. IT landscape topology

In order to store each building block of an common IT landscape and to visualize the topology in the most generic way, we conducted an extensive analyses of the exposed IT landscape meta-models of modern APM vendors including Dynatrace, AppMon, AppDynamics and New Relic. According to Gartner (De Silva et al., 2019), those competitors are currently leading the market for application performance monitoring. Based on our analysis, we elaborate in the next step a general IT landscape meta-model that matches most of the vendor's specific entities and relationships. The definition of those entities is oriented on the ArchiMate 3.0.1 (The Open Group, 2016) specification in order to correspond to a well-known reference model.

We extract the vendor-specific meta-models by analyzing the exposed RESTful APIs. During this process, we recognized that not every useful information that describes the IT landscape from a certain perspective can be consumed by API requests. Especially,

transaction traces that define communication paths between infrastructure components are often not directly accessible via exposed APIs. However, we realized that the frontend clients of the APM tools utilizes APIs which are only available to the user interface, but are not documented or made public. For that reason, we also incorporate in our analysis the client-based user interfaces in order to derive a generally valid meta-model. We refrain from accessing directly the underlying databases of the monitoring vendors to retrieve more information as this led in our investigations to unforeseen performance issues. In addition, accessing the databases is mostly not possible in production environments. In the following sections, we describe the vendor specific IT landscape meta-models in more detail.

5.1.1. AppDynamics

AppDynamics provides four different monitoring agents and development SDKs for various languages, including Java, Node.JS, .NET, PHP and Python. The agents instrument *Applications, Databases, Browser* and *Mobile Clients*, and *Server* including cloud platforms. The agents itself are categorized into *App agents*, that instrument runtime processes (applications, databases and clients) and *Machine agents* which primarily collects hardware and network metrics from virtual machine, operating systems and cloud platforms. The agents report the collected metrics to the AppDynamics *Controller*, which basically represent the monitoring server. The *Controller* retrieves, stores, calculates baselines for, and analyzes performance data reported by the particular agent. AppDynamics exposes various APIs that can be categorized as server-side APIs, which are served by the *Controller*, and agent-side APIs.

AppDynamics provides multiple ways to specify the names of the instrumented applications and physical devices. The recommended way is to set up a configuration file "controller-info.xml" that is stored in each application's path. This file contains a unique combination of *Tier*, and *Node* name. In case the application should be a member of a specific *Business Application*, the according name must also be provided. Besides the configuration file AppDynamics also supports system properties and environment variables (in particular for Docker deployments) for name specifications.

Via the exposed APIs, we are capable to recover the IT landscape meta-model depicted in Figure 5.1. In the following, we describe the entities in more detail:

Business Applications: A *BusinessApplication* encompasses all components that are required to fulfill the functionality of a specific information system. It is a complete set of interacting runtime artifacts including web and mobile applications, backend components like databases or message queues, as well as tiers and business transactions that serve the *BusinessApplication* mission. Based on the recorded tracing data, AppDynamics groups entities to one *BusinessApplication* that frequently occur in the same transaction flows. *BusinessApplication* are stored in the database and can be named via manual input. Performance metrics among the *BusinessApplication* components are aggregated as well in order to provide the DevOps a complete picture of the overall *BusinessApplication* performance. In AppDynamics, there are no correlations between separate *BusinessApplications*.

However, it is possible to create copies of the same *BusinessApplication*. A typical example of using multiple *BusinessApplications* is when you have separate staging, testing, and production environments for the same website. In this case the three business applications are essentially copies of each other.

Tiers: *Business Applications* contain *Tiers*. A *Tier* entity represents an instrumented application such as a web application, a Node.JS server, a JAVA application, or a virtual machine, etc. A *Tier* encompasses multiple application instances that perform the exact same functionality. *Tiers* helps to organize and manage logically related applications. A *Tier* is composed of one *Node* or multiple redundant *Nodes*. One example of a multi-node *Tier* is a set of clustered application servers. There is no interaction among *Nodes* within a single *Tier*. The traffic within a *Business Applications* flows between *Tiers*. The "originating" *Tier* defines the application that receives the first request of a *Business Transaction*. A "downstream" *Tier* is an tier that is called from another tier. The different flows of transactions within a *Business Applications* is modeled by *Business Transactions*.

Nodes: A *Node* is the basic unit of processing that AppDynamics monitors. Basically, a *Node* represents an AppDynamics agent that is either an *App agent* or a *Machine agent*. *Nodes* belong to *Tiers*. A *Node* cannot belong to more than one *Tier*. An application that is represented by a *Tier* could consist several *Nodes* if the application is load balanced.

Machines: A *Machine* consists of hardware and an operating system that hosts applications. The operating system can be virtual. A *Machine agent* instruments a *Machine* to collect runtime data about hardware and network devices, such as CPU activity, memory usage, disk reads and writes, etc.

Business Transactions: A *Business Transaction* is a transaction trace, i.e. a collection of user requests that accomplish a logical user activity within a *Business Application*. A single request is a *Business Transaction* instance. A *Business Transaction* defines the request flow through the *Business Applications* defined by an entry point and a processing path across *Tiers* like application servers, databases, and other infrastructure components.

Service Endpoints: *ServiceEndpoints* define specific application services and detail entry points of requests. AppDynamics provide metrics about *ServiceEndpoints* which represents a subset of the metrics for a *Tier*. *ServiceEndpoints* are similar to exposed APIs that can be accessed by other applications.

Backends: Any detected out of process components that are involved in *Business Transaction* processing, such as *Databases*, *Message Queues*, *Remote Services* or unknown components are collectively known as *Backends*. If AppDynamics detects a request to an unknown application it records it as *Uninstrumented*. *Backends* are recovered from exit point instrumentation placed in the application code. An exit point is the location in the code where an outbound call is made from an instrumented *Node*.

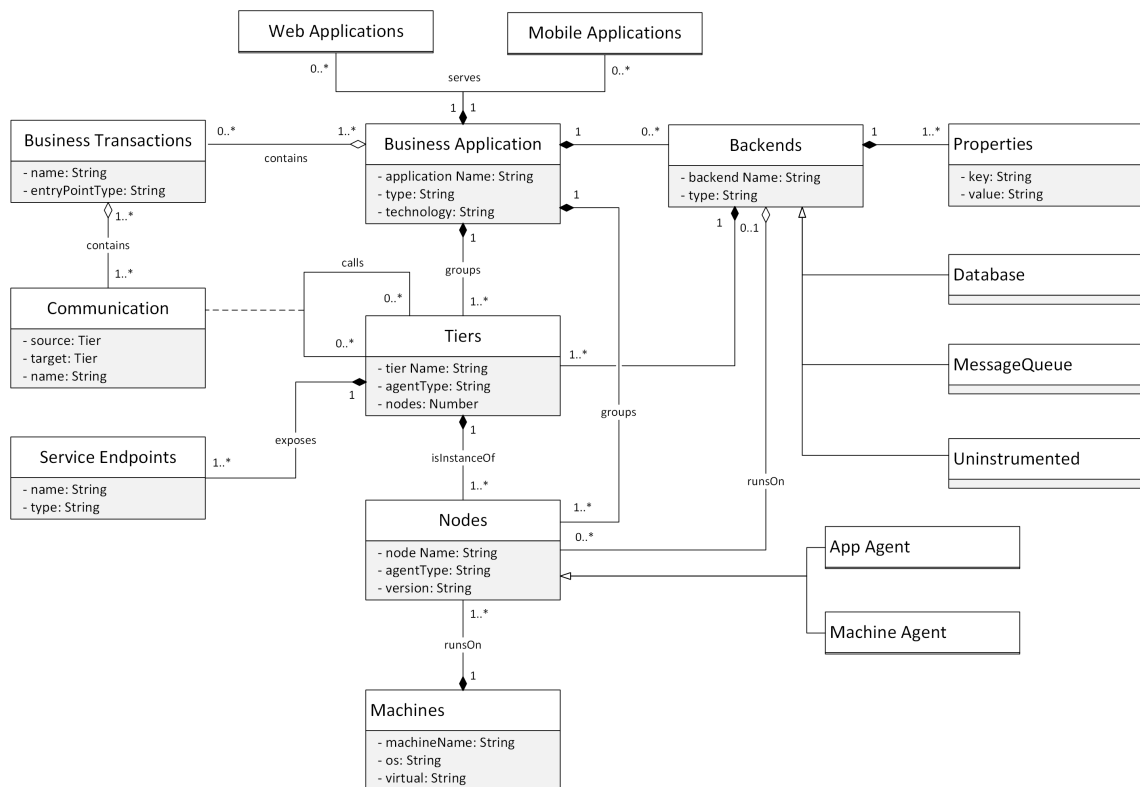


Figure 5.1.: AppDynamics meta-model (A. Inc., 2020)

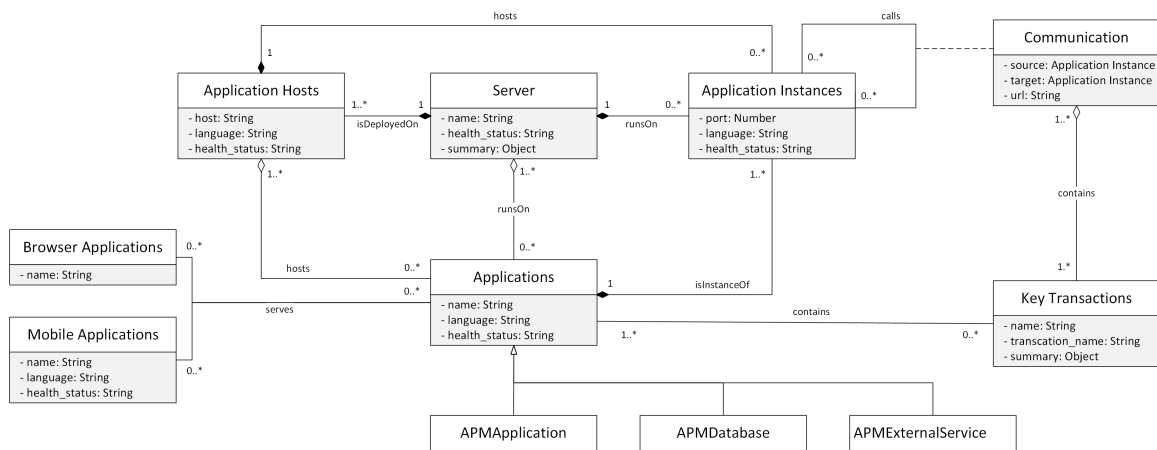


Figure 5.2.: New Relic meta-model (N. Inc., 2020)

5.1.2. New Relic

The New Relic agents can be grouped into four categories. 1) APM agents collect application specific metrics (APM) and distributed traces. Currently, the following languages are supported: Java, .Net, C, Go, Node.JS, PHP, Python and Ruby. 2) The Browser agent monitors Web Browsers by injecting JavaScript snippet into the front-end pages, regardless which framework is used. It collects metrics about page load performance, Ajax calls, geographic localization, JavaScript errors, and others. 3) Infrastructure agent are installed on operating systems (Windows, Linux) and cloud based environments (Amazon Web Services, Google Cloud Platform, and Microsoft Azure). The agent provides host specific runtime data including hardware and network performance, and resource utilization as well as event logs. Database monitoring is also covered by the Infrastructure agent. 4) Mobile agents are responsible to deliver crash reports, request performance and user behavior.

New Relic provides different ways to name applications. The recommended approach is the specification of the application's name in a configuration file "newrelic.yml" that is assigned to the application agent's directory. Further options are set up of system properties with JVM arguments, or environment variables during deployment of containers. Based on the exposed server API, we are capable to recover the IT landscape in the Figure 5.2 illustrated scope. In the following, we describe the entities in more detail:

Applications: The *Application* details all generic software applications that are instrumented by the APM agent. Each monitored *Application* is identified by its assigned name and an unique ID. The *Application* is categorized in 1) *APMAApplication* which represents an application component written either in Java, C, GO, PHP, Python, and Ruby, 2) *APMDatabase* groups all instrumented databases, and 3) *APMEExternalService* indicate non-instrumented applications that occur in recorded tracing data but cannot be assigned

to known applications. In addition, the *Applications* entity contains a self-relationship in order to indicate communication between applications in request processing.

Application Instance: The *Application Instance* entity provides general information about each specific instance of an *Application*. The instances represent the runtime processes of an *Application*. Every *Application* contains at least one *Application Instance*.

Application Host: The *Application Host* resource provides information about the host in which *Applications* reside. It also represents an aggregation of all the *Application Instances* that might be running on the *Application Host*. New Relic summarizes physical machines, virtual machines, and cloud instances to *Application Hosts*. Whereas physical machine primarily means the operating system on which *Applications* run and has dedicated physical resources, including memory, cpu and storage.

Server: A *Server* describes a physical device that has one or more *Application Hosts*. The *Server* entity provides, besides device meta-information, general information about real-time hardware related metrics. In addition, *Servers* encompass cloud environments like Amazon Web Services or Microsoft Azure.

Key Transaction: The *Key Transaction* entity represent transaction traces that has been marked as particularly important, since they either indicate defined key business events, such as signups or purchase confirmations, or transactions with a high performance impact – a transaction that is regularly slower than other transactions and might affect the user experience. *Key Transactions* represent only a subset of transaction traces. In order to obtain all traces for recovering any communication path between *Applications* the required data must be collected via accessing the browser-based user interface.

Mobile and Browser Applications: Those both entities provides general information about mobile and browser-based applications monitored by New Relic, including active users, interaction time, crash rate, and http error rate.

5.1.3. Dynatrace

The monitoring vendor Dynatrace provides a full-stack monitoring covered by one single monitoring agent called *Oneagent* that is responsible for collecting all monitoring data produced on one host, supporting on-premise and cloud environments. Dynatrace leverage bytecode injection in a huge scope in order to reduce agent configuration to a minimum. *Oneagent* that is installed on the host injects itself in processes of known technology patterns such as Node.JS, Java, Python, .Net, PHP, Go, C/C++ and others. Browser applications are instrumented as well by injecting JavaScript tag into the HTML of each application page that is rendered by the web servers. With one agent that is capable to instrument a large amount of different technologies from scratch, Dynatrace generates a complete picture of the IT landscape by default. This is different to other solutions, e.g. AppDynamics and NewRelic, as they require a manual integration into application and configuration effort.

In addition, Dynatrace provides an agent SDK that enables manual instrumentation of applications to extend end-to-end visibility for frameworks and technologies for which there is no default support available yet.

Dynatrace automatically detects names of *Processes* based on basic properties of the application deployment, configuration or code level inside. For instance, some technologies allow to give deployed application explicit names. In Spring Boot applications this name is stored in the *spring.application.name* property included in *application.properties* file. In Node.JS the application name is provided in the *package.json* file. For other technologies, the default name can be provided by the environment variable *DT_APPLICATIONID=<name>*. Figure 5.3 illustrated the meta-model we are able to recover. In the following, we describe the entities in more detail:

Applications: In Dynatrace the term *Application* refers to the front-end part that can be accessed via a browser or a mobile app. It serves as the user interface. *Application* are built upon *Services* that process incoming requests. Dynatrace monitors the trace of each request to uncover all individual components like services, databases, processes, hosts, etc. that work together in order to collectively deliver what the end users view as a complete application. The metrics collected on *Application* level encompass all KPIs that has an affect on the user experience. *Web Applications* are automatically named on domain level. *Mobile Applications* based on the app name.

Service: The *Service* entity basically represents server-side methods that are accessible via application interfaces and are responsible to consume and process requests like web requests, web service calls, remote procedure calls, and messaging. For instance, Dynatrace considers an API interface controller of a microservice to be a *Service*. In case Dynatrace detects database requests executed by an *Application*, the request itself is defined as a database service. *Services* may also call other services indicating a communication relationship between two processes. *Services* are categorized into web request services, web services, database services, messaging and queueing services, remote services, as well as background activity services.

Process: A *Process* is an instance of an executed computer program. It defines a runtime artifact that consumes hardware resources. *Processes* serve as containers that host *Services*. In general, *Processes* provide topology information, whereas *Services* uncover code-level insights. For example, a Tomcat *Process* hosts a web application in the form of a server-side *Service*. A *Process* is associated to a single *Hosts* and therefore host-centric, whereas *Services* are request-centric and therefore typically span across multiple *Hosts* in a *Data Center*. The technology the *Process* is based on, like Java, Java application servers (Tomcat, WebSphere, Weblogic, Glassfish, JBoss), Node.js, .Net, PHP, C/C++, NGINX, Apache HTTP, IIS, etc., including container information like Docker is assigned directly to the *Process* entity.

Process Groups: *Process Groups* exists of several *Processes* that perform the same function.

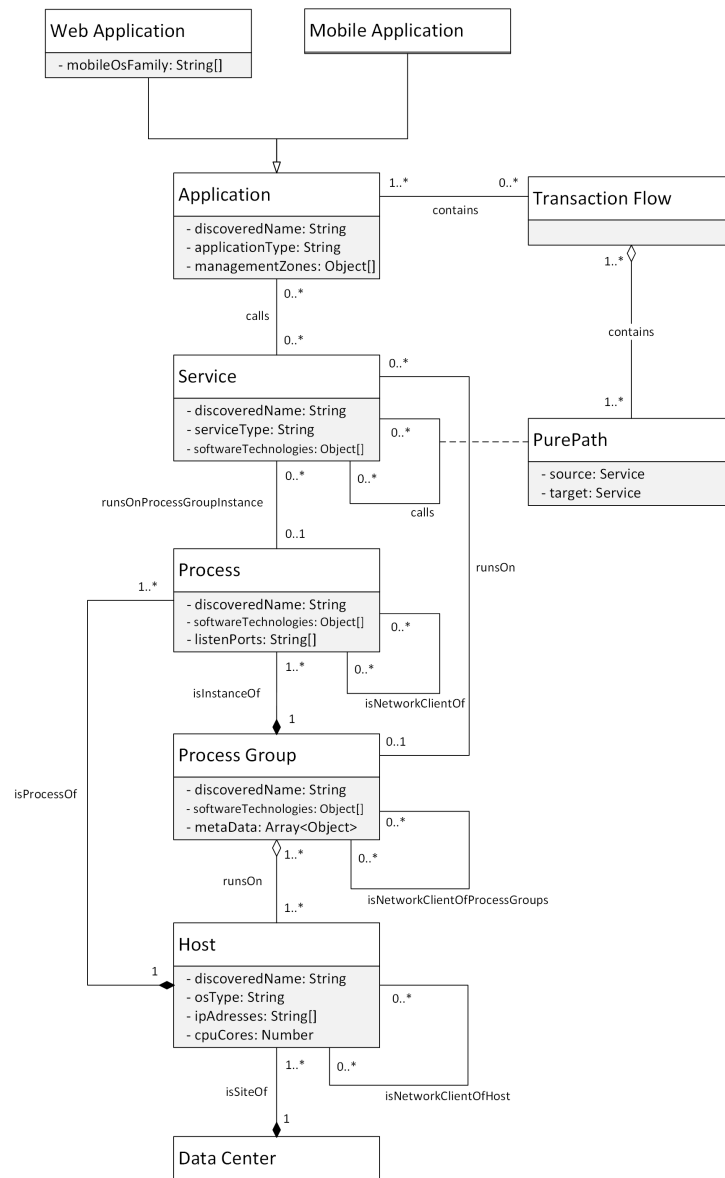


Figure 5.3.: Dynatrace meta-model (D. Inc., 2020)

Process Groups span across multiple *Hosts*. For instance, a load-balanced application is deployed on a cluster of servers with each server running the same *Process* in support of multiple *Hosts*. Dynatrace recognizes automatically related *Processes* and assign them to a *Process Group*. When Dynatrace detects multiple *Process Group*, it assumes that the *Process Group* represent separate applications, or at least separate logical parts of a single application. Therefore, *Process Group* represent boundaries for the *Services* they contain.

PurePaths: *PurePaths* are traces of transactions through the backend system. Multiple *Services* can participate in a single *PurePath*. The observation of a transaction trace covers called interfaces and include asynchronous communication over message queues. In terms of data structures, *PurePaths* are realized as trees with arbitrary width and depth whereby each node represents a *Service* and each edge represents a call between connecting nodes. Dynatrace is able to aggregate transaction traces in order to unveil communications between higher order elements, i.e. *Process*, *ProcessGroups* or *Hosts*.

Host: A *Host* in Dynatrace is a virtual or physical machine on which *Processes* run and hardware resources are assigned. They are mostly regarded as operating systems. Each *Host* is mapped to one *Data enter*. Dynatrace generally names the detected hosts based on their DNS names.

Data Center: *Data Center* primarily specifies the real-world geographic location on which *Hosts* are running and *Processes* are deployed. From the perspective of Dynatrace, a *Data Center* is either a grouping of virtual machines running in an cloud platform or a set of vCenter-based virtual hosts that transmit data to Dynatrace via a single ActiveGate.

5.1.4. AppMon

AppMon is the precursor of Dynatrace and mainly offers application performance monitoring. The core technology of AppMon is identical to Dynatrace. Monitoring agents are injected into application processes, from where they collect performance metrics and send them to the monitoring server. However some monitoring features are missing and parts of the IT landscape entities are named differently. The meta-model of AppMon is illustrated in Figure 5.4. It is rather similar to the meta-model of Dynatrace. However, some terms were renamed: *Agents* represent *Processes*, *AgentGroups* are *ProcessGroups*, and *Sites* reflects *DataCenters*. An exception represents AppMon *Applications*. Whereas Dynatrace defines *Applications* as the front-end part that can be accessed via a browser or a mobile app, AppMon's *Applications* are logical groupings of an arbitrary amount of *Agents* that cooperate to perform some business task.

5.1.5. Meta-Model Transformation

In the next step, we transform the APM vendor specific meta-models to a generally accepted meta-model that will represent the persistence layer of MICROLYZE. For this purpose, we elaborate a mapping table that translate the meta-model entities of the APM

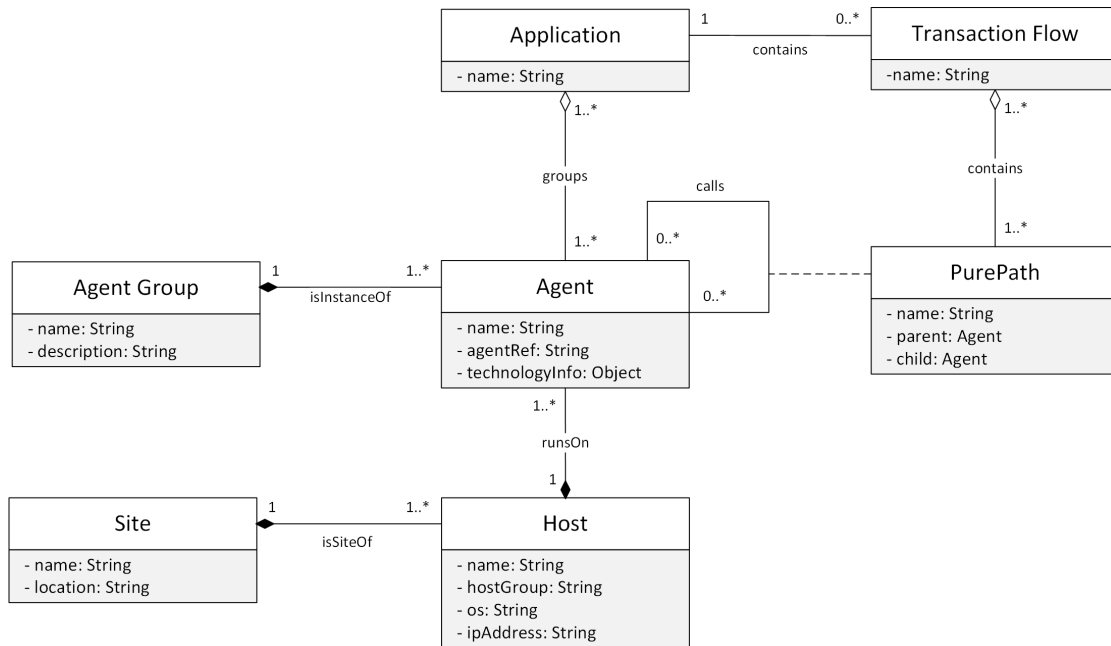


Figure 5.4.: AppMon meta-model

vendors to the ArchiMate specification. The mapping table is shown in Table 5.1. The Table unveils that every meta-model follows mostly the same schema but uses different names for describing identical entities. Only a few entities do not have one concrete counterpart. For instance, *Business Applications* as defined in AppDynamics cannot be extracted from NewRelic or Dynatrace via API request or direct access to the Web UI. We consider, that *Business Applications* are not defined in those tools. Furthermore, transaction traces are mostly not directly accessible via API request. Hence, we must access directly the user interface in order to extract tracing information. In NewRelic, retrieved *Key Transactions* represent only a subset of all traces. This also applies for AppDynamics. In Dynatrace and in its previous version AppMon, tracing data is defined in combination of *PurePath* and *Transaction Flows*. However, they obtained data is also restricted to a certain maximum timeframe.

The entities that define runtime processes (*Application Instance, Nodes, Process, Agents*) are not translated into an ArchiMate model, as we consider this information as 1) a runtime aspect that changes frequently due to scaling behavior and 2) it is already aggregated in *Applications, Tiers, Process Groups and Agent Groups* and can be easily obtained by accessing the related API of the monitoring tool.

As stated in Section 2.2.4 ArchiMate differences between, inter alia, structure elements and behavior elements. Structure elements are the subjects that can perform behavior. They are primarily defined as "nouns" whereas behavior elements are listed as "verbs". Structure elements represents resources and behavior elements represent the dynamic aspects of the enterprise. In the following, we describe the structure and behavior entities that form our general meta-model in more detail:

Table 5.1.: Meta-model transformation for deriving a generally accepted meta-model for Microlyze

NewRelic	AppDynamics	Dynatrace	AppMon	Microlyze
Mobile Application	Mobile Application	Mobile Application	n/a	Mobile Device
Browser Application	Web Application	Web Application	n/a	Browser Device
APMApplication	Tiers	Process Group	Agent Group	Application Component
Application Instances	Nodes	Process	Agent	n/a
n/a	Service Endpoints	Service	n/a	Application Service
n/a	Business Applications	n/a	Applications	Application Collaboration
Key Transactions	Business Transactions	Pure Path / Transaction Flow	Pure Path / Transaction Flow	Application Interaction
Application Host	Machines	Host	Host	Node
Server	Machines	Data Center	Site	Facility
APMDatabase	Database	Service	n/a	Application Component
n/a	Message Queues	Service	n/a	Application Component
APMExternal Service	Uninstrumented	Service	n/a	Application Component

Device: A device is a physical IT resource upon which system software and artifacts may be stored or deployed for execution.

We model browser-based and mobile-based applications as devices, as the user interface of any software is only accessible via a Smartphone or a Desktop Computer. It therefore represents a physical IT resource with processing capability. The user interface is deployed on a *Device* and forwards all requests to the processing applications, such as microservices.

Application Component: An application component represents an encapsulation of application functionality aligned to implementation structure, which is modular and replaceable. It encapsulates its behavior and data, exposes services, and makes them available through interfaces.

Microservices are self-contained units, which are independently deployable, reusable, and replaceable (cf. Section 2.3). Its functionality is only accessible through a set of exposed interfaces. With this definition, they fit into the model description of *Application Components*. We also assign databases, message queues, and any other backend application to *Application Components*, as ArchiMate does not distinguish between those artifacts. However, in order to keep this information, we create an enumeration entity *Application Type* which stores the particular specialization of *Application Components*. This enumeration is referenced by the *application_type* attribute. As *Application* (NewRelic), *Tiers*, *Process Groups* and *Agent Groups* represent an aggregation of runtime processes which perform the same functionality, we map those entities to *Application Components*.

Application Collaboration: An application collaboration represents an aggregate of two or more application components that work together to perform collective application behavior.

Application Collaborations are recognized in the entities *Business Applications* (AppDynamics) and *Applications* (AppMon), as they define an aggregate of two or more runtime processes that cooperate to perform some task. We did not identify such entities in NewRelic and Dynatrace.

Application Service: An application service represents an explicitly defined exposed application behavior.

As defined in ArchiMate, *Application Service* exposes the functionality of components to their environment. This functionality is accessed through one or more *Application Interfaces*. Thus, *Application Services* represent methods that use, produce and yield data objects. In APM, traces between *Application Components* are recorded on method level. Each outbound request is triggered by the caller method and each inbound request is processed and

answered by the callee method. The communication itself is performed through the *Application Interface*.

Not all investigated monitoring tools provide a separate entity for *Application Services*. Dynatrace and AppDynamics represents this context via Service and Service Endpoints respectively. *Application Services* are not directly monitored as separate entities by NewRelic and AppMon. In order to keep the data model as general as possible and allow for conformance with other APM tools, we create two dummy *Application Services* for each recovered *Application Component*. One service represents an interface for incoming requests and the other for outgoing requests. They strictly follow a naming convention where "_IN" or "_OUT" is added to the ID of the respective *Application Component*. To summarize, we define *Application Service* as the methods of *Application Components* that can be accessed via *Application Interfaces*.

Application Interface: An *Application Interface* represents a point of access where application services are made available to a user, another application component, or a node.

An *Application Interface* specifies how the functionality or an *Application Service* of an *Application Component* can be accessed. In microservice-based environments RESTful requests are primarily used for calling interfaces. Other messaging mechanisms are Remote Procedure Calls (RPC), Remote Method Invocation (RMI) the object-oriented equivalent of RPCs, or Remote Function Call (RFC) which represents the standard SAP interface for communication between SAP systems.

Application Interfaces are assigned to *Application Services*, which means that the interface exposes these services to the environment. In case dummy *Application Services* must be created, all *Application Interfaces* are assigned to the "_IN" - *Application Service* of the respective called *Application Component*. A description of *Application Interfaces* is often provided in API documentation and administration tools such as Swagger¹ or Apigee².

Application Interfaces are not directly accessible via the exposed APIs of the investigated monitoring tools. Similar to *Application Services* this information can only be uncovered by analyzing the communication path between applications visualized in the respective UIs. In AppDynamics interfaces are described in the *Service Endpoints* entity.

Application Interaction: An application interaction represents a unit of collective application behavior performed by (a collaboration of) two or more application components.

The communication behaviors of *Application Components* that participate in an *Application Collaboration* can be described in the *Application Interaction* entity. In general, it defines a detail trace of a user request that accomplishes a logical user activity. The trace between *Application Components* are modelled on *Application Service* level which, in turn, realize them based on the exposed *Application Interface*. Hence, each single *Application Interaction*

¹<https://swagger.io/>, last accessed: 2020-10-28

²<https://cloud.google.com/apigee>, last accessed: 2020-10-28

record contains the information of the *Application Collaboration* in which the interaction takes place and two attributes of *Application Service* – the caller and callee service. In sum, a complete *Application Interaction* is described by an amount of hierarchical caller - callee relationships. With this approach, a whole request trace can be stored. In the investigated monitoring tools, the *Application Interactions* are modelled via *Key Transaction*, *Business Transaction*, or is encapsulated in the *PurePath* and *Transaction Flow* visualization.

Node: A node represents a computational or physical resource that hosts, manipulates, or interacts with other computational or physical resources.

Every *Application Component* is deployed on *Nodes*, which represents the host environment that provides required physical resources for data processing and storage. Hereby, we distinguish between operating system, virtualization, containerization, or cloud environments. We map the monitoring entities *Application Host*, *Machines* and *Host* to the *Node* entity.

Facility: A facility represents a physical structure or environment.

In ArchiMate notation, a *Facility* is a specialization of a *Node*, which provides the physical resources like housing or locating for facilitating the use of equipment required for data processing and storing. As it is typically used to model factories, or buildings, we use the entity to generalize *Server*, *Machines*, *Data Center* and *Site*. Each *Node* is assigned to one *Facility*.

The UML representation of the general meta-model is depicted in Figure 5.5. We define three different relationship types in order to express entity connections:

- *hierarchy* relationships represent technological deployments or associations. In general, they express an hierarchical relationship from a deployment perspective. In the meta-model, we name *hierarchies* with *runs on*. For instance, "An Application Component *runs on* a Node".
- *grouping* expresses an aggregation of entities in the form of a parent-child relationship. That means, child entities that rely on the same superior parent entity are grouped together. In the meta-model, we represent *groupings* with the verb *contains*. For instance, "A Product *contains* a Business Service".
- *communication* relationships either represent data exchange between entities, such as communications, or remote execution of application interfaces. The assigned verb for communications is *calls*: "An Application Service *calls* another Application Service". This basically, represents communications between Application Components that happen on Application Service level, which are realized through Application Interfaces.

To summarize, in the meta-model shown in Figure 5.5 *Devices* define the presentation layer of an information system that is directly accessible by end-users. *Devices* call backend

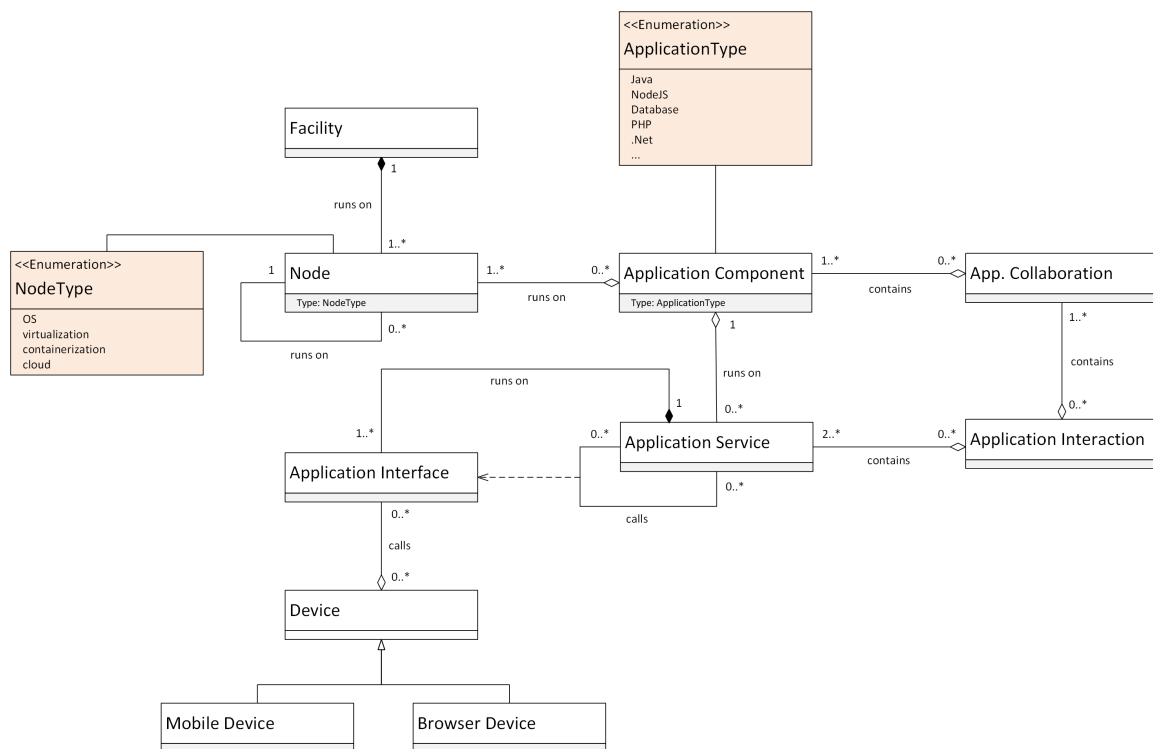


Figure 5.5.: Derived meta-model for MICROLIZE

components represented by *Application Components* entities via *Application Interfaces* that constitute the access points where *Application Services* are made available to end-users, or other *Application Components*. Communications between *Application Components* are realized through their assigned *Application Services*. Hereby, a *Application Service* calls another *Application Service* through the exposed *Application Interface*. *Application Components* run on *Nodes* that represent the host which provide required hardware resources and services. A *Node* is classified in operating system, virtualization, containerization, or cloud space. In case physical hardware, such as mainframe servers are maintained in the company’s infrastructure, the corresponding *Facility* indicate the location where the server reside. *Application Collaborations* contain *Application Components* that work together to realize a certain functionality of a specific information system. An *Application Interaction* represent a certain communication context (trace). It contains all corresponding *Application Services* within a specific *Application Collaboration* that call each other for executing a user request.

In the following sections, we define each entity in the meta-model as an *Architecture Model (AM)* that is extracted from runtime data and causally connected to the particular components of an IT landscape. The *Architecture Models* reflect the running system with respect to functional and non-functional concerns. It defines a concrete entity in one specific EA layer, i.e business, application and technology layer.

5.2. System Design

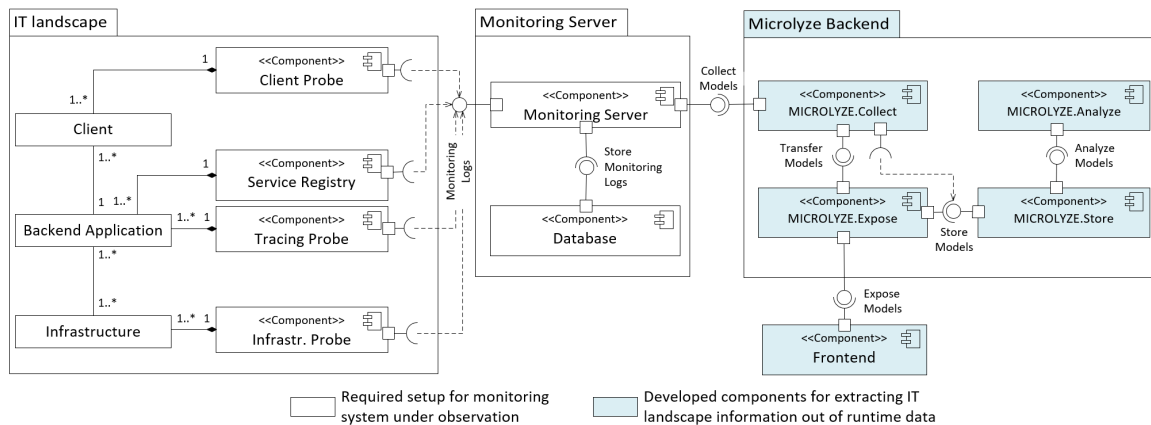


Figure 5.6.: The interplay between the IT landscape under observation (SUO), the corresponding APM server and our developed tool *MICROLYZE*

After designing our meta-model for managing models of microservice-based IT landscapes, we continue with the elaboration on a system design for recovering, analyzing, storing and exposing the models. In the following sections, we outline the fundamental concepts.

We schematically divide the architecture into four main domains. The *IT landscape*

represents the system under observation (SUO) that constitutes the runtime environment whose AMs and their location in the overall IT landscape architecture are supposed to be recovered by *MICROLYZE*. In order to receive the required runtime data, several monitoring probes need to be deployed on the IT landscape, in particular on infrastructure hosts, applications and web clients. Those probes collect measurements and traces that are streamed as *Monitoring Logs* to a central *Monitoring Server* that stores the data for a configured period of time in its *Database*. The *Monitoring Server* exposes those runtime data via APIs. *MICROLYZE* itself is separated into a backend- and frontend-based application. The backend contains a runtime data collection part referred to as *MICROLYZE.Collect*, an analysis part, defined as *MICROLYZE.Analyze*, a component for storing the models in a database named as *MICROLYZE.Store* and finally a component *MICROLYZE.Expose* that provides an interface for exposing the recovered models and its dependencies. *MICROLYZE.Collect* consumes the runtime data provided by monitoring tools through their APIs and browser-based clients. *MICROLYZE.Analyze* provides the corresponding infrastructure to analyze runtime data and to recover architecture-related models and their relationships, which cannot be obtained by the conventional APIs. The *MICROLYZE.Store* persists the recovered architecture in its current and previous states. *MICROLYZE.Expose* serves as entry point for accessing the stored EA models. It exposes a GraphQL interface with the required query definitions. The frontend application provides several views for visualizing the recovered IT landscape from different perspectives. Figure 5.6 depicts the core components and their assembly.

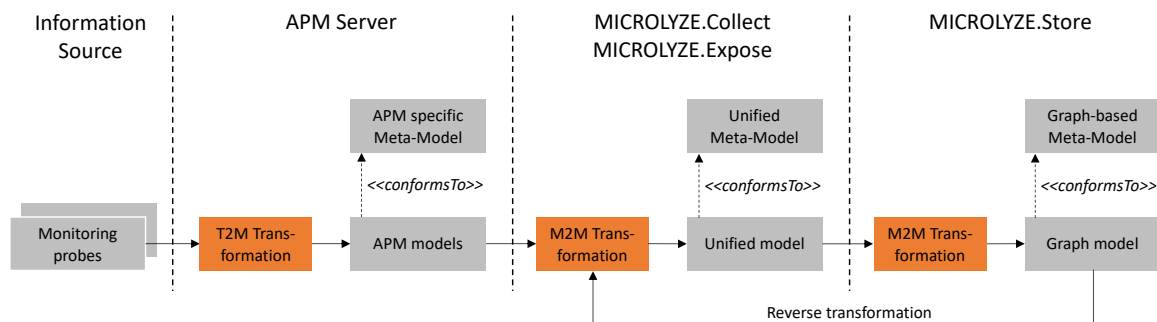


Figure 5.7.: Model transformations during the IT landscape recovery process

In relation to the model transformation theory stated in Section 2.1.2, the collection of runtime data, as well as the performed reconstruction and management of EA models is a continuous process of model transformations. This transformation takes place in four different steps as Figure 5.7 illustrates. First of all, the *Monitoring Logs* that are streamed from the monitoring probes to the central *Monitoring Server* apply a text-to-model transformation. This takes place in the APM system itself. By collecting the models through the exposed API, we transform the (APM) source models to our defined target model in order to achieve a unified model representation. This transformation is performed in the *MICROLYZE.Collect* component. Afterwards, the models are again transformed into a graph-based representation for an efficient persistence. The *MICROLYZE.Store*

is responsible for this model-to-model transformation. Finally, in case the models are requested from the database, we transform the graph-based meta-model back to our target meta-model.

Figure 5.8 summarizes the most important classes and their relationships in each component in high detail, which is needed to better understand the architecture of *MICROLYZE*. The following Sections mainly refer to this Figure.

5.2.1. Monitoring probes

Microservices are very different to monolithic application, as detailed in Section 2.3. Due to agile practices and emphasizing change tolerance and continuous deployment (Dingsøy et al., 2012), microservices are introduced very quickly into the current infrastructure or removed when they are no longer needed. In these scenarios, it is crucial to keep track of the current microservice architecture orchestration and their service dependencies. In addition, microservice architectures are mostly distributed that means they run on different hosts but act as a composition to serve a specific request (Dragoni et al., 2018). For those reasons, monitoring microservices remains challenging. In the following sections, we elaborate which types of monitoring probes need to be installed on the system in order to instrument the IT landscape in its full essence.

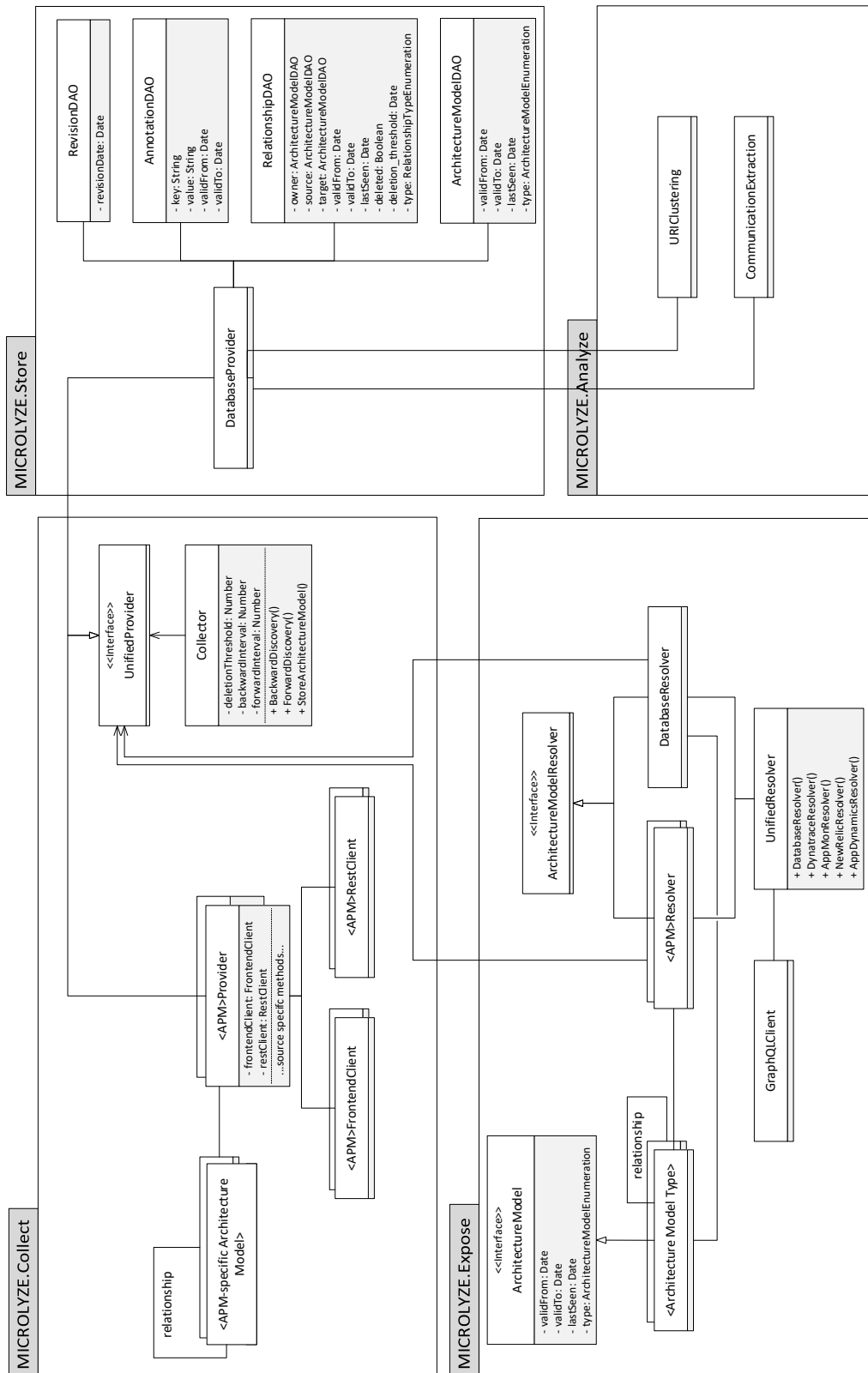


Figure 5.8.: Detailed class diagram of the core components of MICROLIZE

Service Discovery

As described in Section 2.3.2, a fundamental pattern for building microservice architectures is the usage of **Service Discovery** (Newman, 2015) that serves as a repository to find the network location of a specific microservice dynamically. Since, microservices frequently change their status and IP-address due to reasons like updates, autoscaling or failures the **Service Discovery** serves as a gateway that always provides the current network locations of the registered instances of microservices. 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 via accessing the exposed API, we are able to reveal the existence of microservices and the current status of each service instance.

```
1 <applications>
2   <application>
3     <name>EXAMPLE-SERVICE</name>
4     <instance>
5       <instanceId>
6         PC192-168-2-50:example-service:6003
7       </instanceId>
8       <hostName>PC192-168-2-50</hostName>
9       <app>EXAMPLE-SERVICE</app>
10      <ipAddr>192.168.2.50</ipAddr>
11      <status>UP</status>
12      <port enabled="true">6003</port>
13      <dataCenterInfo>
14        <name>MyOwn</name>
15      </dataCenterInfo>
16      <metadata>
17        <appType>functional-service</appType>
18      </metadata>
19      ...
20    </instance>
21  </application>
22  <application>
23    ...
24  </application>
25 </applications>
```

Figure 5.9.: GET list of processes from Dynatrace API

Listing 5.9 provides an example *Monitoring Log* produced by the **Service Discovery**. In this example, we use the application Netflix Eureka³ and call the endpoint "GET

³<https://github.com/Netflix/eureka>, last accessed: 2020-10-28

`/eureka/v2/apps"`

The application name inside the `<name>` tag is delivered by the application during registration on start up. The `<instanceID>` attribute is concatenating with `<hostName>`, `<app>` and `<port>`. It identifies a concrete instance of an application. The `<ipAddr>` defines the IP address in which the application instance can be found by other applications for direct communication. In case the application is deployed in a docker container, the IP address defines the public container address. The `<status>` represents the health status of the application.

In order to install the **Service Discovery** probe on any application, it is required to inject a subroutine into the start-up sequence. An example is provided in Listing 5.1 for Spring Boot⁴ based applications. The agent will be activated by adding the `@EnableEurekaServer` annotation before the main class of the application.

```

1 package registry;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
6
7 @SpringBootApplication
8 @EnableEurekaServer //Start Service Discovery
9 public class Application {
10
11     public static void main(String[] args) {
12         SpringApplication.run(Application.class, args);
13     }
14 }
```

Listing 5.1: GET list of processes from Dynatrace API

Infrastructure Probe

The **Service Discovery** mechanism already provides useful data about the status of microservice instances. However, it is restricted in discovering mainly microservices. Other application types like databases are not reported. In addition, further interrelationship information like on which specific host (operating system, virtualization) and physical infrastructure the microservice is running remain unclear. For that reason, the SUO must be instrumented with a monitoring probe, that runs on host level and collects infrastructure-related information for all processes running on that particular host. A key feature to consider is the ability to support multi-component analytics (Heger et al., 2017; Kleehaus et al., 2016), i.e. the identification of different information systems like database, web service, server, etc.

⁴<https://spring.io/projects/spring-boot>, last accessed: 2020-10-28

APM serves to automate the process of mapping applications to underlying infrastructure components. Data measurements delivered by APM probes reveal 1) infrastructure related information, like host type, container type and address, database type and address, operating system or information about the cloud provider, and 2) interrelationship information like schematic connections between microservices and other infrastructure elements, like on which host the microservice is running, or in which specific container or operating system the microservice is deployed.

Most of the APM vendors expose APIs for extracting the IT landscape topology. The whole topology including the relationships can be retrieved via API calls. For instance, according to Dynatrace a request to the endpoint `GET/entity/infrastructure/processes` fetches the list of all processes in the environment, along with their parameters and relationships. An excerpt of the response JSON file including attributes and attribute types is shown in Listing 5.2.

Listing 5.2: GET list of processes from Dynatrace API

```
1  [{
2  "entityId": String,
3  "displayName": String,
4  "customizedName": String,
5  "discoveredName": String,
6  "firstSeenTimestamp": Date,
7  "lastSeenTimestamp": Date,
8  "fromRelationships": Object,
9  "toRelationships": Object,
10 "azureSiteName": String,
11 "versionedModules": [
12 "monitoringState": {
13   "actualMonitoringState": Boolean,
14   "expectedMonitoringState": Boolean,
15 },
16 "softwareTechnologies": Array<String>,
17 "agentVersions": Array<String>,
18 "managementZones": Array<String>,
19 "azureHostName": String,
20 "listenPorts": Array<String>
21 }]
```

The attribute *discoveredName* describes the recovered name of the entity by the monitoring agent. This is mostly the name of the application component. The attribute *fromRelationships* and *toRelationships* constitutes vertical and horizontal relationship types, i.e. it represents either a deployment or a communication relationship. A definition of the meta-model of Dynatrace was shown in Figure 5.3. The description of all attributes can be found in the API documentation of Dynatrace.⁵

⁵<https://www.dynatrace.com/support/help/dynatrace-api>, last accessed: 2020-10-28

Tracing probes

Both monitoring probes (service discovery, infrastructure probe) uncover the status of running applications and unveil infrastructure-related information including interrelationships. In order to extract the data exchange behavior between applications it is necessary to instrument each microservice with a monitoring probe that tracks request flows through the system. This technique is called **distributed tracing** (Sigelman et al., 2010). 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 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 class and method name, requested port, etc. We leverage distributed tracing in order to detect the communication relationship between applications.

Listing 5.3 provides an example monitoring log produced by the open-source distributed tracing tool Zipkin⁶. We call the API endpoint `GET/traces` for retrieving all recorded traces within a defined timestamp.

Listing 5.3: Get list of traces from Zipkin call

```
1  [[{
2  "traceId": String,
3  "name": String,
4  "parentId": String,
5  "id": String,
6  "kind": "CLIENT|SERVER|PRODUCER|CONSUMER",
7  "timestamp": Date,
8  "duration": Integer,
9  "debug": Boolean,
10 "localEndpoint": {
11   "serviceName": String,
12   "ipv4": String,
13   "ipv6": String,
14   "port": Integer
15 },
16 "remoteEndpoint": {
17   "serviceName": String,
18   "ipv4": String,
19   "ipv6": String,
20   "port": Integer
21 },
22 "annotations": Array<String, String>,
23 }]]
```

⁶<https://zipkin.io>, last accessed: 2020-10-28

As described in Section 2.4.3, a trace is a series of spans which nest to form a latency tree. Spans are in the same trace when they share the same *traceId*, i.e each span represents a processing step of a request. The logical operation this span represents is stored in *name* (e.g. rpc method). The *parentId* field establishes the position of one span in the tree. The root span is defined with a absent *parentId* and usually has the longest duration in the trace. However, nested asynchronous work can materialize as child spans whose duration exceed the root span.

Spans usually represent remote activity such as RPC calls or in-process activity in any position of the trace. In which chronological order the span is positioned in the call tree is defined (among other attributes like span id, parent id and timestamp) by the *kind* attribute. For instance, "client" spans forward the request to "server" spans and wait for a response. If no response is expected due to asynchronous calls, the "sender" span is marked as a producer and the receiver span as the "consumer".

The endpoints define the network context of a node in the trace graph. The local endpoint in combination with name, ip address and port either defines the sender application that calls a remote endpoint, or a local processing step in the trace. The remote endpoint represents the receiver application that consumes the request. Hence, if the span *kind* is of type "CLIENT" the remote endpoint is the server. If the span *kind* is of type "SERVER" the remote endpoint is the client.

Client-side Probe

Last but not least, monitoring of the client applications delivers important information about the user behavior including key metrics like load time and transaction paths. This type of monitoring is also known as real user measurement, real user metrics, end-user experience monitoring, or real user monitoring (RUM). There exists many approaches (Choudhary et al., 2009; Filipe et al., 2016, 2017) how to realize the instrumentation of client-based applications. It is an important component of APM, as it captures and analyzes each transaction executed by users. Those transactions are transferred to the system's backend where they are further analyzed via the tracing concepts. By the support of RUM, monitoring tools are able to disclose which users, browsers or mobile devices communicate with the system. This information are important to collect from a documentation perspective.

In order to instrument client-side applications, a JavaScript tag is required to inject into the HTML head of every application page. It must be ensured that the tag is the first executable script on each page. Otherwise, the probe might collect wrong metrics. As soon as the probe is installed it records the following metrics:

- JavaScript errors: The majority of web applications and websites depend on JavaScript to function. Hence, one of the major goals of a client-side monitoring tool is to identify JavaScript errors, their frequency, and how severe their impact is.
- Network request failures: Web sites and web applications rely on several of external services to function, in addition, they perform continuously requests to the backend

server in which they are hosted on. Client-side monitoring tools record the HTTP requests and responses that a user initiates.

- Framework-specific issues: Frameworks like React or Angular are used more and more frequently, as they make the development of powerful applications much easier. Based on client-side monitoring tools, those frameworks are easier to debug and provide additional reporting for issues encountered with functionality, such as what state the application was in.
- User experience issues: Monitoring collects events that have a negative impact on the user experience, like “rage clicks”, where a user clicks an element multiple times very quickly, or detect if a user is stuck in a navigation loop.
- User behavior: A further use case is the analyzes of the user behavior. Monitoring tools report the click path users are following, or how long users stay on a specific page before they continue with their journey.
- Performance issues: In addition to the aspects above, client-side instrumentation will also track performance metrics like average time to load a page, average server response time, time to display elements, and more.

Commercial monitoring tools

Instrumenting the IT landscape with four different monitoring tools, each covering one of the described monitoring technique would increase the administration overhead unnecessarily. Each agent must be installed and configured and requires its own monitoring server that compete against hardware resources. For that reason, the commercial APM suits integrate all mentioned monitoring techniques into one monitoring agent.

Dynatrace, for example, provides with its technology *OneAgent* a software that is able to instrument most application and infrastructure out-of-the-box. Those monitoring agents primarily consumes metrics that are either delivered by the applications itself, like "Docker stats", "Unix top", or analyzes log events produced by applications or operating systems. A more intrusive runtime analysis can be achieved by integrating those agents into the source code. For this purpose monitoring vendors provide SDKs for most of the common programming languages like C, C++, Java, Python, JavaScript, etc. The complexity of the installation of those agents depends on the runtime environment and used frameworks.

For instance, the Java Virtual Machine (JVM) runtime expose command-line options, also called start-up commands, that can be used to override the default start-up settings for applications running on a JVM. These options define how the application should be run or specify a path to a script that should be executed before the application starts. One command-line option is the `-agentpath` tag which specifies an absolute path from which a monitoring agent should be load and run. This simple approach can also be used for Java frameworks like Spring or Spring Boot and enables the collection of runtime metrics and traces.

The instrumentation of applications that are developed in other runtimes like Node.js or .NET proves to be more complex. The installation of agents must be performed on

code level, as well as the instrumentations of remote calls for collecting trace data. As the example for Node.js illustrated in Listing 5.4 shows, the trace agent must be wrapped around each function that triggers a remote call.

```
1 // Setup monitoring agent
2 const Sdk = require("@dynatrace/oneagent-sdk");
3 const Api = Sdk.createInstance();
4
5 // Issue a traced outgoing remote call
6 async function tracedOutgoingRemoteCall(method, data) {
7   const tracer = Api.traceOutgoingRemoteCall({
8     serviceEndpoint: "ChildProcess",
9     serviceName: "StringManipulator",
10    channelType: Sdk.ChannelType.NAMED_PIPE
11  });
12
13
14  try {
15    //start tracer, wrap tracer around doOutgoingRemoteCall()
16    return await tracer.start(function triggerTaggedRemoteCall() {
17      // getting a tag from tracer needs to be done after start()
18      const dtTag = tracer.getDynatraceStringTag();
19      // now trigger the actual remote call
20      return doOutgoingRemoteCall(method, data, dtTag);
21    });
22  } catch (e) {
23    tracer.error(e);
24    throw e;
25  } finally {
26    tracer.end();
27  }
28 }
```

Listing 5.4: Tracing of outgoing remote calls with Dynatrace

5.2.2. Monitoring Server

The monitoring server consumes the runtime data called metrics delivered by the monitoring agents, processes and aggregates it, and finally persists the runtime behavior in the database. The server exposes APIs for delivering the metrics to other systems, mostly for reporting purposes. As the analysis in Section 5.1 shows, the exposed meta model of the persisted data can be various between the APM vendors. All commercial APM tools provide their own reporting tool for visualizing metrics and creating dashboards. Those frontend tools use further server APIs that are often not reported in the official documentation.

5.2.3. MICROLIZE.Collect: Collecting Architecture Models

In the following, we detail the *MICROLIZE.Collect* component that is responsible to consume monitoring data and to translate the runtime information into the *MICROLIZE* meta-model. Hereby, we assume that it is sufficient to access one monitoring server which provides all required information about models and relationships, i.e. topology information and traces.

The core class of the *MICROLIZE.Collect* component is the *Collector*. It provides the required methods for consuming monitoring data from different sources and extracting the architecture information. Hereby, we leverage the adapter design pattern (Gamma et al., 2015) by wrapping the APM-specific AM with the *UnifiedProvider* class that supports the interface required by the *Collector*.

The *Collector* class exposes two important methods, the *BackwardRecovery()* and the *ForwardRecovery()*. The former methods represents an algorithm for discovering the IT landscape architecture based on historical data, whereas the latter method starts right after the *BackwardRecovery()* and regularly polls the APM server APIs in order to recover newly deployed applications and communications paths. How those algorithms work in detail is explained in Section 5.3.

The *UnifiedProvider* defines all "getter" methods for retrieving the unified architecture meta-model described in Section 5.1.5. The provider classes build the foundation of all the processes that are orchestrated via the API. Each particular *<APM>Provider* class ensures the access to one particular vendor-specific APM server. The *<APM>Provider* class access the APIs exposed from the APM server and transforms the APM-specific meta-model managed by the *<APM-specific Architecture Model>* classes to the our unified meta-model. An extension of *MICROLIZE* in order to support further monitoring tools than the listed ones in Section 5.1 can be done by adding a new provider class that fulfills the *UnifiedProvider* interface.

Furthermore, we separate the API request realization into two client classes. The first class *<APM>RestClient* provides access to rest-based APIs. Those APIs are official and mostly well-documented. They allow to retrieve general metrics but provide only limited access to the vendor-specific meta-model. For instance, during the writing of this thesis 1) the documented API of Dynatrace is not able to get data on Docker images or 2) the AppMon API does not provide details about transaction flows. For that reason, we also developed a *<APM>FrontendClient*, which utilizes APIs that are not documented and only available to the user interface, i.e. the vendor-specific frontend. In general, the *<APM>FrontendClient* imitates the user interface in this regard. All frontend-based APIs are JSON-based and use the HTTP transport protocol.

By developing the *<APM>FrontendClient*, we face the challenge that all APM vendors restrict querying of frontend-specific APIs when there is no valid and active user session available in the cookies. However, generating a user session requires manual authentication via the loaded authentication form. A simulated login process that is performed programmatically got blocked due to security restrictions. In order to solve this issue,

we leverage the Node library Puppeteer⁷ which provides a high-level API to control Chrome/Chromium Browser over the DevTools Protocol⁸. In general, Puppeteer is a product for browser automation. It downloads a version of Chromium and mirrors the browser structure for simulating user behavior. The library is officially supported by Google. With Puppeteer, we are capable to imitate a manual login process and got access to all frontend-specific APIs.

5.2.4. MICROLYZE.Analyze: Analyzing Architecture Models

The monitoring records are transferred from the monitoring collection part to the analysis part *MICROLYZE.Analysis*. The goal of the analysis part is 1) to extract further communication relationship information between application models, 2) to remove request parameters for finding the original RESTful API and 3) to determine AM deletion thresholds. From an implementation perspective, the *MICROLYZE.Analysis* component keeps the recovered AMs in memory until all records are processed. Afterwards, it persists the architecture in the database.

With the support of modern APM tools important correlations between AMs are already recognized and delivered through a simple API call. However, in the context of communication relationships many information like the requested URL endpoints, the communication synchronicity, or involved message broker are not delivered "out of the box" and must be reconstructed by analyzing the available runtime data. For this task, we developed the class *CommunicationExtraction*. A detailed description of the analysis of communication relationships is stated in Section 5.3.2.

The determination of AM communication also involves the extraction of the involved API interfaces. Many APM tools provide information about the methods that are called for request processing and the corresponding URI, in case the HTTP-based communication protocol is used, like for RESTful webservices. However, the URIs reported by the monitoring agents still contain all provided parameters that must be removed first in order to find the originally executed REST API. For this purpose, we integrate a log events clustering algorithm called *LogCluster* (Vaarandi et al., 2015) in the *URIClustering* class. The algorithm recognizes substrings in the requested URI that mainly remain stable and others that change frequently. Stable substrings indicate REST methods and the others define parameters. A detailed description of the applied algorithm can be found in Section 5.3.6.

Monitoring data represent the as-is architecture of a SUO in a specific point in time. Monitoring agents frequently report the health status of the instrumented microservices. No runtime data indicate the microservices is either not running or was removed from the SUO. In both cases, the recovered model is not a part of the IT landscape anymore. However, this cannot be applied for the recover of removed model communications. Model communications represent a behavioral pattern. If the event occurs that triggers this data exchange between models, it can also be observed. That means, no observed

⁷<https://github.com/GoogleChrome/puppeteer>, last accessed: 2020-10-28

⁸<https://chromedevtools.github.io/devtools-protocol>, last accessed: 2020-10-28

communication only indicates, that the required event did not take place. It cannot be concluded that models do not communicate at all. Hence, in order to solve this issue 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, the particular communication path is marked as deleted. We define this process in detail in Section 5.3.7.

5.2.5. MICROLYZE.Store: Storing Architecture Models

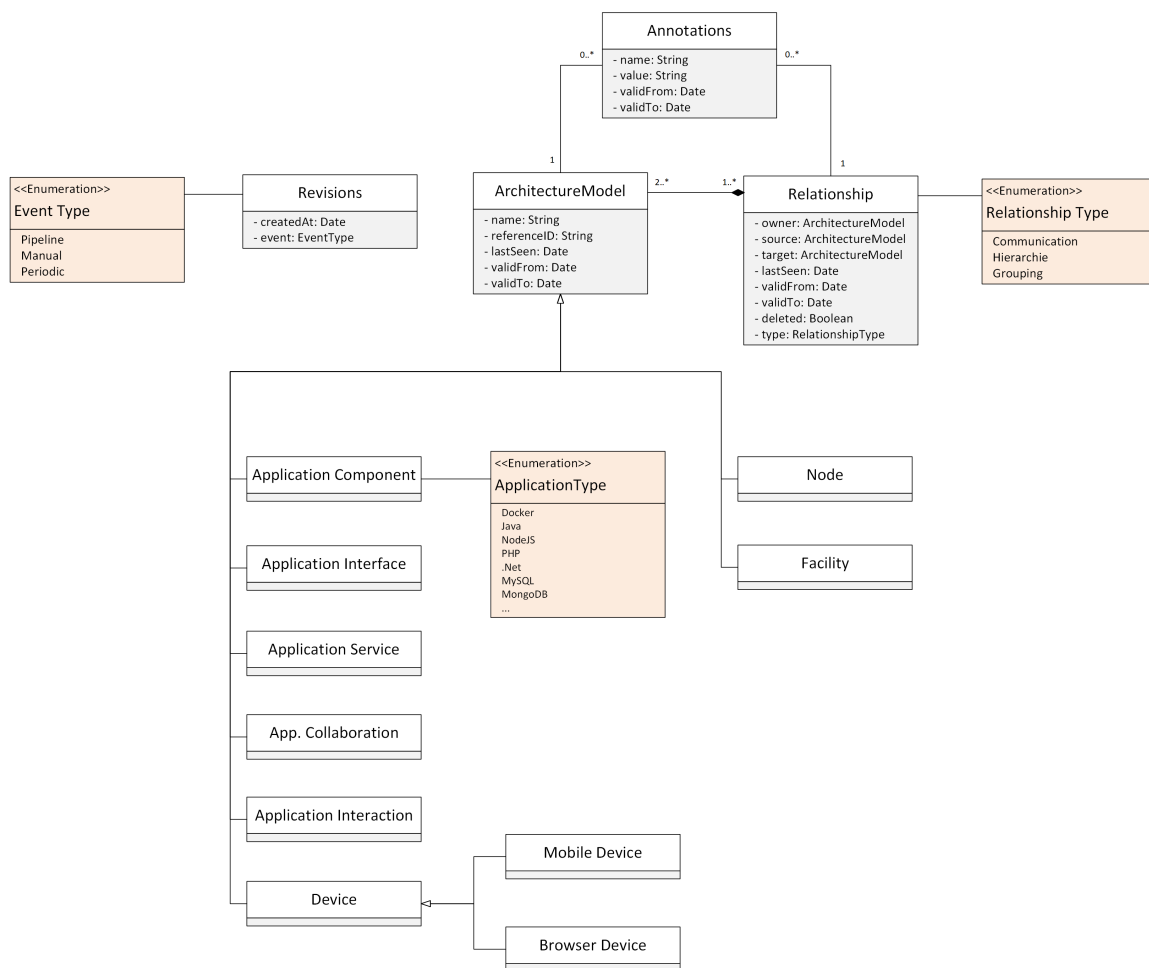


Figure 5.10.: Meta-Model for storing EA models in graph-based representation

The *MICROLYZE.Store* component provides interfaces for storing and retrieving persisted AMs recovered by the *MICROLYZE.Collect* component. The core class of this component is the *DatabaseProvider* that also inherits the *UnifiedProvider* interface in order to be aligned with the defined unified architecture meta-model. However, for storing

the data, we do not follow the elaborated architecture meta-model described in Section 5.1.5. We transform the meta-model into a graph-based representation. Hereby, we follow the idea of an *enterprise topology graph* (ETG) that was introduced by Binz et al. (Binz et al., 2013). The purpose of an ETG is to capture the whole architecture of an IT landscape in a graph-based form. The nodes in the graph define AMs and the edges the logical, functional, and physical relationships between the models.

The final meta-model for storing the IT landscape is depicted in Figure 5.10. It is highlighted that the *Application Models* are grouped into the common EA layers. The information is distributed to the following classes:

- The data access object *ArchitectureModelDAO* defines all AMs that exist in the SUO. The specific entity type like node, application component, application interaction, etc. is defined by the *type* attribute. We define the validity of an AM by the attributes *validFrom* and *validTo*. AMs are valid from the moment they were recovered at the first time and valid until they are removed from the IT landscape or experience a change that has a significant effect from an architecture perspective. In general, the validity attributes describe a revision of an AM. We "only insert" entities into the database and never delete. With this approach, we are capable to move back in time in order to analyze architecture evolution. Furthermore, the *lastSeen* attribute indicates the last time when an AM have been seen by *MICROLYZE*. If the *lastSeen* timestamp is beyond a predefined threshold, it could be marked as removed from the IT landscape. More information about this process is described in Section 5.3.7.
- Within the *RelationshipDAO* class, we realize the relationship between AMs. The concrete relationship type as defined in Section 5.1.5 is stored in the *type* attribute. A relationship is compiled by a *source*, a *target* and an *owner*. The source - target pair define the ends of a directed graph. The owner specifies a group of relationships which belong together via a schematic assignment. An example is the *Application Interaction* AM. The *validFrom*, *validTo* and *lastSeen* attributes apply the same logic described for the *ArchitectureModel* class.
- As the retrieved information about AMs and the relationships between two models differ from APM tool to APM tool, we decided to store every attribute as a key-value pair in the *AnnotationDAO* class. With this structure, we keep flexibility and are able to handle any incoming information. The *AnnotationDAO* class is assigned to the *ArchitectureModelDAO* and *RelationshipDAO* class. That means, we store any further information regarding to AMs and their relationships in the same class.
- Revisions define specific points in time in which the recovered AMs experiences a change. Revisions basically store every modified validity period of an AM, representing an architecture evolution which is worth to keep in database. A change could be the deployment of a new application, the introduction of a new interface, the starting up of a new server, or the like. Those revisions are saved in the *RevisionDAO* class. Based on revisions, we are able to analyze the emerging behavior of an IT landscape, or compare different architecture states.

The reason for a further meta-model transformation are manifold and can be summarized into the following points:

1. We represent the IT landscape in a graph-based form, where the AMs are the nodes and relationships are the edges of this graph. In order to query relationships instantly regardless which AM is used as the entry point and the complexity of the query, the relationships are easier to manage when they are maintained in one single class and stored in a "parent - child" representation.
2. In addition to the aforementioned point, new relationship types can be easily introduced with this approach without having to change the database schema. It is only required to update the relationship type enumeration.
3. Since the information what we receive from APM tools could vary from AM to AM, we decided to store all extracted information as annotations in the *Annotation* class. With this approach, we keep flexibility of the amount of different AM attributes and do not have to change the database schema.
4. The recovered IT landscape can be easily extended with further not yet regarded AMs without having to change the database schema. It is only required to update the AM type enumeration.
5. Since we store the validity of AMs in one single class, the querying of the IT landscape orchestration in a specific point in time is easier to implement.

5.2.6. MICROLYZE.Expose: Exposing Architecture Models

In order to access the recovered AMs including the relationship structure, we create the component *MICROLYZE.Expose*. It represents an interface for querying the data that enables users to retrieve an holistic picture on the IT landscape resources.

For exposing the recovered AMs of our SUO, we use a graph-based query language with the name GraphQL⁹. GraphQL is an API standard that provides a more efficient, powerful and flexible alternative to REST. In comparison to RESTful APIs, GraphQL is a server-side runtime for executing queries by using a type system backed on a predefined data schema, which is finally represented by our elaborated unified meta-model described in Section 5.1.5. Due to its type system, GraphQL is able to check queries for syntactic correctness and validity before execution, thereby allowing the server to make certain guarantees about what response to expect. In addition, RESTful APIs typically suffer from over- and under-fetching. Over-fetching is when a request returns too much data because the addressed endpoint returned fixed data structures including data which is currently not needed. Under-fetching, on the other hand, is when an addressed endpoint returns not enough data, forcing the client to send one or more additional requests to other endpoints to acquire the desired results. GraphQL solves this problem by exposing all of the data from a single endpoint, thereby enabling the client to request precisely the data that is

⁹<https://graphql.org>, last accessed: 2020-10-28

required with a single query. This not only reduces the complexity of creating queries, but also minimizes the amount of data transferred.

The data schema must be written in a graph-based representation, as GraphQL only understands nodes and relationships. An example is depicted in Listing 5.5.

```
1 interface ArchitectureModel {
2     id: ID!
3     name: String!
4     validFrom: Date!
5     validTo: Date!
6     lastSeen: Date!
7     ...
8 }
9
10 type ApplicationComponent implements ArchitectureModel {
11     id: ID!
12     name: String!
13     node: [Node]
14     applicationService: [ApplicationService]
15     applicationCollaboration: [ApplicationCollaboration]
16     calls: [ApplicationComponent]
17     calledBy: [ApplicationComponent]
18     ...
19 }
```

Listing 5.5: Example data schema of GraphQL

Informally, such a GraphQL-schema defines *types* of objects by specifying a set of so-called *fields* for which the objects may have values. A field is synonymous with attributes specified in UML notation. The possible values of the fields can be restricted to a specific type of scalars or objects. An object refers to another type, which represents model relationships. The exclamation mark defines mandatory fields that are not allowed to be null.

It is also possible to define relationships that are not provided in this way in the reflected meta-model. For instance, a communication relationship between *ApplicationComponents* are actually realized through their *ApplicationServices*. That means, the *calls* relationship is included in the *ApplicationService* type by default. However, by implementing an additional method in the *ApplicationComponent* class with the name "calls" and "calledBy", that extracts the communication information from the related *ApplicationService* class, an additional request in the GraphQL query can be bypassed. This enables users to write simple queries for retrieving rather complex structures. In the next Section, we describe the concept of queries in more detail.

Querying Enterprise Topology Graph

With the support of GraphQL, we are capable to query ETGs efficiently and to enable users to select, search, filter, analyze and to modify ETGs. It is worth to mention that GraphQL is not tied to any specific database or storage engine. Instead it is completely backed by the defined graph-based data schema, which makes the query engine powerful. In general, the query executes the respective method in the backend, which retrieves the data from the connected information source and returns the result in JSON format.

In order to allow data to be queried, *Resolvers* for each connected monitoring tool and the database have to be defined and implemented on the root level. The method within the resolver allow to query for an object or collections of objects and work in a similar manner to *RemoteProcedureCalls*. The resolvers inherit from the *ArchitectureModelResolver* interface in order to be aligned with the defined unified meta-model. Each resolver references to a GraphQL-based representation (*<Architecture Model Type>*) of the *<APM-specific Architecture Model>*. With the support of resolvers, we can decide which specific data source should be addressed to resolve the query. The *UnifiedResolver* class serves as entry point of each user request. It forwards the request to the particular resolver based on a passed argument.

The methods in the resolver classes can be parameterized with multiple arguments that serve the query as filters to reduce the amount of objects. Every traversal of the graph is done on behalf of the implemented object types and selections on their properties. We developed for every AM a GraphQL resolver method that retrieves the data either from our database or from the connected monitoring system. With the latter approach, we can also obtain further runtime information by accessing the exposed APIs. We developed four different resolver methods for each AM. Two methods return one single AM defined by an ID and differentiate on basis of an passed date argument. The other two resolver methods are not restricted by an ID and return more than one AM. For instance, the following method returns the *Application Component* with the defined ID that is valid until the current time.

```

1 @Field(returns => GraphQLApplicationComponent)
2 public async applicationComponent(@Arg("id") id: string):
3 Promise<GraphQLApplicationComponent> {
4     const entity = await this._databaseProvider.
5         getArchitectureModel(id);
6     return new GraphQLApplicationComponent(entity, Date.now());
7 }

```

Listing 5.6: GraphQL resolver for retrieving a specific application component

The method below returns all *Application Components* that were valid until the provided date, or is still valid in case the *validTo* attribute is empty:

```

1 @Field(returns => [GraphQLApplicationComponent])
2 public async applicationComponentsAt(@Arg("timestamp") t: number):
3 Promise<GraphQLApplicationComponent []> {

```

```
4     const entity = await this._databaseProvider.  
5         getArchitectureModelsByTypeAtTimestamp("application-component", t);  
6     return applicationComponents.map(  
7         item => new GraphQLApplicationComponent(item, Date.now()  
8     );  
9 }
```

Listing 5.7: GraphQL resolver for retrieving application components for a specific validity period

With the support of GraphQL, we have any AM as starting point for a query and can traverse through the ETG. The following Figures 5.11 and 5.12 provide an example for an ETG traversal and the corresponding result. The query list all valid *Application Components* that are required to process a particular request that is represented by an *Application Interaction*.

```
1 query getCommunications{  
2   database {  
3     applicationInteraction(id: "/svds/vehicle/internal/v2/{VIN}") {  
4       id  
5       applicationComponents {  
6         name  
7       }  
8     }  
9   }  
10 }
```

Figure 5.11.: GraphQL query for retrieving all *Application Components* that are required to process a specific request

```
1 {"data": {  
2   "database": {  
3     "applicationInteraction": {  
4       "id": "/svds/vehicle/internal/v2/{VIN}",  
5       "applicationComponents": [  
6         {"name": "CASA"},  
7         {"name": "AM_USAGE"},  
8         {"name": "AM_PROCESS"},  
9         ...  
10      ]  
11    }  
12  }  
13 }
```

Figure 5.12.: GraphQL result for retrieving *Application Components* within a specific *Application Interaction*

Each GraphQL query starts with the term "query". On the root level of a query,

we pass the information which particular resolver, i.e. data source should handle the request. In this case we request the database for resolving the request. Similar to method calls, resolver accept arguments if implemented. In the above example, the argument `id="/svds/vehicle/internal/v2/VIN"` was used to select a specific *Application Interaction*. Further attributes like `name` define entity properties to display in the result list. GraphQL queries statements could be theoretically expressed with an arbitrary amount of depth. In practice, however, too complex queries rise timeouts or have performance impacts and should be verified by the server before execution. The result of the query is shown in 5.12 and follows the pattern how the initial query was structured, thereby allowing a consumer to expect predictable result structuring.

Querying Runtime Data

To this end the ETG is static and can only visualize EA models that are known by GraphQL. In the next step, we enhance the managed AMs with runtime information in order to uncover behavioral aspects and to transform the static models to models@run.time (Bencomo et al., 2019). As we use monitoring tools to recover the EA models anyway, we are also able to retrieve runtime metrics from the exposed APIs. For this purpose, we add additional fields to the GraphQL schema that point to the related methods in the backend.

In this context, metrics are always assigned to a specific AM and must be aggregated to a higher-order model if necessary. For instance, most resource utilization metrics including CPU, memory, harddrive, etc. can be retrieved on *Node*, and *ApplicationComponent* level. Whereas response time, failed operations, number of requests, idle time, etc. are only available on *ApplicationComponent* level. In addition, several metrics exist for specific application types, like Java, Docker, MySQL, MongoDB, etc. Those metrics describe the number of reads and writes operations, request time, cpu and memory utilization, running containers, etc. A full list of all available metrics can be found in the API documentation of the particular APM tool.

Each APM vendor defines its metrics differently, which makes the creation of a uniform query complicated. An additional transformation would be needed to translate each vendor-specific metric definition to one standard. This can only be achieved with a mapping table. However, taking only Dynatrace¹⁰ into account this table would have more than 1.200 metrics that need to be mapped with the definitions of the other APM vendors. The creation of this table would have gone beyond the scope of this thesis.

For the above reason, we analyzed the metric API specifications of the APM tools and figured out that every vendor offers a general API that provides requested metrics based on defined parameters. Those parameters are mostly the same for all APM providers. Required fields are 1) metric ID and 2) entity ID, which represents the specific AM. Further important optional fields are aggregation type like sum, avg, count, max, min, median and percentile, as well as the observed timeframe.

Hence, in order to retrieve metrics, we add a method *metric* to each AM which can be queried by the field *metric*. We extend the resolver and provider classes accordingly. This

¹⁰<https://www.dynatrace.com/support/help/extend-dynatrace/dynatrace-api/environment-api/metric-v1/available-metrics/managed>, last accessed: 2020-10-28

method requires the aforementioned parameters. The metric ID must be looked up for every APM tool individually. Based on this concept, we are able to transform the static model into a model@run.time. Listing 5.13 illustrates an example for retrieving metrics about a defined *Application Component*:

```
1 query cpu_util {  
2   database {  
3     applicationComponent(id: "MUPI_GW_NBT") {  
4       metric(type: "cpu.system", aggrType: "avg", relTime: "1585825695000") {  
5         data  
6       }  
8     }  
9   }  
10 }
```

Figure 5.13.: GraphQL query for retrieving the average cpu utilization within a specific timeframe for a defined *Application Component*

Querying the Architecture Evolution

The retrieval of the ETG is based on a revision concept backed by the validity attributes. Figure 5.14 shows an example of a chronological sequence of the validity of a revision and how the AM can be retrieved for a selected time. Time t_1 leads to the selection of revision 1.12 and 2.1. Application component C does not yet exist at this time. T_2 contains the revisions 1.14, 2.1 and 3.1 for the applications A, B and C. That means, the AM has been significantly changed after t_1 , as component A was modified two times and component C was introduced into the architecture.

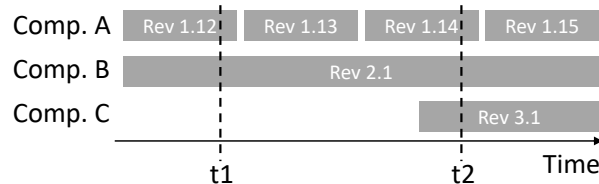


Figure 5.14.: Revision concept

If we query the AMs for time t_1 , all revisions that are valid at this time are selected and the relationships of the selected revisions are transferred to their source and target components. Since revisions have a validity period over time, their relations also have a validity period. Figure 5.15 shows an example of this for two AMs with several revisions.

Modifying Enterprise Topology Graph

It is also possible to use GraphQL for executing requests to create and update AMs. We use this interface for starting, inter alia, the architecture recovery process. An example for such a request is shown in Figure 5.16. The required keyword is *mutation* instead of

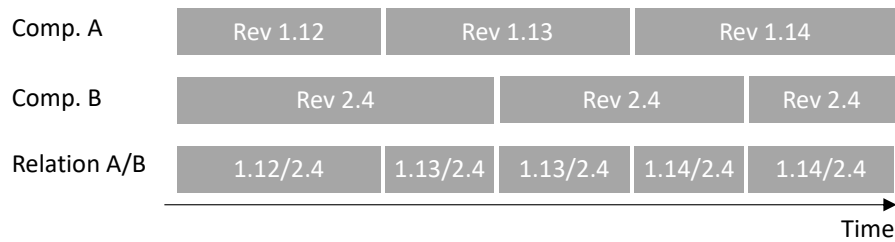


Figure 5.15.: Revision relation concept

query. Similar to a regular queries arguments can be passed as well. In this case, the *forwardRecovery* algorithm runs for the next seven days and checks every 12 hours for unknown AMs. We will describe the architecture recovery algorithms in Section 5.3.1 and 5.3.3 in more detail. After the process has been successfully completed, it returns all newly recovered AMs with their corresponding ID and name.

```

1 mutation {
2   forwardsRecovery(
3     currentTimestamp: Date.Now(),
4     futureTimestamp: Date.Now()+7,
5     timeInterval: 12) {
6     architectureModels{id name}
7   }
8 }

```

Figure 5.16.: GraphQL mutation for recovering and storing new AMs

5.3. Process Design

In the following sections, we describe processes related to EA models recovery in more detail. We published the corresponding algorithms in Kleehaus et al. (Kleehaus et al., 2020). The overall process we elaborated for reconstructing AMs including the corresponding relationships is illustrated in Figure 5.17. It consists of four collapsed subprocesses that are detailed in the following paragraphs.

In general, during model recovery, we set the focus mainly on disclosing the dependency structure between AMs. Runtime data primarily determines a snapshot of the IT landscape architecture for the regarded period of time. As the monitoring server receives frequently the health status of all instrumented AMs, the existence of those models is proven undoubtedly. However, communication dependencies are reactive in nature, i.e. they can only be recognized as soon as they appear in the analyzed runtime data. Hence, we cannot ensure that the communication structure is uncovered completely during runtime data analysis, as it would require all possible communications between applications take place in the

considered period of time. That is rather unlikely to happen. For that reason, our model recovery process, reconstructs an incomplete IT landscape by analyzing past runtime data (Section 5.3.1). This incomplete IT landscape is continuously refined by processing newly incoming runtime data (Section 5.3.3). Hence, the recovered IT landscape will become complete at some point in the future. However, as we cannot guarantee this, we consider our reconstructed IT landscape as eventually consistent (Brewer, 2012). The recognized changes that finally lead to an architecture revision are defined in Section 5.3.4. We store those changes as revisions with a corresponding timestamp in the database. Hereby, we never delete old architectural states, but set them as invalid and insert the currently valid state. Based on this *only insert* approach, we are capable to move through time and disclose the architecture evolution (see Section 5.3.5).

Furthermore, communications between applications are performed at the interface level. Especially in microservice architectures those interfaces are realized with HTTP-based RESTful APIs, which are defined with an base URI and an HTTP method. The URI contains member resources that determine parameters. In order to reconstruct the origin REST API, those member resources must be recognized automatically and replaced with an appropriate placeholder. We describe the corresponding process in Section 5.3.6. Finally, we detail in Section 5.3.7 possible approaches for defining communication deletion threshold. We use those thresholds to determine when communication relationships can be marked as removed in the IT landscape.

During the creation of the aforementioned processes, we have to consider the following challenges:

- Most tracing techniques are based on sampling (Duffield, 2004) in order to reduce network overhead, i.e. only a percentage of, for example, 10% of the requests is traced and forwarded to the monitoring server. For the purpose of creating performance KPIs, an excerpt of the traces is sufficient. However, sampling could also conceal communication dependencies between services, which is a concern for model recovery purposes. In the worst case, specific communication paths are not seen at all.
- Due to resource limitations like available CPU time, most monitoring tools only provide a small timeframe of runtime data, e.g. last 6 hours or 24 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.
- Even though the aforementioned challenge could be solved, 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.
- The runtime history does also contain obsolete AMs and communication paths that had been already removed a long time ago. This legacy data must be filtered in order to uncover the real architecture. This can be easily performed with the general existence of AMs extracted from repository data, as the installed monitoring agents frequently provide application health data. Hence, if no health data is reported

anymore, the AM was certainly removed from the IT landscape. 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 yet.

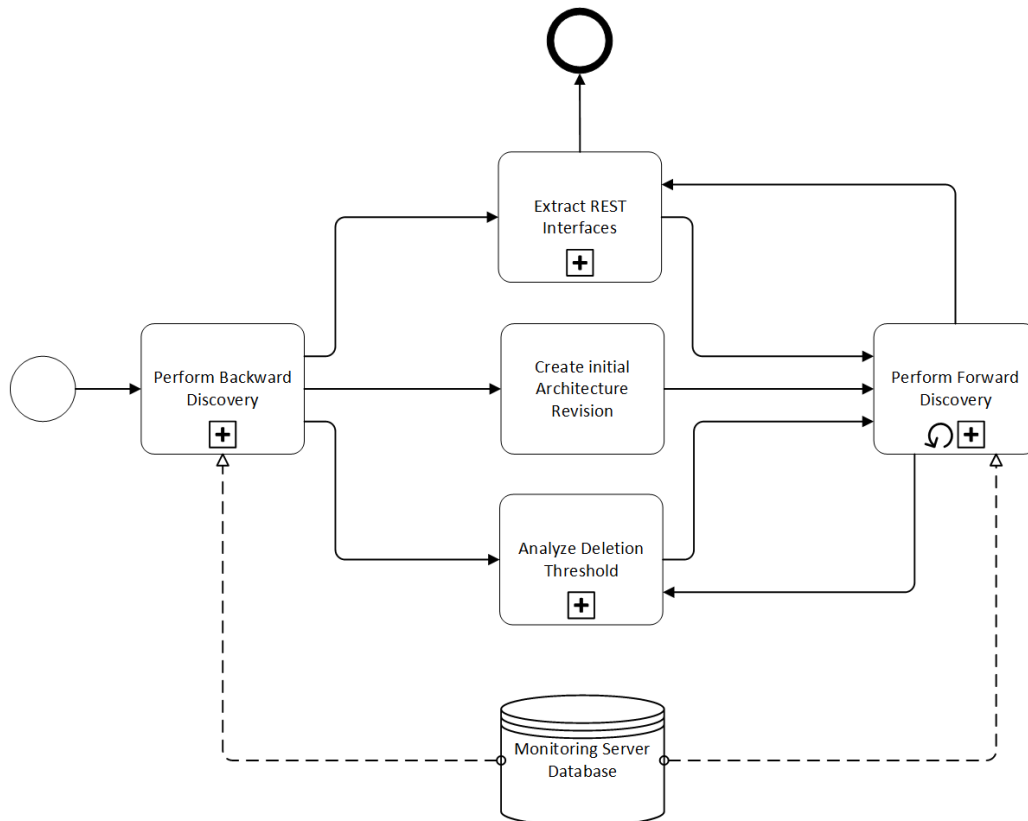


Figure 5.17.: Overall model recovery process

5.3.1. Reconstruction of Architecture Model Dependencies

For recovering the IT landscape based on available historical runtime data, we developed the *backwardRecovery* process. The iterative process is illustrated in Figure 5.18. In each iteration, we extract the topology models from the monitoring server for a predefined timeframe. The size of the timeframe is dependent on the configuration of the monitoring server and the maximum allowed query response time. For each recovered AM a revision record is created that expresses the validity of this model. The *validFrom* attribute is set to the timestamp when the *backwardRecovery* algorithm was triggered. The *validTo* attribute remains empty. The *lastSeen* attribute of the AM is also set to the current timestamp.

The topology records already contain interrelationships between AMs. However, in the available meta-model of the particular monitoring tool, the intrarelations, i.e.

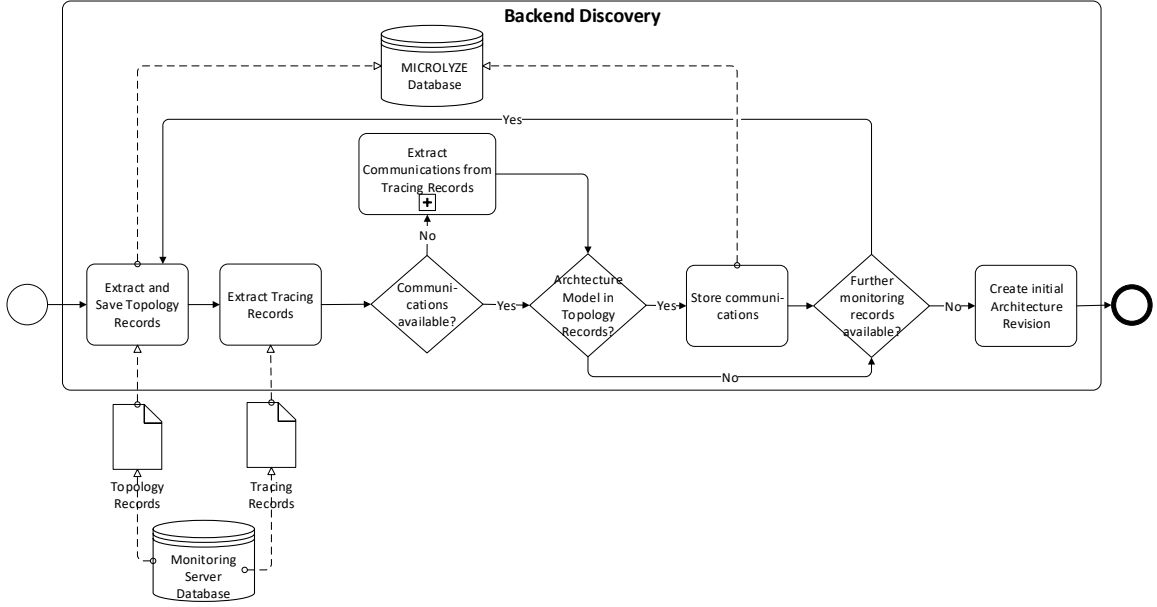


Figure 5.18.: Backward recovery process

communications path between applications might be incomplete. This is often the case with asynchronous communications in which a message broker like *Apache Kafka* is installed between. Hence, as a next step, we extract tracing records from the monitoring server and analyze the AM communications. In order to extract asynchronous communications, we developed an algorithm described in Section 5.3.2. Each identified relationship and communication dependency between AMs is assigned to the particular revision entity.

The identified communications are only stored if the *caller* and *callee* AM is available in the topology records. Missing AMs have been removed in previous architecture versions, but might be still available in the monitoring server. Hence, an additional verification is required.

In general, we assume the architecture $A(E, C)$ is a directed graph with AMs E and dependency paths C , whereas $C \subseteq E \times E$ on a finite set E and $E \times E \in \{\text{contains}, \text{runsOn}, \text{calls}\}$. The algorithm reconstructs the architecture $A'(E', C')$ 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 applications that were already removed in future versions. Both scenarios must be handled accordingly. Hence, we define $A'(E', C')$ as

$$\begin{aligned}
 E' &= E : \iff \forall e(e \in E' \leftrightarrow e \in E) \\
 C' &:= \{c \mid (c \in E') \wedge (c \in E' \cap E)\}
 \end{aligned}$$

The corresponding algorithm in Listing 1 runs recursively and retrieves in every iteration historical tracing data (tD) with a timeframe of $T = t_0 - t_{-1}$. The timeframe T represents the maximum time period that is accepted by the monitoring tool to go back in history. First, we instantiate the architecture $A'(E', C')$ based on the Topology Record $rD(E)$

(line 3 and 4) APM tools provide in order to identify running IT artifacts. We use the function `TOPOLOGYRECORD()` 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 `TRACINGRECORD(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 topology 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.

Algorithm 1 Backward Recovery Algorithm

Require: $T > 0$

```

1: function BACKWARDRECOVERY( $A, t_0, T$ )
2:   if  $A = \emptyset$  then
3:      $rD(E) \leftarrow \text{REPOSITORYDATA}$ 
4:      $A' \leftarrow A(rD(E), C)$ 
5:      $t_1 \leftarrow t_0$ 
6:      $t_0 \leftarrow t_1 - T$ 
7:      $tD(E, C) \leftarrow \text{TRACINGRECORD}(t_0, t_1)$ 
8:     if  $tD \neq \emptyset$  then
9:       for all  $e \in tD(E)$  do
10:        if  $e \in rD(E)$  then
11:           $A'' \leftarrow A'(E, C \cup tD(C_e))$ 
12:        BACKWARDRECOVERY( $A'', t_0, T$ )
13:     else
14:       return  $A'$ 

```

5.3.2. Reconstruction of Communication Dependencies

The analyzed APM tools display communication relationships between monitoring agents mostly in two different granularity levels. In Dynatrace/AppMon for instance, the *Transaction Flow* describes which agents contribute to process a certain transaction. Hence, it displays the trace of the transaction through the backend. However, the information about communication between agents is rather limited and does not include 1) which interfaces were targeted and 2) whether communication took place asynchronously by means of a messaging queue (cf. Section 2.3.2). This message queue is abstracted from the *Transaction Flow*, i.e. if agent A communicated with agent C over message queue B, the *Transaction Flow* will only store "A communicated with C" without mentioning the message queue. Consequently, for the user all communication appears as synchronous communication.

A more detailed analyzes of traces is available in *PurePaths*, which mainly represents all correlated spans of a trace. Unlike *Transaction Flows*, *PurePaths* do capture targeted

interfaces and reveal asynchronous communication over message queues. Most APM tools follow the design of Sigelman et al. (Sigelman et al., 2010) to store traces. That means, in terms of data structures, they are realized as trees with arbitrary width and depth whereby each node represents an agent and each edge represents a call between agents (cf. Section 2.4.3). In order to identify which interfaces microservices expose and how often they are called, it is required to periodically analyze all incoming traces.

We process each *PurePath* record for a specific timeframe in order to extract the information about microservice communications. Since traces are represented as data trees of arbitrary depth, the algorithm recursively calls itself once for every edge in the tree in order to traverse the entire trace. In every execution it checks whether the source (parent node) and target (child node) belong to different microservices to exclude internal communication of microservices. From an architecture point of view, this information is not important. We extract the targeted API and the communication synchronization (synchronous or asynchronous calls). In the last step, we determine trace equivalence classes, i.e. all traces that are equal in terms of the control flow structure. While the request flow within a single trace constitute a view on the sequence of interactions among microservices, it is required to analyze this runtime dependencies in an aggregated form in order to disclose microservice interactions from an architectural perspective. Simply spoken, we prevent to store communications that are already available in the database and only add the target interface if it was not seen yet. Furthermore, the edges are augmented with the total number of call actions among the respective microservices observed in the considered set of traces. Hence, we update the call count in the annotations¹¹. Figures 5.19 illustrate the aggregation of microservice communications.

5.3.3. Validation of Architecture Changes

We regard architecture recovery as a never-ending process. It is a continuous monitoring of the service interaction within an IT infrastructure. Especially microservice architectures evolve over time (Dragoni et al., 2016). That means, new services are added or removed, and newly implemented interfaces lead to a change in the information exchange and dependency structure. Hence, it is important to keep track of architectural changes, especially when new releases cause failures or performance anomalies.

Based on this consideration, we elaborate the *forwardRecovery* algorithm that either gets triggered on a fixed time interval or on special events (see Section 5.3.4). The process is depicted in Figure 5.20 in detail. It continuously extracts the AMs and corresponding communication dependencies from incoming topology and tracing records. Those records are validated against the database. If the extracted models or dependencies are unknown, or the change has a significant impact on the overall maintained architecture then the database is updated accordingly. In case, the *lastSeen* attribute is behind the particular threshold, the entity is marked as deleted.

¹¹We must emphasize that the number of calls between microservices observed in the traces do not represent the real number of interactions. As stated in Section 2.4.3, most APM tools apply sampling to reduce network overhead. That means, the monitoring server only receives a subset of runtime data. In order to derive the real number of interactions the amount of observed calls must be divided by the sampling rate.

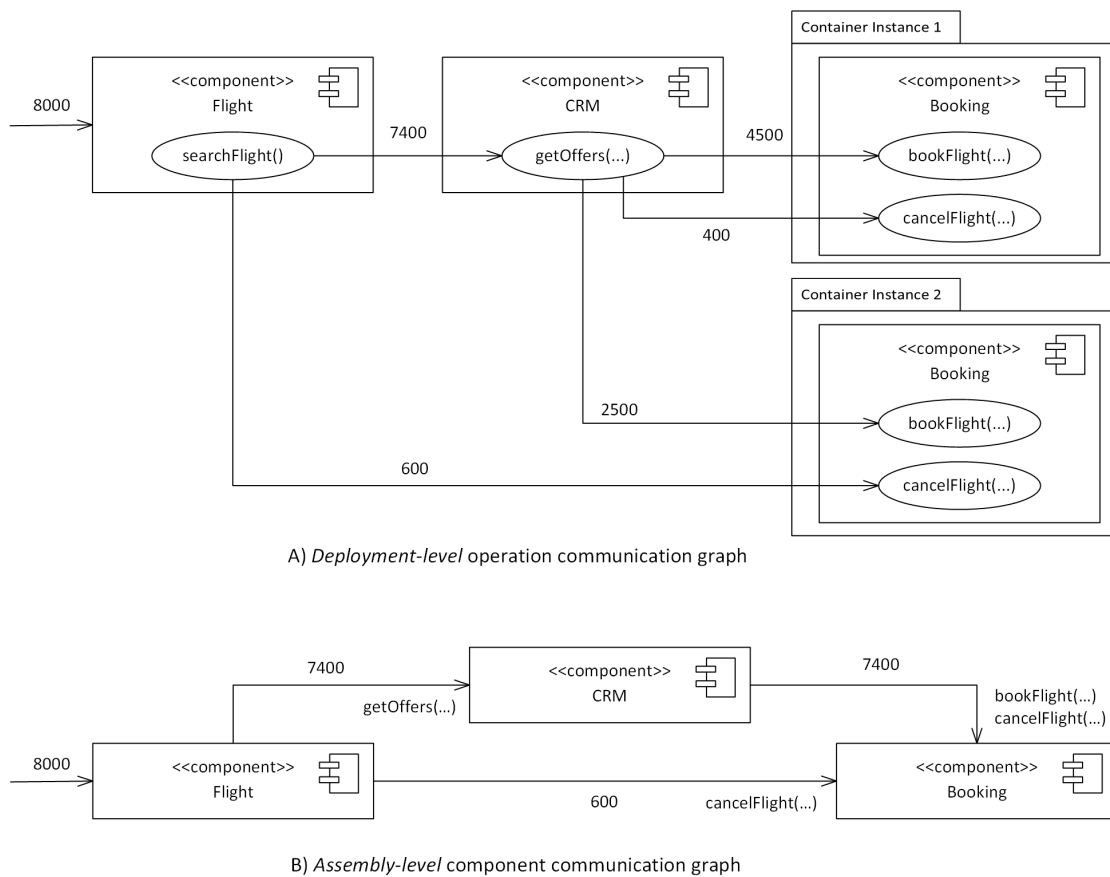


Figure 5.19.: Differences between deployment-level communications and assembly-level communications between microservices. Only the assembly-level are stored in the database.

The corresponding *forwardRecovery* algorithm is described in Listing 2 and consumes 1) a timeframe T for retrieving the monitoring data, 2) the deletion threshold $\tau > 0$ which defines how old a communication path is allowed to be, before it gets removed (see Section 5.3.7) and 3) the obsolete stored architecture that was initially returned by the *backwardRecovery* function. First, the function fetches both the current content of the *TopologyRecords* (line 3) and the *TracingRecords* (line 4) for a specific period of time. Based on the retrieved data the architecture A'' 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. In case an AM has changed from an architecture point of view (line 8), we create a new revision for this entity (line 9) and set the value of the *validTo* attribute of the previous revision to the current timestamp in order to express a validity expiration. Subsequently, all relationships going in and going out from this model are duplicated. If no changes were detected, we update only the

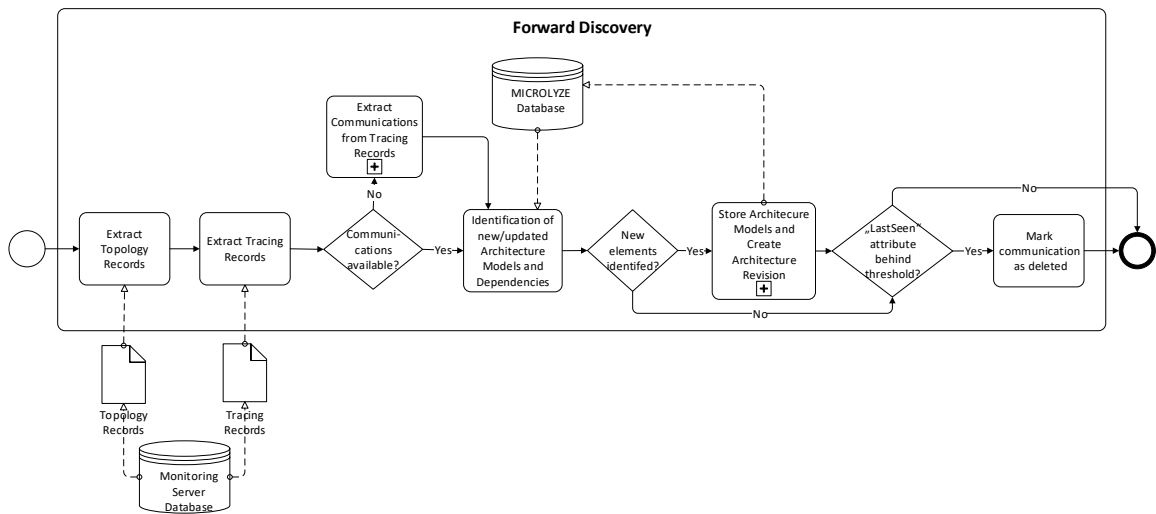


Figure 5.20.: Forward Recovery process

lastSeen attribute for the AM (line 11). The same applies for dependencies between AMs (line 14). As removed communications are still not recognized without any manual input, 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 15), the particular communication path is marked as deleted (line 16) in the annotation entity. Hereby, we leverage the *lastSeen* 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. 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.

5.3.4. Change Events

A major concern of model recovery is the definition of events that trigger an adoption of the maintained EA models. Hereby, we define every event can be considered as an architecture-related change that need to be handled accordingly if it has a significant impact on the maintained EA models including their inter- and intrarelations. In (Kleeaus et al., 2019b), we described the results of a survey conducted in 2018. In this survey, we analyzed the response of 58 survey participants to unveil major challenges of documenting microservice-based IT landscapes. One important question we asked was "Which events lead to an update of the EA models in order to keep the models in-sync with the reality?" Most participants agree (37,9%, n=22) that the development of new applications or the deployment of a major release of an existing application lead to an update of the documentation. 27,6% (n=16) of the participants state that business driven as well as IT driven projects that have an impact on the IT landscape initiate an update of the corresponding models. 27,6% (n=16) of the participants state that their data collection

Algorithm 2 Forward Recovery Algorithm

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

```

1: function FORWARDRECOVERY( $A, \tau, T$ )
2:    $t_1 \leftarrow t_0 - T$ 
3:    $rD(E) \leftarrow \text{TOPOLOGYRECORDS}$ 
4:    $tD(E, C) \leftarrow \text{TRACINGRECORDS}(t_1, t_0)$ 
5:    $A' \leftarrow A(E \cap rD(E), C)$ 
6:    $A'' \leftarrow A'(E', C \cup tD(C))$ 
7:   for all  $e \in A''(E')$  do
8:     if  $\text{isNewOrUpdated}(e)$  then
9:        $\text{CREATEREVISION}(e)$ 
10:    else
11:       $e(\text{lastSeen}) \leftarrow t_0$ 
12:    for all  $c \in A''(C')$  do
13:      if  $c \in tD(C)$  then
14:         $c(\text{lastSeen}) \leftarrow t_0$ 
15:        if  $c(\text{lastSeen}) + \tau \leq t_0$  then
16:           $c(\text{annotations.deleted}) \leftarrow \text{true}$ 
17:        else
18:           $c(\text{annotations.deleted}) \leftarrow \text{false}$ 
19:    return  $A''$ 

```

process is initiated by a change request or incident system that allows triggering tasks for other stakeholders, as it has been also recommended by literature from practice (Hanschke, 2010). Many survey participants (20,1%, n=12) rely on periodic checks with key stakeholders that provide data on specific parts of the architecture. Hence, the update of the models happens in an ad-hoc manner. 10,3% (n=6) of the participants describe automatic triggers that initiate a change in an automatic manner. Those changes are mostly detected by CMDBs and afterwards forwarded to the particular architects which populate the data into the EA tool, or even update the model automatically.

Based on our findings and the findings elaborated by other researcher like Farwick et al. (Farwick et al., 2012b), and Winter et al. (K. Winter et al., 2010), we define the following points as an architecture-related change:

- Deployment-based: Every deployment of new applications or application updates
- Deletion-based: Deletion of AMs regardless from which architecture layer
- Structure-based: Modifications of the interrelationships between AMs, like movement of applications from on-premise to cloud environments.
- Host-based: Modifications in the host environment including operating system, virtualization and containerization
- Hardware-based: Modifications, upgrades or new physical resources

As described in Section 3.2.4, Fischer et al. (Fischer et al., 2007) discuss two different strategies to initiate a new cycle for EA model maintenance:

periodic changes are initiated automatically based on a maintenance schedule. In our scenario, this schedule would trigger the *forwardRecovery* algorithm periodically to update the maintained AMs. However, this strategy comes with two drawbacks. 1) A too long schedule would always follow the change that leads to timeframes in which the IT landscape is obsolete and not up-to-date. 2) A too short schedule would be rather resource intensive as the already heavily loaded monitoring server has to handle additional frequently triggered queries.

non-periodic can be triggered manually by the data owners. A non-periodic cycle is initiated if AMs changed significantly, e.g. due to project work. In our scenario, a non-periodic schedule could be integrated into a *Continuous Delivery Pipeline* which triggers a non-periodic maintenance cycle every time a deployment is performed.

The *forwardRecovery* algorithm is responsible to apply changes to the EA models and can be triggered based on the aforementioned strategies. 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.

5.3.5. Revision Concept

The implementation of the revision concept is based on the assumption that every change in the IT landscape has an impact on the structure and performance of the maintained architecture. If changes on AMs or relationships are identified a revision entity is created in the

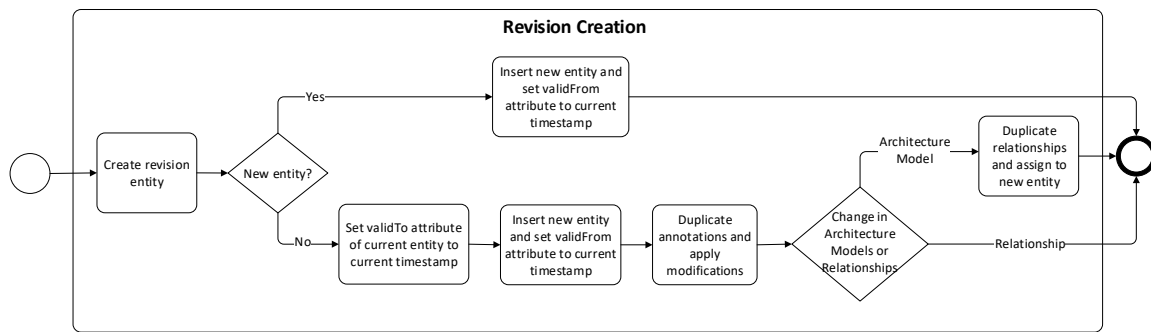


Figure 5.21.: Revision Creation Process

Revisions table indicating a general change in the IT landscape. If the *MICROLYZE.Collect* component recovers a new AM or relationship, a new record is inserted into the database. If the modified entity is already known, then the previous valid record becomes invalid by setting the *validTo* attribute to the current timestamp. Afterwards, the modified entity is inserted in the *ArchitectureModel* or *Relationships* table respectively and the corresponding *validFrom* attribute indicates the timestamp when the change was recognized. In addition, the assigned annotations are duplicated and modified according to the recognized change. As modifications on *ArchitectureModels* has a direct affect on their relationships, all corresponding entities in the *Relationships* table of the invalidated AM must be duplicated and assigned to the new valid entity. This step can be skipped for recognized changes in the *Relationship* entities.

In general, revisions could also be used as a versioning concept. With this approach architects are capable to compare different architecture revisions in order to uncover unforeseen changes. In addition, it is possible to evaluate how the architecture emerged over time and what impact specific changes have on the overall IT landscape performance.

5.3.6. Recovering REST Calls using Runtime Data

Monitoring tools collect log events that are created during the communications between applications. Especially microservices communicate primarily via RESTful interfaces. The executed URL in this HTTP-based communication protocol consists of substrings that represent resources. The URL in addition with the HTTP method like POST, PUT, GET, DELETE, etc. determines the exposed interface of an application, i.e. the RESTful-Webservice. Those interfaces are mainly documented via tool support like with swagger.io¹² or apiary.io¹³.

The log events reported by the monitoring agents contain the complete executed URL with all provided resources. However, resources do not only represent the location of a specific method call but also specific IDs of the addressed resource. In a figurative sense, those IDs determine parameters of an exposed interface. Hence, in order to determine the original used interface that was executed by the application, those parameters must be

¹²<https://swagger.io>, last accessed: 2020-10-28

¹³<https://apiary.io>, last accessed: 2020-10-28

Table 5.2.: Reported log events of RESTful API calls and their corresponding interface description. The parameters are highlighted in bold.

Log Event	127.0.0.1 user-identifier frank [10/Oct/2018:13:55:36 -0700] "GET /api/v1/users/123456/vehicles/789/status HTTP/1.0" 200 2326
Interface	GET: /api/v1/users/{ UID }/vehicles/{ VID }/status <i>Retrieve the status of a specific vehicle</i>
Log Event	127.0.0.1 user-identifier frank [10/Oct/2018:14:01:17 -0100] "PUT /api/v1/users/123456/vehicles/790 HTTP/1.0" 200 2189
Interface	PUT: /api/v1/users/{ UID }/vehicles/{ VID } <i>Create a new vehicle for a specific user</i>
Log Event	127.0.0.1 user-identifier paul [12/Oct/2018:15:02:24 -0600] "DELETE /api/v1/users/123457 HTTP/1.0" 200 562
Interface	DELETE: /api/v1/users/{ UID } <i>Delete a specific user</i>

identified automatically and replaced with a placeholder. An example is provided in Table 5.2. This is a challenging task, as the identification of a resource location and parameter is not obvious at first glance from a programmatic perspective. For that reason, several data mining methods have been proposed in the past to analyze textual log data without well-defined structure. The methods primarily focus on the detection of line patterns from textual event logs. Suggested algorithms have been mostly based on data clustering approaches (A. Makanju et al., 2013; A. A. Makanju et al., 2009; Vaarandi, 2003). The algorithms assume that each event is described by a single line in the event log, and each line pattern represents a group of similar events. Further algorithms (Reidemeister et al., 2011; Reidemeister et al., 2009) uses event log mining techniques for diagnosing recurrent patterns in textual data. However, the applied methods require labeled event logs which is not always available.

A further development of clustering log events and discovering frequently occurring line patterns was conducted by (Vaarandi et al., 2015). The authors introduce the algorithm *LogCluster* that is mainly designed for addressing the shortcomings of existing event log clustering algorithms.

In general, *LogCluster* regards the log clustering problem as a pattern mining problem. As an input parameter *LogCluster* requires a support threshold s ($1 \leq s \leq n$) and divides event log lines into clusters C_1, \dots, C_m , so that there are at least s lines in each cluster C_j (i.e., $|C_j| \geq s$). Each cluster C_j is uniquely identified by its line pattern p_j which matches all lines in the cluster. The *support* of pattern p_j and cluster C_j is defined as the number of lines in C_j : $supp(p_j) = supp(C_j) = |C_j|$. Finally, each pattern consists of words and wildcards, e.g., */api/v1/users/*{1,1}/vehicles/*{1,1}* has words *api, v1, users* and

vehicles, and two wildcards $\{1,1\}$ that matches at least 1 and at most 1 word. The second number of the wildcard would increase to $\{1,2\}$ in case the parameter consist of up to two words. However, this case does not need to be taken into account in our scenario, as URLs do not allow words separated by empty space. How the pattern is constructed is described in detail in (Vaarandi et al., 2015).

LogCluster is originally designed to cluster whole log events that recognizes words based on empty spaces as delimiter. Hence, in its original form, the algorithm would convert the URL string in the log event as wildcard as its appearance certainly exceed the passed support threshold. For that reason, we modified the algorithm in that way, it initially extracts the reported URL and uses the slash sign as the main word delimiter. In general, the URL string starts after the recognition of HTTP methods like GET, POST, PUT, DELETE, etc. and ends before the next empty space appears. As we are interested in the reconstruction of the RESTful Webservice interface, only the URL part in the log event is sufficient to consider.

As soon as we constructed the patterns for each log cluster, we translate the pattern into a regular expression. Those expressions are stored as the *Interface* type of an *AM*. Afterwards, we establish the relationship between *Interface* and *Application Service*. In case the *Application Service* is unknown as the installed Monitoring Server is unable to retrieve this information, we create dummy entities. The name of the dummy entity is built upon the *Architecture Component* name and *_IN* postfix for representing the processing of incoming transactions and *_OUT* postfix for outgoing transactions. As soon as *MICROLYZE* processes new incoming runtime data via the *forwardRecovery* algorithm, we validate each new log event in that specific timeframe against the stored regular expression. If the validation is successfully, we increase the API access counter with 1, otherwise we store the log event as unknown API call. In a frequent period of time, the modified *LogCluster* algorithm is executed on the unknown log events in order to update the event pattern repository. A complete visualization of this process is illustrated in Figure 5.22.

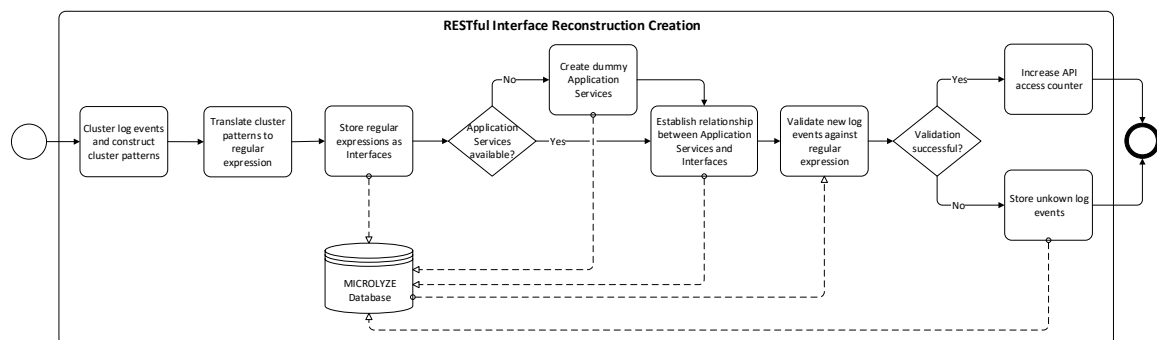


Figure 5.22.: RESTful Interface Reconstruction Process

5.3.7. Elaboration of a Deletion Threshold

The selection of an appropriate deletion threshold strategy is fundamental to keep a high model recovery 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 marked as removed could be based on simple manual input. That means, the user defines the period of time 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.

Statistic based: The drawbacks of the manual method could be neglected based on statistics. Hereby, we analyze past runtime data and extract maximum communications pauses between applications. Once this maximum is exceeded in future runtime data, the concerned communication will be marked as deleted. Even though this approach has benefits as it calculates the threshold automatically and is determined for each individual communication path, it does not take behavior changes into account that result from new deployments. In addition, communication pauses do not necessarily lead to architectural changes.

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.

As soon as we detect a communication that might have been removed due to the threshold excess, we set the *deleted* attribute to true but we do not close the validity by setting the *validTo* attribute to the current timestamp. The reason for this approach is that we want to keep potentially removed communications in the query result as we can never make sure those communications were definitely removed. Based on the *deleted* attribute, we emphasize those communications visually in our frontend tool. As a follow-up the Architects can finally remove those communications manually from the ETG.

5.4. Visualization Design

After we detailed the backend architecture of *MICROLYZE* and how we process and analyze runtime data for recovering the AMs, the following sections describe the visualization framework. In EA management and especially in modeling of IT landscapes,

visualizations are a common mean not only to identify and prioritize problems but also to understand the current state of an EA. Visualizations often form the basis for knowledge sharing, discussing planned states and therefore are indispensable for developing strategies and transition plans that implement changes. However, visualizing relationship and dependency information can be regarded as a complex task, as too much information adversely affect the recognition value. Such a complex task can be facilitated by advanced user interfaces that make use of different approaches to prevent information overload. Still, human cognition is especially strong in identifying patterns in complex information represented visually¹⁴.

As stated in Section 5.2 the IT landscape architecture is based on the ETG model accessible via GraphQL. Hence, we also provide the different views on the IT landscape in a graph-based representation. The conceptual framework builds on the works of Wittenburg (Wittenburg, 2007). Although Wittenburg presents a comprehensive Visualization Model, we put the focus of the IT landscape representation on schematic relationships, like communications, deployment dependencies and schematic groupings. In addition, we decided to apply the well-known Archimate notation (The Open Group, 2016) for styling all ETG nodes and edges. With this approach, we ensure that the IT landscape visualization is comprehensible from the start and requires less explanation.

The remainder of this section is as follows: First, we present the architecture of the visualization component and outline general principles. Then, we detail how users can interact with the visualization framework. Finally, we show different IT landscape views and the rationale behind them. Each visualization addresses and renders dependency information from a particular perspective. Hereby, we distinguish between *Compositions*, *Associations* and *Flow* dependencies.

5.4.1. Visualization Architecture

We developed the frontend of *MICROLYZE* with the JavaScript Library ReactJS¹⁵. The library is maintained by Facebook and a community of individual developers and companies. We use ReactJS as a base for developing a web-based single-page application. ReactJS code is made of stateful components, that can be rendered to a particular element in the DOM using the ReactJS DOM library. For visualizing the nodes and edges of the ETG, as well as applying different layouts for the graph, we leverage the visualization library *yFiles*(Wiese et al., 2001)¹⁶. In general, *yFiles* is a library for the visualization and automatic layout of graphs. Included features are data structures, graph algorithms, a graph viewer component and diverse layout and labeling algorithms. The main layout styles are orthogonal, tree, circular-radial, layered and force-directed. For the representation of the graph, we provide all layout styles to the user. In the following, we describe the particular components of our visualization framework in more detail. Figure 5.23 sketches the architecture.

¹⁴For general design guidelines for visualizations we refer the interested reader to Tufte (Tufte, 2001), and Moody (Moody, 2010); for design guidelines for information dashboards we refer to Few (Few, 2006)

¹⁵<https://reactjs.org>, last accessed: 2020-10-28

¹⁶<https://www.yworks.com/products/yfiles-for-html>, last accessed: 2020-10-28

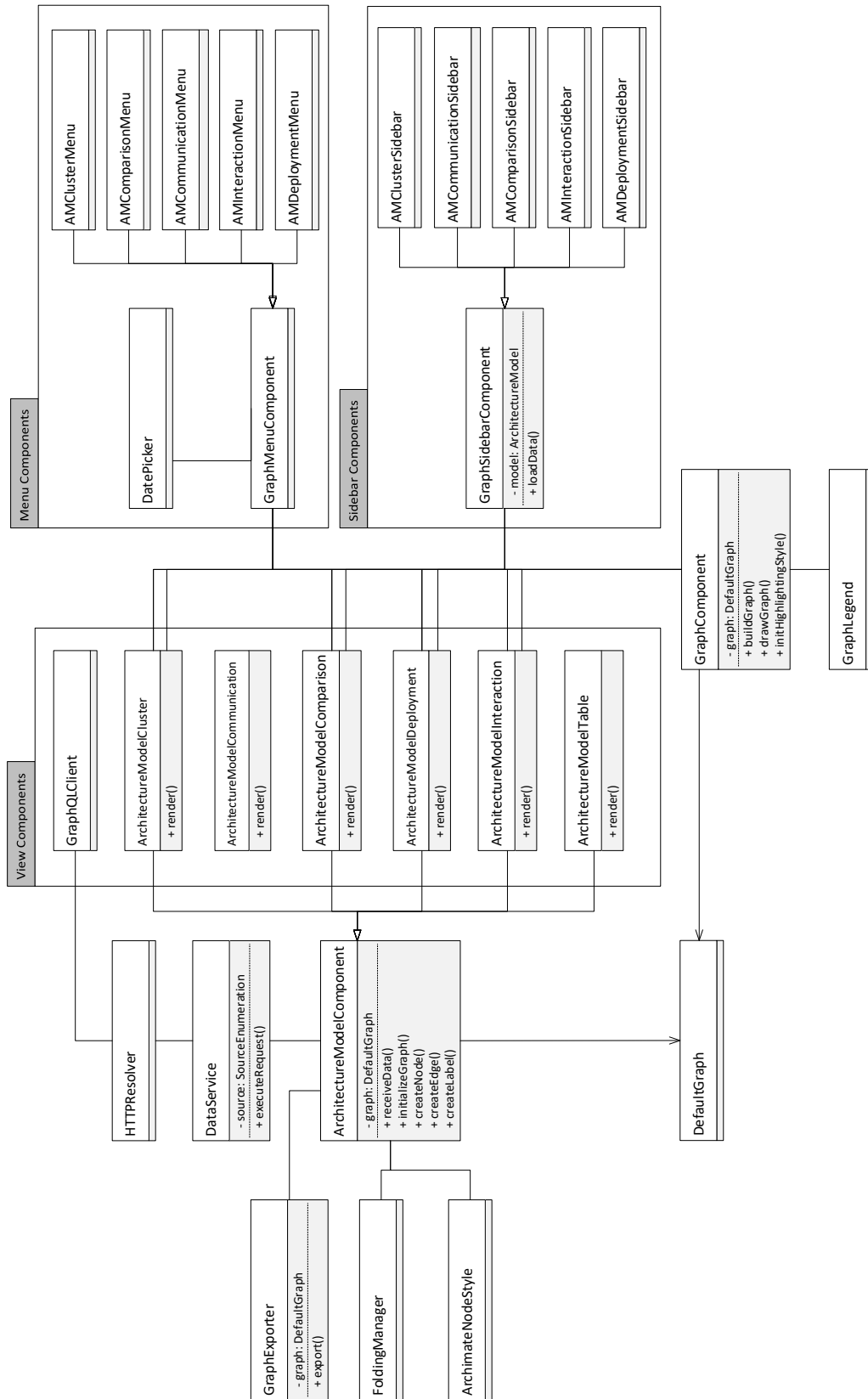


Figure 5.23.: Architecture of the visualization components of MICROLYZE

- The *ArchitectureModelComponent* serves as the parent class for all visualization component classes that transform information represented in the general meta-model (see Section 5.1.5) to a target structure that is convenient to traverse in the viewpoint. Within the *receiveData()* method of each subclass, this model-to-model transformation is performed. The method executes GraphQL queries on the source model, applies filters and transforms the model into a view model. This view model denotes the intended part of a model that is visualized later on. *ArchitectureModelComponent* provides required methods for creating nodes, labels and the edges of the ETG that represents the IT landscape from an particular viewpoint. In addition, it manages the styling towards the Archimate notation. The graph styling classes are described in Section 5.4.2 in more detail.
- *DataService* is a static class managing the client-server communication via GraphQL-based query statements.
- The *View Component* group contains all ReactJS components that render the particular visualization pages. They inherit attributes, methods and in particular style formats from the *ArchitectureModelComponent* class.
- Most visualization views are compiled with a graph component, a menu component and a sidebar component. The *GraphMenuComponent* provides functionality for interacting with the ETG by adding dropdown items for filtering, time picker for choosing an architecture revision or search fields.
- The *GraphSidebarComponent* contains further static and dynamic information about each AM and their assigned relationships.
- The *GraphComponent* class is responsible for building the ETG and rendering it to the DOM. It receives the required information of the ETG from the particular visualization view components. All user interactions with the graph is processed by the *GraphComponent*.

For each visualization type, a concrete *View Component* exists that inherits from the *ArchitectureModelComponent*. Each view component executes a GraphQL statement that responds in a JSON-based format. During the execution the GraphQL resolver translates the relationship information into the unified meta-model. The type of each returned AM is defined by a type attribute integrated in each GraphQL statement. Based on the returned AM type, the correct node style can be selected. The same approach applies to the edge style. With this approach any dependency information can be visualized via the Archimate taxonomy.

5.4.2. Graph Styling

After describing the core concepts of the visualization framework, we introduce some additional classes that make up the styling and layouting of the ETG. The Archimate taxonomy already introduces symbols, visual styles and notations for visualizing the

IT landscape. We apply this taxonomy for creating our ETG. The corresponding class diagram is illustrated in Figure 5.24 and detailed in the following.

- The *NodeStyleBase* is the abstraction for different node representations. We separate between *NodeStyle* that visualizes a node in its basic format and *GroupStyle* that renders a folder-based structure of nodes. We commonly use *GroupStyles* for grouping information that either belong together semantically, or represent hierarchies. For instance, *Application Components* that are build upon the same technology like Java, NodeJS, .Net, etc. are grouped together. A further example is the definition of hierarchical relationships. Therefore, a *Product* that consists of several *Business Services* is visualized as a group node. In addition, we developed a *NodeStyleDecorator* that provides methods for individual node decoration. We use this class for styling our nodes based on the Archimate taxonomy provided in the *ArchimateNodeStyle* class.
- The purpose of the *EdgeStyleBase* class is to provide methods for styling the edges of the ETG. Those methods are inherited by the child classes *EdgeStyle* and *EdgeStyleDecorator* which represent edges either in a basic format or decorated towards the Archimate taxonomy.
- The ETG is compiled by nodes, edges, as well as edge labels and node labels. Labels are used to visualize text on nodes and edges. For this purpose, we developed the abstract class *LabelStyleBase* and three subclasses. The *HtmlLabelStyle* renders HTML markup for formatting label text. This class is used for displaying all Archimate models that are not collapsed in a folder. Whereas, the *GroupLabelStyle* class displays the label of folder nodes. Last, but not least, the *EdgeLabelStyle* class is responsible for superimposing labels on edges.
- The *LayoutBase* class and its subclasses provide methods for applying layouts for the ETG. The classes control how the ETG is build and which algorithm is used to arrange the nodes and edges. This also includes how the graph representation changes while uses are interacting with it, like zooming, expanding or collapsing group nodes, etc.

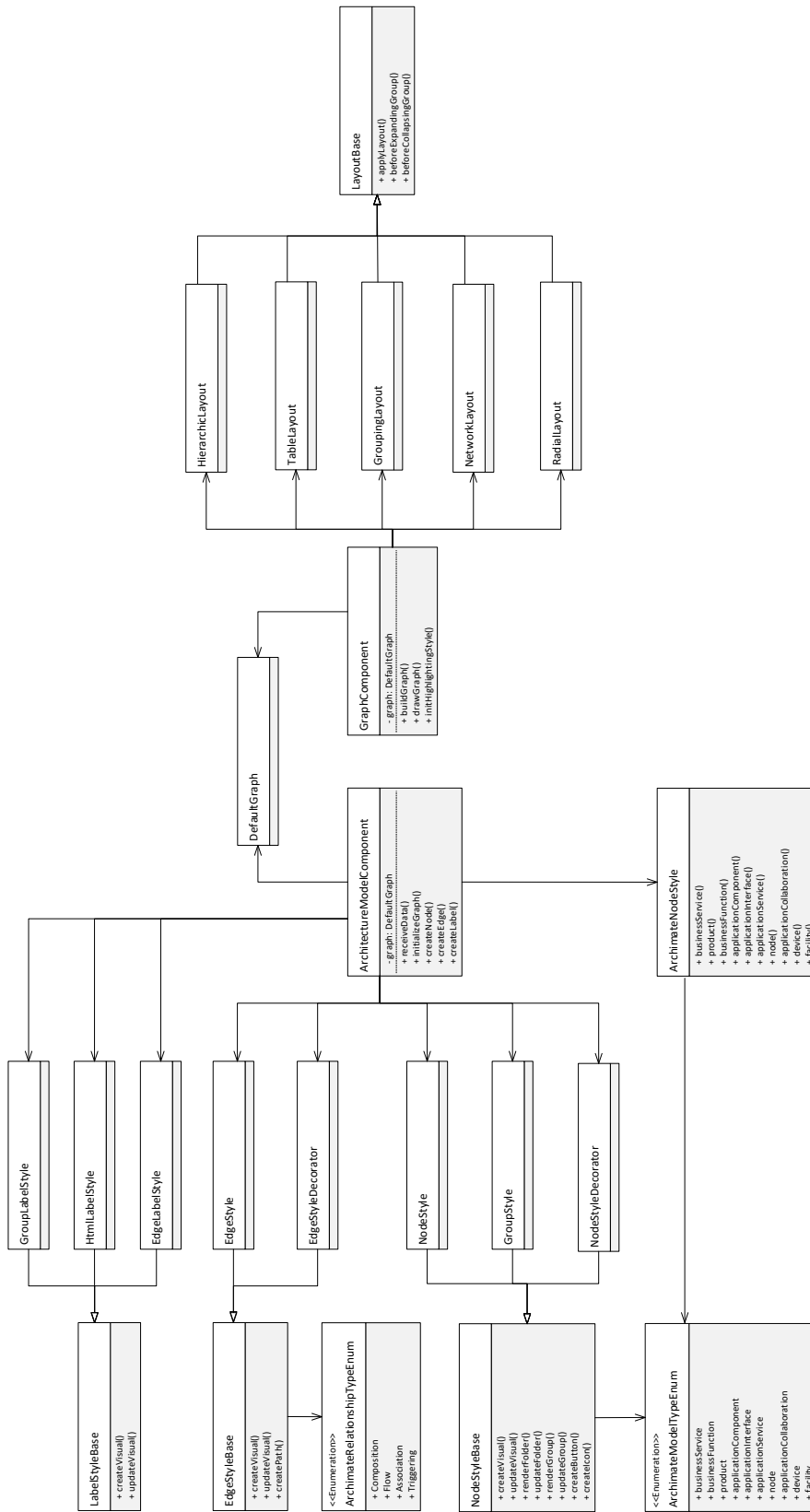


Figure 5.24.: Style and layout architecture of MICROLIZE

Figure 5.25 illustrates the visual representation of the different nodes we use to build the ETG. Each node consists of a label and a symbol. The symbol and the color of the node represents the corresponding AM which is based on the Archimate taxonomy. We separate between leaf nodes and group nodes. Group nodes group schematically-related nodes and can be collapsed or expanded like a folder. Collapsed group nodes visualize the number of child nodes in the middle of the node. Collapsed and expanded group nodes position the label on the top left side of the node. Leaf nodes show the label in the middle of the node. Each node contain the Archimate corresponding symbol on the top right side. In addition, *Application Components* draw two rectangles in the middle of the node that show the number of instances and exposed interfaces.

Each leaf node and collapsed group node is 140px wide and 80px high. This size is fixed and only changes by expanding group nodes. The label width is determined by the width of the node minus the space which is required for drawing the grouping button and the Archimate symbol. If a *Label* object requires more space than it has available, it gets shrunk to its width and the removed text is replaced by three dots.

As stated in Section 5.1.5, we classify dependencies between AMs in *grouping*, *hierarchy*, and *communication* relationships. The corresponding Archimate taxonomy determines those relationships in *Composition*, which indicates that an element consists of one or more other elements, *Association*, which models an unspecified relationship and finally *Flow*, that details a dynamic relationship which indicates a transfer of information from one element to another. The visual representation of those relationships is shown in Figure 5.26.

The information which node or edge type must be visualized by the frontend is directly provided by the backend and derived by the relationship meta-model. How the visualization process is performed in detail is described in the next Section.

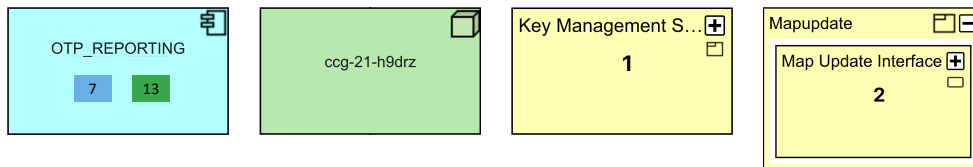


Figure 5.25.: AM Visualization: (left) *Architecture Component* with 7 instances and 13 exposed interfaces, (middle-left) representation of a *Node* element, (middle-right) collapsed *Product* with 1 subelement and shrunk *Label*, (right) expanded *Product* with one *Business Service* as subelement. Further AMs are displayed in a similar way and with the correct Archimate symbols.

5.4.3. Visualization Process

Figure 5.27 gives a high level overview of the visualization process. In a first step, the required data for visualizing a particular perspective on the IT landscape is retrieved via the *DataService* class. The method *receiveData* is used to execute a predefined GraphQL

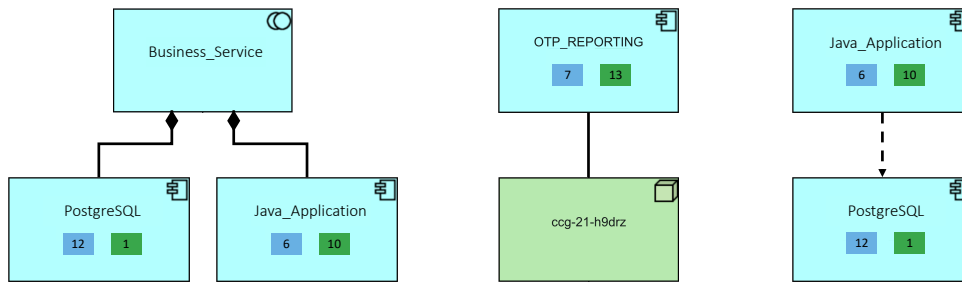


Figure 5.26.: Relationship Visualization: (left) *Compositions* are marked with a rhombus and determine *contains* relationships, (middle) *Associations* are represented by a solid line, indicating *runs on* relationships, and (right) *Flows* are drawn with a dashed arrow and defines *calls* relationships

statement that responses in JSON format. Subsequently, the retrieved data records are analyzed within the *ArchitectureModelComponent*, which also creates the ETG in its raw form. Hereby, the following steps are performed: First, an empty default graph is initialized and all the taxonomy for visualizing the ETG in the Archimate styles is loaded. Next, we iterate through the JSON data and create all nodes, group nodes, edges and labels for the ETG. The information on how the nodes and edges have to be styled according the Archimate taxonomy is stored in the attribute *type*, which is a main integral of the GraphQL statement.

In the next step, the *GraphMainComponent* is used for building and layouting the ETG. In the method *buildGraph*, the ETG including its nodes, groups, labels and edges is compiled. Afterwards, we apply a specific algorithm for arranging the graph. We can choose between 1) hierarchic layout which distributes the nodes into layers so that most of the edges point to the main layout direction, 2) the tabular layout style arranges the nodes in rows and columns, 3) the circular layout style emphasizes group and tree structures within a network. It creates node partitions by analyzing the connectivity structure of the network, and arranges the partitions as separate circles. The circles themselves are arranged in a radial tree layout fashion. 4) The radial layout style arranges the nodes of a graph on concentric circles. The overall flow of the graph is visualized similar to hierarchic layouts. The layout algorithm itself runs in three main phases. In the following, we describe those phases with hierarchic layouting as example:

1. Layering: The nodes are distributed into layers by means. If the layout orientation is top-to-bottom, the nodes in each layer are arranged horizontally while the layers are ordered vertically top-to-bottom.
2. Sequencing: The order of the nodes in each layer is determined such that the number of edge crossings is as small as possible.
3. Drawing: The layout algorithm assigns the final coordinates to all nodes and routes the edges.

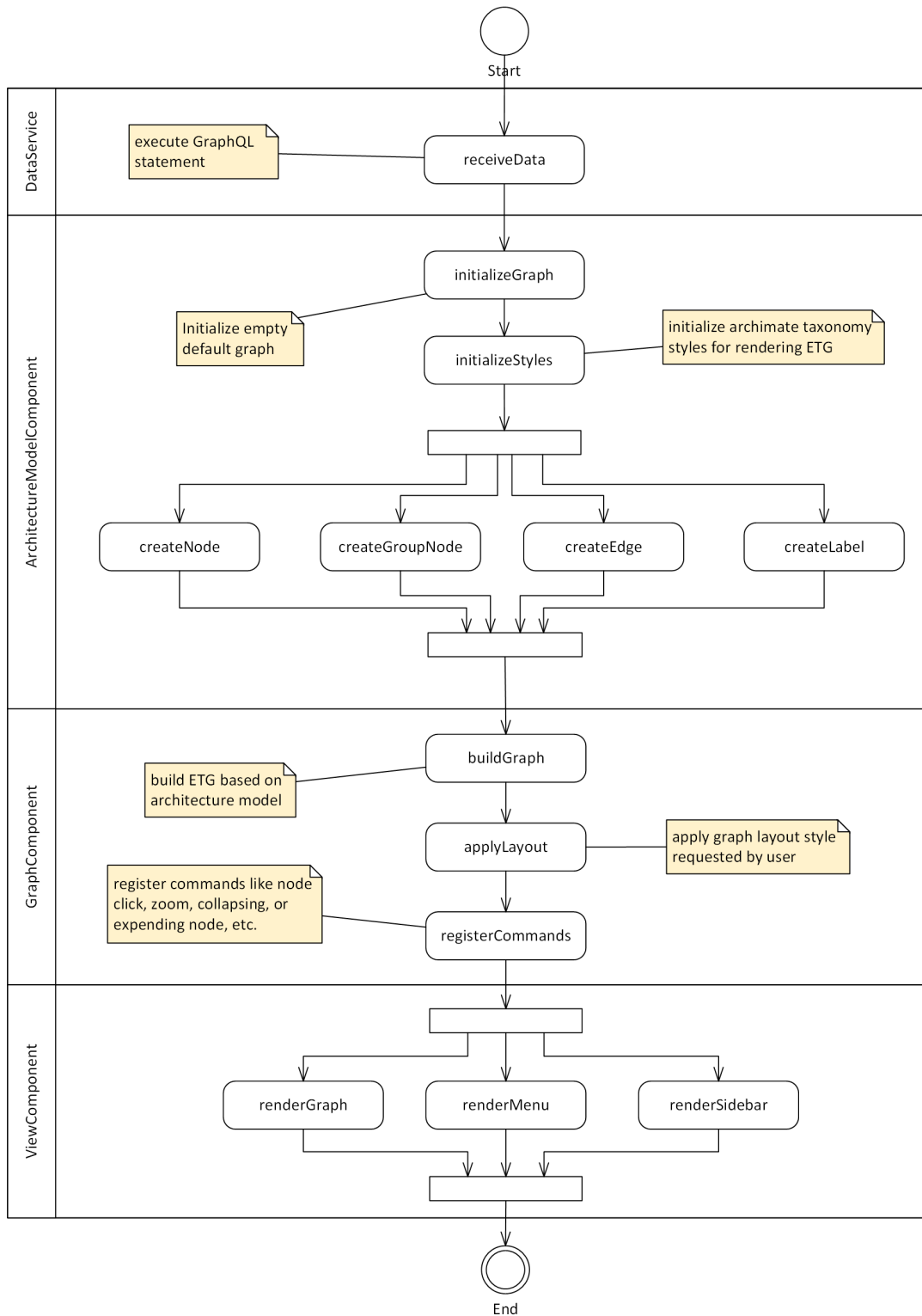


Figure 5.27.: Overview of the Visualization Process represented in an activity diagram

Finally, we register commands required for interacting with the graph. This includes node and edge selection, zoom functionality, collapsing and expanding group nodes, as well as hovering and drag and drop features.

In a final step, the particular *ViewComponent* which is selected for visualizing the graph renders all three main components of a *View*, the ETG, the menu and the sidebar.

5.4.4. Architecture Model Deployment

The following visualization is a first demonstration that highlights the potential of querying the ETG with GraphQL. We query AMs from different layers and visualize their hierarchical dependency. The query statement is provided in Listing 5.28.

```

1 query getAMDeployment{
2   database {
3     applicationCollaborationAt(id:"${id}", timestamp:${timestamp}) {name
4       applicationComponents {name
5         nodes {name
6         facilities {name
7   }}}}}

```

Figure 5.28.: GraphQL query for *Architecture Model Deployment* visualization

Goal: The *Architecture Model Deployment* view shows the interrelationship of the application- and infrastructure layer on the basis of *Application Collaborations*. This view can be used to determine on which server microservices are deployed and which server technology or environment (cloud or on-premise) is employed. In this context, the *Application Collaboration*, *Application Components*, *Nodes* and *Facilities* present the different layers in descending order.

Visualization: Each of those AMs represents nodes in a graph-based structure as Figure 5.29 details. The root node represents the selected *Application Collaboration*. The leaf node is always the *Facility*. The positioning of the nodes and group nodes are determined automatically and cannot be changed manually. *Nodes* with the same technology (Linux or Windows) are grouped in a folder. When collapsing group nodes the out-going and in-going edges are aggregated accordingly in order to reduce unnecessary duplicates. The shown number of instances on every *Application Component* corresponds to the number of out-going edges.

Interaction: The menu component of the *Architecture Model Deployment* view contains three input fields. The datepicker shows the selected architecture timestamp. The single-select dropdown field contains all *Application Collaboration* that can be selected. Performing a search in the search bar lead to a highlighting of the nodes which names are within the research result. Group nodes can be expanded or collapsed and moving the mouse wheel performs zooming. A sidebar is opened immediately after clicking on any node.

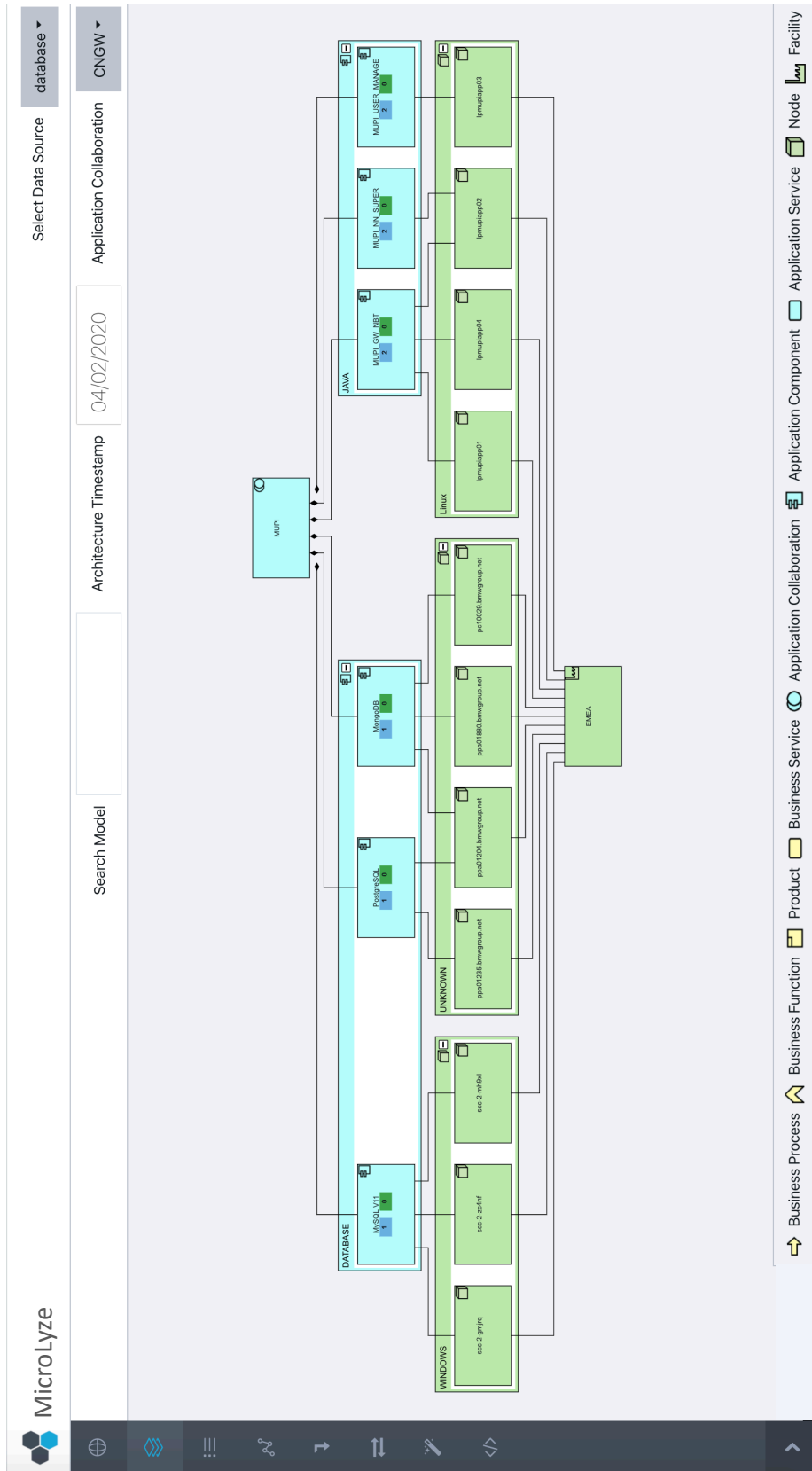


Figure 5.29.: Visualization of *Architecture Model Deployments* by structuring the model dependencies in a tree-based representation. Grouped nodes can be collapsed or expanded.

5.4.5. Architecture Model Communication

The aforementioned view represent only hierarchical relationships but no communications between AMs. In the following view, we detail how *MICROLYZE* empowers users to identify communication dependencies between *Application Component*. In this scope, communications are collected on *Application Service* level by observing the called *Interfaces* and afterwards aggregated up on *Application Components*.

Goal: The *Architecture Model Communication* views shown in Figure 5.30 respectively, enables users to analyze which AMs communicate with each other in general, i.e. exchange information and how often they communicate with each other. This is especially important to understand architecture complexity. The communication direction is displayed via a directed arrow. Further information about the communication details are provided in the sidebar. This includes the synchronicity as well as the addressed interfaces.

Visualization: The nodes are arranged tree-like. The positioning of the nodes are determined automatically via an algorithm provided by the yFiles library. The positioning cannot be changed manually. Those nodes which have neither in-going nor out-going communication paths are arranged at the very top. Nodes with communications are layouted underneath. Each child nodes are positioned in levels. By moving the mouse pointer over a node, the node will be highlighted in red as well as all assigned communication paths. The hovering of an edge leads to the highlighting of the edge inclusive of the adjacent nodes.

Interaction: Similar to the *Architecture Model Deployment* view, group nodes can be expanded or collapsed and moving the mouse wheel performs zooming. A sidebar is opened immediately after clicking on any node. The menu component of the *Architecture Model Communication* view contains two input fields. The date-picker shows the selected architecture timestamp. Performing a search in the search bar lead to a highlighting of the nodes which names are within the research result.

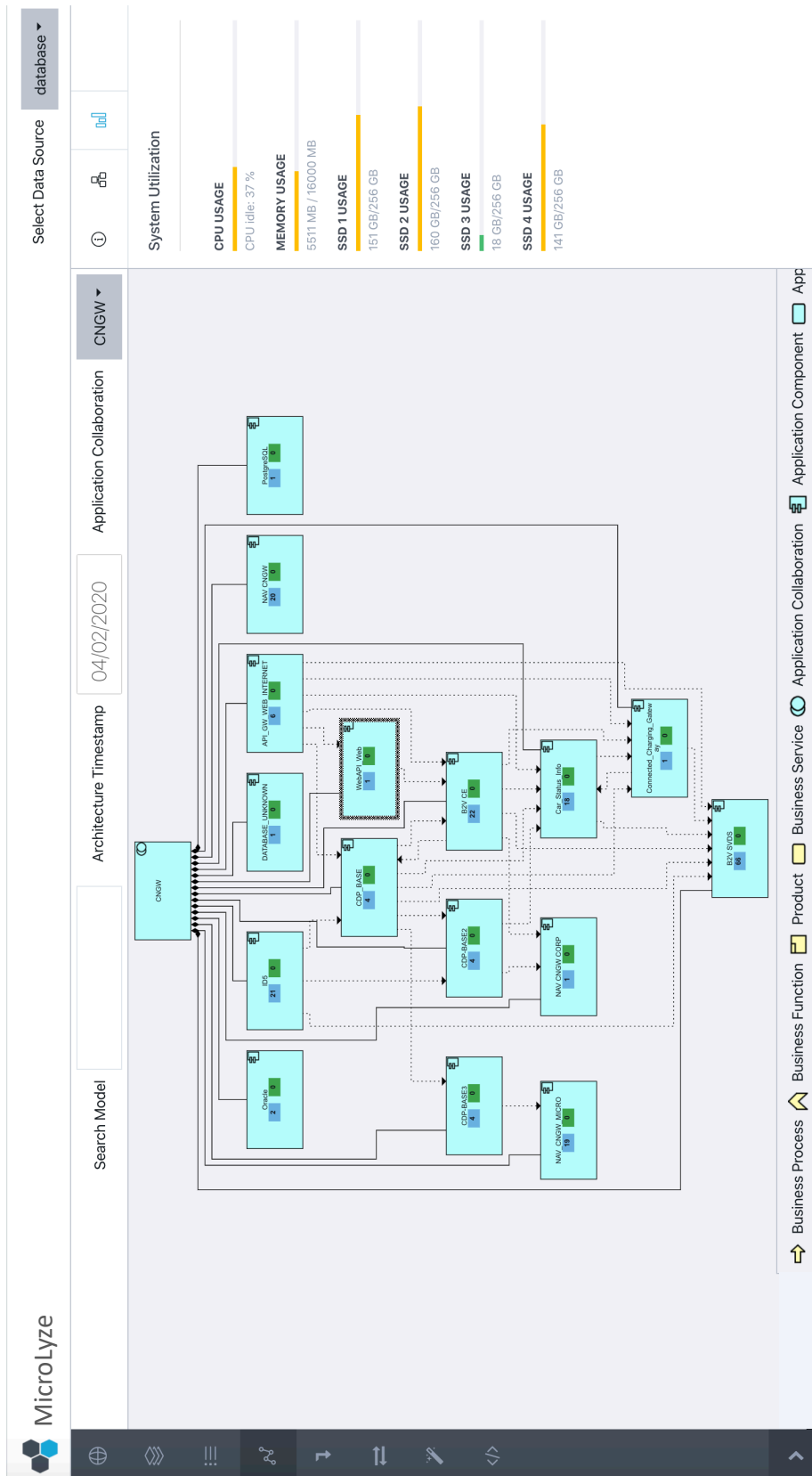


Figure 5.30.: Architecture Model Communication visualization by leveraging tree-based representations.

5.4.6. Architecture Model Interaction

The previous described visualizations represent the IT landscape in a rather static way without any dynamic characteristics. However, in order to understand how the IT landscape behaves and emerges over time, we developed a visualization that focus on dynamic aspects.

Goal: The *Architecture Model Interaction* visualization uncovers how *Application Components* are interacting with each other to process a certain business request. This is especially important for troubleshooting and architecture optimization. In detail, it shows the transaction flow through the system and which services, interfaces and communication paths are used. Figure 5.31 illustrates this interaction. In this example, the root node represents a *Application Collaboration*. The following child nodes are *Application Components* representing microservices, databases and the like. As the visualization constitutes a very fine granular representation of the IT landscape it is mainly suitable for Solution Architects or Developers.

Visualization: The positioning of the nodes are determined automatically and follow a tree-like layouting. The positioning cannot be changed manually, in order to achieve a high recognition value by the users. The transaction flow is shown via two different approaches. The first approach leverages the tree-based preparation of the microservice architecture and highlights used communication paths. The highlighting is achieved by changing the edge and node border color to blue and increasing the line thickness. The second approach is showing the microservice interaction via a table. This table lists all microservices that are required for processing the transaction in a chronological order. The table has four columns as illustrated in Figure 5.31. The first field contains the name of the microservice that communicates with the microservice in field two. The third field describes through which interface (in URL representation) the communication is established. The interface is exposed by the called microservices. The parameters have been removed from the interface. The last field indicates the synchronicity of the communication, i.e. whether the communication is performed through a message broker.

Interaction: The menu component of the *Architecture Model Interaction* view contains three input fields. The date-picker shows the selected architecture timestamp. By changing the value of the *Application Collaboration* dropdown field the architecture of the selected *Application Collaboration* is retrieved. The second dropdown field contains all transactions that were processed within the selected architecture. The selection of a particular transaction leads to the visualization of the transaction flow. In parallel, the aforementioned table is opened. Furthermore, the mouse wheel changes the zoom level of the graph and a sidebar is opened immediately after clicking on any node.

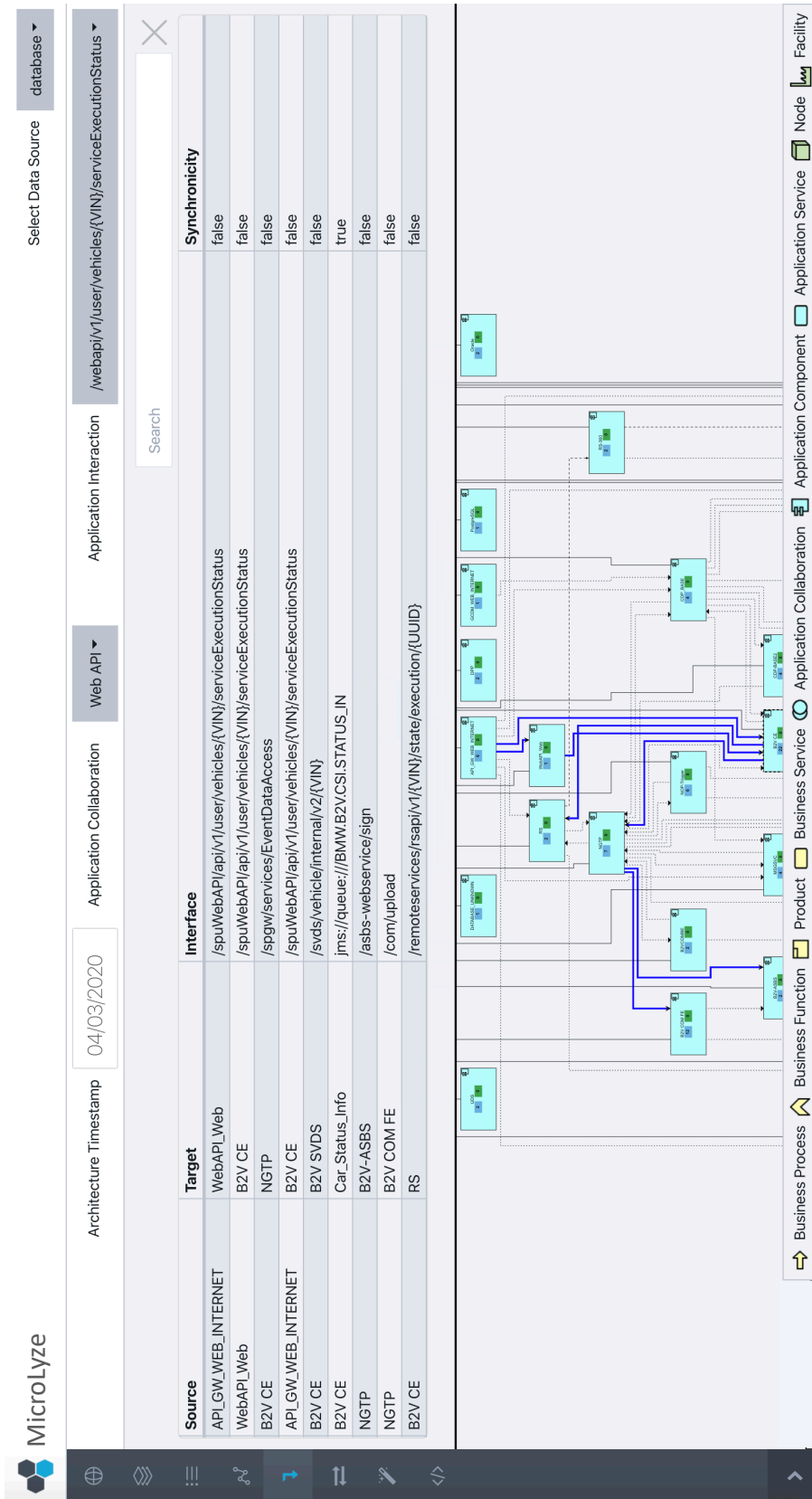


Figure 5.31.: Revealing *Architecture Model* interactions by highlighting transaction paths within a tree-based representation. The table above the graph details the used interfaces for transaction processing.

5.4.7. Architecture Model Comparison

Whereas the *Architecture Model Interaction* visualization uncovers how transactions are processed with the system, the following visualization describes how the IT landscape changes over time.

Goal: The *Architecture Model Comparison* clarifies to what extent the regarded section of the IT landscape has changed over time by comparing two different architecture timestamps. This enables Architects to validate whether the changes correspond with the planned architecture and can intervene at an early stage if necessary, in case the changes are going in the wrong direction. Figure 5.32 illustrates an example.

Visualization: The view is divided into two parts. Each part represent an excerpt of the IT landscape in a specific time. The left part contains an older representation of the IT landscape. In case new AMs have been introduced into the landscape the particular nodes on the right side of the view get a green border. The same applies for new relationships. If AMs or relationships between models have been removed this transition is expressed in red. As in all other tree-based visualizations, the positioning of the nodes are determined automatically and the positioning cannot be changed manually.

Interaction: The menu component of the *Architecture Model Comparison* view contains two timepicker elements which enables the user to change the architecture timestamps of both views for the aforementioned comparison purposes. Further interactions like zooming and the visualisation of the sidebar via clicking on any node is implemented as well.

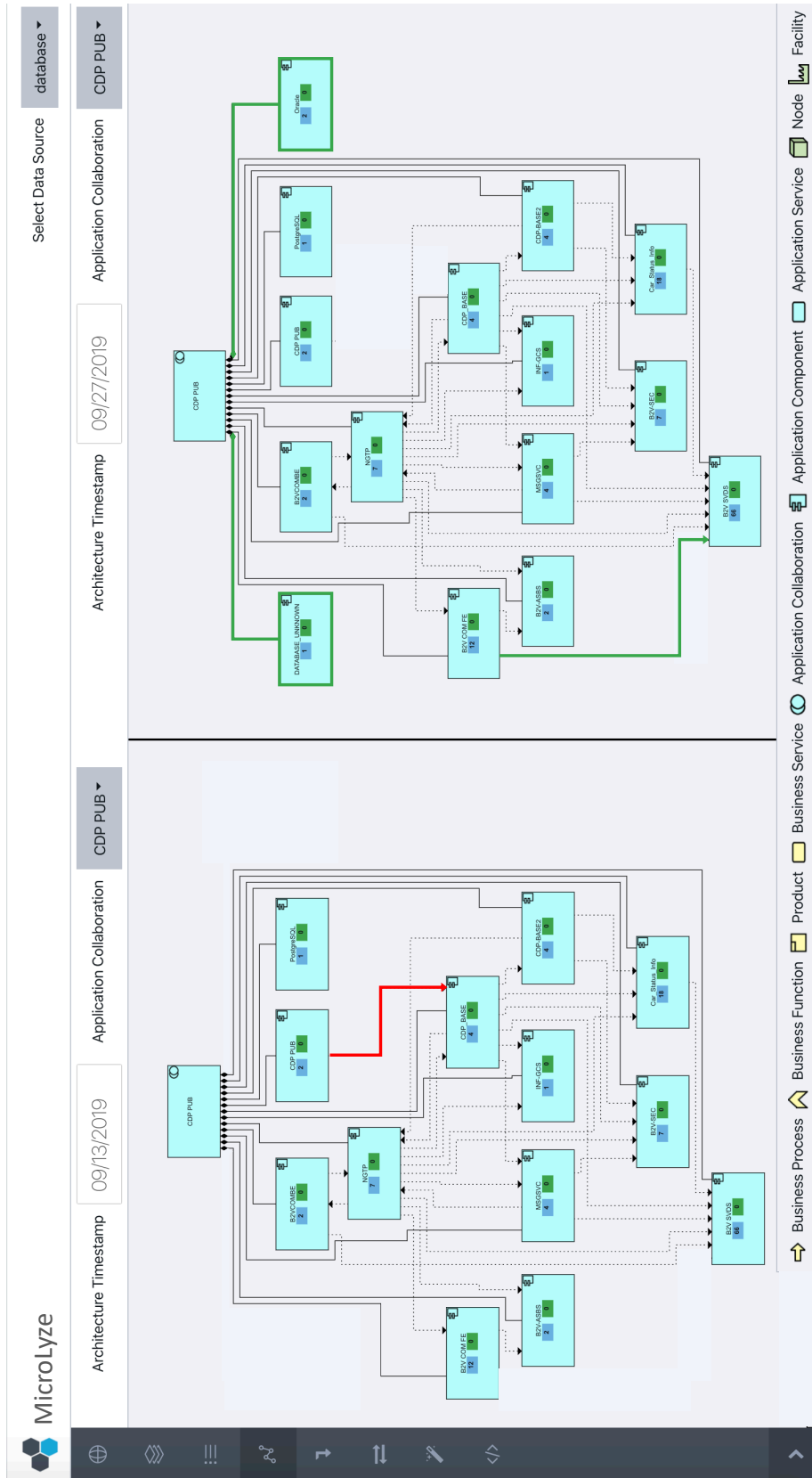


Figure 5.32.: Architecture Model Comparison visualization by splitting the window layout. The left hand side displays the obsolete architecture of the selected *Application Collaboration*. The right hand side unveils the updated architecture. Removed relationships or AMs are highlighted in red. New relationships or AMs are visualized in green.

5.4.8. Architecture Model Sidebar

Each described IT landscape visualization contains three components 1) the main component, 2) the menu component and 3) the sidebar component. All three components are required to provide the user sufficient information about the IT landscape and the best user experience. In the following, we detail the sidebar, illustrated in Figure 5.33.

Goal: The sidebar provides further information about a selected AM or a communication relationship between two AMs. Based on the selected element the provided information is either retrieved from the database or extracted directly from runtime data. The sidebar for the application and infrastructure layer is filled with runtime information. In addition, the sidebar opens for *call* relationships between AMs. The displayed information comprises primarily runtime data as well as communication details like used interfaces and synchronicity.

Visualization: The sidebar component consists of three tabs as Figure 5.33 shows. The first tab "General Information" assembles all static attributes provided by the monitoring tool. This includes inter alia technology type, OS type, cloud provider, containerization technology and virtualization vendor, as well as IP address, ports, CPU cores and further meta data. The second tab "Relationship" details the caller and callee relationship of the selected AM. For instance, if a microservice is called by another microservice this information is listed in the *called by* category. If the same microservice calls other microservices this relationship is detailed in the *calls* table. By clicking on the "communication" button a table modal is displayed which visualizes the communication relationship in more detail. The third tab "Monitoring" shows how the selected AM or *call* relationship performs in runtime. We consume different KPIs from the monitoring tool like CPU-, memory-, and disk utilization, data throughput, requests per seconds or response time. Which model KPIs are shown depends on the architecture layer in which the selected node is assigned.

5.4.9. GraphQL Client

All aforementioned visualizations provide different perspectives on the IT landscape. However, all approaches face several issues: the scalability suffers by increasing data volume which leads inevitably to bad transparency. Information will always be missing as either not all requirements can be fulfilled or the graphical representation is difficult to achieve. For that reason, we also integrate an interface that enables users to write their own GraphQL statements for querying any perspective on the IT landscape which they currently require. The result of the queries are provided in json-based format but can also be exported to Microsoft Excel. For this purpose, we integrate the open-source GraphQL client "GraphQL Playground"¹⁷ developed by prisma¹⁸. The menu component contains one button which exports the query result to Microsoft Excel. An example query including the output result is depicted in Figure 5.34.

¹⁷<https://github.com/prisma-labs/python-graphql-client>, last accessed: 2020-10-28

¹⁸<https://www.prisma.io>, last accessed: 2020-10-28

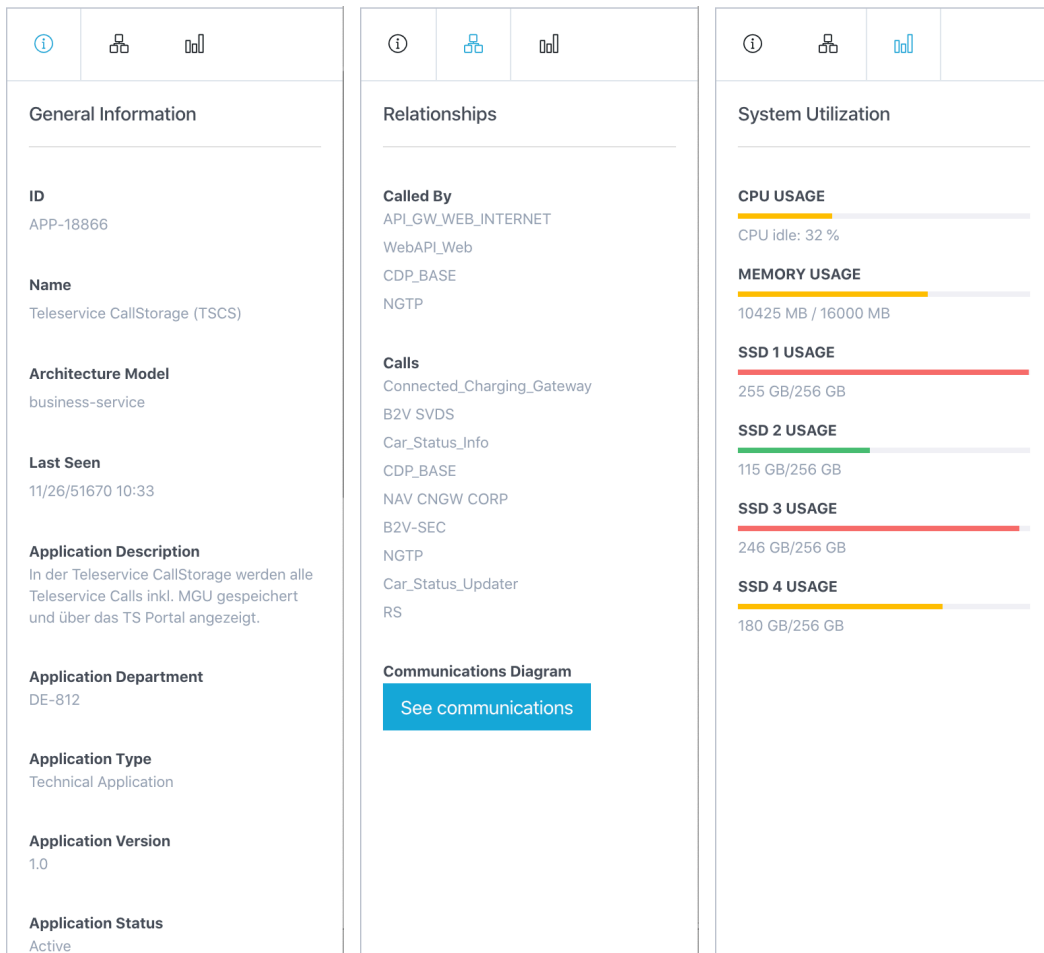


Figure 5.33.: Sidebar that opens after clicking on a *Architecture Model* or communication relationship. **Left:** static information about the AM that is stored as annotations in the database. **Middle:** communication relationship information separated in *calls* and *called by*. **Right:** runtime information about the AM

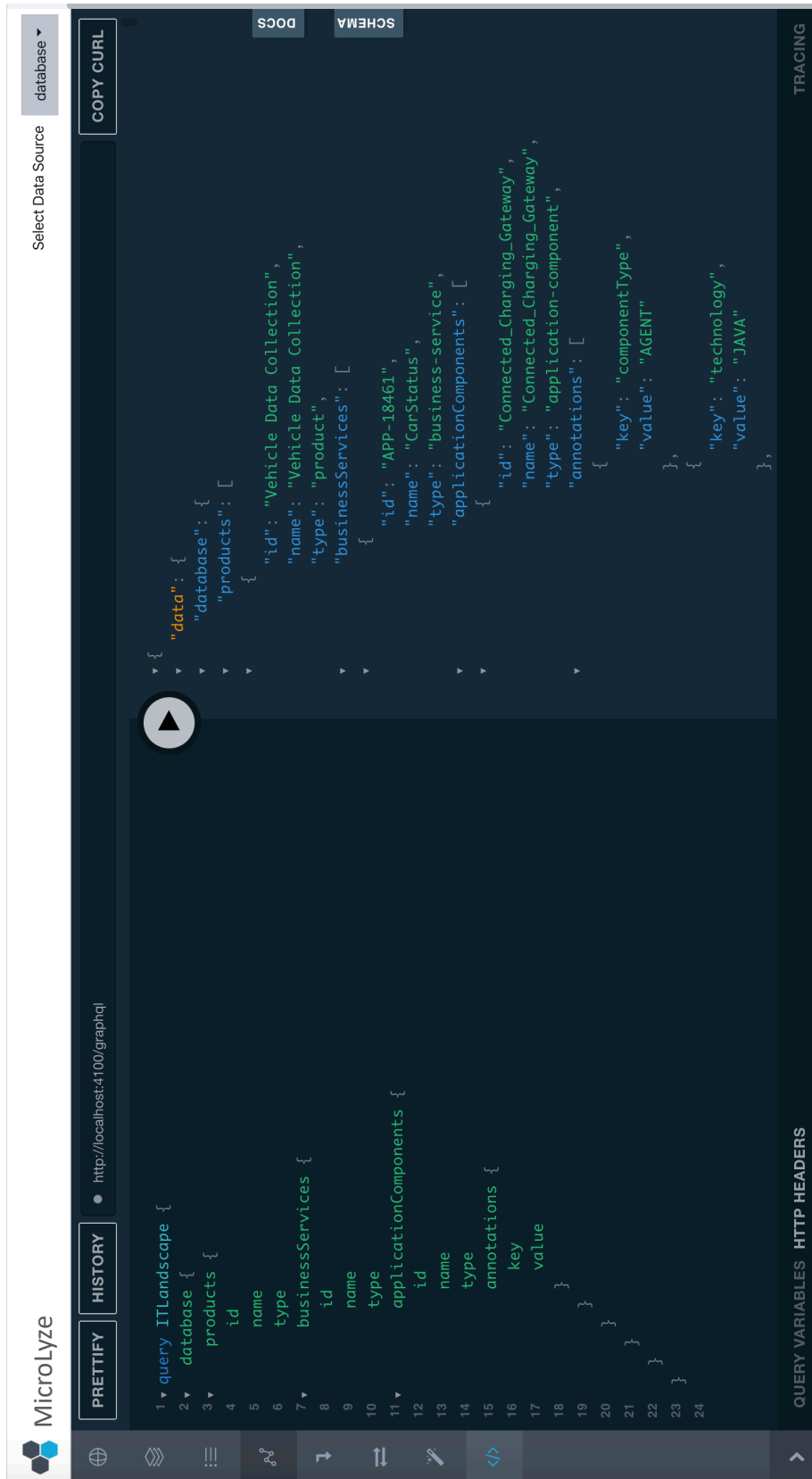


Figure 5.34.: GraphQL client developed by Prisma. The view illustrates an example how to write GraphQL queries and how the result is displayed.

6. Recovery of Business-related Models

The described concept in Chapter 5 that uses runtime data to automate the architecture recovery focuses primarily on technical aspects of an IT landscape. However, an overall management of EA models does not only cover the application and infrastructure layer but also encompass the allocation of models within the business layer. Having a thorough understanding of all parts and pieces of the enterprise's IT including their inter-relationships, lead to profound strategic decisions and a smoother transformation of the EA into the desired direction which is in line with business needs (Farwick et al., 2011b).

As demonstrated in Chapter 5 runtime data can only provide information about the specialization and execution phase of technical models. However, business-related information covering the status of ongoing projects, software development progress, performance of products, the relationship to business services and business domains as well as team definitions are not available in runtime data. This information resides in federated information systems and are used along the IT value chain. Consequently, it is necessary to extract this information and map it with the runtime data in order to enhance the common meta-model with further business-related information. With this approach, we aim to achieve a complete and holistic view on the model representation. In the following Sections, we present a concept of how to enhance the common meta-model with further business-related information that cannot be extracted out of runtime data. A detailed description of the concepts are presented in (Achhammer, 2019; Corpancho, 2019; Kleehaus et al., 2019a, 2021; Machner, 2019).

In Section 6.1, we describe to what extent we extend our backend for supporting further AMs that represent the business-layer of an organization. In this scope, we propose the creation of a configuration file that must be manually maintained and assigned to each deployed microservice. This file contains all required information for establishing the relationship between business- and application layer.

As the maintenance of the configuration file introduces a manual process, several stakeholders must be involved that contribute to the solution concept. Which stakeholders are addressed and what tasks they must undertake is stated in Section 6.2. In addition, we detail how the concept can be integrated into a common agile development process and which artifacts in SCRUM need to be adapted accordingly.

The previous Sections outline the system and organizational design of our solution approach. In Section 6.3, we focus on technical processes how to import the content of the configuration file into *MICROLYZE*. In particular, we highlight the process of validating the content of the configuration file and which prerequisites must be fulfilled.

Finally, we present three visualizations in Section 6.4, that we use to visualize the application- and business layer relationship. For this purpose, we leverage clustermaps and table views.

6.1. System Design

The field of software engineering has changed in the recent years to become as agile as possible by leveraging various social and technical techniques that improve the development process pace and quality (Hanssen et al., 2011). Social techniques are the ongoing adoption of agile process models like scrum (Schwaber et al., 2002) or kanban (Ohno, 1988) and even techniques directly related to the development itself like pair programming.

Continuous delivery (Humble et al., 2010) (CD) is considered as an emerging paradigm that aims to significantly shorten software release cycles and improve feedback loops by bridging existing gaps between developers, operators, and other stakeholders involved in the delivery process. A prerequisite to establish continuous delivery, is a high degree of technical automation. This is typically achieved by implementing an automated CD pipeline (also known as deployment pipeline) (Shahin et al., 2017), covering all required steps such as retrieving code from a repository, building packaged binaries, running tests, and deployment to production.

As CD pipelines directly indicate a non-periodic change in the IT landscape, we regard this technology as an important starting point to enhance our meta-model with business-related architecture models. The overall system integration concept described in Archimate 3.0.1 is depicted in Figure 6.1 and detailed in the following.

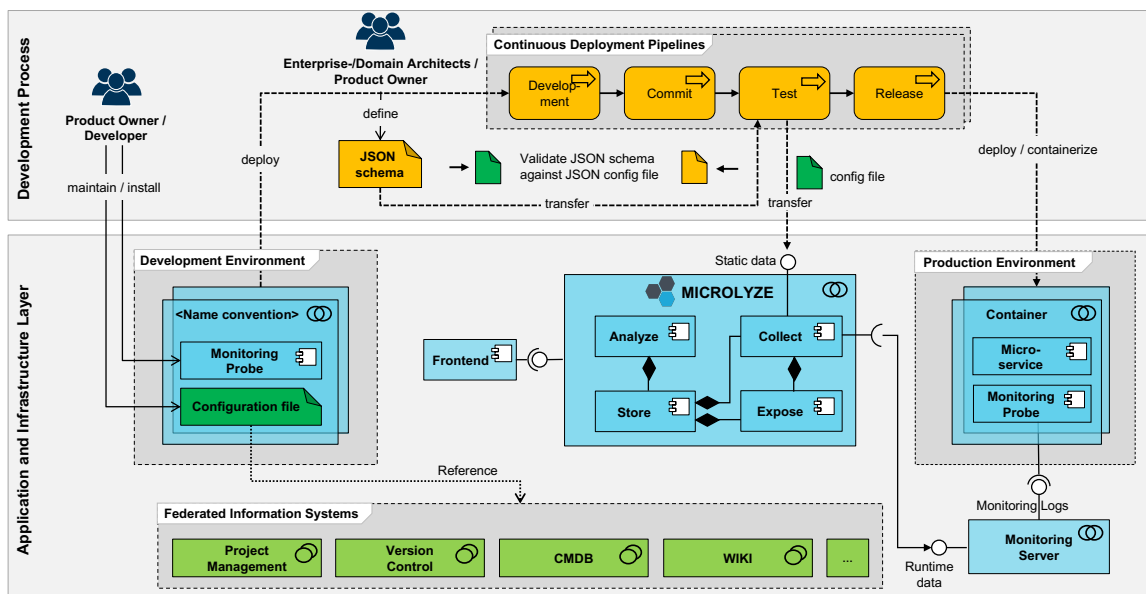


Figure 6.1.: Overall System Integration Concept

The Figure shows two layers that together represent a general agile-based software development process. The bottom layer covers the application and infrastructure layer in which software systems are developed and process business transactions. This layer is separated into different environments. In the development environment all new applications are created by the development team. New or updated applications are deployed into the

production environment. The top layer of Figure 6.1 details all activities of the software deployment process that is supported by a CD pipeline. Our concept for recovering EA models is arranged between the development and production environment and is an integrated part of the agile development process. It is performed with each run of the deployment pipeline without any further manual intervention. In an iterative and incremental software development process the models are updated at least with each product increment release. Hereby, the following prerequisites must be fulfilled in order *MICROLYZE* becomes fully operational:

1. First of all, a monitoring agent must be installed on all applications and server instances. Those agents provide us with runtime data, so that we are able to recover the technical aspects of the IT landscape.
2. Second, in order to extend our meta-model with business-related information, we propose a configuration file (cf. Section 6.1.1) that is placed in the root directory of the application's source code repository. This configuration file contains business-relevant information including model assignment in the business context.
3. Third, the CD pipeline is extended with a further quality check within the test stage that validates the content of the configuration file, transfers the content to *MICROLYZE* and cancels the deployment in case the content does not confirm to the provided schema (more details about this process in Section 6.1.5).
4. Fourth, *MICROLYZE* is allowed to access federated information systems through references and our software has the required rights to consume exposed data during deployment. As IT4IT defines, a newly developed IT service leaves a digital twin in each federated information system that is used along the value chain. Farwick et al. (Farwick et al., 2013) elaborated which specific information system provides architecture-relevant information, especially for EA model maintenance. By accessing this information, we are able to recover the whole context in which the application was developed. This concept is detailed in Section 6.1.3.

In the following Sections, we describe the single components of the concept in more detail.

6.1.1. Configuration Files

As state above, all deployed applications integrate a configuration file that is placed in the root directory of the application's source code repository. The file is maintained manually by the development team. The integration of a configuration file serves two purposes:

- **Contextual information:** The configuration file contains contextual business-related information about that application. This encompasses on the one hand a naming and a description of the primarily goal of the particular application, and on the other hand the general allocation in the business context, like product assignment, supported processes or business capabilities.

- **References to federated EA information sources:** Federated information systems store further important architecture-relevant information. For that reason, it is necessary to extract those information and map them with the runtime processes, i.e. deployed applications. However, mapping of different data sources demands indispensably a foreign key relationship. Hence, we propose to establish the mapping on the application or where applicable on the microservice level as the smallest unit. In order to accomplish this goal, the configuration file additionally contains URL-based references to the application's representation in the federated information systems.

The configuration file itself is specified in JSON¹ format. It contains mandatory key-value json-based attributes, but can also be filled with arbitrary additional attributes. We hereby leverage the technology pivio.io² and modified it according to our needs. The mandatory attributes are the following:

- **Name:** The name of the application represents an human-understandable unique definition of the artefact. We recognized during our survey described in (Kleehaus et al., 2019b) that many Enterprise Architects claim that "*same things (apps, servers, tools) have different names in different sub-organizations*". In most cases the name of an application depends on the particular user role that determines the name. For instance, developers recognize applications based on the name given in the development framework configuration file. Administrators or Operators see the name provided by the monitoring tool and the particular departments have their own names as well. Hence, the application name specification in the configuration file could lead to a standardized name assignment.
- **Description:** The *Description* property contains further information about the purpose of the application and which task it is supposed to solve.
- **Business Service:** A *Business Service* is an explicitly defined exposed business behavior and primarily defines the according application from a business perspective.
- **Product:** Archimate defines a *Product* as a *coherent collection of services, accompanied by a set of agreements, which is offered as a whole to customers*. The *Product* itself is mostly defined by the business departments. Hence, the product assignment is very important to understand in which business-specific relation the application was built. In particular when service-oriented architectures are applied.
- **Business Function:** The Archimate definition of a *Business Function* is *A collection of business behavior based on a chosen set of criteria, closely aligned to an organization, but not necessarily explicitly governed by the organization*. In our scenario this attribute represents the business domain in which the application is allocated.
- **Business Capability:** A *Business Capability* represents an ability that an organization, person, or system, possesses. On the one hand, they provide a high-level view of

¹<https://www.json.org>, last accessed: 2020-10-28

²<http://pivio.io>, last accessed: 2020-10-28

the current and desired abilities of an organization, in relation to its strategy and its environment. On the other hand, they are realized by various elements like people, processes, systems, and so on that can be described, designed, and implemented using Enterprise Architecture approaches.

The aforementioned mandatory attributes are at least required to establish the mapping between the application- and the business layer. For this purpose, we extend our meta-model with further AMs. In general, a *Business Capability* is provided by a *Business Service*, which consists of one-to-many *Application Components*. A *Product* is compiled with one or many *Business Services* and one or several *Products* group a *Business Function*. The adapted meta-model is illustrated in Figure 6.2.

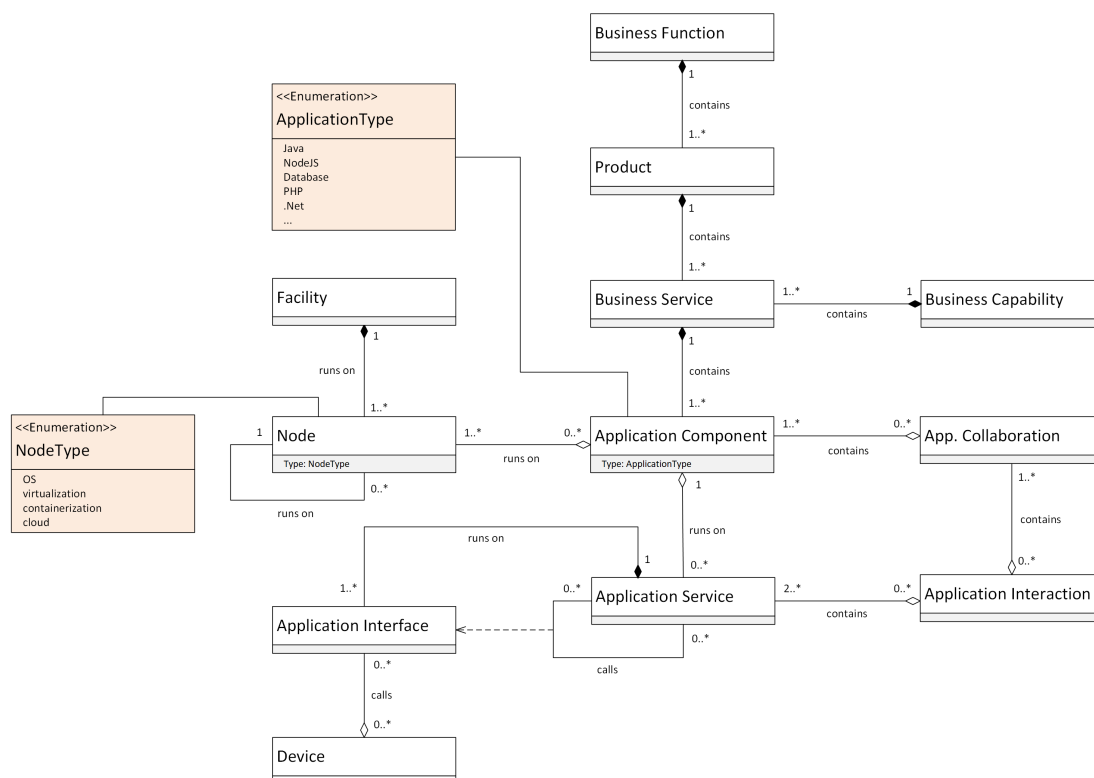


Figure 6.2.: Adapted Meta-Model including AMs from the business layer

6.1.2. General Extension of the Configuration File Content

The listed attributes in the configuration file represent the minimum requirement to achieve a mapping between the application and business layer. In addition, we enhanced the implementation of the processing logic of the configuration file in a way it can be easily extended and incorporate further arbitrary information, either in a key-value or in a key-object format. Those extensions can represent either *Architecture Model* entities

which replaces already existing *Architecture Models*, or general information that are stored as annotations in the database. An example for such an extension is shown in Listing 6.1.

Listing 6.1: Extended configuration file

```
1 {
2   // mandatory attributes
3   "name": "COD Booking Service",
4   "description": "A human understandable description of the service",
5   "business_service": {
6     "name": "COD Booking", //mandatory
7     "ID": "BS-123" //optional
8     "any": "further information" // optional
9   },
10  "product": "...",
11  "business_function": "...",
12  "business_capability": "...",
13
14  // optional attributes
15  "application_collaboration": {
16    "name": "COD Application",
17    "overwrite": true
18  },
19  "product_owner": {
20    "name": "John Who",
21    "contact": "john_who@mail.com"
22  },
23  "used_ports": [5000,5001,5002]
24 }
```

The Listing contains one key-object property that extend the maintained AMs. They are initialized by the definition of the particular AM type. The corresponding object is compiled by one mandatory property, which is an unique "name". However, further optional properties can be added as well, which are stored as assigned annotations. A special feature is the optional property "ID". It does not define the primary key in the *MICROLYZE* database as this key is created automatically. It reflects the entity in the particular federated information system, which is primarily used to create and maintain it. For instance, business-layer related entities like *Business Service*, *Product*, *Business Function* or *Business Capability* are maintained in EAM tools like Iteraplan, PlanningIT or LeanIX. These tools also create unique IDs for these entities. By using the same IDs, we are able to establish a mapping between *MICROLYZE* and the mentioned EAM tools. However, as it is often the case, those entities are not up-to-date or incomplete, so we cannot assume that there will be always a match. For that reason, this property is declared as optional.

A further special feature is the boolean property "overwrite". If it is set to *true*, the previously assigned AM entity is updated with the provided one. A detail description of

this process is presented in Section 6.3.3. For instance, a monolithic application is split into microservices. Hence the *Application Collaboration* represents the related parent application. Many APM tools recognize automatically which microservices belong together logically and suggest a possible name of the parent application. However, this information must be provided beforehand in the monitoring configuration in most cases, as the automatically identified name is mostly not understandable. Overall, with this property, we can make sure that the parent application follows a predefined standard.

The last two attributes "product_owner" and "used_ports" determine further information that is added to the annotation table in a key-value or key-object format. It is possible to extend the annotation table with arbitrary information. As we use a document-based database like MongoDB, objects with several attributes can be easily managed.

6.1.3. References to Federated Information Systems

Farwick et al.(Farwick et al., 2013) conducted a survey in 2013 for elaborating a list of potential tools that provide important architecture-related information that can be used for IT model maintenance. Those tools are also used inter alia by Enterprise Architects to collect information for their EA model maintenance endeavours. Examples are Configuration Management Databases (CMDBs) that provide data about server, database and application instances and their relationships. Project Portfolio Management (PPM) tools cover information about ongoing projects including its start and end date, budget information, as well as the artifacts affected by the projects. Change Management (CM) tools are used in the context of ITIL in order to optimize the processes of implementing changes in the IT landscape. The participants in the survey mentioned change projects, and the statistics of tickets for applications as the data types that could be collected. Furthermore, Granchelli et al.(Granchelli et al., 2017a) demonstrate a tool that leverages code repositories like Github to extract further architecture-relevant information. This data covers information about developers that contributed to the development of an application. Based on our experience with previous projects many companies use Wikipages to describe and report architectural concepts of an application. Those concepts are either written in plain text or with modeling language support like UML.

In general, the life cycle of an application, starting with the architecture planning, through the development and operation to the deletion, leaves a digital twin in many collaboration tools that are required to support each phase of the value chain of the application (The Open Group, 2019). In order to recover the EA models in its full essence, we suggest to document each reference between the running application and the collaboration tool that maintains architecture information from a certain perspective about that application.

Listing 7.1 illustrates an example how those references could be modelled. First of all, the beginning of a reference section is defined with the keyword *references*, which contains an array of key-object pairs. Each array item represents a reference to a known collaboration tool. The reference is described in detail with at least four properties. *Tool* defines the tool itself that is to be accessed. *Domainurl* points to the domain in which the tool is running and accessible. In the current state, we can only extract information that

are exposed via a RestAPI. The corresponding resource is defined in the property *apiurl*. Finally, the *id* points to the exact instance that represent the particular application.

Listing 6.2: Extended configuration file with references

```
1 {
2   // mandatory attributes
3   "name": "...",
4   "description": "...",
5   "business_service": "...",
6   "product": "...",
7   "business_function": "...",
8   "business_capability": "...",
9
10  // references to federated information systems
11  "references": [{
12    "pm": {
13      "tool": "jira",
14      "domainurl": "http://131.159.30.142:5000",
15      "apiurl": "/rest/api/2/component",
16      "id": "10002"
17    },
18    "cmdb": {
19      "tool": "servicenow",
20      "domainurl": "http://131.159.30.300:16001",
21      "apiurl": "/api/now/cmdb/instance/cmdb_ci_apps",
22      "id": "539ff0c0a8016407ec",
23      "apiToken": "fH45c45F"
24    }
25  ]},
26
27  // further optional attributes
28  ...
29 }
```

In order to store references, we add a new class to the storage-specific meta-model illustrated in Figure 6.3. For simplicity, we omit all child classes that inherit from the *ArchitectureModel* class. The new class is called *References* and applies the same schema of the *Annotation* class. In the current version, the references can only be assigned on *Application Component* level.

6.1.4. Importing and Processing of Configuration Files

In order to process the received content of the configuration file, we adapted the components *MICROLYZE.Collect* and *MICROLYZE.Expose*. The updated class diagram for

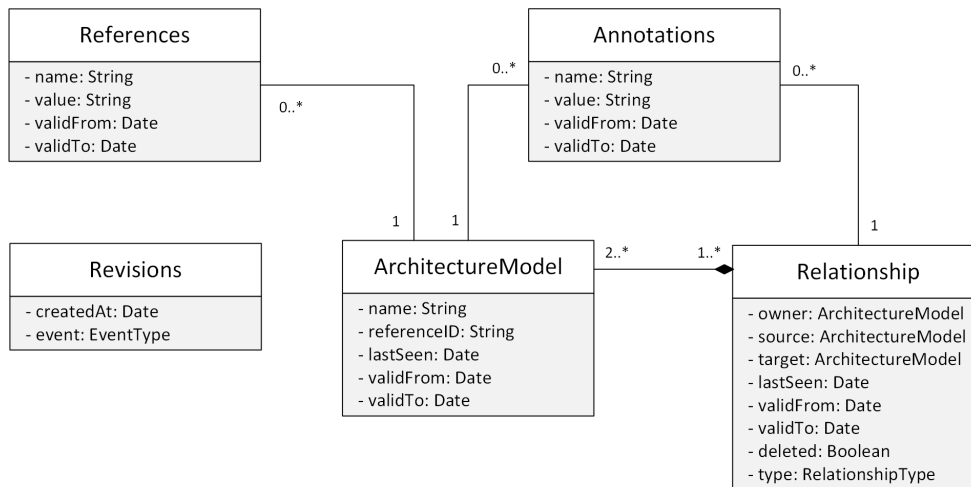


Figure 6.3.: Extended meta-model for storing entities within the *MICROLIZE.Store* component

those two components is depicted in Figure 6.4. New classes are highlighted in yellow. In the following, we describe the adaptations in more detail:

MICROLIZE exposes a REST interface that must be called via a POST request to process the content of the configuration file. The related class is named *ImportConfiguration*. The class contains four methods that are responsible to create AMs, relationships between AMs, references to federated information systems and annotations. The creation of relationships executed in the method *createArchitectureModelRelationship()* is performed iteratively in a hierarchical order. That means, from the lowest level represented by *Application Components* up to *Business Functions*. A detailed description of this procedure is provided in Section 6.3.2.

In addition, we create for each provided reference a provider class that inherits from the *UnifiedProvider* interface. Those providers contain methods that retrieve the reference-specific meta-model exposed by REST APIs. In order to establish the required connection, we create the corresponding *RestClients*. Any information of the regarded application that reside in the referenced tool is accessed via the provided *id* and *apiUrl* maintained in the configuration file. That allows to exactly address the instance in the respective source system independent of any changes made to it such as renaming.

MICROLIZE keeps the data it manages down to a minimum to avoid any synchronization or update conflicts that arise with managing data that is owned by a different system. While it might be considerable to cache some types of data to enable more efficient querying of the enterprise graph, the maintaining of own copies of objects should be avoided, as performed changes need to be collected too frequently which would stress the systems unnecessarily. *MICROLIZE* realizes these constraints by only saving the references to the application instances. With the support of the GraphQL framework, we are able to realize adhoc queries to the federated information systems, retrieving only up-to-date data.

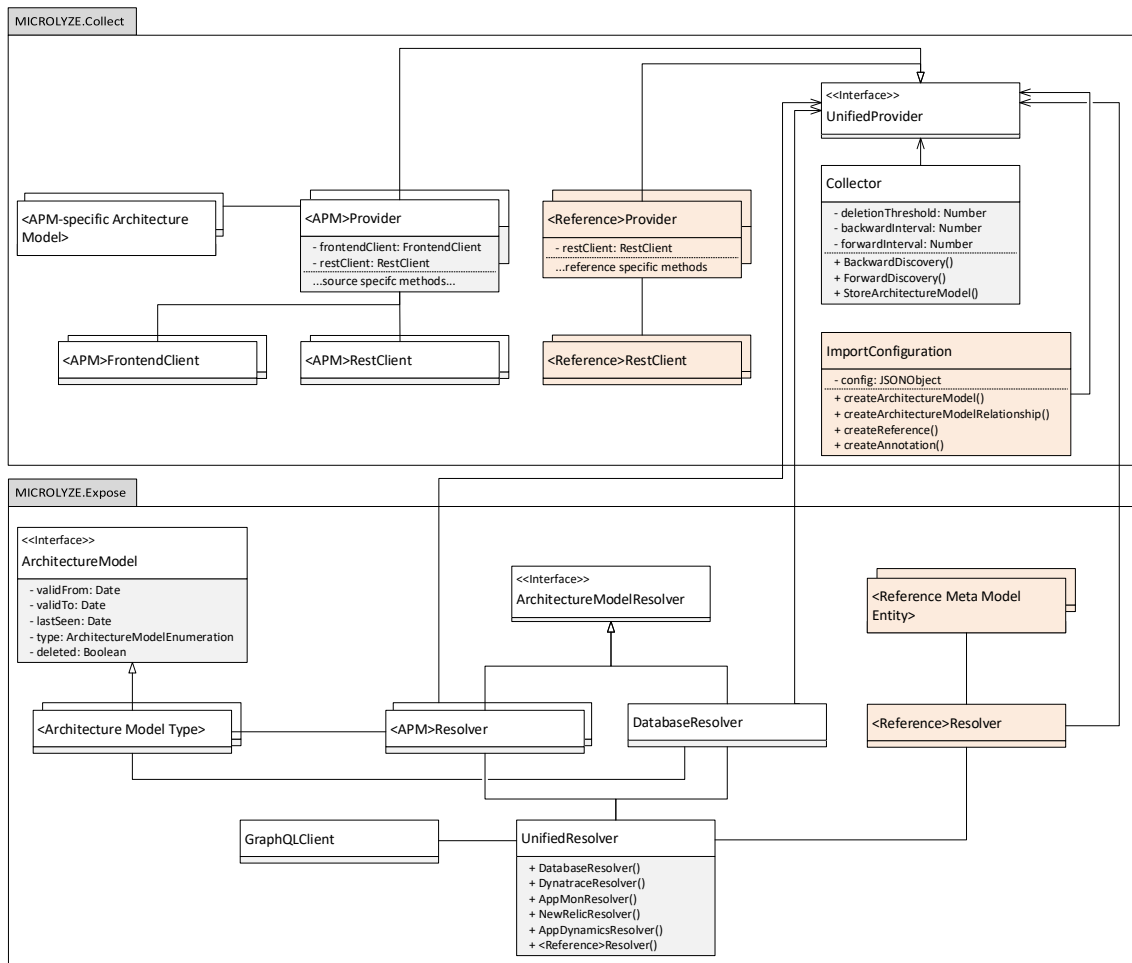


Figure 6.4.: *MICROLYZE.Expose* and *MICROLYZE.Collect* component extension for supporting model import from federated information systems

For this purpose, we created new *Resolvers* for each referenced information system that abstracts the meta-model exposed by the particular system. The particular entities of the meta-model is maintained in the classes *<Reference Meta Model Entity>*. For example, in order to retrieve all projects that involve the regarded application and are currently ongoing, we access the project management tool JIRA. Applications are mostly modelled as *Components* and assigned to one or more *Projects*. *Components* can be referenced in *Stories* or *Epics*. The corresponding API for retrieving information about *Components* including assigned *Projects* is: *<url>/rest/api/2/component/<componentID>*. Afterwards, we can retrieve more information about a specific project via *<url>/rest/api/2/projects/<component.projectID>*

All those aforementioned classes build the foundation for querying application-related information that reside in the collaboration tools. In order to support more tools, those classes need to be duplicated and adapted accordingly.

6.1.5. Continuous Delivery Pipeline

Since the configuration file must be maintained manually, we must ensure that the content is validated in order to guarantee a high data quality. For this purpose, we integrated a separate test case into the CD pipeline that checks whether 1) the configuration file is available in the defined application path, 2) all mandatory attributes are available, 3) the references point to an available and accessible endpoint in the particular federated information system, and 4) the content of the configuration file follows a predefined schema. For this purpose, we validate the content against a *JSONSchema* file. In the following, we describe how we developed the additional test case based on the open-source CD pipeline Jenkins³. However, any other CD pipeline can be used for this purpose. A detailed description can be found in (Achhammer, 2019).

In order to adopt the pipeline, we create a *SharedLibrary*⁴ that is useful to share parts of pipelines between various projects to reduce redundancies. The shared library can be defined in external source control repositories and imported into existing pipelines. A *SharedLibrary* is defined with a name, a source code retrieval method, and optionally a default version. The most important benefits of using a library in the context of the suggested solutions are as follows:

- **Central management and maintenance:** A library can be centrally managed by the responsible department. They can maintain the source code, extend its functionality and provide different versions to cope with different technologies.
- **Ease of integration:** Libraries are easy to import and to integrate into pipelines. The centrally managed code logic is loaded into the CD server instance and executed decentrally. The code requires only the necessary parameters for executing methods.
- **Reuse of existing credentials:** As the exposed methods of the library are loaded into the pipeline, it allows to reuse of the credentials that are needed to run the

³<https://jenkins.io>, last accessed: 2020-10-28

⁴<https://jenkins.io/doc/book/pipeline/shared-libraries>, last accessed: 2020-10-28

deployment. Credentials are required in particular for accessing the referenced information systems.

- **Reduced roll-out and running costs:** The aforementioned benefits reduce the roll-out costs significantly. In case of updates, the development teams are not required to integrate the logic anew. The only exception is when the method's signature is changed.

The additional *Model Exposure* stage, we integrate into the pipeline, is responsible to validate the configuration file against a predefined *JSONSchema* and export the content to *MICROLYZE*. How the library is used in detail is presented in Listing 6.3. First of all, the procedure of the new stage must be added into the pipeline script before the final deployment stage starts (line 14–37). The reason for this order is that a failed test case should block the final deployment of the application in order to ensure the configuration file is validated and follows the predefined schema. In the *Microlyze documentation process* stage, we first extract the *microlyze.json* configuration file and the *JSONSchema* file and translate it to a *JSONObject* (line 19–21). The paths must be given as a parameter. The *JSONSchema* file should be stored in a shared folder on which every CD pipeline has access to it. In the lines 24–26, we start the validation process by executing the method *microlyze.validate()*. In case the validation is successful and the provided attributes follow the predefined schema, we continue with the model exposure process by executing the method *microlyze.expose()* (line 31). Basically, this method forwards the content of the configuration file to *MICROLYZE*. *MICROLYZE* consumes the content and creates/updates all corresponding AMs including assigned relationships. Afterwards, *MICROLYZE* returns the IDs of the affected entities. Those entities are stored in the variable *updatedAM*. This variable represents an array with key-value pairs, whereas the keys are not unique but reference to the main table *Revision*, *ArchitectureModel*, *Relationship* and *Annotation*.

After finishing the exposure process, the final release is executed (line 40). However, a failed final deployment task would lead to an inconsistent model documentation, as the new or updated application is not yet in production but already available in *MICROLYZE*. In order to prevent this situation, we suggest to add a post section. The post section defines one or more additional steps that are run upon the completion of the stages. In the post section, we define conditions that allow the execution of steps depending on the completion status of the pipeline. In case any stage is failed and the *updatedAM* parameter contains a list of IDs, then the rollback method *microlyze.rollback()* (line 48) is triggered which removes all created records and corresponding relationships in the database. This process is necessary as otherwise we experience an inconsistency between the documented EA models and the real as-is IT landscape.

Besides assuring the availability of the configuration file, the validation process executed in *microlyze.validate()* encompasses the following checks:

- *JSONSchema* validation: This check verifies whether all provided key-value properties follow the predefined schema. For implementing the validation logic, we take advantage of the open-source Mozilla Jenkins pipeline library⁵ that integrates all

⁵<https://github.com/mozilla/fxtest-jenkins-pipeline>, last accessed: 2020-10-28

methods written in groovy required for a full *JSONSchema* validation.

- Null pointer references: We allow exclusively read-only access. Hence, this check pings all exposed REST APIs in the referenced information system and fails if the HTTP respond status code is not 200, which determines a successful HTTP request. In a GET request, the response will contain the entity corresponding to the requested resource. Returned empty resources will also lead to a failed check.
- Repository matching: Our concept is powerful in recognizing schematic errors, but not in the verification of the delivered information. That means, spelling mistakes could lead to the creation of completely new AM entities although the true entity already exists in the database. In order to restrict this scope of error, we suggest to connect EA repositories to the validation process to verify the regarded business-related entity is found in the EAM tool. However, this assumes that the entity is created beforehand, which might not always be the case. Therefore, this check is regarded as non-critical which will not lead to a failed deployment. In any case, the result is reported to the user for possible manual adoption.

Listing 6.3: Example pipeline script

```

1 // import the microlyze-library into the pipeline script
2 \@Library(['microlyze-jenkins-library@master'])
3
4 // define return value for updated Architecture Model
5 parameters {
6     string(name: 'updatedAM', defaultValue: '')
7     boolean(name: 'validConfig', defaultValue: false)
8 }
9
10 // original pipeline stages (Checkout, Build, Test)
11 [...]
12
13 // pipeline stage to be added
14 stage('Model Exposure') {
15     steps {
16         script {
17
18             // read microlyze.json to a JSONObject
19             def microlyzeConfig = readJSON file: "${PATH_TO_CONFIG.JSON}"
20             // read microlyzeSchema.json to a JSONObject
21             def microlyzeSchema = readJSON file: "${PATH_TO_SCHEMA.JSON}"
22
23             // validate configuration file against JSON schema
24             env.validConfig = microlyze.validate(
25                 config: microlyzeConfig, schema: microlyzeSchema

```

```
26         )
27
28         // if config file is valid execute final documentation process.
29         if (env.validConfig) {
30             // the IDs of the updated AM are returned
31             env.updatedAM = microlyze.expose(config: microlyzeConfig, name
32                 : NULL)
33         }else {
34             sh 'exit 1'
35         }
36     }
37 }
38
39 // start final deployment stage
40 stage('Release') {
41     // execute deployment steps
42     [...]
43 }
44
45 // in case deployment fails, rollback affected Architecture Models
46 post {
47     failure {
48         microlyze.rollback(affected_IDs: env.updatedAM)
49     }
50 }
```

6.1.6. JSON Schema validation

*JSONSchema*⁶ is a specification for defining the structure of JSON-based data. It represents basically a contract that specifies the required JSON data-definition format, and how to interact with that data-object structure. *JSONSchema* is intended to define complete structural validation for JSON data and describes existing data format in a clear, human- and machine-readable documentation. In order to validate JSON data, the schema specification defines a vocabulary that can be used to describe the meaning of JSON documents, provide hints for user interfaces working with JSON data, and to make assertions about what a valid document must look like. A complete terminology can be found in <https://datatracker.ietf.org/doc/draft-handrews-json-schema-validation/>

The JSON Schema terminology asserts constraints on the structure of JSON data instances. If the instance satisfies all asserted constraints, then the instance is said to be valid against the schema. In the following, we describe the most important keywords:

⁶<https://json-schema.org>, last accessed: 2020-10-28

- **required:** This keyword specifies the minimum required JSON keys that must be available in the JSON data.
- **properties:** Properties are key-value pairs that define the format of each key and value of a JSON object. Properties contain inter alia the id of the JSON key, the value data type, corresponding format and the pattern validated against.
- **type:** The keyword type determines the data type that a key is allowed to accept as a value. Examples are *object*, *string*, *number*, *boolean*, etc.
- **pattern:** The pattern keyword uses regular expressions to express value constraints.
- **format:** The format keyword allows for basic semantic validation on certain kinds of string values that are commonly used. This allows values to be constrained beyond what regular expressions can do. Examples are *email*, *hostname*, *ipv4/6*, *uri*, *date*, *time*, etc.

In Listing A.1 provided in Appendix Section A.1, we determine the required JSON Schema for validating the JSON example provided in Listing 7.1.

6.1.7. Distribution and Location of JSON Schema Files

The distribution of the *JSONSchema* files follows a hierarchical regulation as depicted in Figure 6.5. At the highest level the Enterprise Architects create the master of the *JSONSchema* file which represents the basis from which all further schema files derive and are distributed to the lower levels including domain-, product-, and application level. That means, the next lower levels must accept the specifications provided by the upper levels and are only allowed to add more, but never remove specifications. As the figure illustrate, the Enterprise Architects create the initial file and communicate it to the Domain Architects. They might extend the schema with further specifications that need to be fulfilled by the configuration files. Thereafter, the Domain Architects report the extended schema to the Product Owners which are finally responsible to ensure the configuration file is created, assigned to the particular application and fulfills the defined schema specifications. Also on this level, the Product Owner are allowed to extend the schema according to their needs.

After all, the *JSONSchema* files are stored in a shared location with access control. The Developer Teams create the configuration files that correspond to the particular JSON schema that is applied for their application domain. The Developers are only able to provide the demanded information if there is an active exchange between all involved stakeholders, which entails an important sensitization of EA-relevant concerns. In many enterprises, those concerns are often not accessible or comprehensible, especially on development level due to missing stakeholder collaboration (Armour et al., 1999; Henttonen et al., 2009; Lam, 2004; Meilich, 2006; Shah et al., 2007; Templeton et al., 2006). Finally the CD pipeline retrieves both file types and validates the content of the configuration files against the predefined schema.

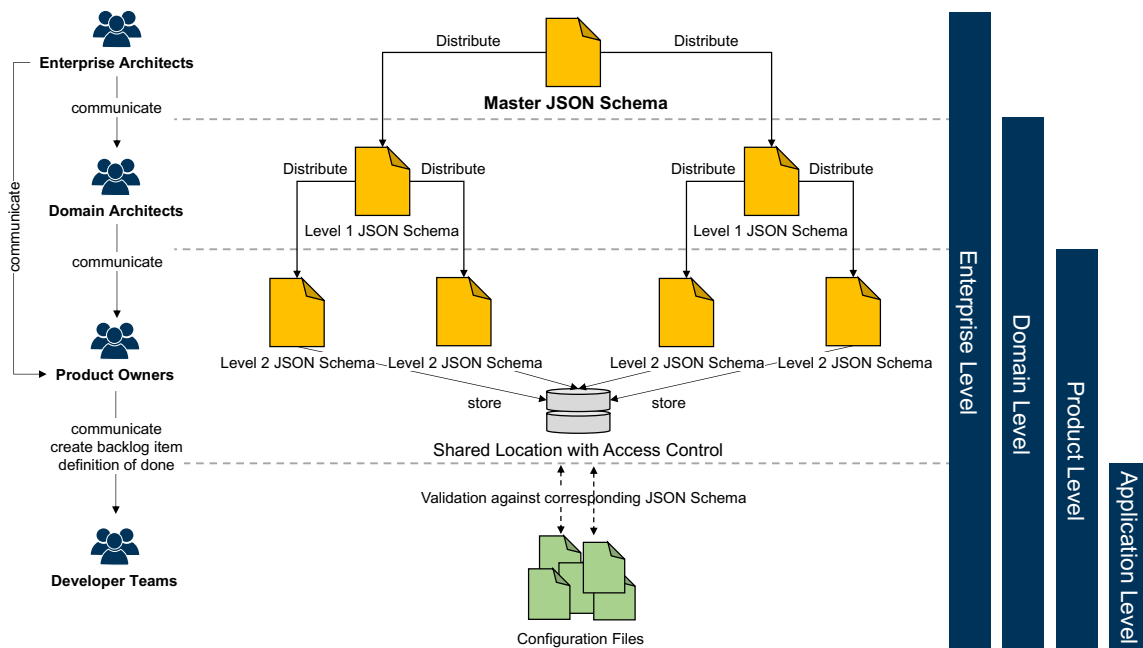


Figure 6.5.: JSON Schema location and distribution overview

6.2. Organizational Design

Microlyze maintains the AMs of the current IT landscape that is supplied with different information systems, above all the APM system and the maintained references in the configuration file. With the extraction of architecture-relevant information from different sources, we follow a decentralized setup originally proposed by Fischer et al. (Fischer et al., 2007) who stated that each stakeholder in an enterprise (e.g. Enterprise Architect, Systems Operation, Project Manager, etc.) needs their own set of tools which they are familiar with. However, every solution concept or software is only successful if the involved stakeholders who are required to bring this solution in operation 1) accept the additional effort and 2) can easily integrate it into their daily work. In the following Sections, we outline which stakeholders need to be involved in the proposed concept. Thereafter, we detail how the concept can be integrated into a common agile development process and which artifacts in SCRUM need to be adapted accordingly.

6.2.1. Roles and Responsibilities

The determined roles represent an integrated part of the solution concept. They need to interact with each other and assume certain tasks to achieve the overall goal of automated EA model maintenance. The roles are defined as follows:

Enterprise Architects: The Enterprise Architect is responsible for aligning the business mission, strategy, and processes to its IT landscape. In terms of EA documentation Enterprise Architects are responsible for defining central guidelines and rules for modeling.

The department centrally defines and models the most central business layer elements including Business Domains, Business Functions, Business Capabilities, Products, etc. in close collaboration and alignment with business departments. Hence, the EA department has the overall accountability for the EA repository, its operation and maintainability.

Domain Architects: A Domain Architect is a specialist with deep knowledge within a particular domain of their expertise. In the course of documentation, they cover most of manual modeling efforts and ensure that the business and application layer are reasonably documented and interconnected by the most important relationships. This information is mainly obtained by the means of meetings and intensive collaboration with Developer teams.

Product Owners: The Product Owner role was introduced during the creation of the agile development process framework SCRUM (Schwaber, 2004). The Product Owner is a member of the Agile Team responsible for defining user stories and prioritizing the Backlog items. The Product Owner has a significant role in quality control and is the only team member empowered to accept stories as done. Product Owners take the responsibility for the documentation of the application they own including the relationships to other applications, provided and consumed interfaces, transferred business objects and the related infrastructure platforms they are operated on.

Development Team: The Development Team carries out all tasks required to build increments of an application. While Domain Architects are specialists in one facet of their domain, and Enterprise Architects keep an eye on the bigger picture, Developers focus on one solution at a time, and have a deep focus on technical details. In the course of documentation, the Development Team is mainly responsible to document the interplay of created software components, as well as to comment the software code in detail.

In general, those roles can be clustered into three user groups. 1) Enterprise- and Domain Architects represent the **Enabler** group. They are responsible to define which AMs and relationships need to be recovered and documented in an automated manner. Not all models and relationships are equally important or change so frequently, that is impossible to document them manually. As we will see in Section 7, it depends on the company's assessment which information must be included in the automated documentation process or can be omitted. Overall, the required information pool is modelled in the *JSONSchema* format and reported to the development team. 2) The **Worker** group has the task to fulfill the defined requirements. The Product Owner as part of the Worker group ensures that the configuration file follows the predefined schema, all information is provided and adapts the definition of done (DoD) for the SCRUM development process. The Developer team creates the corresponding JSON-based configuration file for each application and integrates the additional validation stage into the CD pipeline. Furthermore, the Developers install the monitoring agent on each applications. 3) Finally, we define the **Beneficiary** group that benefits most from the solution. As the management of EA models and the automated recovery of such models is important and useful for every user role that takes part of the IT service value chain, we assign every aforementioned role into this group. The Enterprise- and Domain Architects might be mainly interested into the business-related AMs and the business- and application layer assignment for EA model maintenance.

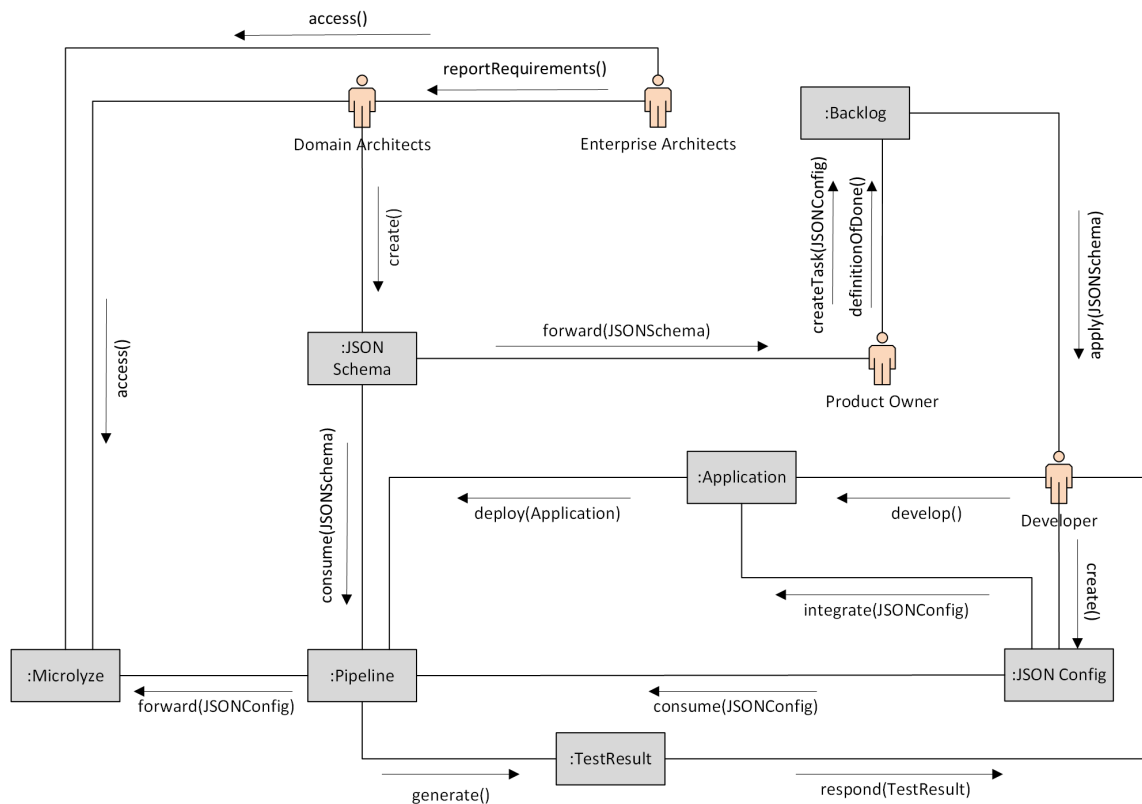


Figure 6.6.: Interplay of the different roles and components involved in the overall solution concept.

The Product Owner and Development Team requires a more granular insight into the application communication and deployment relationship for performance assessment and troubleshooting. However, we regard the Development Team to take most of the effort to make the solution concept runnable.

6.2.2. Adapted Agile Development Process

The interplay of all those roles and components and how they are allocated into the overall solution concept is illustrated in the communication diagram shown in Figure 6.6. Once the preconditions are established, the IT landscape documentation becomes a natural part of the agile development process, in which the particular AM is updated at least with each product increment release.

First of all, the Enterprise Architects report their requirements to all Domain Architects which AMs and relationships are incomplete or outdated in their EA repository and need to be included in the automated documentation process. The Domain Architects translate the requirements into the *JSONSchema* format. It is important that Domain Architects decide which information system contain important architecture-relevant information and should be referenced by the configuration file. Of course, Domain Architects can extend

the requirements to their needs, resulting in a more comprehensive schema.

Afterwards, the *JSONSchema* is forwarded to the Product Owners of each Agile Team. They define a new *UserStory* that contains the requirements for the solution's integration. For further facilitation and reduction of comprehension problems, the Product Owners must stay in active exchange with Domain- and Enterprise Architects. This improves the communication between those roles which used to be rather low in most companies. The *UserStory* is added to the *ProductBacklog* and at some point selected for execution as part of a *SprintBacklog*. Depending on how fast the roll-out should proceed, the backlog items priority may vary according to the organization's needs.

During the *Sprint*, the Development team create the JSON configuration files and fills them with support of the Product Owner. Once the files are assigned to the applications, they finally commit the adapted source code repository. In parallel, they integrate the additional stage for validating the configuration file into the CD pipelines. This completes the *UserStory*. The provision of the models itself is automatically performed with each run of the deployment pipeline. There is no further need for manual intervention. The CD pipeline finally reports the test results back to the Development team. If the Developers do not adhere to the defined schema, the application cannot be deployed to production.

Once the solution is integrated, it has to be ensured that the information contained in the configuration files is kept up-to-date. In order to achieve an up-to-date model management, we suggest to redefine the *Definition of "Done"* (DoD) for a product increment. The DoD is a Scrum artifact that ensures a common understanding of what "done" means and what criteria have to be fulfilled (Schwaber, 2004). By including this aspect in the *SprintReview* it is possible to ensure that the information is maintained before any product increment release.

6.3. Process Design

In the previous Sections, we outline the system and organizational design of our solution approach. In the following Sections, we focus on technical processes, we elaborated on to import the content of the configuration file to *MICROLYZE*. We publish those processes in (Achhammer, 2019; Kleehaus et al., 2019a, 2021). First, we detail the performed sequences in the CD pipeline which are necessary to validate the content of the configuration file and to transfer the content to *MICROLYZE*. Next, we elaborate the steps that are carried out to process the content and on which challenges we have to pay attention.

6.3.1. Performed Sequences in CD Pipeline

Figure 6.7 depicts the overall sequence diagram of the model provision process. Grey activities express the normal continuous deployment steps that are common in most pipeline configurations. The activities marked in yellow indicate new steps that must be introduced into the continuous deployment. In general, the process can be broken down into three sequences, which we detail in the following:

Sequence 1: Extraction and Build: The deployment process starts as soon as the

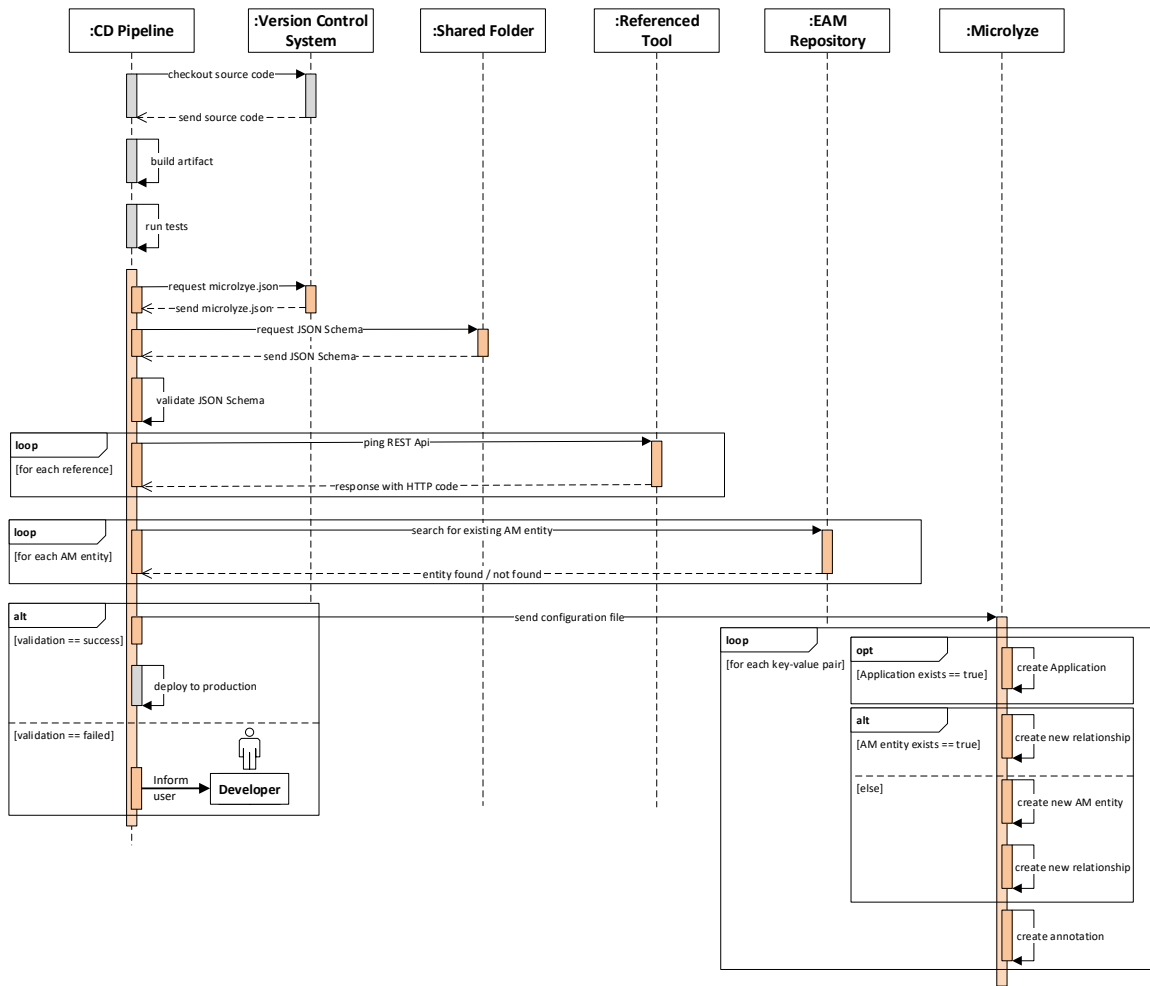


Figure 6.7.: CD pipeline validation process

Developer uploads the modified source code to the Version Control System (VCS). This event triggers the automation of the CD pipeline. Initially the regular pipeline stages are executed, which encompasses:

- **1a)** The checkout step loads the artifact's source code from the VCS system into the CD server's working directory
- **1b)** The CD server builds the source code into an executable artifact
- **1c)** Functional tests such as unit tests or integration tests are performed to ensure the built artifact works correctly

Sequence 2: Validation: If all previous steps performed successfully, the validation sequence continuous before the application is finally deployed to the production environment. In case one of the following activities are not successful the validation process end with an error report and the Developer are notified accordingly. The validation sequence consists of the following steps:

- **2a)** The first stage verifies the existence of the JSON-based configuration file *microlyze.json* in the specified path. It is retrieved from the VCS. Furthermore, this step obtains the application's technical name, i.e. the name that is assigned to the artifact, by analyzing the configuration file of the development framework which is either provided in *POM.xml*, *application.properties*, or *package.json*, etc. files, depending on the used framework. The technical name is required as an ID to establish the mapping between the runtime data and the static data provided by the configuration file. If the stage is not able to retrieve this information automatically, the artifact's name can also be provided as a parameter in the *microlyze.validate()* method.
- **2b)** Next, the JSON Schema file is retrieved from a shared folder. We suggest to store the schema file in a location at which all developer teams in the specific domain has access to it.
- **2c)** In this step, the pipeline validates the schematic correctness of the configuration file with support of the *JSONSchema*.
- **2d)** Thereafter, we loop through each provided reference in the configuration file and ping the addressed information systems. The step can only be regarded as successful if the HTTP respond code is 200 and the returned resource is not empty.
- **2e)** Finally, we validate all delivered AM information by searching the entity in the connected EA repository. However, this is the only step, which will not lead to a failed stage, since non-existent entities only means an incomplete documentation.

Sequence 3: Model Creation: In the final sequence, the actual model creation process starts by sending the content of the configuration file in JSON format to *MICROLYZE*. The steps of the upcoming sequence is performed in the *MICROLYZE* backend and can be grouped into three main activities. A detail description of the following activities is provided in Section 6.3.2:

- **3a)** In the first step, *MICROLYZE* consumes the content of the configuration file and iterates through all key-value pairs. Applications that are not in the database yet are inserted as a new entity. In case the application is already known the *lastSeen* timestamp is updated.
- **3b)** In the next step, all provided AM entities are processed. Unknown entities are newly added to the database and already available entities get an updated *lastSeen* timestamp. Furthermore, the relationship table is updated as well.
- **3c)** In the last step, all remaining key-value pairs are inserted as annotations or as references to the *Annotation* and *Reference* table, respectively. Also in this step a comparison with existing entities is carried out in order not to transfer duplicates into the database. However, with support of the *overwrite* parameter, the existing value of a provided property can be overwritten.

In case all activities of the above sequences perform successfully (with exception of activity 2e), the application is deployed to the production environment and *MICROLYZE* starts with processing of the received runtime data. It is important to highlight that the mapping between the information delivered by the configuration file and delivered by runtime data is achieved based on the technical name of the application extracted from the development configuration file, i.e. the technical name represents the primary key. If this name differs in the monitoring tool, then it must be adapted manually.

6.3.2. Processing the content of the configuration file

The detailed workflow for processing all information delivered by the configuration file is depicted in Figure 6.8. First of all, the *MICROLYZE.Collect* component receives the information and stores it temporarily in the cache. Thereafter, the content is processed chronologically in three steps:

- **Step 1:** The first step covers the mandatory keys *Name* and *Description*. If there already exists a corresponding *Application Component* for the delivered application in the database, the *lastSeen* attribute is updated to the current timestamp. If not, *MICROLYZE* creates a new *Application Component* entity.
- **Step 2:** The second step is an iterative process and focuses on provided AMs including the mandatory keys *BusinessService*, *BusinessCapability*, *Product* and *BusinessFunction*. In each iteration the next AM is selected and processed as follows: First, the system checks whether the stored *Application Component* is new, i.e. the *lastSeen* timestamp equals the *createdAt* timestamp. If it is not new, the meta-model is updated based on a decision tree detailed in Section 6.3.3. Otherwise, a further check is performed to determine whether the selected AM entity have been already created in *MICROLYZE*. If it has been, the *lastSeen* timestamp for the AM is updated, as well as new relationships are established, most notable between the *Application Component* and the *Business Service*. In case the AM entity is new, then it is

inserted as the appropriate type before the relationship table is updated accordingly. Afterwards, the procedure begins over with the next provided AM.

- Step 3: In the third step, we store all remaining keys in the *Annotations* table and the references to the federated information systems in the *Reference* table. Same as Step 2, this process is performed iteratively. We proceed as follows: First, we select a remaining key and store it if the *Application Component* is new. Otherwise, the property *overwrite* indicates whether the value of the selected key should be overwritten. In case *overwrite = true*, a new revision of the *Application Component* is created before the key-value pair is stored. The same process is also applied with references, with the exception that references are stored in the *Reference* table.

6.3.3. Meta-model update based on decision tree

In the previous Section, we described the procedure, how provided AMs are processed for new applications. In the following Section, we elaborate on how the process must be extended for updating existing applications. The workflow is based on a decision tree (Quinlan, 1987) and illustrated in Figure 6.9.

Decision tree is a tree-like model of decisions that help to identify which paths or decisions lead to specific goals. In our scenario decisions are represented by conditions and goals are sequences of activities. We separate between three conditions: the root condition validates the existence of the particular AM entity in *MICROLYZE*. The second condition checks whether there is already an available parent-child relationship from the provided AM type that lead back hierarchically to the *Application Component*. The third condition checks whether the *overwrite* property is set to true.

For example, let us consider the following scenario. A defined AM entity in the configuration file is not yet available in *MICROLYZE*. However, there already exists a relationship to another entity of the same AM type. Hence, in case no *overwrite* property was set, then the user receives an error respond with the parameterized notification "A relationship to a <Architecture Model> with the name <name> was already defined." Otherwise, the following sequence is performed. First, the AM entity is created in *MICROLYZE*. Thereafter, new revisions for all AM entities are created that are subordinate in hierarchical terms. Finally, new relationships are created for all concerned AM entities.

6.4. Visualization Design

In the previous Sections, we detailed the system design how to enhance the EA model management with further business-related information via support of configuration files. This enhancement bridges the gap of missing or incomplete documentation of the dependencies between application- and business layers. It discloses where applications are located in the business domain. In the following Sections, we describe approaches how this additional application- and business layer relationship information can be represented visually. For this purpose, we leverage clustermaps and table views. The visualizations

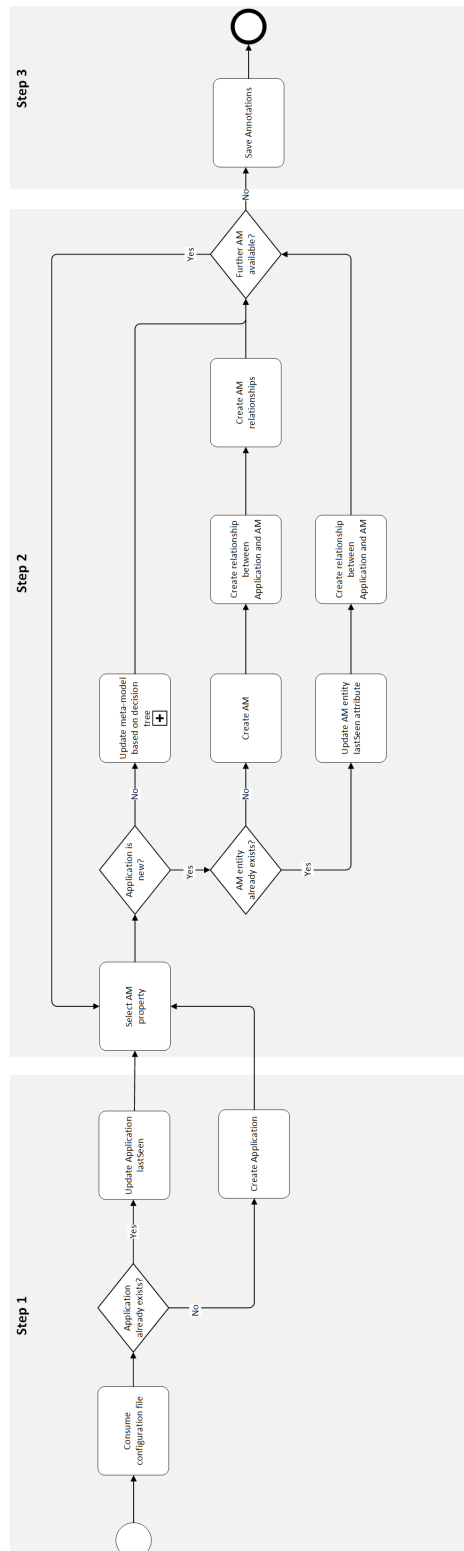
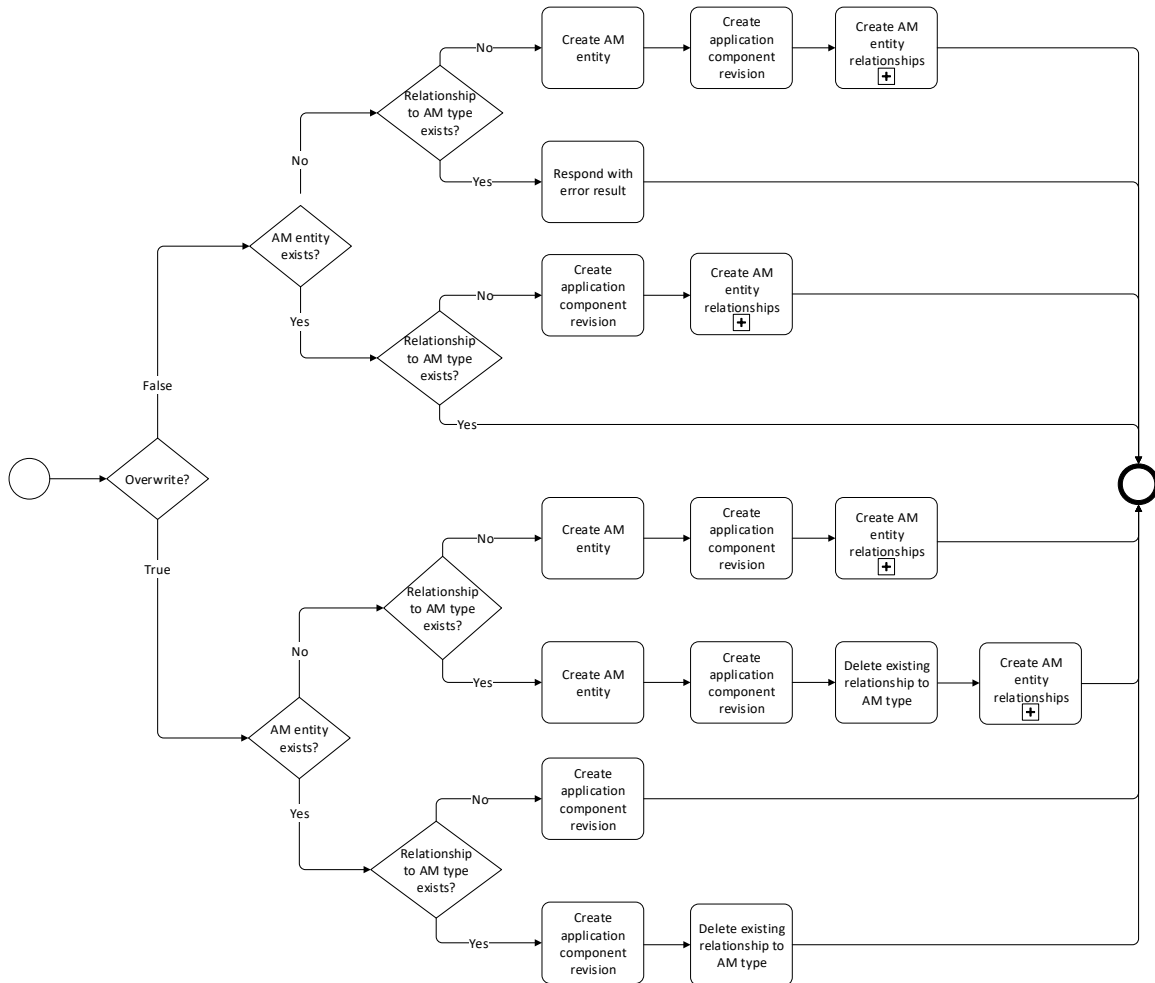


Figure 6.8.: Workflow for processing information delivered by the configuration file

Figure 6.9.: Decision tree for *ArchitectureModel* updates

are based on the same architecture detailed in Section 5.4.1. Again, we use the Archimate taxonomy to visualize AMs and their dependencies and address the research conducted by Wittenburg (Wittenburg, 2007).

6.4.1. Architecture Model Cluster

In general, a cluster map combines elements (e.g. information systems, databases, products, etc.) into logical domains. The domains are derived from functional areas, organizational units or locations. The visualized elements can be nested to adapt the information granularity. The representation of relationships between elements are facilitated by hierarchically overlapping layers. That means, the schematic relation of the elements is represented by locating the elements in the logical border of the parent element to which it belongs to. A further way to distinguish between the various logical elements is the use of a colour code. A cluster map does not specify how the domains are placed and how the different elements are arranged within a domain. A space-optimizing positioning on the map tries to create a map with minimal size. Thus, when such a card is created, rules are already established. The drawback of this visualization are duplicated elements. There is the possibility that an element which belongs to more than one logical parent elements appears several times on the map.

Goal: The *Architecture Model Cluster* visualization illustrates a management overview of the hierarchical dependencies of all AMs in an organization. It describes the dependencies of all *Business Functions*, *Products* and *Subproducts*, as well as *Business Services* and which *Application Components* are assigned to the aforementioned AMs. It mainly represents the business layer of a company and can be easily extended with further information like the application type, *Nodes* or *Facilities* as well as *Business Capabilities* in case those information are relevant for the particular use case.

Visualization: Figure 6.10 presents the *Architecture Model Cluster* view in detail. In this example, the topmost parent folder node represents *Business Functions* and the following child nodes represents those AMs that are subordinated in a hierarchical perspective. Since *Business Services* are manually defined by the business departments and *Application Components* are automatically recovered using runtime data, the subsequent mapping of those layers must be achieved via the configuration file.

The positioning of the nodes and group nodes are determined automatically via an algorithm provided by the yFiles library. The positioning cannot be changed manually. With this approach, we achieve a high recognition value by the users. The top-down and left-right ordering of the nodes is performed randomly and does not indicate any prioritization.

Not every AM is assigned to a parent node within the same hierarchical level. If this is the case, *MICROLYZE* skips automatically missing layers and establish the connection to the next available parent node. However, if the relationship information to the topmost parent node, i.e. the *Business Function* is not available, a dummy *Business Function* is created and named as "unknown".

Interaction: The menu component of the *Architecture Model Cluster* view contains three input fields. The datepicker shows the selected architecture timestamp. By selecting another date, the visualization process described in Section 5.4 is restarted with an adjusted timestamp attribute in the GraphQL statement. Future dates are deactivated and cannot be chosen. The same applies for dates that are older than the last known architecture timestamp. The "select level" dropdown field is a multi-selection. The values are fetched from the database and represent basically the hierarchy structure maintained in the configuration file, plus the *Architecture Component* recovered using the runtime data. By changing the values selected layers are shown or hide in the graph, in order to reduce the granularity of information. Performing a search in the search bar lead to a highlighting of the nodes which names are within the research result.


Apart from using the menu component, a direct interaction with the graph is also possible. Group nodes can be expanded or collapsed by clicking on the  button. By using the mouse wheel the graph can be zoomed in or zoomed out. A sidebar is opened immediately after clicking on any element node. This sidebar is showing further static and dynamic information about the selected node.



Figure 6.10.: Visualizing hierarchical dependencies of *Architecture Models* via a domain cluster representation. Grouped AMs can be opened or collapsed.

6.4.2. Architecture Model Table View

Although graph-based and cluster-based visualizations are frequently used in EA management, both visualizations face scalability issues with large amounts of data. Filter, grouping and search functionality can help in the beginning, but they also reach their limits at some point. For that reason, we propose a table-based visualization of dependency information. Hereby, relationships are represented by using tabs and filter combinations.

Goal: The *Architecture Model Table* view list all AM entities in a table structure. With this approach a large amount of data can be easily managed and saves a lot of space. Furthermore, the readability of large data sets is also supported by tables by using pagination.

Visualization: Each particular AM is shown in horizontally arranged tabs. The sequence is based on the hierarchical order of the AMs. Each tab content contains a table which lists all entities and attributes of the particular AM. Below the tables, a pagination supports how many records are shown at once.

Interaction: In order to constitute the dependency between AMs one can select any row in the particular table and a filter is set on this entity. This filter is also applied to any following table, that means, only those records are shown that are related to the filter selection. It is also possible to set more than one filter. The filter itself is indicated by a symbol shown on the tab.

Each attribute in the particular table can be ordered in ascending or descending order. The ordering is also applied when a filter is set. The ordering is shown via an arrow symbol on the attribute header.

The menu component of the *Architecture Model Table* view contains two input fields. The search component consists of a dropdown field that lists all AMs and a search field. By selecting a value in the dropdown field the search is only applied on the selection. In the current state only full text search is supported. The datepicker indicates the shown architecture revision and can be changed by selecting another date.

6. Recovery of Business-related Models

Domains	Name	Products	Subproducts	Business services	App collaborations	App components	Nodes	Facilities	Department
	Default: Application		10.9.2019 - 02:24:41	Hans Peter	Hans Peter	NGTP Sec I	Team	AI Connected Services	AI Connected Services
	Monitoring		10.9.2019 - 02:24:41	Hans Peter	Hans Peter	B2V Operation		AI Connected Services	AI Connected Services
	B2V COM		10.9.2019 - 02:24:41	Hans Peter	Hans Peter	B2V Operation		AI Connected Services	AI Connected Services
	TSSB		10.9.2019 - 02:24:41	Hans Peter	Hans Peter	NGTP Sec I		AI Connected Services	AI Connected Services
	B2V SEC		10.9.2019 - 02:24:41	Hans Peter	Hans Peter	B2V Operation		AI Connected Services	AI Connected Services
	NGTP		10.9.2019 - 02:24:41	Hans Peter	Hans Peter	NGTP Sec I		AI Connected Services	AI Connected Services
	CCC		10.9.2019 - 02:24:41	Michael Wurst	Michael Wurst	CCC/ECS		AI Connected Services	AI Connected Services
	B2V SVDS		10.9.2019 - 02:24:41	Michael Wurst	Michael Wurst	CCC/ECS		AI Connected Services	AI Connected Services
	ECS		10.9.2019 - 02:24:41	Michael Wurst	Michael Wurst	CCC/ECS		AI Connected Services	AI Connected Services

Figure 6.11.: Visualization of Architecture Models in a table-based representation. The columns of the particular table can be specified according to the available annotation information.

6.4.3. Aggregated Architecture Model Communication

The AM communication visualization displayed in Figure 5.30 shows which *Application Components* exchange information. If this aggregation level is too fine granular to fulfill certain use cases or to answer specific questions, GraphQL queries can also be designed with maximum flexibility, ensuring the execution of queries on any aggregation level of the IT landscape architecture. In this case, on *Business Service* level, as illustrated in Figure 6.12.

Goal: An aggregated AM communication visualization enables users to analyze data exchange behavior on a higher aggregation level. The goal of this feature is to remove unimportant details and to provide a more holistic perspective of the IT landscape communication architecture. Hereby, users are able to uncover communication turntables and corresponding bottlenecks.

Visualization: The nodes are arranged tree-like and the position is determined automatically. The positioning cannot be changed manually. Those nodes which have neither in-going nor out-going communication paths are arranged at the very top. Nodes with communications are layouted underneath. Each child nodes are positioned in levels. By moving the mouse pointer over a node, the node will be highlighted in red as well as all assigned communication paths. The hovering of an edge leads to the highlighting of the edge inclusive of the adjacent nodes.

Interaction: The interaction possibility of this view is the same as in the *Architecture Model Communication* visualization. That means, a sidebar is opened immediately after clicking on any node. The menu component contains two input fields, i.e. the date-picker and the search bar.

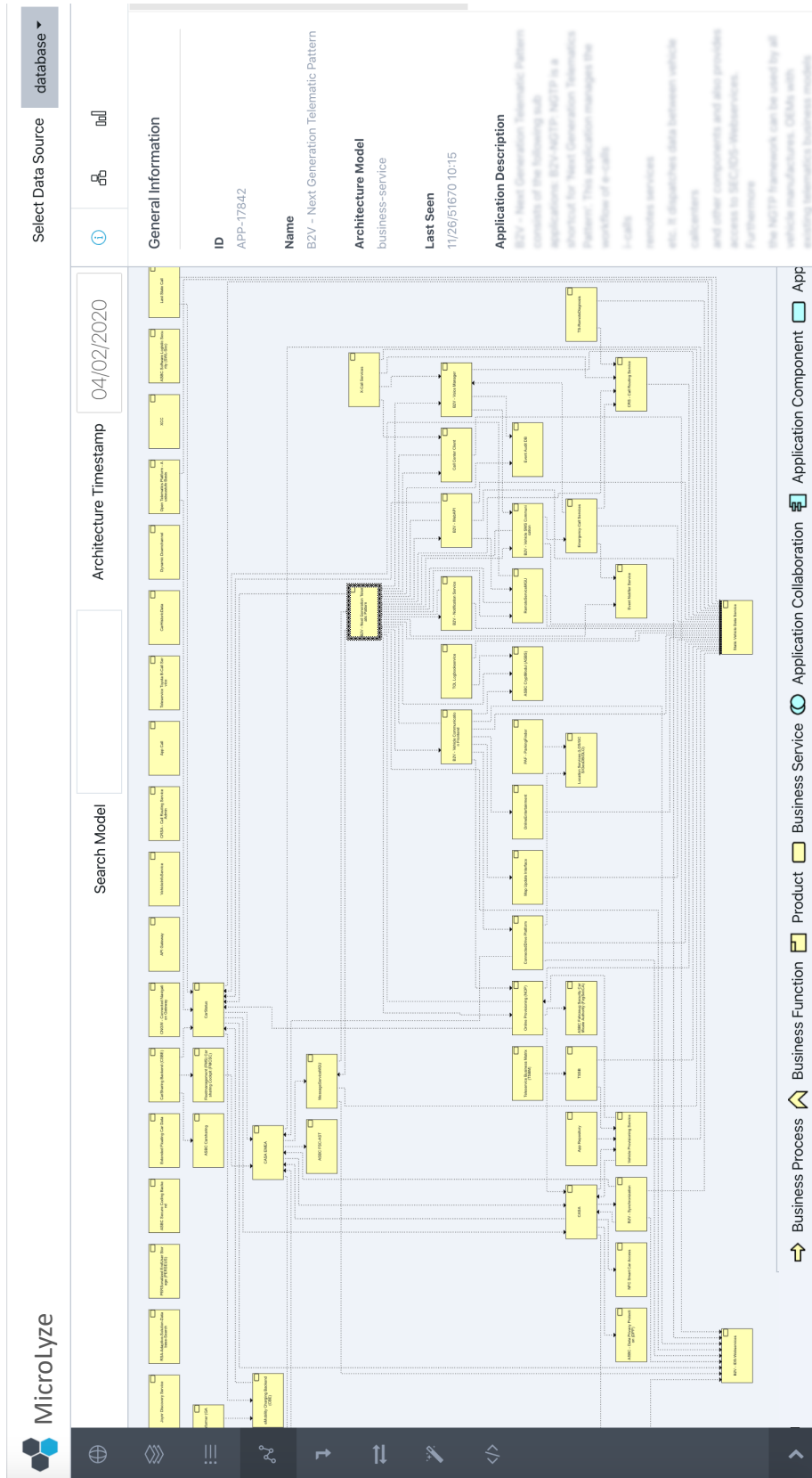


Figure 6.12.: The visualization of *Architecture Model* communications can also be performed on an higher aggregation level in order to address different stakeholder concerns. In this example, we aggregate the communication dependencies on *Business Service* level to unveil which services communicate in general.

7. Evaluation

In the previous chapters, we presented the system, process and visualization design of *MICROLYZE*. Based on a prototypical implementation, we conducted multiple kinds of empirical evaluations to assess different aspects of our contribution.

During our evaluation, we performed surveys, experiments, case studies and interviews as Wohlin et al. (Wohlin et al., 2012) suggests for conducting empirical research. In general, surveys as well as interviews collect information from practitioners to describe, compare, or explain knowledge, preferences, and behavior of individuals (Fink et al., 2006). Surveys and interviews are accomplished by handing out written or online questionnaires that are filled by individuals. Experiments takes place in controlled environments, in order to get an understanding of relationships between specific factors. Furthermore, case studies empower researchers to observe a software tool within an industrial setting, and to study its performance in real-life situations (Kitchenham et al., 1995; Yin et al., 2003).

The combination of different feedback enriched by several evaluation methods result in a profound empirical evaluation (Shull et al., 2001; Wohlin et al., 2012). As argued by Yin (Yin et al., 2003) only multiple sources of evidence lead to an acceptable conclusion in the context of an empirical study. As a consequence, we chose to perform different kinds of empirical strategies as part of this thesis' evaluation. In the following, we detail our evaluation process with support of Figure 7.1 and point out the core publications.

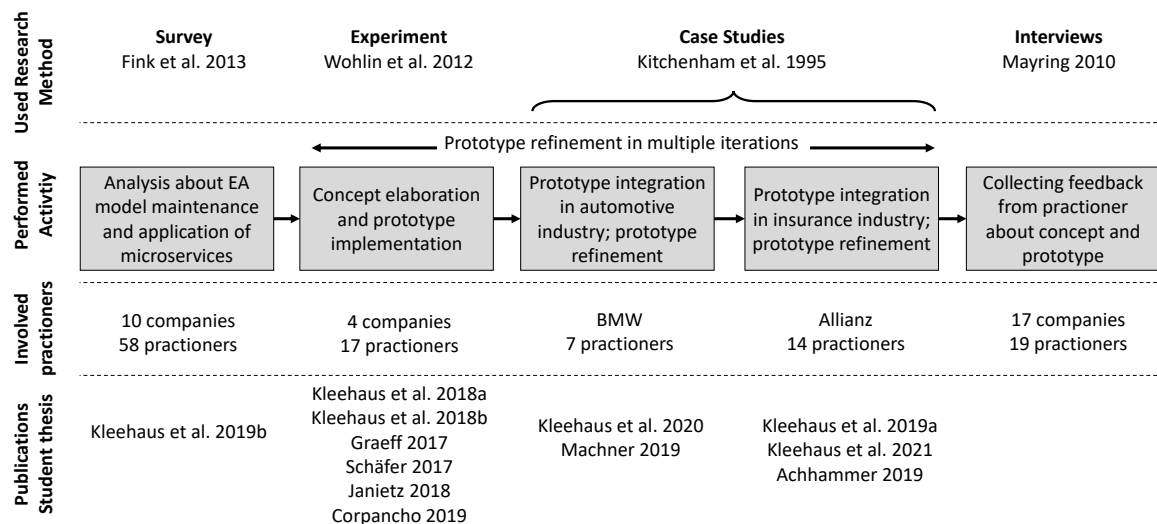


Figure 7.1.: Performed evaluation activities combining different research methods

Initially, we started with an online survey (Kleehaus et al., 2019b) among 58 IT practitioners in the German market to analyze the status quo in the adaption of microservices and

what challenges organizations face while documenting microservice-based IT landscape from an EA perspective.

Next, we performed multiple experiments (Corpancho, 2019; Graeff, 2017; Janietz, 2018; Kleehaus et al., 2018a,b; Schäfer, 2017) to test our elaborated concepts and developed first prototypes in cooperation with industry partners. Four experiments were evaluated in an industrial environment. The insights gained from the experiments were enriched additionally with feedback from practitioners. The gained knowledge were used to refine our concepts that ultimately resulted in the development of *MICROLYZE*.

In the next step, we integrated the tool in industrial settings in the scope of two case studies (Achhammer, 2019; Machner, 2019). We found the industry partners for the case studies after an initial one-hour phone call, in which we intended to figure out if the organization is an interesting candidate and in turn is interested to evaluate *MICROLYZE* within their IT infrastructure. During the phone call, we focused on the following aspects:

- problem description, solution proposal and concept description
- unveil status-quo of the organization's microservice utilization and IT landscape documentation endeavours
- align expectations of a case study on both sides
- scope of the case study, define responsibilities and define follow-up activities

The outcome of the phone call was a precisely formulated scope, i.e. on which part of *MICROLYZE* the industry partner is setting the focus and what is the realistic timeline to employ the prototype. For both case studies the follow-up activity was a live demonstration of the first prototype in front of all involved stakeholders and the clarification which organizational and technical obstacles must be solved upfront.

The first case study mainly focused on the evaluation of the EA model recovery concept based on runtime data analysis. This case study was conducted in the automotive industry. We highlight obtained key findings in Section 7.2. The second case study covers the feasibility of maintaining business-related models and the connection of business- and application layers via the introduction of configuration files and leveraging CD pipelines. For this purpose, we received feedback from the insurance industry. The case study is presented in detail in Section 7.3. Both case studies provided insights about 1) the general feasibility of the approach, 2) which challenges we faced during integration and 3) to what extent *MICROLYZE* can support the management of EA model endeavours.

As a last step to finalize the empirical evaluation of *MICROLYZE*, we conducted 19 interviews with practitioners from 17 different companies to receive overall feedback about the proposed software-, process- and visualization design of the tool. We analyzed the experts' feedback with support of the qualitative content analysis technique introduced by Mayring (Mayring, 2010). Based on this feedback and the results of the previous performed evaluation methods, we were finally able to derive limitations of our concept and to propose future research questions that need to be solved by upcoming researchers. We summarize in Section 7.4 the findings of our conducted interviews.

7.1. Evaluation Design

The way through the evaluation process of the elaborated concept is depicted in Figure 7.2. The case study is structured along the following elements:

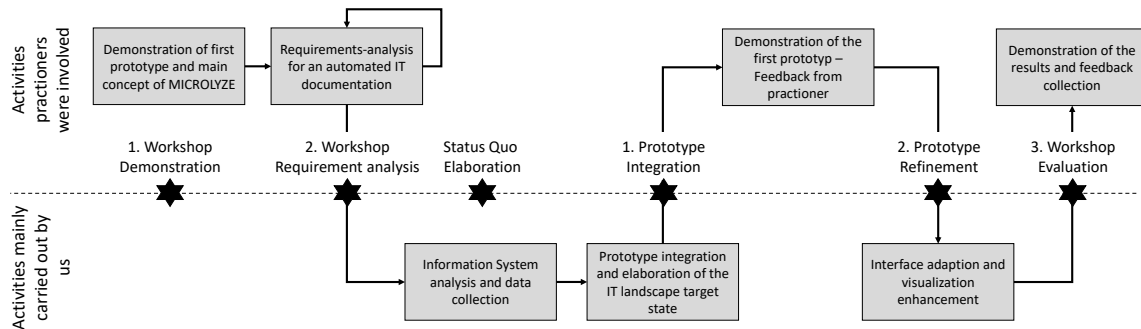


Figure 7.2.: Performed evaluation activities and their milestones

In a first workshop, we clarify the concept of *MICROLYZE* and demonstrate the prototype. The outcome of this workshop was the clarification, which employees must be involved, which infrastructure setting can be used for evaluation, for which applications are access rights required, who is the primary contact person, and what is the exact timeline as well as defined milestones.

In the second workshop, we conducted interviews to derive automation requirements, i.e. AMs, that are important for the industry partner to be recovered and maintained in an automated manner. The objective of this workshop was two-folded: First, we aimed to identify how IT landscape documentation is currently performed at the industry partner's side in detail. Second, we analyzed the present documentation weaknesses and quantified the documentation gap. For this purpose, we created a questionnaire to collect the as-is situation of the IT landscape documentation with regards to the criteria 1) importance, 2) completeness, 3) actuality and 4) change frequency degree of a particular AM:

- The **importance** criteria aims at determining how important it is to document a given artifact. IT artifacts that are not perceived as very relevant also do not necessarily need to be documented automatically.
- The **completeness** criteria indicates how complete the already existing documentation of a given artifact is. Already completely documented IT artifacts have a lower need for automated recovery than incompletely documented ones.
- An IT artifact with an high **actuality** rate is perceived as up-to-date and signalize that the current documentation process is sufficient and the need for automation is low.
- Last but not least, the **change frequency** criteria determines how often an artifact is subject to change and therefore indicates an high need for automated updates.

These criteria dimensions were structured along the three main EA layers (business layer, application layer and technology layer), as well as the inter- and intra-relationships between those layers. In total, the experts rate 41 different AMs on a likert scale (Likert, 1932) of four labels. Even though four labels are not suitable to calculate mean and standard deviation from a mathematical point of view, we still chose this scale in order to identify tendencies of the experts. The 41 AMs are collected during the analysis of Archimate model elements and the examination of well-known EAM tool meta-models including Iteraplan¹ and Alfabet². Afterwards, we prioritized those requirements based on provided dimension ratings. The top 15 requirements represent us the basis to decide whether the solution approach is successful or not.

The next goal aims to determine the case study context, including the SUO in which we were allowed to integrate *MICROLYZE*, as well as the industry partner's current state of IT landscape documentation. The outcome of this evaluation step serve as a baseline to measure achieved improvements after deploying and executing *MICROLYZE*. Data collection methods used in this phase are quantitative analysis of the EA repository, CMDB and used cloud platforms. Based on the status quo analysis, we developed a priority list of those AMs that are mostly required to be recovered and maintained automatically.

For this purpose, we build an *automation score* over the questionnaire's criteria importance, completeness, actuality and change frequency. This score reflects the need for an automated recovery and maintenance of EA models. That means, the score identifies which IT artifacts would benefit most from being maintained automatically. The higher a given IT artifact scored within a category the higher the need for automated maintenance is perceived to be. Based on received feedback from the industry partners, we also applied weights over the criteria importance and completeness, i.e. importance with factor 2.0 and completeness with factor 1.5. Both case study partners perceived those criteria more important than the others. In addition, we reverse the criteria rating of importance and change frequency to normalize the scaling. That means, an importance or change frequency rate of four denotes a low automation need. Whereas, a completeness or actuality rate of four indicates a high automation need. Hence, the scoring is calculated as depicted in formula 7.1.

$$score_e := \frac{\sum_{n=1}^N 2.0x \underbrace{|a_n - 5|}_{\text{rating reversal}} + 1.5xb_n + c_n + \underbrace{|d_n - 5|}_{\text{rating reversal}}}{N} \quad (7.1)$$

where:

$score_e \in \{5.5, \dots, 22.0\}$, defines the need for automation for a specific architecture element
e. The higher the rate, the higher the need for automation.

$a \in \{1 \text{ (essential)}, \dots, 4 \text{ (irrelevant)}\}$, defines the importance rating

$b \in \{1 \text{ (complete)}, \dots, 4 \text{ (not documented)}\}$, defines the completeness rating

$c \in \{1 \text{ (up-to-date)}, \dots, 4 \text{ (not documented)}\}$, defines the actuality rating

$d \in \{1 \text{ (very often)}, \dots, 4 \text{ (seldom)}\}$, defines the change frequency rating

¹<https://www.iteraplan.de>, last accessed: 2020-10-28

²https://www.softwareag.com/au/products/aris_alfabet/eam/default.html, last accessed: 2020-10-28

In the next phase, we finally integrate *MICROLYZE* into the SUO and quantitatively assesses its capabilities with regards to its automated IT landscape recovery degree. In this phase, we also adapted the code of *MICROLYZE* where required in order to better address the target landscape of the case study partner. The outcome of this phase unveils to what extent the industry partner's target meta-model can be covered and how much of it can be automatically reconstructed, including inter- and intra-relationships.

The outcome of the next two steps covers a qualitative assessment against the results obtained by *MICROLYZE*. We use the first feedback in order to start a next iteration to improve the solution. The improved version was afterwards deployed to the same environment. We bundle all findings acquired in the previous phases and present them to the organization's experts in the field. For this purpose, we conducted a semi-structured interview to obtain feedback about the solution's capabilities, value proposition, ideas for improvement and overall judgment. All feedback received as part of the open question was compiled into a set of items classified as *concerns*, *limitations* and *future work*. Concerns denote skepticism or doubts expressed regarding a certain topic. Limitations denote restrictions of different kinds which the solution currently does not cover. Both criteria allow room for future work.

7.2. Case Study in the Automotive Sector

The first case study was conducted in cooperation with a large German-based international automotive company. As of 2019, the company counts over 130,000 employees and generates nearly 100 billion euro revenue. As documented by Machner (Machner, 2019), the case study took place between July, 2019 and November, 2019. The design follows the guidelines and best practices suggested by Runeson et al. (Runeson et al., 2008).

Due to technical and organizational limitations, we had no possibility to 1) instrument the microservices with *microlyze.json* configuration files and 2) to integrate the software into a Continuous Integration pipeline as detailed in Chapter 6. This evaluation mainly covers the concept described in Chapter 5. However, in order to be still able to bridge the gap between application and business layer assignments, the industry partner provided us a Microsoft Excel sheet that already contains a mapping between microservice IDs, domains, (sub)products, business services, as well as further relevant attributes like product owner. The information within the Excel sheet is maintained manually and represent a collection from different sources like CMDB and other federated information systems. With the support of the industry partner, we map the recovered *Application Components* with the microservice IDs maintained in the provided Excel sheet in order to obtain the required mapping table. In general, our industry partner considers microservices too small of a unit and too large in numbers to justify documenting them within a centralized information system. Hence, regarding microservices and their relationships to other AMs there exist no official single source of truth that could be used to validate the accuracy of our recovery algorithms.

Table 7.1.: Status quo: Average as-is EA model documentation rating per EA layer. N=1.
1=fully agree, 2=rather agree, 3=rather disagree, 4=fully disagree

	Importance	Completeness	Actuality	Change Freq.
Business Layer	1,90	2,10	2,40	3,70
Application Layer	1,00	2,50	3,00	3,25
Technology Layer	1,33	1,50	2,00	3,33
Intra-Relationships	1,00	2,00	2,50	3,00
Inter-Relationships	1,43	3,14	3,14	3,86

7.2.1. Requirement Analysis and Status Quo

The second workshop aims to assess the current state in the scope of EA model documentation and which mandatory requirements must be fulfilled in order to describe the project as successful. The workshop was conducted with one *Enterprise Architect*. In the following, we briefly summarize the results, while Machner (Machner, 2019) reports on the case study extensively.

The EA model documentation is mostly performed on a manual basis, which is still the normal case in big organizations (Kleehaus et al., 2019b; Lucke et al., 2010; Roth et al., 2013a). In order to keep the EA models up-to-date, the case study partner installed a robust deployment process. This process ensures that every application which is ready to deploy to production obtains a unique ID, the *APP-ID*. The creation of this ID is part of an holistic quality management process. In general, the *APP-ID* defines an *Application Collaboration*, i.e. a logical aggregation of one or more *Application Components* that perform a specific task. The particular *Application Components* are not documented, nor is their communication structure. As different roles name applications differently, the only way to uniquely identify the applications is via the assigned *APP-IDs*.

The completed questionnaire about the as-is situation of the EA model documentation serve as a baseline to identify the perceived documentation gaps most painful to the organization. An aggregated summary of the result is shown in Table 7.1 including the average rating tendency per element.

The model documentation of the **business layer** is perceived as least important in comparison to the other layers, even though an average completeness of 2,1 indicates a tendency that there are still deficiencies. The stored business information is reported as not always current (2,4). However, the information is not changing very often. Business processes, use case, as well as products are considered the most important models that need to be documented.

All models within the **application layer** are perceived as one of the most important information that need to be collected and kept up-to-date. An average rating of 2,5 indicate that all elements face the highest documentation deficiency except *Application Collaborations*.

Table 7.2.: MICROLYZE execution result: Result of the Top 15 of those *ArchitectureModels* with the highest automation score. $score_{min} = 5,5$, $score_{max} = 22,0$

Rank	Architecture model	score	# recovered models	# unknown models
1	Data flow and dependencies	18,5	2250	2250
2	Intrarelationships (application layer)	18,5	3420	3420
3	Instance (running process)	18,0	884	n/a
4	Interrelationships (application - technology)	17,5	6135	6135
5	Application component	17,5	221	62
6	Actors (customers, partners, employees)	17,0	n/a	n/a
7	Business functions (marketing, accounting, etc.)	17,0	4	0
8	Interrelationships (business - application)	16,5	129	0
9	Interface (external application behavior)	16,5	3.485	n/a
10	Use Cases	16,5	n/a	n/a
11	Intrarelationships (technology layer)	15,0	5807	5807
12	Business processes	14,5	n/a	n/a
13	Application (application collaboration)	13,5	73	n/a
14	Communication technology (e.g. protocols)	13,5	2	2
15	Database (Mysql, MongoDB, etc.)	13,5	8	8

The information about applications should be complete, as the documentation is a required part of the deployment process as stated above. Hence, it is not surprising that the information base of **application layer** models are perceived as rather obsolete (3,0).

Similar to **application layer** models, the information within the **technology layer** is rated as very important and the most completed one. In addition, with an average actuality rating of 2,0 and change frequency rating of 3,33 there is no particular need for a system that automate the maintenance of **technology layer** models.

The **inter-relationships** of elements, i.e. the architectural dependency between different architectural layers, as well as the **intra-relationships**, i.e. the relationship between elements within an architectural layer are also both perceived as one of the most important information that needs to be documented. Hence, especially for **inter-relationships** it is essential to improve data completeness and actuality which is currently reported both with 3,14.

After collecting the ratings, we created the architecture model priority list by calculating the *automation score* detailed in Section 7.1. Table 7.2 lists the top 15 of those models that have the highest score.

As Table 7.2 shows, all dependency and relationship information including data flow between **Application Components** as well as **intra-specific relationships** and **inter-relationships** are highly ranked. This indicates potential in the automated recovery

and maintenance of relationship information between AMs. In addition, also most of the **Application Layer** elements are found within the Top 15. This emphasize the poor documentation of applications at the industry side.

7.2.2. Prototype Integration

The following sections describe the SUO in which *MICROLYZE* was integrated, as well as the execution results in detail.

Evaluation Environment

We evaluated our system at the department for vehicle data connectivity, which develops and operates several IT services that are required in the context of connected cars. The IT landscape which is in responsibility of our industry partner is mostly based on microservice architecture. The infrastructure is monitored by *Dynatrace* Application Monitoring (AppMon). The version in use during the conduction of this thesis was *AppMon 2018 April*. During the evaluation of *MICROLYZE*, we had full access to the whole monitoring data. Unfortunately, AppMon exhibits a few limitations that restrict the effectiveness of the proposed solution approach. These limitations and the applied workarounds will be presented and discussed hereinafter.

- **API restrictions:** As of most APM tools do not publish or even document all available APIs, we are forced to use the additional REST APIs that are only accessible via the AppMon web interface. As described in Section 5.2.3 leveraging undocumented APIs that are not designed to be freely available by developers require specific security credentials that need to be included in every HTTP request header. Without those credentials, we receive a *HTTP 401 Unauthorized* response. In a trial and error approach, we identified two parameters that are absolutely necessary to successfully perform HTTP requests. 1) the *WEBUISESSIONID* needs to be included as a cookie parameter. 2) a CSRF-Token needs to be added as a *X-XSRF-Header* in each request. This token serve as security mechanism designed to prevent CSRF attacks (De Ryck et al., 2011). We are able to retrieve both parameter by simulating user authentication with the support of Puppeteer (see Section 5.2.3).
- **Filter restrictions:** In order to circumvent timeouts in requests or massive resource loads, *MICROLYZE.Collect* is originally developed to retrieve data batch-wise and limited upon timeframes. However, after the first run of our recovery algorithms (see Section 5.3.1), we realized that requesting a timeframe of six hours for all applications took a heavy toll on the algorithm's runtime. It required more than 24 hours to process the data. Hence, we integrated a second request iteration by using each application information as additional filter. This yielded an acceptable trade-off between retrieved amount of data and overall runtime.
- **Timeframe restrictions:** We observed during the execution of the recovery algorithms that the longer we go back in history, the longer the request against the

AppMon API takes to respond. After all, detailed tracing data could only be retrieved for roughly the last ten days relative to the point in time the *backwardRecovery* algorithm was initiated. This restriction could be removed by modifying the AppMon configurations, however, we were not allowed to perform this change.

- **Request response restrictions:** AppMon restricts the response of tracing data to the 100 most recent transactions. As a result, depending on how busy certain microservices are, requesting tracing data for a timeframe of, e.g. one hour, might effectively retrieve data pertaining only to the last few minutes of that timeframe. This could be countered by minimizing the requested timeframe, but in turn, the overall runtime of the algorithm would increase. After performing various tests, setting the requested timeframe to six hours proved to provide an acceptable trade-off between accuracy of individual timeframes and overall runtime of the recovery algorithms. Ideally, since there exists a finite set of possible tracing information to be observed, running the recovery algorithms for a sufficiently long period of time would eventually lead to recovering each unique tracing record at least once and, hence, also each unique microservice communication.
- **Request amount restrictions:** The deployed instance of AppMon in the evaluation environment is reportedly running close to full capacity. AppMon prioritizes available resources for storing incoming data, streamed by deployed agents. During heavy load, manually executed API requests are sometimes canceled via timeout or an empty data set was returned. In order to circumvent this issue, we 1) ran our recovery algorithms between 8pm - 5am during business days and full day during weekends. 2) all requests were performed in a sequential manner, which evidently increased the overall runtime of the recovery algorithm. However, as a positive side effect, requesting information only sequentially reduced the complexity of synchronizing access to the database when storing relevant tracing information.

7.2.3. MICROLYZE Execution Result

MICROLYZE was installed on a personal computer with Ubuntu 16.04 as operating system. The PC is configured with a Quad-Core Intel Core i7 with 2,7 GHz and 16GB DDR5 RAM. The connection to the AppMon server was only available within the internal network of our industry partner and has been established via a VPN connection.

A total of 174 iterations were completed successfully starting September 12th and ending September 27th, thereby covering a time window from August 15th (00:00) to September 27th (12:00). The algorithms did not run continuously but were stopped and resumed various times due to the limitations detailed in Section 7.2.2. One iteration took one to two hours to complete. All requests were executed sequentially in order to prevent too heavy load on the monitoring server.

- Start: September 12th
- End: September 27th

- Timeframe Size: 6 hours
- Completed Iterations: 174
- Iterations backwards: 112 (28 days)
- Iterations forwards: 62 (15.5 days)
- Duration per iteration: ~60 - 120 minutes

Table 7.2 lists the amount of recovered AMs and their relationships. We can only show the total amount of AMs recovered and new AMs found, but not the complete coverage ratio as the ground truth is for most AMs unknown.

In addition to the table above, we recovered 1.141 *Application Interactions*, 5.805 *Nodes* and one *Facility*. Except of the *Facility* model none of those models are maintained in any repository. In addition, we extract 46 *(Sub)Products* and 79 *Business Services* from the provided Excel sheet. All recovered AMs are structured based on 12.544 hierarchy-, 3.086 grouping- and 2.250 communication dependencies. Last but not least, we were able to extract 15.432 annotations that represent further AM attributes.

Business Layer

In this case study, we had no possibility to instrument microservices with *microlyze.json* files in order to establish the relationship between application and business layer elements. Hence, technically speaking, we did not recover such elements, but only extracted them from the provided manual documentation with support of the mapping table.

Application Layer

196 of the 221 recovered *Application Components* (~88.7%) were identified during the first iteration covering merely a timeframe of 6 hours. Within the first day of recovery (4 iterations), the total amount increased to 211 (~95%). The remaining ten *Application Components* were recovered dispersed over the remainder of the recovery process.

MICROLYZE identified automatically 159 of the 167 manually documented microservices (~95%). While this number represent a high coverage and exceedingly satisfying accuracy, the even more interesting observation is that MICROLYZE recovers 221 *Application Components*, which is a lot more than the 159 that could be matched. Further analysis revealed that 8 of the 221 recovered *Application Components* represent databases which are not documented as microservices by our industry partner.

Figure 7.3 illustrates the amount of recovered *Application Components* during the recovery process. The numbers on the x-axis indicate a timeline represented by iterations. Whereas the algebraic sign denotes whether it was an iteration during the backward recovery (negative sign) or the forward recovery (positive sign). The recovery process starts at iteration -1 and initially recreates the IT landscape based on history data (backwards recovery) until iteration -112. Afterwards, the forward recovery takes over by iteration 113 and updates the modeled IT landscape based on new incoming monitoring data until iteration 174.

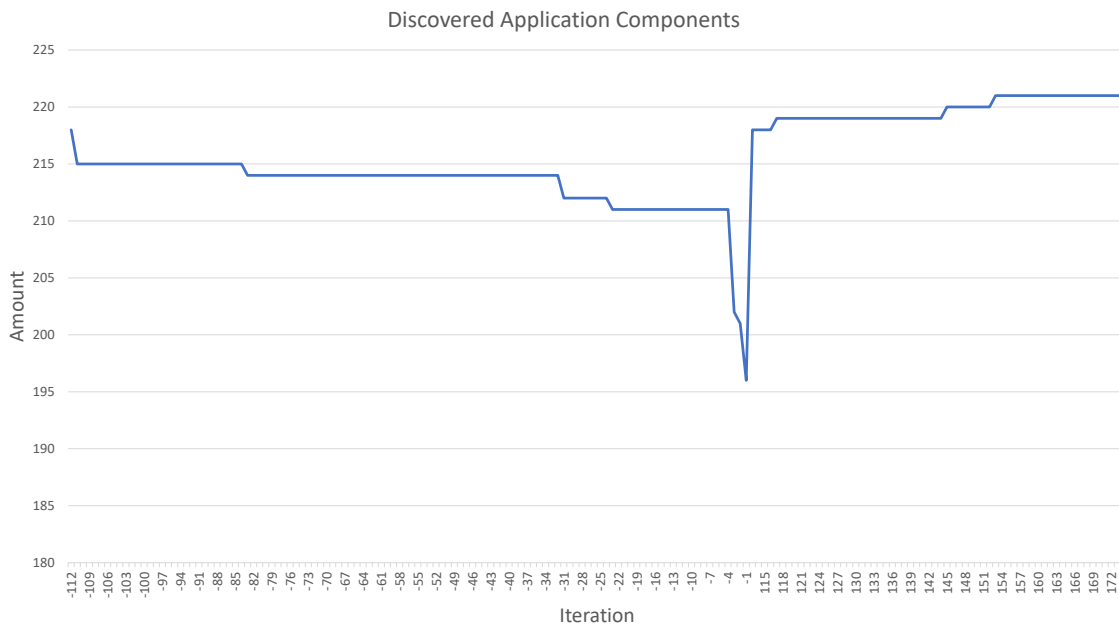


Figure 7.3.: Recovered *ApplicationComponents* during recovery process. The backward recovery process starts at iteration -1 and ends at iteration -112. The forward recovery process takes over at iteration 113.

Recovered *Application Collaborations* were of no particular interest of the industry partner because they do not directly represent *Application Collaborations* as maintained within the federated information systems. Therefore the amount of identified elements bears no expressiveness. However, the industry partner realizes during the evaluation that AppMon should be indeed configured to directly match maintained *Application Collaborations*. This might entail a change of the underlying meta-model.

Technology Layer

Quite a lot of *Nodes* were identified during the recovery process (a total of 5.805), which seems to be disproportionate considering only 221 *Application Components* were uncovered during the same time span. In particular, *MICROLYZE* recognizes multiple spikes in iteration -57, -85 and -112 as Figure 7.4 illustrates, which do not correspond to other recovered elements, when comparing the data.

According to EA practitioners, this phenomenon is accounted for by the usage of pods. A pod is a concept of Kubernetes³. It represents a deployable unit that is used to run a single instance of an application by encapsulating and managing its container. Our industry partner uses OpenShift technology, which leverages the concept of pods⁴.

The sudden rise of *Nodes* can be explained through the lifecycle of pods. Kubernetes'

³<https://kubernetes.io/docs/concepts/workloads/pods>, last accessed: 2020-10-28

⁴https://docs.openshift.com/enterprise/3.0/architecture/core_concepts/pods_and_services.html, last accessed: 2020-10-28

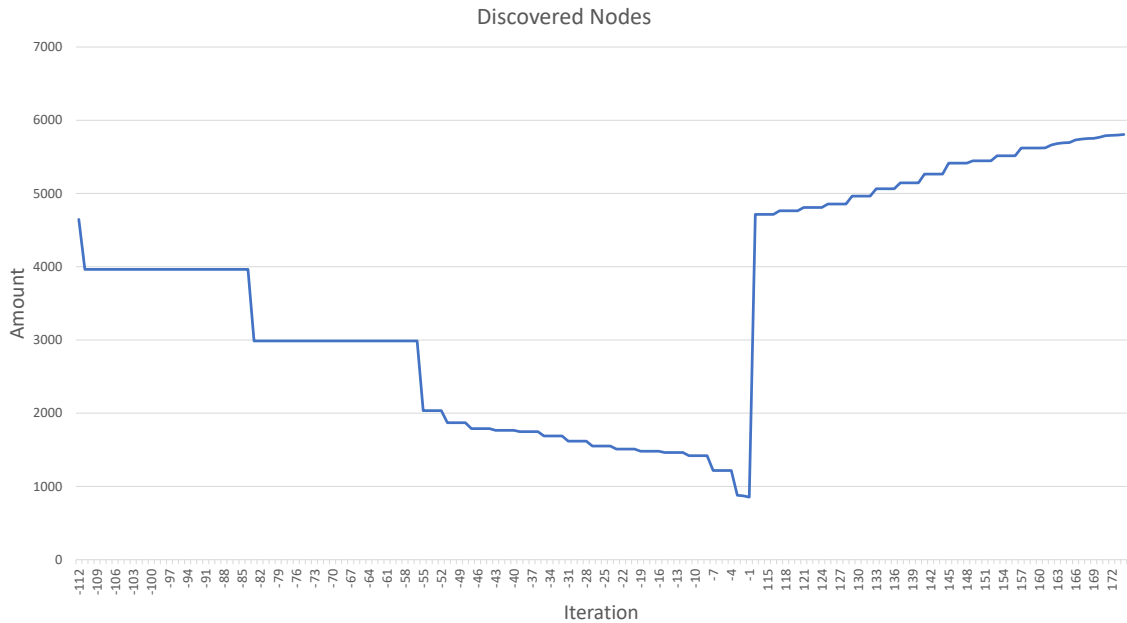


Figure 7.4.: Recovered *Nodes* during recovery process. The backward recovery process starts at iteration -1 and ends at iteration -112. The forward recovery process takes over at iteration 113.

Pods can not be modified while running. They must be terminated, recreated with adjusted configuration to reflect intended changes and finally redeployed. As pods do not maintain state during redeployment, AppMon is unable to recognize redeployed pods and treat them as new *Nodes*. Consequently, *MICROLYZE* inserts those already known *Nodes* into the database with different names, which leads to unwanted duplicates.

Eventually, the "lastSeen" value of the old *Node*, which would no longer get updated, would indicate the old *Node* is most likely no longer available. However, until a reasonable amount of time has passed to justify such assumptions, the *Nodes* could have been redeployed multiple times, which poses a problem in the long run, because the database gets flooded with outdated data. This issue needs to be addressed in future research.

Dependencies

The observed *Application Interactions* were plotted in Figure 7.5 in conjunction with their corresponding communication relationships. Considering that tracing data can only be retrieved for a restricted timeframe into the past, the majority of recovered interactions and communications were identified during the first iterations of the backwards recovery. Further elements were identified with forwards recovery, which was still detecting unknown data even in its final iteration. Hence, it can be suspected that there are a lot of *Application Interactions* which were not yet observed during the evaluation.

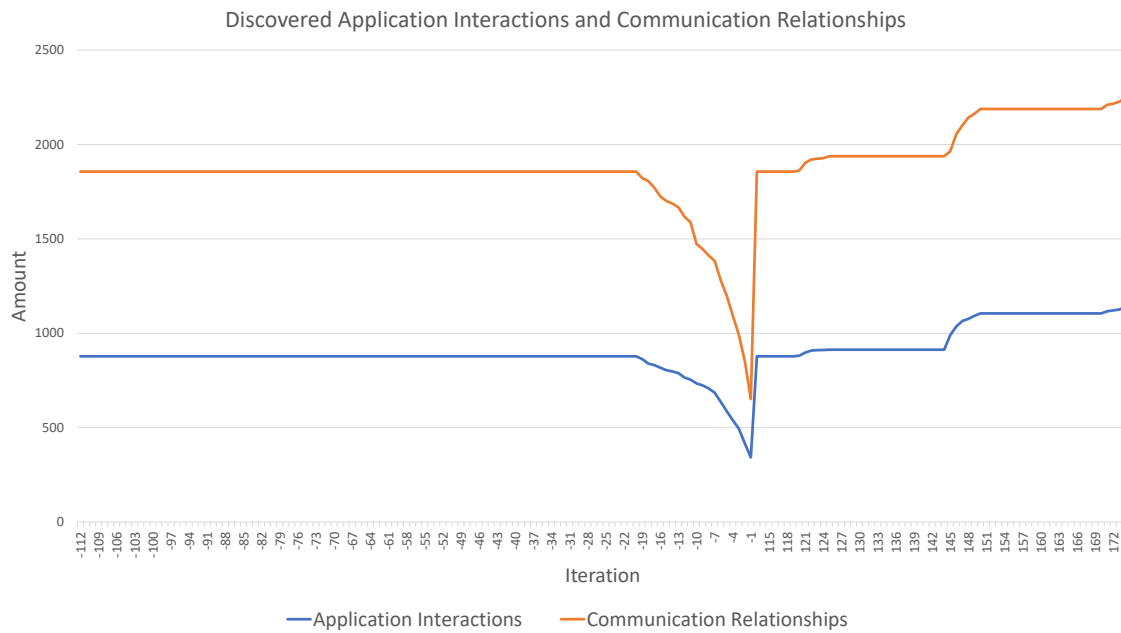


Figure 7.5.: Recovered *ApplicationInteractions* and communication relationships during recovery process. The backward recovery process starts at iteration -1 and ends at iteration -112. The forward recovery process takes over at iteration 113.

7.2.4. Feedback from Practitioners

In the following section, we present the results of our conducted semi-structured interviews with seven experts from the case study company including three Enterprise Architects and four Solution Architects. We structure each of our findings with 1) the question we asked 2) the core feedback received from the practitioner, and 3) a brief discourse providing further insights and discussions. For a detailed report on the case study interviews, we refer to Machner (Machner, 2019). Even though, we had no possibility to evaluate the instrumentation of microservices with *microlyze.json* configuration files, we presented our interview partners the whole solution approach including every aspect described in Chapter 5 and 6. However, in this feedback round we set the focus on the concepts of IT landscape recovery based on runtime data processing and the according visualization approaches.

Overall assessment

Question Q1: The first question intends to obtain an overall assessment of the suggested solution. We asked the following question: *"To what extent does the solution approach contribute in general to improve the EA model documentation?"*

Feedback: Multiple practitioners expressed uncertainty as to how the presented solution approach could also cover the automated documentation of monoliths and legacy systems

besides microservices (limitation L1). Some concerns were raised towards completeness and reliability of the data. Since it can not be guaranteed that the solution recovers all relevant elements during a specific timeframe, no claim can be made that the recovered AMs and relationships are complete (concern C1).

Reflection: In general, most of the interviewees agree that *MICROLYZE* is an innovative approach to realize an automated EA model documentation, especially in a microservice architecture-based environment. The Enterprise Architects stated that every piece of information that can be extracted automatically is considered an improvement compared to manual documentation prevailing at the moment. It was claimed that manual documentation becomes outdated at the moment it is created. Especially in the aspect of actuality and consistency runtime data is considered to be the right choice to uncover reality. In this regard, one Enterprise Architect stated that *MICROLYZE* might be a good tool to validate the currently existing manual documentation and thereby assist the documentation process in general. The Solution Architects argue that *MICROLYZE* with its visualizations and standardized query language provide a good baseline for supporting root-cause-, triage- and even impact analysis. From their point of view, those use cases are even more important than the model documentation itself.

According to stated limitation L1, the presented solution approach is primarily scoped for the usage of microservice-based IT landscapes. We heavily rely on the runtime data exposed by APM tools. In case the connected APM tool is also able to instrument monolithic application and legacy systems, we are also capable to integrate those information systems in our recovery process. During this evaluation, we had no access to legacy systems or received related runtime data. Hence, it is part of future work (future work F1) to cover this question. C1 is a reasonable concern. We cannot guarantee in any point in time whether the recovered AMs and related relationships are complete. An assessment of data completeness is always based on the assumption that the ground-truth is known. Since in this case the ground-truth is unknown, we can only guarantee that our recovery process is a good approximation. Expressed in simple words, since runtime data always have a time dimension, we only see what happens in the regarded timeframe.

Question Q2: This question aims at assessing the value proposition of the solution approach by contrasting the estimated costs with the expected benefits. *"How do you assess the cost-benefit ratio of the solution approach in general?"*

Feedback: All interviewees estimate the costs for integrating and managing the solution as being low. APM tools are already in place and therefore would not create additional costs. Creating and maintaining a JSON configuration file is also not perceived as a huge effort even for external contractors. Still, it is advised to keep the content of the configuration files as minimal as possible. No information should be added just for the purpose of adding additional information. Hence, every piece of information needs to have an added value. Furthermore, it was argued that the integration needs to be conducted incrementally and not as a big bang approach. Initially, only mission critical

microservices should be incorporated in order to save costs and assess the provided results.

Reflection: In general, it was argued that costs and benefits strongly depend on the organization in question. In case of our industry partner, where APM tools are already in widespread usage and therefore do not represent additional costs, the perceived benefits clearly outweighed the overall cost estimations. However, even if monitoring tools are not in place, it was stated that the automatic extraction of architecture information from the JSON configuration files represents a tremendous benefit.

Incremental roll-out would indeed minimize the risks in general. However, we see two issues that need to be considered. 1) An incremental roll-out requires multiple procedures for documentation to be used in parallel while integrating all microservices into the solution approach. 2) Even though only incorporating mission critical microservices brings the most important elements of the IT landscape into focus, it also eliminates the ability to provide a complete view of the IT landscape. This contradicts with one of the goals of the proposed solution, which is to provide an holistic recovery approach.

Technical assessment

Question Q3: The next question focuses on receiving feedback about how reasonable runtime data is for the architecture recovery process: *"How do you assess the approach of extracting Architecture Models and relationship information from runtime data?"*

Feedback: Besides stated limitation L1, the practitioners correctly identified the challenge of recognizing removed models based on runtime data. *MICROLYZE* will always show the presence and therefore existence of certain models because they were observed within a specific timeframe. However, it can never conclusively prove the absence of models, as missing runtime information only indicates a lack of instrumentation in the first place (limitation L2). Furthermore, while runtime data can show that a communication took place between two AMs, it can not explain the reasons why these models communicated with each other. Therefore, it was argued by the practitioners that an actual understanding of the underlying architecture cannot be revealed (limitation L3). In addition, one Enterprise Architect raised the concern that runtime data is naturally technical and fine-grained and therefore probably not suitable to be used as a basis for communicating with stakeholders outside the technical domain (e.g. management). It needs to be abstracted and translated into terms and KPIs appropriate for the stakeholder's role (concern C2). Last but not least, another limitation is that runtime data can only uncover the as-is EA models, but does not provide information about planned or future states (limitation L4). For many EA stakeholders it is important, due to strategic reasons, to see how the IT landscape will look like after the next deployment (n+1 analysis).

Reflection: In general, extracting architecture information from runtime data was perceived as a useful approach. One interviewee even called it the only feasible approach in the long run. In order to solve limitation L2, we introduced deletion thresholds described in Section 5.3.7. However, most of the interview partners were not satisfied with the idea of

using time dimensions to detect obsolete AMs. In addition, we were not able to evaluate our approach during this case study. Consequently, further research is required in this case (future work F2).

An extraction of semantic information from runtime data in order to understand the reason for certain communications between AMs is currently not possible. Even though, semantic information could be added manually in the configuration files, this would cover only certain aspects of the IT landscape. There exists academic projects like MIDAS (Holl et al., 2019), that provide methods to enhance data sources and the underlying schema with semantic information. One goal of this project is to figure out what specific data (from an semantic perspective) certain applications access and process. An integration of MIDAS into *MICROLYZE* could support the identification of this semantic relationships. Overall, limitation L3 must be addressed in future work (future work F3)

According to the limitation L4, especially Enterprise Architects claimed a lack of integration of the EA planning process. At this phase, AMs are planned and initially registered in the EA repository. A link between the current runtime model and the planned model is not available. In order to solve this issue, we propose as future work to extend the *MICROLYZE* meta-model with different AMs life-cycle phases (future work F4). Based on our revision concept, we already store "current" (valid) and "obsolete" models. It would be conceivable to add a third "planned" state.

Question Q4: This question aims at evaluating the integration of the solution approach into a CD pipeline by assessing advantages and possible limitations: *"To what extent do you perceive the integration of the approach into a CD pipeline as useful?"*

Feedback: The usage of a deployment pipeline is perceived as necessary. However, reliability of the approach needs to be ensured at all times as one practitioner stated. For instance, the roll out of an important hotfix cannot be blocked due to an unsuccessful model exposure task (concern C3). A further mentioned concern was that pipeline integration requires additional effort which might be circumvented by some developer teams. To that effect, adherence to the approach needs to be enforced so that all teams follow the rule and do not implement their own solutions outside of a pipeline (concern C4).

Reflection: In general, pipeline integration is seen as very useful and reportedly the best way to get people to comply with the overall approach. It covers the validation of the *microlyze.json* configuration file and achieves an enforcement of the governance regulations. We regard concern C3 as a little evil. The *microlyze.json* configuration files must not be changed frequently. They primarily represent static data that are created once. Consequently, we are convinced that a successfully completed test cases of *microlyze.validate()* and *microlyze.compose()* will still be successful in future deployment executions. Therefore, hotfixes should not represent exceptions.

Question Q5: This question tries to gauge the feasibility of the solution approach with respect to the efforts that need to be undertaken on the technical side in order to imple-

ment *MICROLYZE*. "What barriers do you recognize regarding the technical integration of the approach, especially with respect to the technical requirements that need to be met?"

Feedback: During the case study, we had only access to on-premise deployed microservices. However, in some departments microservices are partially deployed on-premise and partially migrated to the cloud. Hence, uncertainty was expressed regarding how well the solution approach can handle a massive distributed environment (concern C5). This uncertainty is justified in particular when communications with legacy systems occur. A further concern was raised towards security and privacy issues. Since *MICROLYZE* processes and stores sensible information that are included in runtime data, appropriate measures need to be taken to ensure compliance with prevailing laws and regulations (concern C6). Last but not least, a big concern was mentioned by many practitioners regarding the integration of multiple APM tools (concern C7). In the presented case study, we had only access to one monitoring solution. However, especially in big companies, different commercial and even open-source APM tools might be in use to accommodate the agile mind.

Reflection: Even though some concerns were raised, all practitioners agreed that the technical integration is manageable as no identified barrier is perceived as insuperable. All interviewees regarded the technical integration as technically feasible without any major obstacles, even some are going as far as calling it trivial. The usage of monitoring tools is already widespread within the organization and does therefore not require additional effort to set up. However, there was an agreement that a dispersed usage of different APM tools could lead to operational issues in the long run. JSON is considered a standard and therefore not an issue.

Regarding concern C5, most APM solutions including AppMon provide support for both on-premise and cloud deployments. From a technical point of view, it makes no difference on which environment the monitoring agents are installed, as long as they are allowed to communicate with the required monitoring server. As described by Kleehaus et al. (Kleehaus et al., 2018a), further information about the data origin including deployment environment can be simply delivered by extending runtime annotations. If the information system is not supported by the APM solution which might be the case with legacy systems, then the recovery process will provide an incomplete picture. This limitation was already covered in L1.

Even though stated concern C6 does not have to be a barrier per se, it does require additional implementation and configuration effort. For instance, some countries do not allow sensible information to leave the country and be stored elsewhere. Considering many organizations operate data centers in different parts of the world, this is indeed an issue that needs to be addressed (future work F5). One possible solution would be to store the collected data locally within the allowed boundaries and query them from their respective storage locations in order to retrieve the overall IT landscape picture.

MICROLYZE supports different meta-models of several APM solutions as illustrated in Section 5.1. In addition, using the adapter design pattern, it is possible to extend the

support of further APM tools with just a little implementation effort. Hence, the support of several APM tools deployed in parallel should not be an issue. However, during this thesis, we have not evaluated this feature and considering it as future work (future work F6).

Organizational assessment

Question Q6: The following question tries to identify whether shifting the responsibility towards developer teams is considered the right approach: *"How do you assess the approach of shifting the documentation responsibility towards developer teams?"*

Feedback: Some practitioners stated that a centrally stored documentation which is conducted in a decentralized way might be more complicated to keep consistent. An expressed concern is that not every team might use the same notation and structure (concern C8). In addition, the teams might lack in motivation to actually fulfill their responsibility for documentation (concern C9).

Reflection: Overall, shifting the documentation responsibility to the agile teams was perceived as a necessary measure to improve overall documentation quality. The interviewees mostly agreed with this approach as it is already a reality within the organization. The documentation should be conducted by those who are closest to the information source, i.e. mostly developer teams and product owners. They possess the necessary information to comprehensively document the IT artifacts, which is not necessarily the case for Enterprise Architects.

Regarding concern C8 and C9, *MICROLYZE* already provides a solution for enforcing developer teams to use the right notation by validating *microlyze.json* files against *JSON Schema* files within the CD pipeline. The enforcement of rules of conduct is task of the EAM team.

Question Q7: This question aims at identifying which stakeholders or roles need to be taken into consideration when integrating the solution approach into an organization: *"Which stakeholders do you identify as drivers and blockers regarding the integration of MICROLYZE into the existing organization?"*

Feedback: A variety of people and roles were mentioned that need to be involved (cf. Section 6.2). Architecture departments and management positions were identified as drivers. These stakeholders profit most from the integration of *MICROLYZE* and do not necessarily have to spent much effort in the integration of the solution besides approving and controlling it. Possible blockers are considered departments that are responsible for data protection and security related concerns (concern C10). As the proposed solution processes possibly sensible information including user behavior, a variety of measures such as access management need to be applied in order to ensure compliance with applicable laws and regulations. This represents additional effort and might limit the acceptance of *MICROLYZE* by the responsible stakeholders. Developers and Product Owners who are

finally responsible for integrating the solution are not regarded as blockers in fact. It was argued that the integration and operation of monitoring tools are required anyway, so the additional effort keeps within limits.

Reflection: In a final reflection, it can be concluded that every stakeholder who would potentially benefit from the solution would act as a driver. On the other hand, everybody who would perceive additional effort for integrating and adapting the solution would act as a blocker. This is why the benefits of *MICROLYZE* needs to be made transparent and communicated clearly to the respective people in a convincing way.

Visualization assessment

The next questions focus on assessing the different visualizations created with support of GraphQL. We separate the visualizations into two different groups. 1) AM dependency views cover *Architecture Model Cluster*, *Architecture Model Deployment* and *Architecture Model Table*. 2) AM communication views encompass *Architecture Model Communication*, *Architecture Component Interaction*, as well as *Architecture Model Comparison*.

Question Q8: In this question, we wanted to know how the interview partners rate the AM dependency views. Hence, we asked the following question: *How do you assess the AM dependency views?*

Feedback: It was stated by the practitioners that the *Architecture Model Cluster* view contains valuable information in general. However, it does not add more value to already existing visualizations (concern C11). It shows information dependencies that does not change very often. Therefore, from the practitioner's point of view, the manual documentation of such overviews is usually sufficient.

The *Architecture Model Deployment* view was mostly criticized for its level of complexity. Depending on how many *Application Components* are associated with the selected *Application Collaborations*, the resulting view can become quite large with too many edges and nodes in general (concern C12). Grouping *Application Components* by their technology and *Nodes* by their operating systems was perceived as confusing and unnecessary by some practitioners. However, the view was deemed useful for technical users who are responsible for deployments and more interested in the underlying IT infrastructure.

The *Architecture Model Table* view was considered as an important data representation for quickly accessing information. It was argued that more analysis functionality is required to regard it more useful (limitation L5).

Reflection: Overall, the practitioners were overwhelmed with the amount of information those visualizations provide. In a certain way, they were impressed to see how *MICROLYZE* is able to manage and visualize all those dependency information. However, it requires clear use cases and more focused information representation to have an added value to the practitioners. Especially, search and filter methods, as well as analysis functionality would lead to a higher level of abstraction (future work F7).

For instance, one practitioner stated the *Architecture Model Cluster* view is valuable when a company goes through a reorganization, but then only the information of one specific domain or department should be visualized. In this context, the displayed hierarchy level should be selectable as well. Another use case was mentioned according to the table-based representation of dependency information. One practitioner suggested of having a service catalog including assigned interfaces, descriptions and responsible persons of contact. If those information and relationships are recovered automatically, it would really have an added value. Search and filter functionality would round-off this use case.

Question Q9: Finally our last question aims at collecting feedback regarding the developed communication views. We asked the following question: *How do you assess the AM communication views?*

Feedback: The *Architecture Model Communication* view made mostly a good impression on the practitioners. It was argued that more metrics should be included and hotspots should be highlighted, e.g. elements with many incoming and outgoing edges must be visualized more prominently (limitation L6). In this context, additional functionality is deemed necessary to drill the information up and down to the desired level of aggregation (limitation L7).

The *Architecture Component Interaction* view made an exceptional impression on the EA practitioners. In order to further improve the view, it was suggested to remove the *Application Collaboration* from the visualization since it does not provide any valuable information. Further potential was identified by associating the extracted requests with business use cases and business processes (limitation L8). This would unveil what happens on a technical level when a certain use case or business process is triggered.

Regarding the *Architecture Model Comparison* view, it was mentioned, that filtering states by date should be replaced or supplemented by release deployment cycle since this is the only trigger that applies changes (limitation L9). Furthermore, it was argued that future (planned) states need to be included in the comparison (limitation L10). A further concern was raised according to reliability. Since runtime data does not necessarily provide a complete data set as discussed in previous questions, the *Architecture Model Comparison* view can contain false positives, i.e. due to incomplete data differences might be detected which do not exist in reality (see also concern C1). Reduced reliability leads to more manual effort to double check whether a difference is actually true or not.

Reflection: The AM communication views were regarded as more important than the AM dependency views. Especially, the *Architecture Component Interaction* view was praised for providing information which no other tool provided in this manner. The possibility to select an individual request and visualize its way through the system including the called interfaces was perceived as one of the most important added values.

Limitation L6 requires only small changes on the frontend. According to limitation L7, aggregation is already supported by the *MICROLYZE.expose* component, but was not implemented in the frontend yet. Both limitations must be addressed in future work F8

and F9 respectively.

Limitation L8 could be accomplished by adding further dependency information to the *microlyze.json* configuration file and should be addressed in future work F10.

Comparing different states within the *Architecture Model Comparison* view was generally considered useful. The implemented highlighting of added and removed elements was identified as a necessary feature. As we already store the exact time of a deployment cycle, a change in the state selection can be easily achieved in future work F11. According to limitation L10, especially Enterprise Architects claimed a lack of integration of the EA planning process. We see potential in the extension of the *microlyze.json* instrumentation. The JSON file could be already initialized during the application planning phase and is passed through the implementation and release stages in order to bridge the gap between all model life-cycle phases. Once an application is released to production, the state of this model would change from "planned" to "current". An adaption of MICROLYZE's meta-model is indispensable to apply a further AM life-cycle phase. This must be part of future work F12.

7.2.5. Critical Reflection of Results

This case study particularly shaped how we perceive the process support for an automated maintenance of EA models via a continuous analysis of runtime data. Many expectations and ideas of the practitioners center around features that have a productive character and go beyond the purpose of our prototype. Sometimes the Enterprise Architects compared MICROLYZE with a fully-fledged extended EA repository. While this certainly was none of our intentions, it shows the maturity of the prototypical implementation. Taking into consideration all the findings of the qualitative analysis, one can assert that the proposed solution approach seems promising.

However, many concerns and limitations were raised and discussed that need to be addressed in future research. In the following, we summarize the additional prerequisites that need to be fulfilled prior to a productive roll-out:

1. **Reliability and data completeness – C1-C3, L2:** Further research must be conducted in order to ensure reliability of the solution approach and completeness of the recovered AMs including their relationships. Attention should be paid to the recognition of removed elements (see Section 5.3.7).
2. **Scalability – C5, C7, L1:** In order to increase the solution's scalability, further investigation is required regarding 1) the support of massive distributed environments, 2) the parallel support of different APM tools and 3) the support of legacy systems, as well as monolithic applications.
3. **Compliance with guidelines – C4, C8, C9, C10:** The solution requires manual effort to become fully operational like microservice instrumentation, as well as CD pipeline integration. Hence, several stakeholders must be involved in this process. It must be ensured that those stakeholders comply with the defined guidelines and are motivated enough to maintain the system. From our point of view it is indispensable

to communicate the benefits of *MICROLYZE* clearly to all involved stakeholders in order to achieve far-reaching acceptance.

4. **Data privacy and security – C6:** Precautions must be taken in order not to jeopardise data privacy and security. In this case, further assessment in cooperation with data protectionist is required.
5. **General improvement – C11, C12:** As not every developed visualization met with approval, it is necessary to redesign some views on the basis of clear and small use cases.
6. **General extension – L3-L10:** Last but not least, the practitioners elaborated several missing functionalities that must added. Especially, the integration of future IT landscape states (L4) and the rational behind certain microservice communications (L3) were mentioned several times as missing important features. Further limitations can be assigned to the frontend application, e.g. hotspot visualization (L6), drill-up and drill-down features (L7), deployment cycles instead of time dimensions (L9), etc.

7.3. Case Study in the Insurance Sector

The second case study was conducted in cooperation with a large German-based international insurance enterprise. As of 2019, the company counts over 140,000 employees and generates over 140 billion euro revenue. As documented by Achhammer (Achhammer, 2019), the case study took place during April, 2019 and October, 2019. The design follows the guidelines and best practices suggested by Runeson et al. (Runeson et al., 2008).

This case study considers mainly the assessment of the concept to extend the automated recovery and maintenance of EA models with business-related information. This information is provided via configuration files that are exposed by CD pipeline integration as detailed in Chapter 6. Due to technical and organizational limitations the access to runtime data in this case study is limited to the cloud platform repository, which only provides the current state and no historical data. For that reason, in this case study, we mainly focus on evaluating the concepts described in Chapter 6.

7.3.1. Requirement Analysis and Status Quo

The requirement analysis workshop was conducted with seven experts covering the *Enterprise Architect* role. In the following, we briefly summarize the results, while Achhammer (Achhammer, 2019) reports on the case study extensively:

Similar to the first case study, the EA model documentation is performed on manual basis. Exceptions are automated data imports from federated information systems like the CMDB and the PPM system. From those systems relevant architecture-related information are imported to the EA repository on a daily basis in order to keep a central model documentation up-to-date. However, the imported data stored in the federated information systems are created manually as well. The integration is unidirectional, i.e. an executed change is not synchronized with other systems, which sometimes lead to import conflicts.

Table 7.3.: Status quo: Average as-is EA model documentation rating per EA layer. N=7.
1=fully agree, 2=rather agree, 3=rather disagree, 4=fully disagree

	Importance		Completeness		Actuality		Change Freq.	
	avg	var	avg	var	avg	var	avg	var
Business Layer	1,67	0,36	2,63	0,79	2,65	0,62	3,17	0,61
Application Layer	1,78	0,38	3,23	0,25	3,00	0,11	2,69	0,58
Technology Layer	2,54	1,15	3,07	2,34	3,15	1,03	3,24	0,46
Intra-Relationships	2,10	0,30	3,34	1,66	3,15	0,13	2,94	0,56
Inter-Relationships	1,35	0,40	3,09	0,45	3,00	0,25	3,01	0,67

Furthermore, no naming conventions or global, system-independent identifiers are in place for application. Since different roles name applications differently, the used naming does often not match across information systems.

The completed questionnaire about the as-is situation of the EA model documentation serve as a baseline to identify the perceived documentation gaps most painful to the organization. An aggregated summary of the result is shown in Table 7.3 including the average rating tendency per element.

The maintenance of the **business layer** models is perceived very important and in relation to the other layers most complete, even though an average rating of 2,63 indicates a tendency that there are still deficiencies. Important to mention is that the documentation actuality (2,65) of the business layer models is considered rather low, even though the data pool of this layer is not changing very often (3,17). Business domains, -processes and -capabilities as well as products and product domains are considered the most important models that need to be documented.

The models within the **application layer**, including applications (microservices), interfaces and communication relationships face the highest documentation deficiency. The completeness is rated between 2,60 to 3,60 and the actuality between 2,57 to 3,80. A complete repository of those models are considered as essential, as all entities are rated as very important with an average of 1,78 points. By now, it has been a conscious decision by the industry partner, not to model microservices as this would cause too much manual modeling effort. The application layer experiences the most frequent changes in comparison to the other layers with an average of 2,69 points.

The **technology layer** is of least relevance to the industry partner, even though the completeness and actuality of those models is rated rather low with an average of 3,07 and 3,15 respectively. It is notable, that the variance of received importance and completeness rating is high for technology layer models, which might indicate that the experts do not have a uniform perception here.

The **inter-relationships** of AMs, i.e. the architectural dependency between different architectural layers, as well as the **inter-relationships**, i.e. the relationship between AMs within an architectural layer are both perceived as the most important information that needs to be maintained. Hence, it is severe that the rated completeness of this information

ranges between 3,09 to 3,34 in average. As all kinds of relationships are modeled fully manually, the experts stated doubts regarding the actuality of the data repository.

In the next step, we create a priority list of AMs by calculating the *automation score* as detailed in Section 7.1. Figure 7.4 shows the TOP 15 of those AMs that have the highest score.

Table 7.4.: MICROLIZE execution result: Result of the Top 15 of those *Architecture Models* with the highest automation score. $score_{min} = 5,5$, $score_{max} = 22,0$

Rank	Architecture model	score	# recovered models	# unknown models
1	Interrelationships (application - technology)	17,9	3404	2798
2	Interface (external application behavior)	17,7	1133	793
3	Data flow and dependencies	17,6	0	n/a
4	Application (application collaboration)	17,3	2	2
5	Intrarelationships (application layer)	17,2	5906	3393
6	Interrelationships (business - application)	17,1	15	15
7	Instance (running process)	16,7	n/a	n/a
8	Intrarelationships (within business layer)	16,6	3	3
9	Actors (customers, partners, employees)	16,5	2	0
10	Business processes	16,4	0	n/a
11	Use Cases	16,0	0	n/a
12	Roles (product owner, developer, etc.)	16,0	0	n/a
13	Business functions (marketing, accounting, etc.)	15,8	1	0
14	Business capability	15,5	1	0
15	Application component	15,3	1660	1660

As Table 7.4 uncovers, communication dependencies between *Application Collaboration*, i.e. *data flow and dependencies* as well as further **intra-specific** and **inter-specific** relationships are highly ranked and listed within the Top 15. This indicates a high need for automating the recovery and the maintenance of relationships between AMs.

In addition, also most of the **Application Layer** models are highly ranked. This emphasize the poor documentation of *Application Collaboration* at the industry side. An exception are *Application Components* that are ranked on the last position. This corresponds to the current circumstance that the industry partner focuses on the modeling of *Application Collaborations* only, and skips more fine-grained components like microservices. After feedback from the industry partner, we learned that those components are too expensive for manual modeling and especially for Enterprise Architects not important enough.

Most business-related models can be found in the TOP 15. However, it is challenging to automate the maintenance of those models, as they represent virtual elements that must be defined on management level.

Despite being perceived as poorly documented, models from the technology layer, except for *Instance (running process)*, take lower positions in the ranking. The reason behind this are rather low ratings for the criteria *importance*, which has an average of 2,54.

7.3.2. Prototype Integration

The following sections describe the SUO in which *MICROLYZE* was integrated, as well as the execution results in detail.

Evaluation Environment

The available IT landscape for the case study is depicted in Figure 7.6. It consists of an Enterprise Private Cloud layer that is subdivided into a *militarized zone* and a *de-militarized zone*. Frontend-based applications are running in the *militarized zone* and are decoupled from the backend application components located in the *de-militarized zone*. In each of these zones dedicated cloud platform instances are located which host the applications. Cloudfoundry⁵ and Redhat OpenShift⁶ are used as the cloud platform technologies. Communications between both zones are performed either via message queues or via a central API gateway. The core business systems of the case study partner reside in proprietary, traditional data centers. This part of the IT landscape is out of scope for the evaluation.

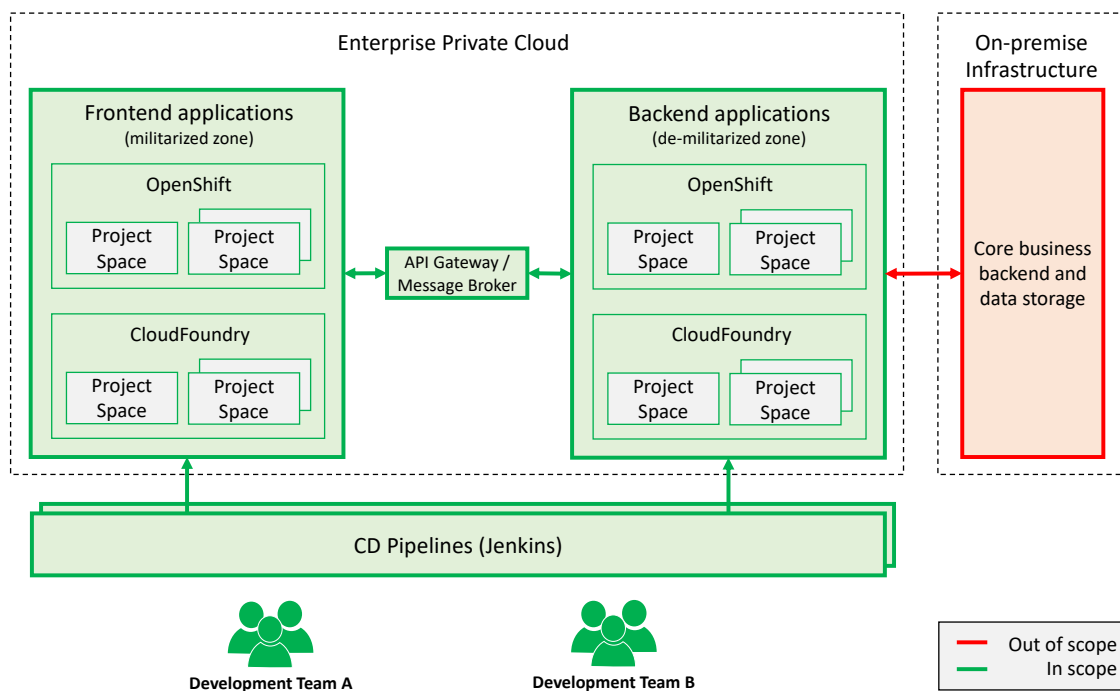


Figure 7.6.: Case Study IT landscape overview

⁵<https://www.cloudfoundry.org>, last accessed: 2020-10-28

⁶<https://www.openshift.com>, last accessed: 2020-10-28

During the evaluation, we had access to two agile development teams that are assigned to each of those zones. Both teams utilize the CD pipeline *Jenkins*⁷ to deploy applications to the cloud environment. We received access to the cloud platforms internal monitoring tool for extracting runtime models. In addition, we were allowed to instrument the CD pipeline and to integrate an additional test case for validating the *MICROLYZE* configuration file. We received feedback from the following experts: 1) The agile development teams including Product Owners and Developers who take a central role in order to make the solution work. 2) The central Enterprise Architecture Management department, that is both, provider of the EA repository and key consumer of EA information. 3) Domain Architects, that currently bear the most manual EA modeling activities.

Configuration File Content

Before *MICROLYZE* becomes operational, we asked the case study partners to decide which federated information systems should be referenced in the configuration file and what additional attributes, beside the mandatory ones, should be added. Since CMDBs and PPM systems are mainly used for EA model documentation and provide important architecture-relevant information, those tools are referenced by *MICROLYZE*.

The reference to the CMDB is important, as this database is currently used as the single point of truth for EA model documentation. The Product Owners maintain all applications within their area of responsibility as *Business Services* in the CMDB. This is part of a mandatory process, as without a valid record the Product Owners do not obtain approval to deploy their applications to the production stage. In addition, the CMDB is closely connected to operative IT service management processes such as change, incident or request management. The CMDB data is also imported on daily basis into the EAM tools to be accessible for Enterprise Architects. Relationships are not imported which is one of the main reasons for a lack of documentation at this place. In general, the CMDB is considered the leading system over the EA repository.

The case study partner uses Apigee⁸ as an API-management system for building scalable APIs. It serves as the API gateway between the frontend-, and backend applications. A reference to the Apigee platform allows to recover which APIs a given application consumes over the gateway. Based on this information, we are able to recover application communications to a certain extent.

Finally, the project management tool *PlanView* is required to retrieve assigned projects to a particular *Application Component*. The experts also considered integrating the license management system into the configuration file. However, the endeavor failed due to issues in establishing a reliable mapping between the systems' models.

In conclusion, based on the expert's feedback and information demand, the configuration file contains the following properties that contribute most to cover the missing AMs listed in Table 7.1.

Listing 7.1: Extended configuration file with references

⁷<https://jenkins.io>, last accessed: 2020-10-28

⁸<https://docs.apigee.com>, last accessed: 2020-10-28

```
1 {
2   // mandatory attributes
3   "name": "...",
4   "description": "...",
5   "business_service": "...",
6   "product": "...",
7   "business_function": "...",
8   "business_capability": "...",
9
10  // further business-related attributes
11  "product_owner": "email of the product owner",
12  "business_subdomain": "the business subdomain the component belongs to",
13  "business_unit": "the business unit the application supports",
14  "business_process": "the business process the application fulfills",
15
16  // references to federated information systems
17  "references": [{
18    "ppm": {
19      "tool": "planview",
20      "domainurl": "...",
21      "apiurl": "...",
22      "id": "..."
23    },
24    "cmdb": {
25      "tool": "servicenow",
26      "domainurl": "...",
27      "apiurl": "...",
28      "id": "..."
29    },
30    "api_gateway_client": {
31      "tool": "apigee",
32      "id": "..."
33    }
34  }],
35 }
```

7.3.3. MICROLIZE Adaption

The configuration file in its current version only supports 1:n business layer assignments. However, some experts of the case study company request the possibility for n:m assignments to reflect the full essence of the IT landscape. As a consequence, we adapted the GraphQL resolvers to support n:m business layer assignment relationships. A modified

microlyze.json is depicted in Listing 7.2.

Listing 7.2: Updated configuration file for n:m business layer assignment support

```
1 {
2   // mandatory attributes
3   "name": "...",
4   "description": "...",
5   "business_service": [],
6   "product": [{
7     "name": "...",
8     "id": "..."
9   }, {...}],
10  "business_function": [],
11  "business_capability": []
12 }
```

As mentioned above, we had only access to the cloud platform APIs, which does not provide full APM functionality but only the status of the repository data. What we can receive are all registered *Application Components* including the application technology, e.g. database, runtime environment, container, etc. as well as their health status, request metrics, resource utilization and the number of instantiations.

Based on the API management tool ApiGee, we are able to extract exposed APIs provided by *Application Components*. This information is normally obtained by distributed tracing. Unfortunately, we are not able to recover which specific *Application Components* consume the exposed APIs. This information is not directly available in ApiGee. We model the exposed APIs as *Interfaces*, which can only be recovered in case the *Application Component* is registered in the API Gateway. As we are not able to obtain detailed information about *Application Services*, we are forced to create dummy records for each *Application Component* in order to assign interface information.

At the industry partner, there is no reliable way to automatically recover *Application Collaborations*. Initially, we aim to extract this information based on spaces (in Cloud Foundry) or namespace (in Kubernetes) that are used to technically organize *Application Components*. However, this was rejected by the industry partner due to a lack of reliability, as the naming of such spaces does often not correspond to a certain *Application Collaboration's* name. Hence, we add this information into the *microlyze.json* configuration file.

The established references to the other information systems *Planview* and *ServiceNow* deliver further attributes, in particular assigned projects that represent valuable information for the Enterprise- and Domain Architects.

7.3.4. MICROLYZE Execution Result

Table 7.4 lists the amount of recovered AMs and their relationships. We can only show the amount of new models found, but not the complete coverage ratio as the ground truth is for most models unknown. The same applies for *Relationships* that have a many-to-many or

one-to-many multiplicity. As we had only access to two development teams that manage in total 15 microservices, we could only recover the relationships that are maintained in the 15 created configuration files.

Business Layer

Regarding the business layer, *MICROLYZE* has not found new AMs in the provided environment, as the documentation of all recovered models were already up-to-date, most probably due to their static nature. Exceptions are three new intra-relationships within the business layer.

Application Layer

MICROLYZE was able to recover two new *Application Collaborations* and 1.660 *Application Components*, which expose 793 undocumented *Application Interfaces*. The huge number of *Application Components* is derived from the exposed cloud platform API. This is a significant improvement as this information is currently not documented at all. Unfortunately, we were not able to extract new communication relationships between *Application Components* since the access of an appropriate APM tool was not available. Furthermore, over 6.084 intra-specific relationships within the application layer could be recovered which mainly represent assignments between *Application Components* and *Application Interfaces*. Important information about *Application Components* that are stored as annotations can be recovered from the cloud platforms itself. This includes inter alia bound cloud services (e.g. databases, autoscaler, monitoring agents, etc.), technology stack, programming languages and software dependencies.

Dependencies

Relationships of *Application Components* towards business-, and infrastructure layer models can be established most likely up to 100 percent. Although, this heavily depends on the availability and completeness of the microservice instrumentation with *microlyze.json* files. In the current case, we were only able to identify 15 new relationships corresponding to the 15 instrumented microservices. In addition, at the case study partner there is no reliable way to technically recover *Application Collaborations*, because no naming conventions or other global identifiers are in place that would allow to map *Application Components* to *Application Collaborations*. The instrumentation of microservices with *microlyze.json* represents currently the only possible way to achieve this mapping. The amount of dependencies between *Application Components* and application instances varies dynamically according to the configuration made in the applied load balancer.

In order to summarize this section, the results in Table 7.4 has shown that most crucial advantage of *microlyze.json* is the automated establishment of inter- and intrarelationshps via a decentralized approach. However, the suggested solution strongly depends on the availability and correctness of the *microlyze.json* configuration file. In any case, the suggested process must be enforced by an established IT governance and technical means to ensure the success of the solution. Also quality assurance mechanisms are crucial to validate the quality of information contained (cf. section 6.1.6). The API gateway as a

source to recover *Application Component* relationships proved to be valuable, but is far from providing a complete picture. In order to close the remaining gap in *Application Component* communications, the analysis of distributed tracing data is indispensable.

7.3.5. Feedback from Practitioners

During the integration of *MICROLYZE* into the SUO including the required setup of all involved systems like the CD pipeline, we documented the time the development teams spent for all implementation tasks. The result of the integration effort is detailed in Section 7.3.5. After the integration process of *MICROLYZE* was finished, we conducted semi-structured interviews with 14 experts from the case study company including six Enterprise Architects, four Domain Architects, two Product Owners and two Software Developers. All of those experts were involved in the integration process and had access to the exposed ETG. While Achhammer reports on the case study extensively (Achhammer, 2019), we summarize his results in Section 7.2.4 and subsequently reflect the case study in the light of additional insights through our interview series. We structure each of our findings with 1) the question we asked 2) the core feedback received from the practitioner, and 3) a brief discourse providing further insights and discussions. As we had only limited access to runtime data, in the following interview series we mainly focus on evaluating the concepts described in Chapter 6.

MICROLYZE Integration Effort

MICROLYZE was integrated by both development teams during a regular *Sprint*. Initially, a *UserStory* was created by the Product Owners in the *Product Backlog* that defines integration requirements. The instrumentation of all 15 microservices with *microlyze.json* files, as well as the creation of the additional test case in the pipeline took for both teams in total 4-5 hours. A challenge was to identify the right business assignments and to model them in the *microlyze.json* file. Whereas the specification of the correct references to the defined federated information systems did not take much time. One Software Engineer had difficulties with adding the shared library to the new *Model Exposure* stage due to missing rights which caused a delay. After the first setup, the teams expect for further instrumentation an effort of 00:30 to 01:00 hour due to learning effects. No impairments were reported by the development teams during the execution of the CD pipelines. The execution of the additional stage took 01:30 to 01:45 minutes to complete. Table 7.7 summarizes the figures.

In general, the development teams perceived the required effort as acceptable, especially compared to other existing governance regulations which require pipeline integration. One Product Owner and Developer appreciated that only a onetime effort is needed, which is predominantly reduced in future instrumentations. However, a four to five hours effort for the initial setup indicate potential improvement. For instance, the number of required method parameters must be reduced as stated by one Developer. Regarding the required time to process the *Model Exposure* stage, one *Product Owner* had little willingness to accept more than 0,5 minutes of extra duration as productive deployments already take

	Development Team 1	Development Team 2
	number of microservices	
	5 microservices	9 microservices
	integration efforts	
effort for pipeline integration	03:00 h	01:00 h
effort for microlyze.json creation	02:00 h	03:00 h
effort for future instrumentation	01:00 h	01:00 h
	pipeline duration (avg)	
documentation stage duration	01:33 min	01:47 min

Figure 7.7.: *MICROLYZE* integration effort compared between two different development teams

a long time. Other experts, in turn, stated that the duration should be kept as little as possible. One practitioner suggested to separate the logic into a dedicated CD server job, that should be triggered from the origin deployment pipeline. This would result to no additional waiting time, however, it would undermine data quality assurance concept (see Section 6.1.6).

Furthermore, additional support for creating the *microlyze.json* file is required as Software Engineers claimed that the manual was not always clear enough. For instance, uncertainties of what the correct path should be have been raised. Developers also requested a more meaningful console based feedback about the success or failure of the *Model Exposure* stage.

Overall assessment

In the following, we discuss the received feedback during the interview series with 14 practitioners involved in the described case study.

Question Q1: Even though, APM tools were not in use in this case study, we presented the complete solution including Section 5.2. We asked the following question: *"To what extent does the suggested approach reduce the amount of manual EA model maintenance effort?"*:

Feedback: Two experts doubt whether APM tools in general and distributed tracing in particular are the right instruments to continuously document the emerging behavior of EA models (concern C1). A complete instrumentation of the whole IT landscape might cause too much overhead and negatively impacts applications' performance. This is seen critically, especially in high performance environments. In addition, Product Owners would benefit only a little as they still have to model their applications in the CMDB in parallel (concern C2). In addition, some experts claimed that the provision of further information about other life-cycle phases are not considered by the solution (limitation L1). The solution focuses on the current state of the IT landscape and does not incorporate planning phases. Hence, the comparison of planned states and current states must be

conducted manually in two different tools. This only leads to unnecessary complexity.

Reflection: In general, all interviewed experts expressed their full agreement, that this solution approach reduces the amount of manual model maintenance effort. In the current state Domain Architects are responsible for most of the EA modeling activities. They collect the required information from agile development teams during meeting, conference calls and emails, which is perceived as very time consuming and bind several resources at once. Those activities can be reduced by the proposed decentralised approach and shift the documentation responsibility to the development teams. Instead of APM tools, we promote two possible alternatives to solve concern C1: 1) the use of cloud specific solutions that are already integrated natively in cloud platforms (for instance AWS XRay). A further alternative might be *Service Meshes*. For instance, Istio⁹ places sidecar-proxies along with deployed services which allows to govern inter-service communication across the mesh uniquely. Such a mesh can be spanned across multiple cloud platforms. The communication behavior can be extracted via REST APIs. However, in order to support both alternatives a huge change in the model recovery algorithms and model transformation logic would be required for *MICROLYZE*. Regarding concern C2, we suggest to leverage *MICROLYZE* as a knowledge source platform for automatically export AMs to CMDBs and other tools (future work F1). Limitation L1 corresponds to limitation L4 in the previous case study (see Section 7.2.4)

Question Q2: In this question, we wanted to evaluate whether *MICROLYZE* represent an important extension of the software portfolio that is needed to successfully fulfill EAM tasks. Hence, we asked the following question: *"To what extent does the suggested solution reasonably integrates into the EAM ecosystem?"*:

Feedback: Two experts disagreed that the solution fit well into the EAM ecosystem. They claim that *MICROLYZE* exposes EA information, that should be actually integrated into existing EAM tools. As a consequence, the recovered EA models and relationships must be imported into the EA repository, which might lead to mapping conflicts and duplicates that need to be handled (concern C3) accordingly. In this respect, one Enterprise Architect also claimed, that automated imports would reduce the flexibility for manual modeling and corrections using the EA repository's user interface (concern C4). Attributes that are affected by automatic import would need to be protected against manual changes. Otherwise they would be frequently overwritten by further imports. The correction support using the deployment pipeline proposed in Section 6.1.2 is perceived as high overhead, as the change request is linked with a deployment release. Hence, proper mechanisms are required to ensure data consistency, which is not provided in the current solution (limitation L2).

Reflection: As of now, the solution is not considered to replace existing EAM tools. Via the proposed ETG *MICROLYZE* concentrates on architecture recovery and knowledge

⁹<https://istio.io>, last accessed: 2020-10-28

provision. Indeed, this knowledge must be imported to EAM tools, to ensure up-to-date and complete EA models. The same applies for CMDB imports, see concern C2 and future work F1. How to solve EA model import conflicts was extensively discussed by Roth (Roth, 2014). During the discussion of concern C3, two Enterprise Architects saw the opportunity to overcome this issue by introducing a system overarching global identifier of applications. In the current state, no naming conventions or global IDs exist at the industry partner's environment. This constitutes an issue whenever elements from different information sources should be matched due to the lack of unique, shared attributes. *MICROLYZE* could help to realize this global ID, by extending the *microlyze.json* file with such an identifier (future work F2). Hereby, we assume that the creation of the configuration file is initialized via a web-based form. A unique identifier for an application is generated automatically and placed into the configuration file. Along with the *microlyze.json* file this ID becomes an integral part of an artifacts source code repository. Through its integrated nature with other federated information systems, this unique ID is propagated to other relevant information systems it connects to, e.g. CMDB, EA repository, etc. Regarding concern C4, one expert argued that this is not an issue but only a matter of process change. In his opinion, agile teams have to know what business layer elements their application corresponds to. In case such assignments are incorrectly modelled, agile teams have to be made aware of this mistake and forced to correct it appropriately.

Technical Assessment

Question Q3: The following questions addresses technical aspects of the solution approach, with a focus on the configuration files. *"How do you assess the approach of binding static information to runtime models within configuration files?"*

Feedback: The general idea was well perceived by the practitioners. In particular, the possibility to link all business-related models to the *Application Component* was well accepted. A slight majority fully supported this idea as software artifacts become directly associated with business layer relationships instead of having this information maintained decoupled in an EA repository. Several interview partners came up with ideas how to extend the configuration file for realizing further use cases, like the analysis of cloud-ready applications. One Enterprise Architects stated, that the configuration file adds a high value for EAM as it allows to interpret, filter and report on *Application Components* recovered at runtime from a business layer perspective.

However, the following drawbacks were recognized during the creation of *microlyze.json*: 1) Most of the agile development teams suffer from missing knowledge about business layer assignments. Such knowledge would have to be build upfront to ensure data quality and correct assignments (concern C5). 2) Some experts stated that the maintenance of configuration files does not lead to real automated recovery of dependency information between business- and application layers (limitation L3).

Reflection: In order to solve concern C5, we suggest as future work to develop an additional frontend application that provides a form-support for facilitating the main-

tenance process for all stakeholders that are either unfamiliar with the JSON format or need support for the correct choice of business layer assignments (future work F3). Form-support could also achieve to reduce efforts by generating the *microlyze.json* file once per application and replicate it for each of its microservices. As a result, the risk of manual errors and inconsistencies is mitigated. Further improvements were proposed from some experts. They suggest binding the *microlyze.json* content directly to the runtime artifact e.g. as environment variables or tags. Doing so would allow to retrieve this information at any time from the artifact's runtime environment after deployment. Regarding the mentioned limitation L3, we admit that this solution approach is rather a semi-automatic recovery of dependency information. Manual input is still required.

Question Q4: With the following question we aim at receiving feedback regarding the concept described in Section 6.1.3. *"How do you assess the integration of references to federated information systems into the configuration files in order to retrieve further architecture-relevant information?"*

Feedback: The automated extraction of further architecture-relevant information from other data sources is regarded as an valuable approach. Experts highly valued the idea to include those references in the *microlyze.json* configuration file, as it represent static information that is easy to maintain. However, some experts complain, that for each new data source an adapter including GraphQL resolver must be written that exposes the additional information (concern C6). This lead to further implementation and maintenance efforts, in particular when the accessed REST APIs might change after an update. An interesting aspect was raised by one expert. The practitioner demanded to assess all tools along the application life-cycle whether they can be integrated into *microlyze.json* or not.

Reflection: The automated extraction of architecture-relevant data from federated information systems is not an easy task. There is not always a direct mapping between an *Application Component* and the counterpart models contained in other systems in many cases. For instance, during project portfolio planning it is not yet clear of what microservices an application will consist of. In CMDBs, *Application Components* might not be represented as well. Other tools might only register the parent application but not its microservices. Therefore an individual assessment is always necessary for a given tool. As future work, the mapping between the configuration files and federated information systems could be achieved on *Application Collaboration* level and not on microservice level (future work F4). However, in the worst case, this would lead to a one to many mapping which must be handled accordingly.

Question Q5: Besides the usage of configuration files to achieve the recovery of business-related EA models and the application mapping to referenced federated information systems, the correct moment when to trigger the recovery process is also crucial to assess. Hence, we asked the following question: *"To what extent, is the instrumentation of deployment pipelines to trigger the data collection process a practicable and reasonable approach?"*

Feedback: Most experts endorsed the idea to instrument CD pipelines for triggering the model maintenance process for two reasons: 1) It brings the process close to the actual knowledge carriers, i.e. the agile development teams, and 2) it ensures AM updates are performed at the most important points in time, which is the deployment of application releases. However, the Enterprise Architects also raised the concern that the solution could lead to a complex fragmentation of the overall model maintenance process, due to the decentralized approach (concern C7). In addition, they again claim that the solution does not consider important preceding EA phases like planning stages (see limitation L1).

Reflection: Regarding the concern C7, some experts proposed 1) to integrate the solution even deeper into the CD pipeline, by delivering the shared library as default when being included in the CD server image. 2) A standardized, team-overarching deployment pipeline that encompasses obligatory stages needed across the entire organization including the model exposure stage would enforce a standardized documentation process. This approach could also be enforced technically. However, they also admit, that this is currently not possible due to the numerous independent CI/CD servers that limit the solution's scalability.

Organizational Assessment

Question Q6: The following question tries to identify whether shifting the responsibility towards developer teams is considered the right approach: *"How do you assess the approach of shifting the documentation responsibility towards developer teams?"*

Feedback: All experts agreed that this approach is the correct way to implement the concept. The maintenance of AMs must be performed by the knowledge carriers, i.e. the agile teams. In the current state, this is not the case. One Enterprise Architect stated that most of EA modeling is currently covered by Domain Architects who have to gather information from agile teams in labor-intensive and error prone work. By shifting the model maintenance responsibility to the development teams, awareness is raised for the need of a proper understanding of business-related models and their relationships. Based on the feedback received from the development teams, the solutions preconditions are easy to integrate as part of regular sprint planning and sprint execution processes, described in section 6.2.2 which was inspired in exchange with the agile teams that adopted the solution. A Product Owner and a Developer appreciated that only a onetime effort is needed.

Reflection: By integrating the model maintenance endeavour into the agile teams natural development process and tool environment, it leads to a higher acceptance than imposing the development teams to use an EA repository for modeling. In particular, by making them responsible for documentation, the current gap between agile teams and EA teams might be reduced and leads to increased understanding at both sides. As a result, better collaboration could be enabled. In this sense, some Enterprise Architecture

experts see an intrinsic motivation with agile teams: They might quickly realize the benefit of the suggested solution as they can save time consumed by meetings, calls and email conversations which are currently needed to align manual EA modeling. Hence, they might integrate the solution on their own without pressure from Enterprise Architects.

Question Q7: For question Q7, we presented the integration effort evaluation covered in Section 7.3.5 and asked the following: *"How do you assess the effort it takes to get the tool operational and do you think it pays off quickly?"*:

Feedback: The majority of experts confirmed the manageability of the solution. The determined integration effort was regarded as feasible. Since the solution is easy to adopt with small amounts of onetime efforts, some experts argued that there is no need to fund a roll-out but only the implementation cost. Still, such a project could probably not be funded by the EA department on its own, as some experts stated (concern C8).

Achhammer estimates for this project with support of one involved Enterprise Architect an amortization period between four to five years, solely considering saved cost due to reduced manual modeling efforts (Achhammer, 2019). After presenting this result, some experts stated that this might constitute an issue as the organization usually demands a faster payback (concern C9). In parallel, they highlighted that the presented estimation did not include other value propositions beside time savings. Hence, the results are perceived as rather pessimistic.

Reflection: The payback of IT projects is always difficult to measure (Devaraj et al., 2001). For that reason, we are convinced that the success of this project is determined by the favour of several involved stakeholders which recognize an added value for their daily work. Beside the access to hidden knowledge, which is obviously a benefit for many users, we see the following motivation to realize this project: 1) Agile development teams and Domain Architects can save time for alignment meetings and workshops with the purpose of manual EA modeling. 2) Domain Architects need to spend less time in manual verification/rework of EA repository based reports due to higher data reliability.

7.3.6. Critical Reflection of Results

This case study particularly assesses our solution to extend the automated recovery and maintenance of EA models with business-related information. This information is provided via configuration files that are exposed by CD pipeline integration as detailed in Chapter 6. The practitioners gave us useful insights how *MICROLYZE* can be integrated into the case study environment in a meaningful way. Reflecting on the feedback of the case study outlined above, from our perspective, only minor adaptations were required, i.e. the major design decisions seem to be promising with respect to this specific use case.

In (Achhammer, 2019), Achhammer additionally sketches a preliminary, revised process which incorporates the most important experts' feedback. The process should overcome most of the concerns and limitations mentioned above. In the following, we summarize the additional prerequisites that need to be fulfilled prior to a productive roll-out or to be

caught up on as soon as possible.

1. **Performance impact assessment – C1:** Prior to roll-out, it is necessary to analyze the additional resources required for monitoring the IT landscape and the exposure of the runtime data.
2. **Reliability and data consistency – C2-C4, L2:** Further research must be conducted in order to resolve data consolidation and conflicts between overlapping data imports by different source systems. This corresponds to the concerns, that the extraction of additional architecture-relevant data might lead to mapping conflicts and duplicates that need to be handled appropriately. In addition, in the current state it remains unclear how automated EA model imports to EA repositories does not reduce the flexibility for manual modeling and data corrections.
3. **Software and operation documentation – C5, C6:** It is necessary, that all involved stakeholders adhere to the modelling rules for the configuration and the json schema validation files. In addition, developers must know how to extend the adapter pattern in order to support further referenced information systems. Hence, a proper software documentation and operation instructions is needed for *MICROLYZE*.
4. **Organizational acceptance – C8, C9:** The experts pointed out that it is mostly the involvement of all required key stakeholder that made the project a success and not the overcoming of technical hurdles. Hence, prior to roll-out, the benefits of *MICROLYZE* must be communicated transparently in order to achieve a common organizational acceptance of the solution.
5. **CD pipeline standardization – C7:** Prior to the roll-out, distributed CD servers should be consolidated to enable standardization of deployment pipelines which is a precondition to further reduce the pipeline integration efforts and allow technically effective process enforcement.

7.4. Interview Series

In order to answer our research question RQ6, we conducted interviews with 19 experts and analyzed the feedback according to the qualitative content analysis technique proposed by Philipp Mayring (Mayring, 2010). For the interviews, we initially compiled a questionnaire with open response format, in which we asked the experts to assess the system based on three subject areas 1) the overall solution architecture, 2) the model visualization approaches, and 3) the technical and organisational integration. Additionally, it was important for us to assess which further use cases can be addressed by *MICROLYZE* besides the automated recovery and management of models created along the IT value chain. Moreover, we also wanted to understand how a detailed project plan for integrating *MICROLYZE* into the organization could look like. For this purpose, we collect all required steps received from the experts and arranged them into a plan of action.

Prior to every interview, we presented our solution approach in detail with support of a Microsoft Powerpoint presentation. In this presentation, we discuss the problem description, our research goal, the overall solution concept, the technical prerequisites that need to be fulfilled as well as a live demo of our prototype. The compiled material is in textual and audio-recorded form, i.e. during the interview, we wrote a protocol to gather the main points and in parallel we recorded the conversation with the interviewee. Afterwards, we transcript the recorded material and finally merged the protocol with the created transcription.

147 interview invitations were sent by e-mail. This list of experts was compiled during EA research we performed with industry partners in recent years. We received 29 commitments for an interview (19,7% of the invitation pool). During a first phone call with all 29 interview partners in which we sketched a rough schedule of the interview 10 participants, i.e. 34,4%, dropped out due to schedule conflicts or because they did not feel skilled enough to assess the system and to answer all questions. As a result, we were able to analyze the interview material of 19 experts from 17 different companies and 10 different industries. Table 7.5 illustrates the distribution of the industry sectors of the organizations with Insurance as the largest sector followed by Automotive Supplier, Automotive Industry, Information Technology and Telecommunications. Table 7.6 shows the job title of the participants. The largest groups in our interview series consist of Enterprise Architects with 68,4% and IT Architects with 10,5%, as well as Software Architects with 10,5%.

Table 7.5.: Interview participants grouped by industry sector. N=19

Industry Sector	n	% of all
Insurance	5	26,3%
Automotive Supplier	3	15,8%
Automotive Industry	2	10,5%
Information Technology	2	10,5%
Telecommunication	2	10,5%
Finance	1	5,3%
Food Retailer	1	5,3%
Logistic	1	5,3%
Media	1	5,3%
Pharma	1	5,3%

Table 7.6.: Interview participants grouped by job title. N=19

Job Title	n	% of all
Enterprise Architect	13	68,4%
IT Architect	2	10,5%
Software Architect	2	10,5%
Global IT Architect	1	5,3%
IT Infrastructure Architect	1	5,3%

As a result, the concrete task for the analysis is to identify the main positive and negative aspects of the solution approach, as well as the concrete barriers that must be overcome in the process of integration. Based on this task and the available data material, we regard it obvious and appropriate to apply the interpretation technique of *Summary*, specifically the systematic categorisation of the material with inductive category formation (Mayring, 2010). The to be analysed three areas of system assessment can be understood as "deductive"

subject areas, which were given on the basis of the structure of the questionnaire. Inductive categories were developed for these subject areas in the course of the evaluation.

When we recall the flow chart of inductive category formation illustrated in Figure 1.4, the further procedure is to define categories (selection criterion), as well as the determination of the analysis units and the level of abstraction. In the course of the interview evaluation, we elaborated three categories. Category A describes all technical-related concerns of the solution concept, like technical limitations that prevent *MICROLYZE* to become fully operational. Category system B determines essential organizational aspects and decisive factors that slows down the integration of *MICROLYZE*. The final category system C summarizes all concerns and limitations that focus on the concrete prototype and the visual representations of the recovered models. Hence, all three deductive subject areas of the interview material was analyzed with regard to these three category definitions. As far as the level of abstraction is concerned, all concrete statements in the interview material to all three subject areas should be coded in inductive categories.

Specifically, inductive categories were developed from the interviews in the first run for all three subject areas and the frequencies with which the categories were mentioned were noted. During the revision of the categories in the second run after the analysis of about one third of all questionnaires, categories relating to a similar subject were combined and reduced. In the third phase, all inductively found categories within the three subject areas were then reassigned to upper categories at a higher level of abstraction. At the same time, the frequencies of the mentions of the individual categories were noted in order to evaluate them quantitatively afterwards. In addition, due to the numerous cross-references across all upper categories in a specific subject area, a further topic area "general assessment" was specified. The final result of the qualitative content analysis is detailed in the following Sections.

7.4.1. Assessment of the Solution Architecture

Overall Assessment

Overall, the experts agree that the employees will be relieved by the proposed concepts from keeping the EA models up-to-date and complete. Most experts, however, would not use it as a standalone application but rather as a data pipeline, that exposes the EA models to other tools, like CMDBs or EAM tools. That means, the data collection and model recovery process is still performed by *MICROLYZE*, but the model maintenance and visualization process should be performed by more advanced tools.

MICROLYZE forces the adoption of a specific organizational and technical structure, which follows the Archimate taxonomy (**S-B1**). It introduces an EA modelling standard. This structure is mostly designed in the EAM but challenging to convey in the whole organization. Via the decentralized and collaborative approach this challenge could be overcome. However, not all experts regard this point as an advantage. In case the organization has its own design and do not follow the architecture of Archimate, either the whole organization must be redesigned or the meta-model of *MICROLYZE*.

Even though *MICROLYZE* was primarily designed for recovering and managing models

Table 7.7.: Received feedback on the subject area of "Assessment of the solution architecture" consisting of code, name and frequency of the categories as well as allocation to the five elaborated upper categories.

Code	Category	Count
Solution Architecture (S)		
Overall Assessment		
S-B1	Forces the adoption of a specific organizational structure	9
S-A1	Too complex for heterogeneous IT landscapes	6
S-A2	No reconstruction of use cases and corresponding relationships	2
S-B2	Too much transparency leads to political issues	1
S-A3	No reconstruction of business processes and corresponding relationships	1
S-A4	Costly change in data sources due to read only access	1
Runtime Instrumentation		
S-A5	No full coverage of the IT landscape due to infrastructure diversity	8
S-A6	Lack of technical interpretation of information flows	2
S-A7	Analysis of huge amount of data	2
S-B3	Too fine granular data for strategic architecture roles	2
S-B4	Up-to-date models are not important for strategic architecture roles	2
S-A8	Strong focus of new technologies	1
S-A9	No recognition of removed communication and interfaces	1
S-B5	Misuse of monitoring tools lead to compliance and security issues	1
S-B6	Conflict with IT operation and service management	1
Configuration File		
S-B7	Additional documentation and integration effort	7
S-B8	Error-prone due to manual maintenance	5
S-B9	Duplicate work as information are maintained in other tools	2
S-A11	No support of legacy and monolithic applications	2
S-A12	No database-driven architecture	1
CD Pipeline		
S-A13	Not applicable for legacy and monolithic applications	3
S-A14	Tool diversity lead to high integration effort	3
JSON Schema Validation		
S-A15	No content driven validation lead to duplicate models	2
S-B10	Time-consuming schema updates	1

that are primarily reconstructed from microservice-based IT landscapes, the experts argue that the concept must also cover other applications in order to become practicable. Modern APM vendors do support a lot of different technologies. Still, it must be validated whether a complex heterogeneous IT landscapes can be fully instrumented (**S-A1**). This is certainly not the case for all legacy applications and obsolete technologies.

Two experts claim that *MICROLYZE* is not able to recover use cases (**S-A2**) and business processes (**S-A3**) and the corresponding relationships to other AMs. Use cases define the interaction between users and the systems to achieve a goal. A business process is a collection of related activities performed by users to produce a service or product for customers. Some experts align IT projects according to use cases and business processes with the goal to optimize the related workflow or transaction processing. Hence, both models and the according relationships are important to maintain.

Furthermore, an interesting aspect was raised by one Enterprise Architect. He stated that too much transparency about the IT landscape might also lead to displeasure by managers (**S-B2**). *MICROLYZE* might disclose bad performance in departments, which is due to bad management or "lazy" employees. This can be reflected by slow adoption of new technologies or slow execution of projects. For instance, the AM comparison view could indicate how fast the system is changing in a particular domain or department. Hence, due to the aforementioned reasons some managers might refuse to introduce and support *MICROLYZE*.

Finally, *MICROLYZE* represents a read-only system that is not able to update the information in the particular data sources. It only consumes the data and prepares them accordingly. If the information provided by the data sources is incorrect, which can be the case especially in the configuration files, then *MICROLYZE* recovers wrong models. The same applies for monitoring tools that require manual configuration to identify correct relationships in IT landscapes. If the data source deliver incorrect information and requires modification, this is only recognizable after *MICROLYZE* recovered the AMs. Hence, the necessary subsequent adaptation of the data sources causes an high effort (**S-A4**).

Runtime Instrumentation

Most of the interviewees regard the concept of recovering EA models based on the analysis of runtime data as a promising solution. One expert even defines it as the only available solution for keeping models up-to-date.

However, the interviewees also identified many challenges that must be overcome for productive use. Most experts doubt that monitoring tools are able to cover the whole IT landscape (**S-A5**). They primarily focus on new technologies (**S-A8**) and self-developed applications, in particular microservices. In every case it must be validated if the heterogeneous landscape is fully supported by the monitoring tool, which causes a high validation effort. If we assume that one organization provides a rather homogeneous IT landscape that is fully instrumented, then we have to set up a very high performance system that is able to analyze those huge amount of incoming monitoring data (**S-A7**). Two experts argue that runtime data are too fine granular for strategic architecture roles (**S-B3**) and up-to-date models are not important for strategic decisions anyway (**S-B4**). Hence,

the processing of runtime data and the subsequent aggregation to higher abstractions is complex and probably causes too much overhead. Moreover, one important information that is missing in runtime data is the technical interpretation of information flows (**S-A6**). It is only visible that two AMs are dependant on each other because they exchange data in general or due to deployment relationships, but it cannot be uncovered the rational of this data exchange or which specific data objects are transported. One expert correctly identified that removed communications cannot be recovered, because this information is not available in runtime data (**S-A9**). Hence, manual work is necessary which puts the actual purpose of the solution into perspective. Last but not least, one expert claim that the application of monitoring tools may cause conflicts with the IT operation and service management team (**S-B6**), because a misuse of monitoring tools for model maintenance purposes lead to higher administration overhead, as well as to compliance and security issues (**S-B5**).

Configuration File

The concept of the configuration files was also well received by the experts. It represents a simple way to provide business-related model information and the according relationship with other EA layers. The definitions of the models in business-related terms achieves a better understanding for the business world among the Developers. Especially, the decentralized approach of model maintenance was perceived as very useful as it leads to collaborative work. The references to other information systems opens up completely new possibilities and use cases for some experts. Even though the maintenance of configuration files causes less work for an individual, it represents a big advantage for the EA model maintenance endeavours and the reconstruction of the model life-cycle along the IT value chain. This was confirmed by most of the interviewees.

But still, the Developers must create the file and integrate it into the CD pipelines, what needs to be done during the actual work. Depending on the size of the instrumented IT landscape, this initial effort may represent for some departments a challenge (**S-B7**), even if the maintenance effort becomes less after the integration is accomplished. In addition, Developers and Product Owners may perceive this task as duplicate work as specific information must already be maintained in project management tools, wikis and the like (**S-B9**). All in all, one justified criticism raised by five experts is the fact, that the maintenance of configuration files are still performed manually, which does not solve the original problem. The effort is only distributed among many. Hence, the provided information might still be error-prone and outdated (**S-B8**). Moreover, the configuration files represent physical data that are stored in the particular application paths and are processed via a CD pipeline. Even though *MICROLYZE* focuses on microservice-based IT landscapes, it must be determined whether legacy applications and cloud-based applications are also supported by this concept and if not which changes must be applied (**S-A11**). One expert proposed to change the concept towards a database-driven architecture, i.e. all maintained information are stored in a database and are frequently validated before each application deployment (**S-A12**). In this sense, it must be figured out how the validation process can be performed in detail.

Continuous Delivery Pipeline Integration

As most of our experts had the Enterprise Architecture role, they were not able to provide much feedback regarding the usage of CD pipelines. In general, it was perceived as a reasonable approach to identify change events and to trigger the model update process.

Two points of criticism were expressed. First, CD pipelines are not used for every development process (**S-A13**). Most organizations follow a liberal approach for application development, i.e. the Product Owner and development teams can decide how to develop and operate applications. This also leads to a technology diversity according to CD pipelines, which in turn, increases the effort of configuration file processing and JSON schema validation (**S-A14**).

JSON Schema Validation

Every additional validation of manual work is important and required to keep the quality of EA models high. For that reason, most experts confirm the necessity of JSON schema validation. It specifies a concrete structure of the IT landscape that every Developer must adhere. In addition, it supports the exchange between Developers and Enterprise Architects.

However, the validation of the configuration files against a JSON schema does not achieve a content validation (**S-A15**). Spelling mistakes and wrong information are processed without verification. This leads to duplicated models and incorrect model assignments in the worst case. Furthermore, a change in the JSON schema must be rolled out in the whole organization, which leads to high update efforts (**S-B10**). In addition, the updates are not carried out in time, but are first scheduled in the sprint backlogs with probably low priority.

7.4.2. Assessment of Model Visualizations

Overall Model Visualization Assessment

All experts agree that the developed visualizations are an appropriate initial approach to demonstrate the flexibility of the graph-based representation of the maintained models. Especially, the different ways how to visualize dependency structures and the aggregation possibilities found approval. However, there are still several points that need to be addressed in order to improve the prototype.

First of all, all views misses important features like aggregation, filter and display of a specific section (**V-C1**). That was claimed by eight experts. The integration of those features would also address the recognized scalability issues, which is a problem with all diagrams that have to handle huge amount of data (**V-C3**).

In addition, *MICROLYZE* is able to offer service for many different user roles, however, the prototype has no clear focus according to the developed visualizations (**V-C2**). As stated above, the *Model Deployment*, *Model Interaction*, and *Model Communication* views are rather important for Solution Architects but unusable for Domain- and Enterprise Architects in the current form. A different entry into the ETG, as well as a higher aggregation level

Table 7.8.: Received feedback on the subject area of "Assessment of the visualization approaches" consisting of code, name and frequency of the categories as well as allocation to the seven elaborated upper categories.

Code	Category	Count
Visualizations (V)		
Overall Model Visualization Assessment		
V-C1	Missing important features like aggregation and filter for all views	8
V-C2	No clear focus on EA layers and different roles	6
V-C3	Scalability issues	5
V-C4	No visualization of important KPIs	5
V-C5	No manual adoption of the model visualization	4
V-C6	Only as-is representation of the IT landscape without target states	1
Model Cluster View		
V-C7	Representation of too much information	5
V-C8	Challenging visualization of many to many relationships	1
V-C9	No clear focus on business context	1
V-B11	Vertical ordering of the IT models might lead to displeasure	1
Model Deployment View		
V-B12	Important for technical roles - no business-related model definitions	7
V-C10	Representation of too much information	7
V-C11	No selection of different entries	5
Model Table View		
V-C12	No export to Excel format	3
V-C13	No data analysis possibilities like sum, count, etc.	1
V-C14	No differentiation between last seen of a AM and data traffic	1
Model Communication View		
V-C15	No interface description	3
V-C16	No transparency about data object exchange on business level	2
V-C17	Representation of too much information	2
Model Interaction View		
V-B13	Only important for technical roles	3
V-C18	No visualization of non-functional requirements	2
V-C19	No selection of different entries	2
V-B14	Aggregation level is too low for business roles	1
V-C20	No highlighting of long request processing paths	1
Model Comparison View		
V-C21	No comparison in other EA layers	5
V-C22	No representation of target plans	2
V-C23	No delta visualization	2
V-C24	No visualization of change impacts on business processes	1
V-C25	Missing information about detailed changes	1
V-C26	Comparison must be based on same data quality level	1

would also make those visualizations usable for Domain- and Enterprise Architects. At the same time the *Model Cluster*, *Table* and *Model Comparison* views are highly relevant for *Enterprise Architects*. Hence, the experts propose to elaborate clear uses cases for every role and create visualizations with an adequate aggregation level for each EA layer accordingly.

The representation of static and runtime information in each model was considered as an advantage over other solutions. However, the experts misses further important KPIs like the incurred costs, especially in cloud environments, runtime status of each model, operation within the defined SLAs, or the number of users and transactions that are currently processed (V-C4).

Furthermore, even though the visualization framework is able to reconstruct the ETG always in the same form in order to improve the recognition value for the users, some experts still wanted to modify the node and edge positions manually (V-C5). Besides that, it should also be possible to save the new positions.

Last but not least, the views only represent the IT landscape in its current state, but do not incorporate planned or target states (V-C6). If *MICROLYZE* could be evolved in this direction that it could also manage different architecture states, this would be an big improvement, especially for Enterprise Architects.

Model Cluster View

Many experts confirm that the *Model Cluster* view represents an important entry to visualize the link between the business and IT world. It creates an overview about the whole IT landscape. In particular, the folder-based structure to navigate into the business and IT layers is considered as valuable.

However, it still has some drawbacks that need to be improved. First of all, the view displays too much information at once, as five experts stated (V-C7). The experts would prefer a more focused business context that shows, for instance, only the models of one selected domain and hides information from other domains (V-C9). In addition, many to many relationships between business, application or infrastructure models lead to duplicated nodes, which created confusion among the users (V-C8). Hence, even though the folder structure is regarded as useful, an improvement would be to zoom into a selected model and only display those information that represent child models from a hierarchical perspective.

We apply an automatic layouting of models that does not change after a page refresh. Hence, there is no priority of those models that are positioned in the first lines. However, one expert stated that the vertical ordering of the models may lead to displeasure by other users (V-B11). They might feel disadvantaged because their domains are not positioned in the first place. The possibility to move the models manually would solve this problem.

Missing information is *Use Cases*, *Business Capabilities* and the according relationships, the display of the status of the applications and hosts like run, stopped, crashed and the like, as well as usability functions like model search, filter, export and the possibility to save the displayed IT landscape section as a snapshot.

Model Deployment View

The *Model Deployment* view was rated as the less valuable visualization of AM relationships. It displays way too much information in the current form and is only useful for applications that are not distributed a lot, which is mostly not the case for microservice architectures (V-C10). In addition, this model representation has a strong focus on technical roles (V-B12). Deployment relationships on a more business-oriented perspective would highlight important cost aspects for *Business Services, Products, or Domains*. Hence, the selection of different AMs as the parent node would represent a big improvement (V-C11).

An additional improvement would be the enhancement of grouped nodes with technical-, and business-related KPIs. For instance, the experts seek answers to the following questions: How much time does my application consume on the cloud platform? What data volume and storage space does it require? What costs does the system incur? How many requests does the system process? Where are the outliers that put heavy strain on the budget?

Model Table View

The table-based representation of the AMs was regarded as the most valuable visualization for hierarchical dependencies. The experts confirmed that tables are especially useful for displaying many objects of the same type and level. The hierarchical ordering of the AMs via tabs and the presentation of relationships via setting filters was also well accepted. A missing feature that was raised by three experts is the export of the table into CSV or Microsoft Excel format (V-C12). Furthermore, the table does not provide any data analysis capabilities like sum, count, aggregation etc. (V-C13) An important information that is missing in the table is the differentiation between last seen of a AM and last seen of data traffic (V-C14). With this additional timestamp it is possible to identify microservices that still run in the system and consume resources, but are not used anymore since there is no data traffic recorded. Hence, it is an indication for removing this microservice from the IT landscape and free additional system resources.

Model Communication View

Regarding the *Model Communication* view, we received good feedback. The experts considered it as a good approach to reveal transparency about the communication behavior of a whole business area, as well as about the dynamics of change in an area over a time dimension. Also the automatic positioning of the nodes that do not change after a page refresh was considered beneficial. It provides a high recognition value.

Some improvements were also mentioned: First of all, the communication path does not provide a detail description of the used interfaces (V-C15). This was criticized by three experts. In addition, the communication path should be enhanced with information about the data object that is exchanged between AMs (V-C16). Last but not least, the visualization also face scalability issues (V-C17). If too many nodes and edges are displayed, the visualization becomes complex and unreadable. Hence, filter, search and aggregation functionality to prevent information overload was demanded.

Model Interaction View

The *Model Interaction* view was well appreciated by the experts. The representation of the transaction flow through the system and all applications that contribute to process the transaction addresses several important use cases. For instance, the experts recognized that it uncovers optimization potential like bottleneck identification, long running processes and root cause analysis. Even though, these use cases are intensively addressed by APM tools, *MICROLYZE* is able to display the transaction flow on a higher aggregation level, which opens up further analysis possibilities.

The *Model Interaction* view was also considered as important for rather technical roles like Solution Architects (**V-B13**). In general, the displayed aggregation is too low and the presented data too fine-grained for business roles (**V-B14**). The possibility for a higher aggregation of the transaction flow was requested several times. One expert also proposed to display the processing flow of predefined use cases and not only single transactions (**V-C19**). In addition, the view does not highlight long path of request processing (**V-C20**). This would unveil hidden potential for performance optimization. Last but not least, two experts suggest to enhance the runtime data with KPIs that measure the coverage of non-functional requirements (**V-C18**).

Model Comparison View

The last view, *Model Comparison* discovers the areas where a lot of development work is conducted. As the expert stated, this information is important for gaining transparency about the dynamics of change of an specific area. Especially Enterprise Architects perceived the *Model Comparison* view as important to assess whether the IT landscape evolves into the desired direction.

However, the visualization in its current version lacks in providing detailed information about changes, like on which level the change was performed (architecture-based, functional-, non-functional-based, bug fixes, etc.), connection to the backlog, and responsibilities (**V-C25**). Furthermore, it remains unclear what impacts the changes have on the business processes, or the processed use cases (**V-C24**). One expert argued that the comparison must be based on the same data quality level (**V-C26**). If particular AM's relationships actually exist in both architecture states, but were not recovered in the obsolete architecture due to any reasons, then one can draw false conclusions from the comparison. Transparency about the dynamics of changes of an specific area can also be abused to assess the performance of developer teams, or whole departments. On management level this would lead to displeasure among the stakeholders as one expert worried.

In addition, what could be improved is the integration of planned states of the IT landscape (**V-C22**) and to display the delta of both architecture states, i.e. the as-is versus as-planned (**V-C23**). This delta would support the monitoring and the assessment of a project progress and to point out critical differences. Last but not least, the view should cover the comparison of all models from every EA layer and not only the business layer (**V-C21**). This is definitely achievable with the graph-based representation of the IT landscape but was not realized due to the high additional development effort.

Table 7.9.: Received feedback on the subject area of "Assessment of the technical and organizational integration" consisting of code, name and frequency of the categories as well as allocation to the three elaborated upper categories.

Code	Category	Count
Technical and Organizational Integration (I)		
Technical Barriers		
I-A16	Instrumentation of legacy and cloud-based applications	10
I-A17	Interface and meta-model support of different monitoring tools	9
I-A18	Organization-wide rollout of APM tools	8
I-A19	Organization-wide rollout of CD pipelines	8
I-A20	Storage of data beyond national borders and restricted areas	6
I-A21	Instrumentation of applications in closed networks	2
I-A22	High network administration effort due to higher network load	1
Organizational Barriers		
I-B16	Adaption of the development process	8
I-B17	Complete preliminary definition of business layer models	4
I-B18	Involvement of many stakeholders	3
I-B19	High initial effort in case not all prerequisites are fulfilled	2
I-B20	Necessary training of employees	2
I-B21	Management guidelines restrict the technology stack	1
I-B22	Convincement of management that EAM requires runtime data	1
Delegation of Documentation Responsibility		
I-B23	Low self-motivation lead to bad data quality	6
I-B24	High heterogeneity of internal and external developers	2
I-B25	High fluctuation of employees	2
I-B26	High task load of developers	2
I-B27	Frequent change of responsibilities	1

7.4.3. Technical and Organizational Integration

Technical Barriers

Many experts recognized a technical barrier towards the instrumentation of legacy- and cloud based applications (**I-A16**). Many legacy applications are not supported by APM tools or do not even provide any interfaces for runtime instrumentation. Cloud platforms offer their own monitoring technologies or provide interfaces for integrating third-party monitoring tools. In both cases, it must be validated technically if the cloud platform fulfills all requirements. It is also challenging to add configuration files to cloud-native applications, as they run in encapsulated environments. Furthermore, each monitoring tool has its own meta-model for analyzing the IT landscape. If this meta-model is currently

not supported by *MICROLYZE* then an additional meta-model transformation must be written in order to translate the meta-model into the Archimate taxonomy (**I-A17**).

The experts claimed that their IT landscape is not fully instrumented by APM tools and not all departments use continuous delivery or agile methodologies. There exist many gaps. Hence, in order to make *MICROLYZE* fully operational, an organization-wide roll-out of APM tools and CD pipelines is required, which might meet refusal by many managers (**I-A18**, **I-A19**). The most stated concern is the cost aspect. Dynatrace, for instance, charges 4 Cent per host per hour for self-service agents. According to our case study described in Section 7.2, we recovered 5.805 hosts. A simple projection would mean that this IT landscape cost over 167.000 EUR per month. We do not have any information about the actual conditions agreed upon regarding the license costs.

Moreover, six experts argued that the central processing of runtime data and the according maintenance of the recovered models is not possible beyond national borders and restricted areas (**I-A20**). The organizations must obey the laws of the country they are located in. Due to data privacy regulations and other laws, it is not allowed to transfer data across national borders in certain countries (Nigel, 2017). A similar technical barrier was identified in closed networks (**I-A21**). They present not a regulatory but a technical barrier. In both cases, an automated collection of distributed runtime data and configuration file contents, as well as a central processing is impossible.

Last but not least, one expert had concerns that the processing of huge amount of runtime data lead to additional network overhead (**I-A22**). This overhead lead to higher administration cost and eventually to performance issues in peak times.

Organizational Barriers

According to organizational barriers, many experts stated that an adaption of the development process is unavoidable in order to integrate the concept (**I-B16**). The maintenance and validation of the configuration file must become a central task in the deployment process. Without a successful validation against the JSON schema the application should not be allowed to deploy to production. However, this deployment process follows agile practices, which is not conducted in all areas of the organization. This also requires a necessary training of the employees, especially for Developers and Product Owners (**I-B20**). First, they must understand how to apply the agile development process. Second, they must understand the structure and features of the configuration file and third, they must understand the Archimate taxonomy and how it is applied in their own organization.

However, before *MICROLYZE* can become fully operational certain prerequisites must be fulfilled that also represent organizational barriers. The fulfillment of the technical prerequisites already requires high initial efforts in case the technical prerequisites are currently not completely fulfilled (**I-B19**). This barrier becomes even worse in case management guidelines restrict the usage of the required technology stack (**I-B21**). In addition, it is important to have a clear, uniform and complete definition of the business layer and the according models (**I-B17**). Without an available definition of the business layer structure the configuration files cannot be maintained properly. However, some experts admitted that their organization is still working on the business layer definition. Hence,

MICROLYZE would put too much pressure on the business which might lead to a refusal of the system.

All in all, the concept of *MICROLYZE* suggest to delegate several task to Developers and Product Owners in order to achieve a decentralized model maintenance process. Some experts see this critically as too many stakeholders must be involved in the model management (**I-B18**). Indeed there are many supporters who recognize the advantages of this concept. Enterprise Architects and Solution Architects were mentioned frequently as the biggest beneficiaries. But there exist also blockers who fear to have even more work to do, or might not see the benefit of *MICROLYZE*. The experts mentioned Developers, Product Owners, Security and Compliance, Business Owners, and the IT Operation as the potential blockers. Those blockers must somehow convinced to become supporter of *MICROLYZE* which seems to be challenging according to the expert's feedback (**I-B22**).

Delegation of Documentation Responsibility

Most experts agreed that a decentralized approach to maintain EA models is the correct way. We recorded many statements similar to "*the documentation must be made where the data is generated.*" A central approach where one department, mostly the Enterprise Architects, are responsible to carry out the documentation for the whole organization lead to incomplete and outdated models. This was also confirmed by many researchers (Armour et al., 2005; Farwick et al., 2011a,b; K. Winter et al., 2010).

However, a decentralized approach for model maintenance leads to several challenges. Many experts admit that the developer teams have less motivation for model maintenance, especially in keeping the documentation up-to-date that is mostly required by other roles (**I-B23**). Hence, the risk is high that the content of the configuration file is maintained in the beginning and not touched anymore afterwards. It must be set up a strong governance, as well as penalty measures to ensure the model maintenance is carried out frequently. This is very difficult to enforce and also not wanted, because the developer teams should not be controlled too much in order to remain they agility. In addition, due to high workload, many developers complain that there is mostly no time left to update the documentation for other roles (**I-B26**). As a consequence, especially architecture roles have to work with obsolete documentations. To summarize, even though a decentralized approach has already been introduced in the organizations of many interviewees, they still need to validate the provided documentation and fill many gaps manually.

Furthermore, many experts complain a high heterogeneity of internal and external developers (**I-B24**). Especially in big organizations application development is mostly outsourced, even though many experts state that the development process is increasingly shifting towards internal development. This leads to an unstable composition of the developer teams. This problem is further exacerbated due to a high fluctuation of internal employees (**I-B25**) and frequent changes of responsibilities (**I-B27**). As a result, the knowledge how to maintain the models and how the business is structured in detail, as well as the rational behind this approach must be passed over and over again. The experts have concerns that due to this instability of development teams the model maintenance quality will suffer in the long run.

7.4.4. Supported Use Cases

Besides the management and the automated recovery of models, the experts identified further important use cases that can be addressed with *MICROLYZE*. Those use cases can also be regarded as future work, that complement the list with future work identified in the previous case studies. In the following, we discuss those use cases in more detail:

- **F1 – Application life-cycle management:** Four experts identified *MICROLYZE* as an instrument to support the application life-cycle management (Keuper et al., 2011). *MICROLYZE* uncovers the version number of technologies and how long the application or infrastructure components are part of the IT landscape. This would give a hint to replace the technology with a newer version. In addition, based on the communication information, *MICROLYZE* helps to disclose obsolete applications that are not used anymore and, hence, can be removed from the IT landscape. This information is often not available as some experts admit.
- **F2 – IT transformation planning and controlling:** *MICROLYZE* uncovers the as-is IT landscape, which represents an important baseline for identifying problem areas and supports the planning in which direction the IT landscape should be optimized and evolved. Hence, it provides important information for elaborating IT transformation plans and EA roadmaps. In addition, since *MICROLYZE* stores every change applied to the IT landscape in the form of revisions (cf. Section 5.2.5), one can use this information to control the IT transformation and intervene if unwanted changes are recognized.
- **F3 – Support of requirements analysis:** Similar to I-UC2, the identification of problem areas by analyzing the as-is IT landscape recovered by *MICROLYZE* makes also room for supporting requirements analysis (Kotonya et al., 1998). This use case was proposed by two experts.
- **F4 – Cloud migration planning and controlling:** According to cloud migration endeavours, the experts stated that it is challenging to identify which applications are ready to be deployed to the cloud. This accounts especially for distributed microservices that establish a big communication network. It is important to analyze which services form a conglomerate of dependant elements and therefore must be migrated as a whole (Gholami et al., 2016). *MICROLYZE* supports this analysis by recovering the communication relationships and the related dependencies to the business layer.
- **F5 – Failure impact analysis:** According to ITIL v3 Service Management (Office, 2011b), the component failure impact analysis (CFIA) can be used to identify what impacts failure have to business operations and users. The result of the analysis emphasize where additional resilience should be considered to prevent or minimize the impact of failures. In general, the impact assessment is mainly conducted via a CFIA matrix, which represents the dependencies of all IT landscape components including business processes and users. *MICROLYZE* could support the CFIA by

automating the recovery of component dependencies and the according visualization of failure impact paths.

- **F6 – Service catalog management:** The service catalogue describes in a formal way the available services that the organization have to provide (Office, 2011a). The catalogue contains the respective Service Level Agreements (SLA) that should be met, setting expectations between clients and providers of services. In a study (Cole, 2008) conducted with 100 companies that tried to implement a service catalogue, 12% reported that the project was unsuccessful. 34% of those companies mentioned service definition as one of the "top risks" for successful catalogue implementation. According to one expert, *MICROLYZE* could support the implementation of a service catalog by recovering all currently running services, their states, and their runtime behavior.

7.4.5. Action Plan

We asked the experts what steps they would perform to introduce *MICROLYZE* into their organization. Based on the received responses we elaborated a general plan of action that is accepted by most of the experts. Interesting, the experts agreed that it is mostly the involvement of all required key people that made the project a success and not the overcoming of technical hurdles. The elaborated plan of action is detailed in the following:

1. **Validation of prerequisites fulfillment:** First of all, a first validation is required to assess to what extent the prerequisites of *MICROLYZE* can be fulfilled in the target organization. If the IT landscape primarily consists of legacy applications that are developed based on a waterfall approach, then *MICROLYZE* is not the right tool to recover and to manage the EA models. However, if the organization has a huge microservice-based IT landscape and the development process mainly follows the agile principles, then the project responsible can proceed with the next step.
2. **Development of a first Proof of Concept (PoC):** The next step covers the development of a first PoC to validate whether *MICROLYZE* satisfies the required aspects and solves the problems it was designed for. For this task, the project responsible should seek for an appropriate area in which most of the technical obstacles are not present. Maybe the own developer team can be used to accomplish this task. The project responsible should calculate important KPIs, that emphasize the advantages of the system. These are among others 1) time needed for integration, 2) costs that are likely to be incurred, 3) model recovery ration, that means the number of AMs that can be recovered automatically divided through all existing AMs in this area.
3. **Presentation of the PoC to the management:** Afterwards, the result of step two must be presented to the management. According to the experts, all benefits and drawbacks of *MICROLYZE* should be listed transparently. The cost factor is an important aspect that needs to be considered, but also the potential cost savings including employee relief. Only if the management can be convinced of the solution, the project can be continued with the next steps.

4. **Validation of required stakeholders and responsibilities:** Next, as detailed in Section 6.2.1 many different stakeholders are required to bring this solution in operation and must integrate it into their daily work. We identified the roles Enterprise Architects, Domain Architects, Product Owners and Development Team. It must be validated whether those roles are present in the target organization and actually performs the necessary tasks. After the correct roles are found, the persons behind those roles must be identified and contacted accordingly.
5. **Performing motivational work:** Now, the most challenging part comes according to the experts. All identified stakeholders must be convinced to contribute to the project. Having the management on his side is a first important step, however enforcing the people to contribute is not seen as the right way, as this would certainly compromise the quality of the project. Hence, incentives must be found for the stakeholders to use the system out of their own conviction. Otherwise, the performance of EA model maintenance endeavours still remains low. However, this step also encompass the validation to what extent the specific business area of the target stakeholders can be supported by *MICROLYZE*. That means, the area fulfills the technical prerequisites and applies an agile development process. If this is not the case at all then any motivational work is pointless.
6. **(Optional) Introduction of an uniform definition of the business-layer models:** A complete definition of the business-layer models and their relationships is required by *MICROLYZE*. This can only be provided by organizations with a certain level of maturity as one expert stated. Without a clear business structure the configuration files cannot be filled with the required information and *MICROLYZE* is not able to assign the recovered models to the business layer. In case the business layer models are not defined yet, then this should be made up in this step.
7. **(Optional) Rollout of APM tools and CD pipelines:** A further optional step is the rollout of required technologies encompassing APM tools and CD pipelines. As already stated by many experts, not all departments have APM tools and CD pipelines in service.
8. **MICROLYZE integration and definition of the JSON schema:** All the aforementioned steps are preliminary work. If the organization arrives this point, *MICROLYZE* can be finally integrated into the IT landscape. This step covers the following tasks: 1) connecting *MICROLYZE* to the APM tool, 2) creating the configuration files and adding them to the application repositories, 3) defining the JSON schema file, 4) integrating the configuration file validation to the CD pipeline, 5) fixing interfaces and wrong model transformations and 6) adjusting the development process to ensure continuous configuration file maintenance. The experts agree that *MICROLYZE* should be rolled out step by step.
9. **Training of Developers and Product Owners:** Last but not least, Developers and Product Owners must be trained to use and maintain the system. This step is important, especially in organizations with a high staff turnover.

8. Conclusion

In the last chapter of this thesis, we aim at summarizing the thesis' content, reflect on the proposed solution design based on the research questions raised in Section 1.2, reveal known limitations, and give an overview of further research.

8.1. Summary

The thesis at hand starts with the motivation that newly developed IT services leave a digital twin, i.e. a *model* in each phase of the IT value chain that is backed by a particular collaboration tool. The management of those models serve as an important information source to establish a knowledge management and an efficient continuous feedback loop in every model life-cycle phase. However, the problem description in Section 1.1 highlights that an efficient model management is overshadowed by numerous issues: 1) Model management is still conducted manually, which is error-prone and time-consuming. 2) The extraction of models from the tools used along the IT value chain lead to an ambiguous documentation of the architecture. 3) Microservice architectures introduce a high level of complexity with regard to model management. Based on the problem description, Section 1.2 elaborates on the formulation of concrete research questions, while Section 1.3 discusses the adoption of the design science research methodology to the context of this thesis. Sections 1.4, 1.5 and 1.6 summarize the thesis' core contributions, conducted preliminary work on this topic and the documentation structure respectively.

Our solution approach is based on the foundation detailed in Chapter 2. We introduce in Section 2.1 the concepts of model-driven engineering that handles models and their transformations as primary artifacts to evolve software systems. EAM as our problem domain in which we observe challenges concerning the management of models in microservice-based IT landscapes is described in Section 2.2. Hereby, we detail in particular IT-landscape modelling as an important subarea of EAM to recover and to maintain the models of an organization. Microservice architectures represent the frame of our research and introduce a high level of complexity with regard to model management. Therefore, the Section 2.3 requires high attention. Finally, Section 2.4 summarizes characterizing features of DevOps, which aims to shorten the software development life-cycle and covers many concepts that we leverage to elaborate our solution including agile practices, continuous delivery and monitoring.

We summarize in Chapter 3 the state-of-the-art which is related to our solution approach. In this scope, we address related literature on behalf of model-driven reverse engineering (cf. Section 3.1), EA model maintenance (cf. Section 3.2), and automated creation of IT landscape visualization (cf. Section 3.3). Finally, we detail in Section 3.4, to what extent

our approach differs from the related work.

In Chapter 4, we describe how we derived the requirements for the conceptual and technical design of our approach to automate the recovery of EA models by processing runtime data. To this end, in Section 4.1 we first explain the different aggregate states of models covering the instantiation, specialization and execution phase, and how those phases are interconnected along the IT value stream. Afterwards, we sketch the respective conceptual framework of our approach to manage models which captures the process design, storage design and reference design respectively. We further identify different roles of users interacting with the models in each life-cycle phase. Based on our conceptual framework as well as on related work on model reverse engineering, EA model maintenance and IT landscape representation, we describe in Section 4.2 the systematic derivation of 21 requirements categorized along the domains architectural, organizational, functional and visualization requirements.

Based on the derived requirements, we outline the concrete system design of *MICROLYZE* and the core activities for recovering technical models via runtime instrumentation in Chapter 5. Consequently, we analyze in Section 5.1 exposed IT landscape meta-models of modern APM tools in order to understand what specific EA models can be recovered and which remain hidden. Based on those findings, we develop in Section 5.2 the meta-model and the main components of our system design. Further on, we detail the processes on how to recover EA models and the corresponding relationships in Section 5.3. This section covers the most important algorithms, that we created to recover EA models. We continue in Section 5.4 to describe our model visualization framework and detail the implementation of several IT landscape views that represent models in different perspectives.

Since Chapter 5 outlines the highlights of the implementation of the prototype empowering the automated reconstruction of technical EA models via runtime instrumentation, we focus in Chapter 6 on recovering business-related models and the extraction of further architecture-relevant information from federated information systems that are used along the IT value chain. In this scope, we introduce in Section 6.1 the concept of configuration files that contain business-related model information and detail how we intend to integrate those files into the continuous delivery pipeline. In Section 6.2, we elaborate on the organizational design and summarize which stakeholders must be involved. We focus on technical processes in Section 6.3, that must be performed to import the content of the configuration file to *MICROLYZE*. Finally, we describe in Section 6.4 approaches how this additional application- and business layer relationship information can be represented visually.

The prototypical implementation serves as a proof of concept on the one hand, and enables the evaluation of the developed concepts as described in Chapter 7 on the other hand. Thereby, we applied different evaluation strategies to validate different aspects of the prototype. In Section 7.2, we evaluate the concepts described in Chapter 5 in an automotive setting. Section 7.3 outlines the setting and key findings of the evaluation in an insurance environment. Hereby, we primarily assess the solution design detailed in Chapter 6. Finally, Section 7.4 summarizes the result of 19 interviews with practitioners

from 17 different companies that provide overall feedback about the prototype.

After summarizing the thesis and its chapters, we assess its results with respect to the research questions raised in Section 1.2. In this scope, we provide a brief answer to the particular research question and refer to specific sections of the thesis that provide more details.

Research Question 1 (RQ1): *How can a system and a process design look like that automatically reverse engineers models from runtime data?*

The answer of the first research question is primarily covered by Chapter 5. First, we analyzed in Section 5.1 four different APM tools and evaluated based on their provided meta-model what specific information they can deliver. The knowledge gained helped us to understand what models of an IT landscape can be reconstructed at most. Afterwards, we elaborated the required model-transformation process that translates the runtime data into our desired meta-model. In this scope, we described our system design that primarily consist of four main backend components (see Section 5.2), i.e. 1) *MICROLYZE.Collect*, that consumes the runtime data, extracts the EA models and transforms the models in our elaborated meta-model, 2) *MICROLYZE.Analyze* extracts communication relationships between application models, identifies the used interface and determines communication deletion thresholds, 3) *MICROLYZE.Store* is responsible to persist the EA models in the elaborated meta-model and finally 4) *MICROLYZE.Expose* provides an interface to enable users to query the recovered EA models.

In this scope, we further elaborated in Section 5.3 two algorithms the *Backward Recovery Algorithm* and the *Forward Recovery Algorithm* that first reconstructs an incomplete IT landscape by analyzing past runtime data and afterwards continuously refined the EA model architecture by processing newly incoming runtime data. As removed communication relationship poses a challenge as this information is not available in runtime data, we elaborated a concept to detect removed communications by introducing the deletion threshold approach.

Research Question 2 (RQ2): *How to recover business-related models and how to establish a correct assignment of those models to technical layers?*

In Section 6.1, we elaborate on a concept of how to enhance the developed meta-model with further business-related information that cannot be extracted out of runtime data. In this scope, we propose the creation of a configuration file that must be manually maintained and assigned to each deployed microservice. This file contains all required information for establishing the relationship between business- and application layer. In addition, we present a concept in Section 6.1.3 to enhance the configuration file with references to other federated information systems that are used to evolve models along the IT value chain. The detailed workflow for processing all information delivered by the configuration file is detailed in Section 6.3.

Research Question 3 (RQ3): *How can a meta-model of the EA knowledge graph look like that represents the models from all EA layers and what relationship types need to be defined?*

In order to answer research question three, we introduce the concept of *enterprise topology graph* in Section 5.2.5. The purpose of the ETG is to capture the whole architecture of an IT landscape in a graph-based form. The nodes in the graph define AMs and the edges

the logical, functional, and physical relationships between the models. The ETG is the final model manifestation after a chain of model transformations. First, the APM server itself transforms the runtime data into their specific meta-model. Second, we analyzed those meta-models in Section 5.1 and transform the entities into the Archimate taxonomy, which represents our basis for visualizing the IT landscape. Third, we enhance our meta-model with further business-related models provided by configuration files. Finally, we transform this meta-model into a graph-based representation. In Section 5.1.5, we discuss the relationship types between the AMs. We specify them into *hierarchy*, *grouping* and *communication* relationships.

Research Question 4 (RQ4): *How and where to integrate the concept in the software development process and which stakeholders must be involved?*

With the extraction of architecture-relevant information from different sources, we follow a decentralized setup. The model extraction out of runtime data is performed continuously without human intervention. However, the processing of the distributed configuration files must be embedded into the agile development process of every SCRUM team. We suggest to involve the model management into the regular sprint planning, as stated in Section 6.2.2. In this scope, the CD pipelines used to release new application versions can be leveraged to validate the content of the configuration files and to forward it to a central model management.

Several stakeholders must be involved in the proposed concept, as described in Section 6.2.1. We cluster them into the **Enabler**, **Worker** and **Beneficiary** group. The interplay of all those roles and how they are allocated into the overall solution concept is described in Section 6.2.2.

Research Question 5 (RQ5): *How can stakeholders be supported in understanding and exploring the EA knowledge graph?*

We introduce in Section 5.2.6 GraphQL as the query language for requesting the recovered models and their relationships. This language adhere to the meta-model taxonomy detailed in Section 5.1. It enables users to select, search, filter, analyze and to modify the ETG. In addition to simple requesting models, we develop methods that enable users to retrieve runtime metrics for every technical model from the exposed APM server APIs. For this purpose, we add additional fields to the GraphQL schema that point to the related methods in the backend. This approach uncovers behavioral aspects of models and, hence, represents the models@run.time. Furthermore, we integrate the model revision concept into the GraphQL language. Based on revisions, we enable users to analyze the emerging behavior of an IT landscape, or compare different architecture states. Finally, we discuss in Section 5.4 and 6.4 visualization approaches that illustrate the IT landscape in different graph-based representations.

Research Question 6 (RQ6): *What are the benefits and shortcomings of the proposed solution? What additional use cases can be addressed?*

Chapter 7 summarizes the settings and key findings of two different evaluation strategies, namely two case studies in the automotive and insurance industry that assess the prototype's practicability, and an interview series with 19 practitioners of 10 different domains to get feedback on the concept and the developed visualizations. Each Section in

Chapter 7 provides a synthesis of the key findings identified in the particular evaluation.

For further challenges of recovering EA models, we refer to Section 8.3 in which we summarize potential future research opportunities based on the findings of the thesis at hand. We conclude our summary with a recapitulation of the requirements identified in Section 4. Table 8.1 lists the requirements and provides both a brief description of how we addressed them and a reference to the respective section of this thesis.

8.2. Critical Reflection

In the previous Chapters in this thesis, we aim at addressing all defined requirements and summarize our achievements in Table 8.1. However, we are aware that not all of them are fully met. Especially, the evaluation conducted in Chapter 7 revealed main limitations of the developed solution approach. In the following sections, we recapitulate on known limitations and critically reflect both this thesis' contribution and its evaluation.

8.2.1. Functional Limitations of the Prototype

Many functional limitations were already discussed in the evaluation chapters. In the following, we summarize those limitations based on four main categories:

- **Limitations towards scalability:** *MICROLYZE* still has to be examined with respect to its scalability. This aspect was mentioned by several interviewed practitioners. In this context, scalability not only refers to the tool's ability to handle large runtime data sets, but also to handle those data in the respective visualizations. The complexity of huge recovered IT landscapes with many EA models must be reduced with either use case specific visualizations or with user control features like aggregation, filter and search in order to allow end-users to effectively explore the ETG in the light of thousands of nodes and edges. Further scalability concerns address 1) the support of massive distributed environments, 2) the parallel support of different APM tools and 3) the support of legacy systems, as well as monolithic applications. Even though, many of those mentioned scalability aspects are already implemented in *MICROLYZE*, they were not evaluated.
- **Limitations towards completeness:** Information completeness can only be proven if the ground truth is known. However, research regarding to (model-driven) reverse engineering and architecture recovery assumes that the ground truth is unknown or at least undocumented. Hence, it is rather challenging to prove that the solution approach is working correctly. This was recognized, especially in the analysis of removed models or model communications. No runtime information about those models initially only indicate a lack of instrumentation, but do not fully prove the according absence. As a result, as long as we cannot prove *MICROLYZE* is able to recover every model in the observed IT landscape, we also cannot form a general opinion about its practicability, but only discuss its potential benefits and drawbacks.

Table 8.1.: Addressed requirements detailed in Section 4.2

Req	Brief Description	Sections
Architectural Requirements		
REQ 1	Automated identification of models	5.2.3, 5.2.4
REQ 2	Automated identification of structural dependencies	5.3.1, 6.1.1, 6.3.2
REQ 3	Automated identification of model communications	5.3.2
REQ 4	Process support for maintaining relationships between business and technical EA layers	6.1.1, 6.1.2
REQ 5	Decentralized data collection process	5.2.3, 6.1.1, 6.1.7
REQ 6	Model references to federated information systems used along the IT value chain	6.1.3
Organizational Requirements		
REQ 7	Organizational regulation of model management	6.2
REQ 8	Technical support of the organizational maintenance process	6.1.5, 6.1.6
REQ 9	Alignment of model management to superior EA concepts	5.1.5, 6.1.1
Functional Requirements		
REQ 10	Delivery of up-to-date information	5.2.3, 5.3, 6.1.5, 6.3.3
REQ 11	Automated detection of changes	5.3.3, 5.3.4, 5.3.7, 6.1.5
REQ 12	Automated change propagation	5.3.3, 6.3
REQ 13	Generic model transformation	5.1.5, 5.2
REQ 14	Management of model evolution	5.2.5, 5.2.6
REQ 15	Network-based management of models	5.2.5
REQ 16	Automated detection of interfaces	5.2.4, 5.3.6
REQ 17	Definition of KPIs	5.2.6
Visualization Requirements		
REQ 18	Web-based client for model visualizations	5.4.1
REQ 19	Visualization of structural and communication dependencies	5.4, 6.4
REQ 20	Visualization of runtime information	5.4.8, 5.4.9
REQ 21	Query language for retrieving model information	5.2.6, 5.4.9

- **Limitations towards information analysis:** This point summarizes all limiting aspects with regard to model extraction from the available information base. First of all, runtime data only show that communication took place between two models, but cannot explain the reasons why these models communicated with each other. Therefore, it was argued by the practitioners that an actual understanding of the underlying architecture cannot be unveiled. In addition, *MICROLYZE* is not able to recover use cases and business processes from the available information. Furthermore, the validation of the configuration files against a JSON schema does not achieve a content validation which would lead to duplicated models and incorrect model assignments in case spelling mistakes are inserted. All those limitations challenge the practicability of the solution.
- **Limitations towards privacy:** Some interviewed experts stated that their organization would have worries when implementing *MICROLYZE* in their IT landscape. In general, this worry refers to the topic of "surveillance vs. privacy". In concrete terms this means that too much transparency about the IT landscape might disclose bad performance in departments, which is reflected by slow adoption of new technologies, slow execution of projects, bad architecture quality due to unskilled developers and the like. Privacy is a huge concern especially in big and regulated companies. Consequently, *MICROLYZE* would reignite the debate about holistic economic utility for an organization and each individual's privacy claims. In this context, the acceptance of people to make certain concessions regarding their privacy would only grow if they receive benefits as an exchange. For that reason, it is indispensable to show that *MICROLYZE* can be applied in different stakeholder concerns. However, this must be performed in every individual organization. We were only able to list its potential benefits and drawbacks.

8.2.2. Critical Reflection on the Validity

In order to discuss the trustworthiness of the evaluation results we apply the schema described in (Runeson et al., 2008). Different aspects of the approach and the prototype were validated with different evaluation strategies. The first case detailed in Section 7.2 study focused on the recovery of technical models through runtime data analysis and according visualizations. The second case covered in Section 7.3 study validates the recovery approach of business-related models by maintaining them in configuration files. The feasibility of both case studies were validated with quantitative and qualitative methods. Finally, the interview series with 19 practitioners of different domains and different companies assess the overall solution and visualization approaches (cf. Section 7.4).

We see one of the greatest threat in the construction validity. The experts we interviewed might understand the questions differently according to our intention. For instance, the term "microservice" could be interpreted differently since this term is not clearly defined. Some experts might not be aware of the differences between business services, service-oriented applications (SOA) and microservices and could mix up the terms. Since we

interviewed a lot of Enterprise and IT Architects this group of people might understand microservices in a different way as Developers do. Nevertheless, from our point of view, it was appropriate to use this terminology, since most of the experts are familiar with it and the hype about microservices leads to a common understanding about this architectural pattern.

For the demonstration of the prototype, we used the results collected by the case studies and showed the visualizations generated from it to the interviewees. Based on this demonstration, we discussed the potential utility for the particular application domain. This demonstration was indeed successful for the case study interviews, as the practitioners recognize their own IT landscape. However, for the conducted interviews detailed in Section 7.4, the discussions were of a hypothetical nature, since on the one hand the presented visualizations were based on anonymized data and on the other hand the practitioners had only the opportunity to see a foreign recovered IT landscape. This fact represents another threat to the evaluation's validity.

The main limitation regarding external validity is given by the structure of the sample. We see three threats according to the external validity: First, the experts were mainly from Germany, hence, the transferability of the results to other countries may be limited. In addition, the sample might not be representative enough because the experts were obtained by convenience sampling. For the interview series, we were only able to analyze responses from 19 experts. The experts do not represent an equal distribution over every industry sector.

The evaluation strategies in general are primarily of a qualitative nature. Although the case studies provide helpful and in-depth insights into concrete applications of the prototype in specific contexts, the evaluation's empirical foundation implies that its results are not generalizable. A big percentage of the whole sample represent user groups that are responsible for EA management tasks. Therefore, the evaluation is potentially biased towards this specific domain. In addition, even though, these group of persons have enough knowledge about EA management and EA model maintenance, they were not always able to provide high-qualified answers about the assessment of rather technical aspects like CD pipelines integration, data collection from monitoring tools, etc. Another danger stems from the decision to primarily address experts who are already dealing with microservices. While their existing knowledge can reduce the risk of misunderstandings and lead to more qualified answers, a positive bias towards this architecture style can not be eliminated.

A potential danger to the reliability of the results may arise from the fact that a significant number of experts were obtained through personal contact. Their answers could be influenced subjectively, since the experts might not want to give a bad evaluation.

8.2.3. Critical Reflection on the Research Methodology

In this thesis, we applied the design science research framework (Hevner et al., 2004) in order to systematically develop the solution design and the prototypical implementation. Even though, this research framework is widely used in the research community, it also must be questioned whether this method is the right approach to process the research

topic. In this context, we refer to Frank (Frank, 2006) which discusses four deficiencies the design science research is suffering from. These are 1) a lack of accounting for possible future worlds, 2) insufficient conception of a scientific foundation, 3) a mechanistic world view, and 4) a lack of appropriate concepts for describing the IT artifact.

For instance, Hevner et al. (Hevner et al., 2004) propagate to focus on problems which are motivated by current business needs. Business needs which might potentially emerge in future are not captured by the framework. According to this, the motivation of this thesis is indeed based on current problems, experiences, and business needs, and thus is certainly suffering from a lack of accounting for possible future worlds. In addition, we derive the requirements for our IT artifact based on related work (cf. Chapter 3) and foundations (cf. Chapter 2). Hence, the derived requirements are certainly contingent. This fact is described by Frank (Frank, 2006) as one of the key issues of the design science research framework's mechanistic world view. As a consequence, it is important to interpret the results and conclusions of this thesis in the light of those flaws.

8.3. Future Work

In the final section of this thesis, we discuss future research opportunities which are enabled by this thesis' contribution and findings. Table 8.2 summarizes all future work we identified during the conducted case studies in Section 7.2 and 7.3, as well as during the interview series in Section 7.4. Those future work are mainly derived from limitations we collected as a response by the interviewed practitioners. In addition to Table 8.2, in the following we focus on research opportunities which are not direct implications from those technical limitations but still represent high research potential.

8.3.1. Business Process Recovery via Process Mining

MICROLYZE processes runtime data for recovering technical EA models. Business-related EA models are not recovered automatically but provided by configuration files that are placed in each application's repository. The configuration file can be extended manually with arbitrary information like business domains, capabilities, products or business processes. We see potential for future research especially in the automated reconstruction of business processes. For this objective, process mining could be applied (W. M. P. v. d. Aalst et al., 2000; W. M. P. v. d. Aalst, 2015). This research aims at identifying trends, patterns and further activity-related details that user performs by analyzing event log data recorded by an information system. One process mining technique is the discovery of event sequences that represent a specific business process. By incorporating this techniques with our model discovery technique, we would also be able to discover business-related models automatically. However, the researcher would probably face one big challenge: At this stage, it remains unclear how to establish the relationship between the discovered business processes and the *Application Components* that are responsible to handle the business process.

Table 8.2.: List of future research derived from identified limitations from the automotive case study (ACS), insurance case study (ICS) and conducted interviews (INT)

Source	Code	Brief Description	Section
ACS	F1	Recovery of monolithic applications via runtime instrumentation	7.2.4
ACS	F2	Application of deletion thresholds for detecting removed communications	7.2.4
ACS	F3	Extraction of semantic information from runtime data in order to understand the reason for application communications	7.2.4
ACS	F4	Integration of planned states in the IT landscape recovery	7.2.4
ACS	F5	Recovery of IT landscapes across national borders	7.2.4
ACS	F6	Support of runtime data analysis exposed from several different APM tools	7.2.4
ACS	F7	Integration of search and filter features into the <i>MICROLYZE</i> frontend	7.2.4
ACS	F8	Integration of highlighting features to emphasize certain hotspots	7.2.4
ACS	F9	Integration of aggregation features to drill the information up and down to the desired aggregation level	7.2.4
ACS	F10	Association of recovered requests with business use cases and business processes	7.2.4
ACS	F11	Add release deployment cycle to <i>Architecture Model Comparison</i> visualization	7.2.4
ACS	F12	Add planned architecture states to <i>Architecture Model Comparison</i> visualization	7.2.4
ICS	F1	Export recovered models to CMDBs, EAM tools or other applications	7.3.5
ICS	F2	Extension of the configuration file with a global identifier that uniquely determines applications	7.3.5
ICS	F3	Additional user frontend for supporting configuration file maintenance	7.3.5
ICS	F4	Extension of configuration file mapping on <i>Application Collaboration</i> level	7.3.5
INT	F1	Application live-cycle management	7.4.4
INT	F2	IT transformation planning and controlling	7.4.4
INT	F3	Support of requirements analysis	7.4.4
INT	F4	Cloud migration planning and controlling	7.4.4
INT	F5	Failure impact analysis	7.4.4
INT	F6	Service catalog management	7.4.4

8.3.2. Assessment of Architecture Quality

The assessment of the quality of the developed architecture also gives potential for further research. Especially, microservice-based architectures rely on the interaction of self-contained services. In order to assess an architecture's quality and validate its conformance to behavioral requirements, those models must be subjected to sophisticated static analyses. Those analysis are mostly limited to performance analysis but not to architecture quality analysis. Since *MICROLYZE* is able to discover the whole IT landscape, we suggest to use the recovered information as the base for performing quality analysis in order to obtain its strengths and weaknesses. This provides a sound foundation for the future evolution of the architecture as well as for decision-making regarding new projects. In this scope, we refer to related work published by the University of Augsburg (Bauer et al., 2015; Langermeier. et al., 2017).

8.3.3. Failure Root Cause and Failure Impact Analysis

Due to the inherent complexity of EA models and their interconnections, the task of generating additional value from these models is very challenging without suitable analysis methods. Failure root cause analysis (Gupta et al., 2003; Wilson et al., 1993), that determine the root causes of faults or problems as well as failure impact analysis (Bagchi et al., 2001; Hanemann et al., 2005), that uncover or predicts the effects of changes or failures on other architectural elements, can provide valuable information for architecture roles. Whether a model is affected depends on its context, i.e. its connections to other models and their semantics with respect to the analysis. Those analysis techniques are mostly applied by operations with support of monitoring tools. However, the big picture is often not taken into account. The goal is to fix the failure to get back into normal operation quickly. An holistic analysis that also takes further EA layer into account is mostly not applied. Hence, it remains either hidden what impact a failure had on customers, business processes and business domains or it will be discovered much later. *MICROLYZE* could support those analysis techniques by automating the recovery of component dependencies and the according visualization of failure impact and root cause paths.

A. Appendix

A.1. JSON Schema

The following Listing A.1 details the required JSON Schema for validating the JSON example provided in Listing 7.1 in Section 6.1.3.

Listing A.1: JSON Schema validation file

```
1 {
2   "$schema": "http://json-schema.org/draft-07/schema#",
3   "$id": "microlyze json validation",
4   "type": "object",
5   "required": ["name", "description", "business_service", "product", "
      business_function"],
6   "properties": {
7     "name": {
8       "$id": "#/properties/name",
9       "type": "string",
10      "pattern": "^(.*)$"
11    },
12    "description": {
13      "$id": "#/properties/description",
14      "type": "string",
15      "pattern": "^(.*)$"
16    },
17    "business_service": {
18      "$id": "#/properties/business_service",
19      "type": "object",
20      "required": ["name"],
21      "properties": {
22        "name": {
23          "$id": "#/properties/business_service/name",
24          "type": "string",
25          "pattern": "^(.*)$",
26        },
27      }
28    },
29    "product": {
30      "$id": "#/properties/product",
```

```
31     "type": "object",
32     "required": ["name"],
33     "properties": {
34         "name": {
35             "$id": "#/properties/product/name",
36             "type": "string",
37             "pattern": "^(.*)$",
38         },
39     }
40 },
41 "business_function": {
42     "$id": "#/properties/business_function",
43     "type": "object",
44     "required": ["name"],
45     "properties": {
46         "name": {
47             "$id": "#/properties/business_function/name",
48             "type": "string",
49             "pattern": "^(.*)$",
50         },
51     }
52 },
53 "references": {
54     "$id": "#/properties/references",
55     "type": "array",
56     "items": {
57         "$id": "#/properties/references/items",
58         "type": "object",
59         "properties": {
60             "pm": {
61                 "$id": "#/properties/references/items/properties/pm",
62                 "type": "object",
63                 "required": ["tool", "domainurl", "apiurl", "id"],
64                 "properties": {
65                     "tool": {
66                         "$id": "#/properties/references/items/properties/pm/properties
67                             /tool",
68                         "type": "string",
69                         "pattern": "^(.*)$"
70                     },
71                     "domainurl": {
72                         "$id": "#/properties/references/items/properties/pm/properties
73                             /domainurl",
```

```
72         "type": "string",
73         "pattern": "^(.*)$",
74         "format": "uri"
75     },
76     "apiurl": {
77         "$id": "#/properties/references/items/properties/pm/properties
78             /apiurl",
79         "type": "string",
80         "pattern": "^(.+)\\[([^\]]+)\]$"
81     },
82     "id": {
83         "$id": "#/properties/references/items/properties/pm/properties
84             /id",
85         "type": "string",
86         "pattern": "^(.*)$"
87     }
88 },
89 "cldb": {
90     "$id": "#/properties/references/items/properties/cldb",
91     "type": "object",
92     "required": ["tool", "domainurl", "apiurl", "id", "apiToken"],
93     "properties": {
94         "tool": {
95             "$id": "#/properties/references/items/properties/cldb/
96                 properties/tool",
97             "type": "string",
98             "pattern": "^(.*)$"
99         },
100        "domainurl": {
101            "$id": "#/properties/references/items/properties/cldb/
102                properties/domainurl",
103            "type": "string",
104            "pattern": "^(.*)$",
105            "format": "uri"
106        },
107        "apiurl": {
108            "$id": "#/properties/references/items/properties/cldb/
109                properties/apiurl",
110            "type": "string",
111            "pattern": "^(.+)\\[([^\]]+)\]$"
112        },
113        "id": {
```

```
110         "$id": "#/properties/references/items/properties/cmdb/
           properties/id",
111         "type": "string",
112         "pattern": "^(.*)$"
113     },
114     "apiToken": {
115         "$id": "#/properties/references/items/properties/cmdb/
           properties/apiToken",
116         "type": "string",
117         "pattern": "^(.*)$"
118     }
119 }]]]]],
120 "application_collaboration": {
121     "$id": "#/properties/application_collaboration",
122     "type": "object",
123     "required": ["name"],
124     "properties": {
125         "name": {
126             "$id": "#/properties/application_collaboration/properties/name",
127             "type": "string",
128             "pattern": "^(.*)$"
129         }
130     }
131 },
132 "product_owner": {
133     "$id": "#/properties/product_owner",
134     "type": "object",
135     "required": ["name", "contact"],
136     "properties": {
137         "name": {
138             "$id": "#/properties/product_owner/properties/name",
139             "type": "string",
140             "pattern": "^(.*)$"
141         },
142         "contact": {
143             "$id": "#/properties/product_owner/properties/contact",
144             "type": "string",
145             "format": "email"
146         }
147     }
148 },
149 "used_ports": {
150     "$id": "#/properties/used_ports",
```



```
151     "type": "array"  
152   }  
153 }  
154 }
```

List of Figures

1.1.	Model representation along the IT value stream described by IT4IT (The Open Group, 2019). Each model undergoes various phases including the <i>Instantiation, Specialization</i> and <i>Execution</i> phase, whereas each phase in the value stream is supported by specific information systems.	2
1.2.	Design science research framework by (Hevner et al., 2004) adapted to the present thesis' contribution	7
1.3.	Steps of deductive category assignment (Mayring, 2010)	13
1.4.	Steps of inductive category development (Mayring, 2010)	13
1.5.	The main contributions of this thesis.	15
2.1.	Core topics that represent the foundation of related work and the thesis' approach.	21
2.2.	Four-layered meta-modeling stack (Brambilla et al., 2012)	23
2.3.	Model transformation schema (based on (Czarnecki et al., 2006))	24
2.4.	Model-driven reverse engineering process (based on (Brambilla et al., 2012))	26
2.5.	Fundamental layers of an enterprise architecture	27
2.6.	ArchiMate core framework (The Open Group, 2016)	31
2.7.	ArchiMate behavior and structure elements meta-model (The Open Group, 2016)	32
2.8.	ArchiMate relationship meta-model (The Open Group, 2016)	33
2.9.	Monolithic Architecture vs. Microservice Architecture	35
2.10.	Microservice Architecture Reference Model	38
2.11.	Proportion of IT that is based on microservices grouped by industry sectors. N=58 (Kleehaus et al., 2019b)	42
2.12.	DevOps life cycle phases (Bass et al., 2015)	44
2.13.	The causal and temporal relationships between four spans in distributed tracing (Sigelman et al., 2010)	54
3.1.	Layered architecture of a software map (Lankes et al., 2005)	62
3.2.	IT landscape visualization with ExplorViz (Fittkau et al., 2015)	63
3.3.	IT domain specific modeling language (Frank et al., 2009)	64
3.4.	Different levels of detail in the C4 model (Brown, 2018)	64
4.1.	Model interconnection along the IT value stream. Each phase in the value stream is supported by specific information systems. An holistic model management is essential for knowledge sharing as several stakeholders are involved in phase in the value stream	68

4.2. Conceptual framework of EA model management in a microservice-based IT environment	69
5.1. AppDynamics meta-model (A. Inc., 2020)	82
5.2. New Relic meta-model (N. Inc., 2020)	83
5.3. Dynatrace meta-model (D. Inc., 2020)	86
5.4. AppMon meta-model	88
5.5. Derived meta-model for <i>MICROLYZE</i>	93
5.6. The interplay between the IT landscape under observation (SUO), the corresponding APM server and our developed tool <i>MICROLYZE</i>	94
5.7. Model transformations during the IT landscape recovery process	95
5.8. Detailed class diagram of the core components of <i>MICROLYZE</i>	97
5.9. GET list of processes from Dynatrace API	98
5.10. Meta-Model for storing EA models in graph-based representation	107
5.11. GraphQL query for retrieving all <i>Application Components</i> that are required to process a specific request	112
5.12. GraphQL result for retrieving <i>Application Components</i> within a specific <i>Application Interaction</i>	112
5.13. GraphQL query for retrieving the average cpu utilization within a specific timeframe for a defined <i>Application Component</i>	114
5.14. Revision concept	114
5.15. Revision relation concept	115
5.16. GraphQL mutation for recovering and storing new AMs	115
5.17. Overall model recovery process	117
5.18. Backward recovery process	118
5.19. Differences between deployment-level communications and assembly-level communications between microservices. Only the assembly-level are stored in the database.	121
5.20. Forward Recovery process	122
5.21. Revision Creation Process	125
5.22. RESTful Interface Reconstruction Process	127
5.23. Architecture of the visualization components of <i>MICROLYZE</i>	130
5.24. Style and layout architecture of <i>MICROLYZE</i>	133
5.25. AM Visualization: (left) <i>Architecture Component</i> with 7 instances and 13 exposed interfaces, (middle-left) representation of a <i>Node</i> element, (middle-right) collapsed <i>Product</i> with 1 subelement and shrunk <i>Label</i> , (right) expanded <i>Product</i> with one <i>Business Service</i> as subelement. Further AMs are displayed in a similar way and with the correct Archimate symbols.	134
5.26. Relationship Visualization: (left) <i>Compositions</i> are marked with a rhombus and determine <i>contains</i> relationships, (middle) <i>Associations</i> are represented by a solid line, indicating <i>runs on</i> relationships, and (right) <i>Flows</i> are drawn with a dashed arrow and defines <i>calls</i> relationships	135
5.27. Overview of the Visualization Process represented in an activity diagram	136
5.28. GraphQL query for <i>Architecture Model Deployment</i> visualization	137

5.29. Visualization of <i>Architecture Model Deployments</i> by structuring the model dependencies in a tree-based representation. Grouped nodes can be collapsed or expanded.	138
5.30. <i>Architecture Model Communication</i> visualization by leveraging tree-based representations.	140
5.31. Revealing <i>Architecture Model</i> interactions by highlighting transaction paths within a tree-based representation. The table above the graph details the used interfaces for transaction processing.	142
5.32. <i>Architecture Model Comparison</i> visualization by splitting the window layout. The left hand side displays the obsolete architecture of the selected <i>Application Collaboration</i> . The right hand side unveils the updated architecture. Removed relationships or AMs are highlighted in red. New relationships or AMs are visualized in green.	144
5.33. Sidebar that opens after clicking on a <i>Architecture Model</i> or communication relationship. Left: static information about the AM that is stored as annotations in the database. Middle: communication relationship information separated in <i>calls</i> and <i>called by</i> . Right: runtime information about the AM .	146
5.34. GraphQL client developed by Prisma. The view illustrates an example how to write GraphQL queries and how the result is displayed.	147
6.1. Overall System Integration Concept	150
6.2. Adapted Meta-Model including AMs from the business layer	153
6.3. Extended meta-model for storing entities within the <i>MICROLYZE.Store</i> component	157
6.4. <i>MICROLYZE.Expose</i> and <i>MICROLYZE.Collect</i> component extension for supporting model import from federated information systems	158
6.5. JSON Schema location and distribution overview	164
6.6. Interplay of the different roles and components involved in the overall solution concept.	166
6.7. CD pipeline validation process	168
6.8. Workflow for processing information delivered by the configuration file . .	172
6.9. Decision tree for <i>ArchitectureModel</i> updates	173
6.10. Visualizing hierarchical dependencies of <i>Architecture Models</i> via a domain cluster representation. Grouped AMs can be opened or collapsed.	176
6.11. Visualization of <i>Architecture Models</i> in a table-based representation. The columns of the particular table can be specified according to the available annotation information.	178
6.12. The visualization of <i>Architecture Model</i> communications can also be performed on an higher aggregation level in order to address different stakeholder concerns. In this example, we aggregate the communication dependencies on <i>Business Service</i> level to unveil which services communicate in general.	180
7.1. Performed evaluation activities combining different research methods . . .	181

7.2. Performed evaluation activities and their milestones	183
7.3. Recovered <i>ApplicationComponents</i> during recovery process. The backward recovery process starts at iteration -1 and ends at iteration -112. The forward recovery process takes over at iteration 113.	191
7.4. Recovered <i>Nodes</i> during recovery process. The backward recovery process starts at iteration -1 and ends at iteration -112. The forward recovery process takes over at iteration 113.	192
7.5. Recovered <i>ApplicationInteractions</i> and communication relationships during recovery process. The backward recovery process starts at iteration -1 and ends at iteration -112. The forward recovery process takes over at iteration 113.	193
7.6. Case Study IT landscape overview	205
7.7. <i>MICROLYZE</i> integration effort compared between two different development teams	211

List of Tables

5.1.	Meta-model transformation for deriving a generally accepted meta-model for Microlyze	89
5.2.	Reported log events of RESTful API calls and their corresponding interface description. The parameters are highlighted in bold.	126
7.1.	Status quo: Average as-is EA model documentation rating per EA layer. N=1. 1=fully agree, 2=rather agree, 3=rather disagree, 4=fully disagree	186
7.2.	MICROLYZE execution result: Result of the Top 15 of those <i>ArchitectureModels</i> with the highest automation score. $score_{min} = 5,5$, $score_{max} = 22,0$	187
7.3.	Status quo: Average as-is EA model documentation rating per EA layer. N=7. 1=fully agree, 2=rather agree, 3=rather disagree, 4=fully disagree	203
7.4.	MICROLYZE execution result: Result of the Top 15 of those <i>Architecture Models</i> with the highest automation score. $score_{min} = 5,5$, $score_{max} = 22,0$	204
7.5.	Interview participants grouped by industry sector. N=19	218
7.6.	Interview participants grouped by job title. N=19	218
7.7.	Received feedback on the subject area of "Assessment of the solution architecture" consisting of code, name and frequency of the categories as well as allocation to the five elaborated upper categories.	220
7.8.	Received feedback on the subject area of "Assessment of the visualization approaches" consisting of code, name and frequency of the categories as well as allocation to the seven elaborated upper categories.	224
7.9.	Received feedback on the subject area of "Assessment of the technical and organizational integration" consisting of code, name and frequency of the categories as well as allocation to the three elaborated upper categories.	228
8.1.	Addressed requirements detailed in Section 4.2	240
8.2.	List of future research derived from identified limitations from the automotive case study (ACS), insurance case study (ICS) and conducted interviews (INT)	244

Acronyms

ADL Architecture Description Language.

AM Architecture Model.

API Application Programming Interface.

APM Application Performance Monitoring.

AR Architecture Recovery.

AWS Amazon Web Services.

CD Continuous Delivery.

CDE Continuous Deployment.

CI Continuous Integration.

CMDB Configuration Management Database.

CPU Central Processing Unit.

CSRF Cross-Site Request Forgery.

DDD Domain-Driven Design.

DoD Definition of Done.

DOM Document Object Model.

DT Development Team.

EA Enterprise Architecture.

EAD Enterprise Architecture Documentation.

EAM Enterprise Architecture Management.

ESB Enterprise Service Bus.

ETG Enterprise Topology Graph.

HOT Higher Order Transformation.

HTTP Hypertext Transfer Protocol.

IaaS Infrastructure as a Service.

IS Information Systems.

IT Information Technology.

ITIL Information Technology Infrastructure Library.

JS JavaScript.

JSON JavaScript Object Notation.

KPI Key Performance Indicator.

M2M Model to Model Transformation.

M2T Model to Text Transformation.

MDE Model-driven Engineering.

MDRE Model-driven Reverse Engineering.

MOF Meta Object Facility.

NIST National Institute of Standards and Technology.

OS Operating System.

PaaS Platform as a Service.

PO Product Owner.

PPM Project Portfolio Management.

QoS Quality of Service.

REST Representational State Transfer.

SaaS Software as a Service.

SM Scrum Master.

SOA Service-Oriented Architecture.

SQL Structured Query Language.

T2M Text to Model Transformation.

TUM Technical University of Munich.

UAT User Acceptance Test.

UI User Interface.

UML Unified Modelling Language.

URL Uniform Resource Locator.

UUID Universal Unique Identifier.

VCS Version Control System.

Bibliography

- Aalst, W. M. P. v. d., J. Desel, and A. Oberweis, eds. (2000). *Business Process Management, Models, Techniques, and Empirical Studies*. London, UK, UK: Springer-Verlag.
- Aalst, W. M. P. van der (2015). “Extracting Event Data from Databases to Unleash Process Mining”. In: *BPM*. Springer, pp. 105–128.
- Abrahamsson, P., O. Salo, J. Ronkainen, and J. Warsta (2017). “Agile software development methods: Review and analysis”. In: *arXiv preprint arXiv:1709.08439*.
- Achhammer, L. (2019). “Assessing the Cost and Benefit of a Microservice Landscape Discovery Method”. Master’s Thesis. Munich, Germany.
- Agrawal, R., D. Gunopulos, and F. Leymann (1998). “Mining Process Models from Workflow Logs”. In: *Proceedings of the 6th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT ’98. Springer-Verlag, pp. 469–483.
- Ahlemann, F., E. Stettiner, M. Messerschmidt, and C. Legner (2012). *Strategic Enterprise Architecture Management: Challenges, Best Practices, and Future Developments*. Management for Professionals. Springer Berlin Heidelberg.
- Aier, S. and B. Gleichauf (2010). “Application of Enterprise Models for Engineering Enterprise Transformation”. In: *Enterprise Modelling and Information Systems Architectures - An International Journal* 5.1, pp. 58–75.
- Aldea, A., M. Iacob, A. Wombacher, M. Hiralal, and T. Franck (2018). “Enterprise Architecture 4.0 – A Vision, an Approach and Software Tool Support”. In: *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 1–10.
- Aleatrati Khosroshahi, P., M. Hauder, A. Schneider, and F. Matthes (2020). *Enterprise Architecture Management Pattern Catalog*. Tech. rep. Technical University Munich. URL: <https://www.matthes.in.tum.de/pages/ugsyi19wmmv1/Enterprise-Architecture-Management-Pattern-Catalog-V2-2015> (visited on 02/14/2020).
- Alegria, A. and A. Vasconcelos (2010). “IT Architecture automatic verification: A network evidence-based approach”. In: *2010 Fourth International Conference on Research Challenges in Information Science (RCIS)*, pp. 1–12.
- Ali, N., S. Baker, R. O’Crowley, S. Herold, and J. Buckley (2017). “Architecture consistency: State of the practice, challenges and requirements”. In: *Empirical Software Engineering* 23, pp. 1–35.
- Allspaw, J. (2008). *The Art of Capacity Planning: Scaling Web Resources*. O’Reilly Media, Inc.
- Alshuqayran, N., N. Ali, and R. Evans (2016). “A systematic mapping study in microservice architecture”. In: *International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, pp. 44–51.
- Armour, F., S. Kaisler, J. Getter, and D. Pippin (2003). “A UML-driven enterprise architecture case study”. In: *36th Annual Hawaii International Conference on System Sciences (HICSS’03)*.

- Armour, F., S. Kaisler, and S. Liu (1999). "Building an Enterprise Architecture Step-by-Step". In: *IT Professional 1*, pp. 31–39.
- Armour, F., S. Kaisler, and M. Valivullah (2005). "Enterprise Architecting: Critical Problems". In: *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*.
- Bagchi, S., G. Kar, and J. L. Hellerstein (2001). "Dependency Analysis in Distributed Systems using Fault Injection: Application to Problem Determination in an e-commerce Environment." In: *DSOM*, pp. 151–164.
- Bahle, S., C. Endres, and M. Fetzner (2013). *Evaluierung von Ansätzen zur Identifizierung und Ermittlung der Enterprise IT in Forschung und Produkten*. Tech. rep. Stuttgart.
- Balalaie, A., A. Heydarnoori, and P. Jamshidi (2016). "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software 33.3*, pp. 42–52.
- Bang, S. K., S. Chung, Y. Choh, and M. Dupuis (2013). "A Grounded Theory Analysis of Modern Web Applications: Knowledge, Skills, and Abilities for DevOps". In: *Proceedings of the 2nd Annual Conference on Research in Information Technology*. New York, NY, USA: Association for Computing Machinery, pp. 61–62.
- Bass, L. J., I. Weber, and L. Zhu (2015). *DevOps: A Software Architect's Perspective*. Pearson Education.
- Bauer, B., M. Langermeier, and C. Saad (2015). "A Flow Analysis Approach for Service-Oriented Architectures". In: *Software, Services, and Systems: Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*. Ed. by R. De Nicola and R. Hennicker. Cham: Springer International Publishing, pp. 475–489.
- Beck, F. and S. Diehl (2011). "On the Congruence of Modularity and Code Coupling". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, pp. 354–364.
- Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.
- Beck, K., M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas (2001). *Manifesto for Agile Software Development*. URL: <http://www.agilemanifesto.org/> (visited on 02/14/2020).
- Bencomo, N., S. Götz, and H. Song (2019). "Models@run.time: a guided tour of the state of the art and research challenges". In: *Software Systems Modeling*.
- Bennaceur, A., R. France, G. Tamburrelli, T. Vogel, P. J. Mosterman, W. Cazzola, F. M. Costa, A. Pierantonio, M. Tichy, M. Akşit, P. Emmanuelson, H. Gang, N. Georgantas, and D. Redlich (2014). "Mechanisms for Leveraging Models at Runtime in Self-adaptive Software". In: *Models@run.time: Foundations, Applications, and Roadmaps*. Ed. by N. Bencomo, R. France, B. H. C. Cheng, and U. Aßmann. Cham: Springer International Publishing, pp. 19–46.
- Bieberstein, N., R. Laird, K. Jones, and T. Mitra (2008). *Executing SOA: A Practical Guide for the Service-oriented Architect*. developerWorks series. IBM Press/Pearson plc.

- Binz, T., U. Breitenbücher, O. Kopp, and F. Leymann (2013). "Automated Discovery and Maintenance of Enterprise Topology Graphs". In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pp. 126–134.
- Blair, G., N. Bencomo, and R. B. France (2009). "Models@run.time". In: *Computer* 42.10, pp. 22–27.
- Bossert, O. (2016). "A Two-Speed Architecture for the Digital Enterprise". In: *Emerging Trends in the Evolution of Service-Oriented and Enterprise Architectures*.
- Brambilla, M., J. Cabot, and M. Wimmer (2012). *Model-Driven Software Engineering in Practice*. Vol. 1.
- Braun, C. and R. Winter (2005). "A comprehensive enterprise architecture metamodel and its implementation using a metamodeling platform". In: *Enterprise modelling and information systems architectures*. Ed. by J. Desel and U. Frank. Bonn: Gesellschaft für Informatik e.V., pp. 64–79.
- Breu, R. (2010). "Ten Principles for Living Models - A Manifesto of Change-Driven Software Engineering". In: pp. 1–8.
- Brewer, E. (2012). "CAP twelve years later: How the "rules" have changed". In: *Computer* 45.2, pp. 23–29.
- Briand, L. C., Y. Labiche, and J. Leduc (2006). "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software". In: *IEEE Transactions of Software Engineering* 32.9, pp. 642–663.
- Brown, S. (2018). *Software Architecture for Developers*. Best Sellers.
- Bruneliere, H., J. Cabot, G. Dupé, and F. Madiot (2014). "MoDisco: a Model Driven Reverse Engineering Framework". In: *Information and Software Technology* 56.
- Brunnert, A., A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, et al. (2015). *Performance-oriented DevOps: A research agenda*. Tech. rep.
- Bubak, O. (2006). "Composing a course book for system and enterprise architecture education". In: *IEEE/SMC International Conference on System of Systems Engineering*.
- Buckl, S. and C. Schweda (2012). *On the State-of-the-Art in Enterprise Architecture Management Literature*. Tech. rep. München.
- Buckl, S. (2011). "Developing Organization-Specific Enterprise Architecture Management Functions Using a Method Base". Dissertation. Munich, Germany.
- Buckl, S., A. M. Ernst, J. Lankes, F. Matthes, C. M. Schweda, and A. Wittenburg (2007). "Generating Visualizations of Enterprise Architectures using Model Transformations". In: *Enterprise Modelling and Information Systems Architectures (EMISAJ) 2*, pp. 3–13.
- Buckl, S., F. Matthes, C. Neubert, and C. M. Schweda (2011). "A Lightweight Approach to Enterprise Architecture Modeling and Documentation". In: *Information Systems Evolution*. Ed. by P. Soffer and E. Proper. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 136–149.
- Buckl, S., F. Matthes, S. Roth, C. Schulz, and C. M. Schweda (2010). "A Conceptual Framework for Enterprise Architecture Design". In: *Trends in Enterprise Architecture Research*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 44–56.

- Bughin, J., M. Chui, and A. Miller (2009). *How companies are benefiting from Web 2.0*. URL: <http://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/how-companies-are-benefiting-from-web-20-mckinsey-global-survey-results?cid=e1-web> (visited on 02/14/2020).
- Buschle, M., M. Ekstedt, S. Grunow, M. Hauder, F. Matthes, and S. Roth (2012). "Automating Enterprise Architecture Documentation using an Enterprise Service Bus". In: *Americas Conference on Information Systems (AMCIS)*.
- Caçado, P. (2014). *Building Products at SoundCloud—Part III: Microservices in Scala and Finagle*. Tech. rep. SoundCloud Limited. URL: <https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-3-microservices-in-scala-and-finagle> (visited on 02/14/2020).
- Canfora, G., M. Di Penta, and L. Cerulo (2011). "Achievements and Challenges in Software Reverse Engineering". In: *Communications of ACM* 54.4, pp. 142–151.
- Cheng, B. H. C., R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle (2009). "Software Engineering for Self-Adaptive Systems: A Research Roadmap". In: *Software Engineering for Self-Adaptive Systems*. Ed. by B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–26.
- Chikofsky, E. and J. H. Cross (1990). "Reverse engineering and design recovery: a taxonomy". In: *IEEE Software* 7.1, pp. 13–17.
- Choudhary, S. R. and A. Orso (2009). "Automated Client-Side Monitoring for Web Applications". In: *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pp. 303–306.
- CIO Council (1999). *FEAF - Federal Enterprise Architecture Framework Version 1.1*. URL: https://web.archive.org/web/20090202182509/http://www.whitehouse.gov/omb/assets/fea_docs/FEA_CRM_v23_Final_Oct_2007_Revised.pdf (visited on 02/14/2020).
- Clements, P. (2003). *Documenting Software Architectures: Views and Beyond*. SEI series in software engineering. Addison-Wesley.
- Cockburn, A. (2002). *Agile Software Development*. Agile software development series. Addison-Wesley.
- Cockcroft, A. (2016). *Microservices Workshop: Why, what, and how to get there*. URL: <https://conferences.oreilly.com/oscon/open-source-2015/public/schedule/detail/44066> (visited on 02/14/2020).
- Cohen, D., M. Lindvall, and P. Costa (2004). "An introduction to agile methods." In: *Advances in computers* 62.03, pp. 1–66.
- Cole, S. (2008). *Service Catalog Trends and Best Practices Survey Highlights*. URL: <https://www.enterprisemanagement.com/research/asset.php/966/Service-Catalog-Trends-and-Best-Practices-Survey-Highlights> (visited on 02/14/2020).
- Conboy, K. (2009). "Agility from First Principles: Reconstructing the Concept of Agility in Information Systems Development". In: *Information Systems Research* 20.3, pp. 329–354.

- Conway, M. E. (1968). "How Do Committees Invent?" In: *Datamation*.
- Corpancho, N. (2019). "Automated documentation of Business Domain assignments and cloud application information from an application development pipeline". Master's Thesis. Munich, Germany.
- Cuadrado, F., B. Garcia, J. C. Dueñas, and H. A. Parada (2008). "A case study on software evolution towards service-oriented architecture". In: *Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on*. IEEE, pp. 1399–1404.
- Cutler, B. (2010). *Firefox Page Load Speed*. URL: <https://blog.mozilla.org/metrics/2010/03/31/firefox-page-load-speed-part-i/> (visited on 09/14/2019).
- Czarnecki, K. and S. Helsen (2006). "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3, pp. 621–645.
- De Ryck, P., L. Desmet, W. Joosen, and F. Piessens (2011). "Automatic and Precise Client-Side Protection against CSRF Attacks". In: *Computer Security – ESORICS 2011*. Ed. by V. Atluri and C. Diaz. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 100–116.
- De Silva, F., C. Rich, and S. Ganguli (2019). *Magic Quadrant for Application Performance Monitoring*. URL: <https://www.gartner.com/en/documents/3904665/magic-quadrant-for-application-performance-monitoring> (visited on 02/14/2020).
- Debois, P. et al. (2011). "Devops: A software revolution in the making". In: *Journal of Information Technology Management* 24.8, pp. 3–39.
- Dern, G. (2009). *Management von IT-Architekturen: Leitlinien für die Ausrichtung, Planung und Gestaltung von Informationssystemen*. Edition CIO. Vieweg+Teubner Verlag.
- Devaraj, S. and R. Kohli (2001). *The IT Payoff: Measuring the Business Value of Information Technology Investments*. Financial Times/Prentice Hall.
- Dietz, J. (2006). *Enterprise Ontology—Theory and Methodology*. Springer Berlin Heidelberg.
- Dingsøy, T., S. Nerur, V. Balijepally, and N. B. Moe (2012). "A decade of agile methodologies: Towards explaining agile software development". In: *Journal of Systems and Software* 85.6, pp. 1213–1221.
- Doest, H. ter and M. Lankhorst (2004). *Tool support for enterprise architecture - A vision*. Tech. rep. Enschede.
- Doucet, G., P. Saha, and S. Bernard (2009). *Coherency Management: Architecting the Enterprise for Alignment, Agility and Assurance*. AuthorHouse.
- Dragoni, N., S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina (2016). "Microservices: yesterday, today, and tomorrow". In: *CoRR* abs/1606.04036.
- Dragoni, N., I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina (2018). "Microservices: How To Make Your Application Scale". In: *Perspectives of System Informatics*. Ed. by A. K. Petrenko and A. Voronkov. Cham: Springer International Publishing, pp. 95–104.
- Dreyfus, D. (2007). "Information System Architecture: Toward a Distributed Cognition Perspective." In: p. 131.
- Ducasse, S. and D. Pollet (2009). "Software Architecture Reconstruction: A Process-Oriented Taxonomy". In: *IEEE Transactions on Software Engineering* 35.4, pp. 573–591.

- Duffield, N. (2004). "Sampling for Passive Internet Measurement: A Review". In: *Statistical Science* 19.
- Duffy, D. (2004). *Domain Architectures: Models and Architectures for UML Applications*. Wiley.
- Dybå, T. and T. Dingsøyrr (2008). "Empirical studies of agile software development: A systematic review". In: *Information and software technology* 50.9-10, pp. 833–859.
- Einav, Y. (2019). *Amazon Found Every 100ms of Latency Cost them 1% in Sales*. URL: <https://www.gigaspace.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales> (visited on 03/04/2020).
- Ensor, P. (1988). "The Functional Silo Syndrome". In: *AME Target* 16.
- Erickson, J., K. Lyytinen, and K. Siau (2005). "Agile modeling, agile software development, and extreme programming: the state of research". In: *Journal of Database Management (JDM)* 16.4, pp. 88–100.
- Espinosa, J. and W. Boh (2009). "Coordination and Governance in Geographically Distributed Enterprise Architecting: An Empirical Research Design." In: pp. 1–10.
- Evans (2003). *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
- Farroha, B. S. and D. L. Farroha (2014). "A Framework for Managing Mission Needs, Compliance, and Trust in the DevOps Environment". In: *2014 IEEE Military Communications Conference*, pp. 288–293.
- Farwick, M., R. Breu, M. Hauder, S. Roth, and F. Matthes (2013). "Enterprise Architecture Documentation: Empirical Analysis of Information Sources for Automation". In: *46th Hawaii International Conference on System Sciences*, pp. 3868–3877.
- Farwick, M., W. Pasquazzo, R. Breu, C. M. Schweda, K. Voges, and I. Hanschke (2012a). "A Meta-Model for Automated Enterprise Architecture Model Maintenance". In: *2012 IEEE 16th International Enterprise Distributed Object Computing Conference*, pp. 1–10.
- Farwick, M., B. Agreiter, R. Breu, M. Haering, K. Voges, and I. Hanschke (2010). "Towards Living Landscape Models: Automated Integration of Infrastructure Cloud in Enterprise Architecture Management". In: *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 35–42.
- Farwick, M., B. Agreiter, R. Breu, S. Ryll, K. Voges, and I. Hanschke (2011a). "Automation Processes for Enterprise Architecture Management". In: *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, pp. 340–349.
- Farwick, M., B. Agreiter, R. Breu, S. Ryll, K. Voges, and I. Hanschke (2011b). "Requirements for Automated Enterprise Architecture Model Maintenance - A Requirements Analysis based on a Literature Review and an Exploratory Survey." In: vol. 4, pp. 325–337.
- Farwick, M., C. M. Schweda, R. Breu, K. Voges, and I. Hanschke (2012b). "On Enterprise Architecture Change Events". In: *Trends in Enterprise Architecture Research and Practice-Driven Research on Enterprise Transformation*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 129–145.
- Few, S. (2006). *Information Dashboard Design: The Effective Visual Communication of Data*. O'Reilly Media.

- Fiedler, M., M. Hauder, and A. Schneider (2013). "Foundations for the Integration of Enterprise Wikis and Specialized Tools for Enterprise Architecture Management". In: *11th International Conference on Wirtschaftsinformatik (WI)*. Leipzig, Germany.
- Fielding, R. T. and R. N. Taylor (2000). "Architectural Styles and the Design of Network-Based Software Architectures". Dissertation. Irvine, California.
- Filipe, R. and F. Araujo (2016). "Client-side monitoring techniques for web sites". In: *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pp. 363–366.
- Filipe, R., R. P. Paiva, and F. Araujo (2017). "Client-side black-box monitoring for web sites". In: *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pp. 1–5.
- Fink, A. and J. Kosecoff (2006). *How to Conduct Surveys: A Step-by-Step Guide*. SAGE Publications.
- Fischer, R., S. Aier, and R. Winter (2007). "A Federated Approach to Enterprise Architecture Model Maintenance". In: *Enterprise Modelling and Information Systems Architectures 2*, pp. 14–22.
- Fittkau, F., A. V. Hoorn, and W. Hasselbring (2014). "Towards a Dependability Control Center for Large Software Landscapes (Short Paper)". In: *2014 Tenth European Dependable Computing Conference*, pp. 58–61.
- Fittkau, F., J. Waller, C. Wulf, and W. Hasselbring (2013). "Live trace visualization for comprehending large software landscapes: The ExplorViz approach". In: *2013 First IEEE Working Conference on Software Visualization (VISOFT)*, pp. 1–4.
- Fittkau, F., S. Roth, and W. Hasselbring (2015). "ExplorViz: Visual Runtime Behavior Analysis of Enterprise Application Landscapes". In: *23rd European Conference on Information Systems (ECIS 2015)*.
- Fitzgerald, B. and K.-J. Stol (2015). "Continuous Software Engineering: A Roadmap and Agenda". In: *Journal of Systems and Software* 25.
- Foltête, J.-C., C. Clauzel, and G. Vuidel (2012). "A software tool dedicated to the modelling of landscape networks". In: *Environmental Modelling Software* 38, pp. 316–327.
- Fonseca, R., G. Porter, R. H. Katz, S. Shenker, and I. Stoica (2007). "X-Trace: A Pervasive Network Tracing Framework". In: *Proceedings of the 4th USENIX Conference on Networked Systems Design Implementation*. NSDI'07. Cambridge, MA: USENIX Association, p. 20.
- Fowler, M. and J. Lewis (2014). *Microservices*. Tech. rep. ThoughtWorks. URL: <https://martinfowler.com/articles/microservices.html> (visited on 02/14/2020).
- Francesco, P. D., I. Malavolta, and P. Lago (2017). "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption". In: *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 21–30.
- Frank, U. (2002). "Multi-perspective enterprise modeling (MEMO) conceptual framework and modeling languages". In: *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, pp. 1258–1267.
- Frank, U. (2006). *Towards a Pluralistic Conception of Research Methods in Information Systems Research*. ICB Research Report. Tech. rep. 7. Essen.

- Frank, U., D. Ferguson, D. Heise, E. Hadar, H. Kattenstroth, and M. Waschke (2009). "ITML: A Domain-Specific Modeling Language for Supporting Business Driven IT Management". In: *In Proceedings of the 9th OOPSLA workshop on domain-specific modeling (DSM '09)*.
- Fu, Q., J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie (2014). "Where Do Developers Log? An Empirical Study on Logging Practices in Industry". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India: Association for Computing Machinery, pp. 24–33.
- Fuchs-Kittowski, F. and D. Faust (2008). "The Semantic Architecture Tool (SemAT) for Collaborative Enterprise Architecture Development". In: *Groupware: Design, Implementation, and Use*. Ed. by R. O. Briggs, P. Antunes, G.-J. de Vreede, and A. S. Read. Springer Berlin Heidelberg, pp. 151–163.
- Fürstenau, D. and N. Kliewer (2015). "Exploring Enterprise Transformation from a Path Dependence Perspective: A Recycling Case and Conceptual Model". In: *Wirtschaftsinformatik*.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides (2015). *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. mitp Professional. MITP-Verlags GmbH & Co. KG.
- Garriga, M. (2018). "Towards a Taxonomy of Microservices Architectures". In: *Software Engineering and Formal Methods*. Ed. by A. Cerone and M. Roveri. Springer International Publishing, pp. 203–218.
- Gartner (2008). *Gartner Research Publication*. Symposium/ITxpo.
- Ghofrani, J. and D. Lübke (2018). "Challenges of Microservices Architecture: A Survey on the State of the Practice". In: *ZEUS*.
- Gholami, M. F., F. Daneshgar, G. Low, and G. Beydoun (2016). "Cloud migration process—A survey, evaluation framework, and open challenges". In: *Journal of Systems and Software* 120, pp. 31–69.
- Graeff, J. (2017). "Enhancing Business Process Mining with Distributed Tracing Data in a Microservice Architecture". Master's Thesis. Munich, Germany.
- Granchelli, G., M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle (2017a). "MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-based Systems". In: *IEEE International Conference on Software Architecture (ICSA)*.
- Granchelli, G., M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle (2017b). "Towards recovering the software architecture of microservice-based systems". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, pp. 46–53.
- Group, O. M. (2017). *Unified Modeling Language (UML)*. URL: <https://www.omg.org/spec/UML/> (visited on 02/14/2020).
- Grunow, S., F. Matthes, and S. Roth (2013). "Towards Automated Enterprise Architecture Documentation: Data Quality Aspects of SAP PI". In: *Advances in Databases and Information Systems*. Ed. by T. Morzy, T. Härder, and R. Wrembel. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 103–113.

- Gupta, M., A. Neogi, M. K. Agarwal, and G. Kar (2003). "Discovering dynamic dependencies in enterprise environments for problem determination". In: *International Workshop on Distributed Systems: Operations and Management*. Springer, pp. 221–233.
- Hacks, S. and H. Lichter (2019a). "Qualitative Comparison of Enterprise Architecture Model Maintenance Processes". In: *Enterprise Modeling and Information Systems Architectures (EMISA)*.
- Hacks, S., A. Steffens, P. Hansen, and N. Rajashekar (2019b). "A Continuous Delivery Pipeline for EA Model Evolution". In: *Enterprise, Business-Process and Information Systems Modeling*. Ed. by I. Reinhartz-Berger, J. Zdravkovic, J. Gulden, and R. Schmidt. Vol. 352. Lecture Notes in Business Information Processing. Cham: Springer International Publishing, pp. 141–155.
- Hafner, M. and R. Winter (2008). "Processes for Enterprise Application Architecture Management". In: *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, pp. 396–396.
- Hanemann, A., D. Schmitz, and M. Sailer (2005). "A framework for failure impact analysis and recovery with respect to service level agreements". In: *2005 IEEE International Conference on Services Computing (SCC'05) Vol-1*. Vol. 2, 49–56 vol.2.
- Hanschke, I. (2009). *Strategisches Management der IT-Landschaft: ein praktischer Leitfaden für das Enterprise-architecture-Management*. Hanser.
- Hanschke, I. (2010). *Strategic IT Management - A Toolkit for Enterprise Architecture Management*. Springer-Verlag.
- Hanschke, I. (2016). *Enterprise Architecture Management – einfach und effektiv: Ein praktischer Leitfaden für die Einführung von EAM*. Carl Hanser Verlag GmbH Co. KG, p. 333.
- Hanssen, G. K., D. Šmite, and N. B. Moe (2011). "Signs of Agile Trends in Global Software Engineering Research: A Tertiary Study". In: *2011 IEEE Sixth International Conference on Global Software Engineering Workshop*, pp. 17–23.
- Haren, V. (2011). *TOGAF Version 9.1*.
- Haselböck, S. and R. Weinreich (2017). "Decision Guidance Models for Microservice Monitoring". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 54–61.
- Hauder, M., F. Matthes, and S. Roth (2012). "Challenges for Automated Enterprise Architecture Documentation". In: *Trends in Enterprise Architecture Research and Practice-Driven Research on Enterprise Transformation*. Springer Berlin Heidelberg, pp. 21–39.
- Hawkins, D. (1980). *Identification of Outliers*. Monographs on applied probability and statistics. Chapman and Hall.
- Heger, C., A. van Hoorn, M. Mann, and D. Okanović (2017). "Application Performance Management: State of the Art and Challenges for the Future". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. L'Aquila, Italy: ACM, pp. 429–432.
- Henttonen, K. and M. Matinlassi (2009). "Open source based tools for sharing and reuse of software architectural knowledge". In: *2009 Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*, pp. 41–50.

- Hevner, A., S. March, J. Park, and S. Ram (2004). "Design science in information systems research". In: *MIS Quarterly: Management Information Systems* 28.1, pp. 75–105.
- Highsmith, J. (2013). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House eBooks. Pearson Education.
- Highsmith, J. and A. Cockburn (2001). "Agile software development: the business of innovation". In: *Computer* 34.9, pp. 120–127.
- Holl, P. and K. Gossling (2019). "MIDAS: Towards an Interactive Data Catalog". In: *International Workshop on Polystore and Other Systems for Heterogeneous Data*, pp. 128–138.
- Holm, H., K. Shahzad, M. Buschle, and M. Ekstedt (2015). "P² CySeMoL: Predictive, Probabilistic Cyber Security Modeling Language". In: *IEEE Transactions on Dependable and Secure Computing* 12.6, pp. 626–639.
- Holm, H., M. Buschle, R. Lagerström, and M. Ekstedt (2014). "Automatic data collection for enterprise architecture models". In: *Software & Systems Modeling* 13.2, pp. 825–841.
- Hoorn, A. van (2014). "Model-Driven Online Capacity Management for Component-Based Software Systems". Dissertation. Kiel, Germany.
- Hoorn, A. van, J. Waller, and W. Hasselbring (2012). "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. Boston, Massachusetts, USA: ACM, pp. 247–248.
- Hoyos, D. von (2017). "Interactive Visualizations for supporting the analysis of distributed services utilization". Master's Thesis. Munich, Germany.
- Hrischuk, C. E., C. Murray Woodside, and J. A. Rolia (1999). "Trace-based load characterization for generating performance software models". In: *IEEE Transactions on Software Engineering* 25.1, pp. 122–135.
- Humble, J. and D. Farley (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series (Fowler). Pearson Education.
- Humble, J. and J. Molesky (2011). "Why enterprises must adopt devops to enable continuous delivery". In: *Cutter Business Technology Journal* 24, pp. 6–12.
- Hüttermann, M. (2012). *DevOps for Developers*. Expert's voice in Web development. Apress.
- Ihde, S. (2015). *From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture*. URL: <http://www.infoq.com/presentations/linkedin-microservices-urn> (visited on 02/14/2020).
- Inc., A. (2020). *AppDynamics APIs*. URL: <https://docs.appdynamics.com/display/PR045/AppDynamics+APIs> (visited on 02/14/2020).
- Inc., D. (2020). *Dynatrace API*. URL: <https://www.dynatrace.com/support/help/dynatrace-api/> (visited on 02/14/2020).
- Inc., N. (2020). *Rest API v2*. URL: <https://docs.newrelic.com/docs/apis/rest-api-v2> (visited on 02/14/2020).
- Israr, T., M. Woodside, and G. Franks (2007). "Interaction Tree Algorithms to Extract Effective Architecture and Layered Performance Models from Traces". In: *Journal of Systems and Software* 80.4, pp. 474–492.

- Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley Sons.
- Janietz, C. (2018). "Enhancing enterprise architecture models using application performance monitoring data". Master's Thesis. Munich, Germany.
- Johnson, P., M. Ekstedt, and R. Lagerstrom (2016). "Automatic Probabilistic Enterprise IT Architecture Modeling: A Dynamic Bayesian Networks Approach". In: *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pp. 1–8.
- Josephsen, D. (2007). *Building a Monitoring Infrastructure with Nagios*. Upper Saddle River, NJ, USA: Prentice Hall PTR.
- Kaner, C., J. Falk, and H. Nguyen (2000). *Testing Computer Software Second Edition*. Dreamtech Press.
- Karmel, A., R. Chandramouli, and M. Iorga (2018). *NIST Definition of Microservices, Application Containers and System Virtual Machines*. Tech. rep. NIST, Information Technology Laboratory. URL: https://csrc.nist.gov/CSRC/media/Publications/sp/800-180/draft/documents/sp800-180_draft.pdf (visited on 02/14/2020).
- Keller, W. (2017). *IT-Unternehmensarchitektur: Von der Geschäftsstrategie zur optimalen IT-Unterstützung*. dpunkt.verlag.
- Kermarrec, A.-M. and M. van Steen (2007). "Gossiping in Distributed Systems". In: *ACM SIGOPS Operating Systems Review* 41.5, pp. 2–7.
- Keuper, F., C. Oecking, and A. Degenhardt (2011). *Application Management: Challenges - Service Creation - Strategies*. Gabler Verlag.
- Kitchenham, B., L. Pickard, and S. L. Pfleeger (1995). "Case studies for method and tool evaluation". In: *IEEE Software* 12.4, pp. 52–62.
- Kleehaus, M., N. Corpancho Villasana, F. Matthes, and D. Huth (2020). "Discovery of Microservice-based IT Landscapes at Runtime: Algorithms and Visualizations". In: *53rd Annual Hawaii International Conference on System Sciences (HICSS)*. Hawaii.
- Kleehaus, M., M. Hauder, Ö. Uludag, F. Matthes, and N. Corpancho Villasana (2019a). "IT Landscape Discovery via Runtime Instrumentation for Automating Enterprise Architecture Model Maintenance". In: *Twenty-fifth Americas Conference on Information Systems (AMCIS)*. Cancun, Mexico.
- Kleehaus, M., J. Landthaler, D. Huth, and F. Matthes (2016). *State of the Art Report: Multi-Layer Monitoring and Visualization*. Tech. rep. Munich, Germany: Software Engineering for Business Information Systems (sebis).
- Kleehaus, M. and F. Matthes (2019b). "Challenges in Documenting Microservice-based IT Landscape: A Survey from an Enterprise Architecture Management Perspective". In: *23RD IEEE International EDOC Conference - The Enterprise Computing Conference (EDOC)*. Paris, France.
- Kleehaus, M. and F. Matthes (2021). "Automated Enterprise Architecture Model Maintenance via Runtime IT Discovery". In: *Architecting the Digital Transformation: Digital Business, Technology, Decision Support, Management*. Springer International Publishing, pp. 247–263.

- Kleehaus, M., Ö. Uludağ, and F. Matthes (2018a). "Towards a Continuous Feedback Loop for Microservice-based Environments". In: *11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pp. 126–134.
- Kleehaus, M., Ö. Uludag, P. Schäfer, and F. Matthes (2018b). "MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-Based Environments". In: *CAiSE Forum*.
- Knight, C. and M. Munro (2000). "Virtual but visible software". In: *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pp. 198–205.
- Knoche, H. and W. Hasselbring (2017). *Treiber und Hindernisse für die Einführung von Microservices in der deutschen Softwareindustrie*. URL: <http://eprints.uni-kiel.de/38682/1/tr-1702.pdf> (visited on 02/14/2020).
- Kotonya, G. and I. Sommerville (1998). *Requirements Engineering: Processes and Techniques*. 1st. Wiley Publishing.
- Kramer, S. (2011). *The Biggest Thing Amazon Got Right: The Platform*. URL: <https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/> (visited on 02/14/2020).
- Krippendorff, K. (2004). *Content Analysis: An Introduction to Its Methodology*. Content Analysis: An Introduction to Its Methodology. Sage.
- Kruchten, P. (2004). *The Rational Unified Process: An Introduction*. Addison-Wesley object technology series. Addison-Wesley.
- Kurapati, N., V. Manyam, and K. Petersen (2012). "Agile Software Development Practice Adoption Survey". In: vol. 111, pp. 16–30.
- Lam, W. (2004). "Technical Risk Management on Enterprise Integration Projects". In: *Communications of the Association for Information Systems* 13.
- Landthaler, J., Ö. Uludag, G. Bondel, A. Elnaggar, S. Nair, and F. Matthes (2018). "A Machine Learning Based Approach to Application Landscape Documentation". In: *11th Working conference on the Practice of Enterprise Modelling (PoEM)*.
- Langermeier, M. and B. Bauer. (2017). "Generic EA Analysis Framework for the Definition and Automatic Execution of Analyses". In: *Proceedings of the 19th International Conference on Enterprise Information Systems - Volume 1: ICEIS, INSTICC*. SciTePress, pp. 316–327.
- Lankes, J., F. Matthes, and A. Wittenburg (2005). "Softwarekartographie: Systematische Darstellung von Anwendungslandschaften". In: *Wirtschaftsinformatik*, pp. 1443–1462.
- Lankhorst, M. (2017). *Enterprise Architecture at Work: Modelling, Communication and Analysis*. 4th. Springer Publishing Company, Incorporated.
- Lankhorst, M. M., H. A. Proper, and H. Jonkers (2010). "The Anatomy of the ArchiMate Language". In: *International Journal of Information System Modeling and Design* 1, pp. 1–32.
- Leite, L., C. Rocha, F. Kon, D. Milojicic, and P. Meirelles (2020). "A Survey of DevOps Concepts and Challenges". In: *ACM Computing Surveys (CSUR)* 52.6, pp. 1–35.
- Lemos, R. de, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S.

- Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, and J. Wuttke (2013). "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap". In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by R. de Lemos, H. Giese, H. A. Müller, and M. Shaw. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–32.
- Li, W., Y. Lemieux, J. Gao, Z. Zhao, and Y. Han (2019). "Service Mesh: Challenges, State of the Art, and Future Research Opportunities". In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 122–1227.
- Likert, R. (1932). *A Technique for the Measurement of Attitudes*. A Technique for the Measurement of Attitudes Nr. 136-165. publisher not identified.
- Lilja, D. J. (2005). *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University.
- Linden, G. (2006). *Marissa Mayer at Web 2.0*. URL: <http://glinden.blogspot.de/2006/11/marissa-mayer-at-web-20.html> (visited on 09/14/2019).
- Linkevics, G. (2014). "Adopting to Agile Software Development". In: *Applied Computer Systems* 16.
- Lucke, C., S. Krell, and U. Lechner (2010). "Critical Issues in Enterprise Architecting - A Literature Review." In: *AMCIS 2010 Proceedings*. Vol. 4, p. 305.
- Ludewig, J. (2004). "Models in software engineering - An introduction". In: *Informatik Forschung und Entwicklung* 18, pp. 105–112.
- Ly, L., F. Maggi, M. Montali, S. Rinderle-Ma, and W. Aalst (2015). "Compliance monitoring in business processes: Functionalities, application, and tool-support". In: *Information Systems* 14.
- Machner, N. (2019). "Assessing the Cost and Benefit of a Microservice Landscape Discovery Method in the Automotive Industry". Master's Thesis. Munich, Germany.
- Maes, P. (1987). "Concepts and Experiments in Computational Reflection". In: *SIGPLAN Notices* 22.12, pp. 147–155.
- Makanju, A., A. N. Zincir-Heywood, and E. E. Milios (2013). "Investigating event log analysis with minimum apriori information". In: *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pp. 962–968.
- Makanju, A. A., A. N. Zincir-Heywood, and E. E. Milios (2009). "Clustering Event Logs Using Iterative Partitioning". In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: Association for Computing Machinery, pp. 1255–1264.
- Mannmeusel, T. (2012). "Management von Unternehmensarchitekturen in der Praxis: Organisatorische Herausforderungen in mittelständischen Unternehmen". In: *Analyse und Gestaltung leistungsfähiger IS-Architekturen: Modellbasierte Methoden aus Forschung und Lehre in der Praxis*. Ed. by C. Suchan and J. Frank. Berlin, Heidelberg: Springer, pp. 35–57.
- Mappic, S. (2011). *The Real Overhead of Managing Application Performance*. URL: <https://www.appdynamics.com/blog/product/the-real-overhead-of-managing-application-performance/> (visited on 03/04/2020).

- Markus, M. L., A. Majchrzak, and L. Gasser (2002). "A Design Theory for Systems That Support Emergent Knowledge Processes". In: *MIS Quarterly: Management Information Systems* 26.3, pp. 179–212.
- Matthes, F., S. Buckl, J. Leitel, and C. M. Schweda (2020). *Enterprise Architecture Management Tool Survey 2008*. Tech. rep. Technical University Munich. URL: <https://www.matthes.in.tum.de/pages/1wdia0twywb0w/Enterprise-Architecture-Management-Tool-Survey-2008-EAMTS-2008> (visited on 02/14/2020).
- Mayer, B. and R. Weinreich (2018). "An Approach to Extract the Architecture of Microservice-Based Software Systems". In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pp. 21–30.
- Mayring, P. (2010). *Qualitative Inhaltsanalyse: Grundlagen und Techniken*. Beltz.
- Mayring, P. (2012). "Mixed Methods. Ein Plädoyer für gemeinsame Forschungsstandards qualitativer und quantitativer Methoden." In: *Mixed Methods in der empirischen Bildungsforschung*. Ed. by M. Gläser-Zikuda, T. Seidel, C. Rohlf, A. Gröschner, and S. Ziegelbauer. Münster: Waxmann, pp. 287–300.
- Mcafee, A. (2007). "Enterprise 2.0: The Dawn of Emergent Collaboration". In: *Engineering Management Review, IEEE* 47, pp. 38–38.
- Medvidovic, N. and R. N. Taylor (2000). "A classification and comparison framework for software architecture description languages". In: *IEEE Transactions on Software Engineering* 26.1, pp. 70–93.
- Medvidovic, N., A. Egyed, and P. Grünbacher (2003). "Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery". In: *Proc. of the 2nd International Software Requirements to Architectures Workshop*.
- Meilich, A. (2006). "System of systems (SoS) engineering architecture challenges in a net centric environment". In: *2006 IEEE/SMC International Conference on System of Systems Engineering*.
- Mell, P. M. and T. Grance (2011). *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, USA.
- Menascé, D. and V. Almeida (2002). *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR.
- Meshenberg, R. (2016). *Microservices at Netflix Scale: Principles, Tradeoffs Lessons Learned*. URL: <https://cloudbestpractices.net/microservices-at-netflix-scale/> (visited on 02/14/2020).
- Mingers, J. (2001). "Combining IS Research Methods: Towards a Pluralist Methodology". In: *Information Systems Research* 12, pp. 240–259.
- Mira da Silva, M., N. Silva, F. Ferreira, and P. Sousa (2016). "Automating the Migration of Enterprise Architecture Models". In: *International Journal of Information System Modeling and Design* 7.2, pp. 72–90.
- Moody, D. (2010). "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Transactions on Software Engineering* 35, pp. 756–779.
- Moran, A. (2014). *Agile Risk Management*. Springer Briefs in Computer Science. Springer International Publishing.

- Morgan, W. (2017). *What's a service mesh? And why do I need one?* URL: <https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/> (visited on 02/14/2020).
- Moser, C., S. Junginger, M. Brückmann, and K.-M. Schöne (2009). "Some Process Patterns for Enterprise Architecture Management". In: *Software Engineering*, pp. 19–30.
- Munaf, R. M., J. Ahmed, F. Khakwani, and T. Rana (2019). "Microservices Architecture: Challenges and Proposed Conceptual Design". In: *2019 International Conference on Communication Technologies (ComTech)*, pp. 82–87.
- Naranjo, D., M. Sánchez, and J. Villalobos (2015). "PRIMROSe: A Graph-Based Approach for Enterprise Architecture Analysis". In: *Enterprise Information Systems*. Ed. by J. Cordeiro, S. Hammoudi, L. Maciaszek, O. Camp, and J. Filipe. Springer International Publishing, pp. 434–452.
- Newman, S. (2015). *Building Microservices*. 1st. O'Reilly Media, Inc.
- Niemann, K. (2005). *Von der Unternehmensarchitektur zur IT-Governance: Bausteine für ein wirksames IT-Management*. Edition CIO. Vieweg+Teubner Verlag.
- Niemi, E. (2007). "Enterprise Architecture Stakeholders - a Holistic View". In: p. 41.
- Nigel, C. (2017). *Cross-Border Data Flows: Where Are the Barriers, and What Do They Cost?* URL: <https://itif.org/publications/2017/05/01/cross-border-data-flows-where-are-barriers-and-what-do-they-cost> (visited on 02/14/2020).
- Nygaard, M. (2007). *Release It!: Design and Deploy Production-ready Software*. Pragmatic Bookshelf Series.
- O'Hanlon, C. (2006). "A Conversation with Werner Vogels". In: *Queue* 4.4, pp. 14–22.
- Object Management Group (2011). *Business Process Model And Notation (BPMN)*. URL: <https://www.omg.org/spec/BPMN/2.0/> (visited on 02/14/2020).
- Office, C. (2011a). *ITIL Service Design 2011 Edition*. Norwich: The Stationery Office.
- Office, C. (2011b). *ITIL Service Operation 2011 Edition*. Norwich: The Stationery Office.
- Office, C. (2011c). *ITIL Service Transition 2011 Edition*. Norwich: The Stationery Office.
- Ohno, T. (1988). *Toyota Production System: Beyond Large-Scale Production*. Taylor & Francis.
- Pahl, C. and P. Jamshidi (2016). "Microservices: A Systematic Mapping Study". In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*. CLOSER 2016. Rome, Italy, pp. 137–146.
- Palmer, S. and J. Felsing (2002). *A Practical Guide to Feature-driven Development*. The Coad Series. Prentice Hall PTR.
- Pargaonkar, V. and K. Ramakrishnan (2012). *AUTOMATIC SELECTION OF AGENT-BASED OR AGENTLESS MONITORING*. URL: <https://patents.google.com/patent/US20120016706A1> (visited on 02/14/2020).
- Peffers, K., T. Tuunanen, C. Gengler, M. Rossi, W. Hui, V. Virtanen, and J. Bragge (2006). "The design science research process: A model for producing and presenting information systems research". In: *Proceedings of First International Conference on Design Science Research in Information Systems and Technology DESRIST*.
- Porter, J., D. A. Menascé, and H. Gomaa (2016). "DeSARM: A Decentralized Mechanism for Discovering Software Architecture Models at Runtime in Distributed Systems". In: *Models@Run.time*.

- Quinlan, J. (1987). "Simplifying decision trees". In: *International Journal of Man-Machine Studies* 27.3, pp. 221–234.
- Rabl, T., S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii (2012). "Solving Big Data Challenges for Enterprise Application Performance Management". In: *CoRR* abs/1208.4167.
- Raibulet, C., F. A. Fontana, and M. Zanoni (2017). "Model-Driven Reverse Engineering Approaches: A Systematic Literature Review". In: *IEEE Access* 5, pp. 14516–14542.
- Reidemeister, T., Miao Jiang, and P. A. S. Ward (2011). "Mining unstructured log files for recurrent fault diagnosis". In: *12th IFIP/IEEE International Symposium on Integrated Network Management*, pp. 377–384.
- Reidemeister, T., M. A. Munawar, M. Jiang, and P. A. S. Ward (2009). "Diagnosis of Recurrent Faults Using Log Files". In: *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*. USA: IBM Corp., pp. 12–23.
- Richardson, C. (2018). *Microservices Patterns: With examples in Java*. Manning Publications.
- Riungu-Kalliosaari, L., S. Mäkinen, L. E. Lwakatare, J. Tiihonen, and T. Männistö (2016). "DevOps Adoption Benefits and Challenges in Practice: A Case Study". In: *Proceedings of the International Conference on Product-Focused Software Process Improvement (PROFES 2016)*, pp. 590–597.
- Ros, J. P. and R. S. Sangwan (2011). "A Method for Evidence-Based Architecture Discovery". In: *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pp. 342–345.
- Ross, J., P. Weill, and D. Robertson (2006). *Enterprise Architecture as Strategy — Creating a Foundation for Business Execution*.
- Roth, S. (2014). "Federated Enterprise Architecture Model Management". Dissertation. Munich, Germany.
- Roth, S., M. Hauder, M. Farwick, R. Breu, and F. Matthes (2013a). "Enterprise Architecture Documentation: Current Practices and Future Directions". In: *Wirtschaftsinformatik*.
- Roth, S., M. Hauder, F. Michel, D. Münch, and F. Matthes (2013b). "Facilitating Conflict Resolution of Models for Automated Enterprise Architecture Documentation". In: *American Conference on Information Systems (AMCIS)*.
- Roth, S. and F. Matthes (2014). "Visualizing Differences of Enterprise Architecture Models". In: *Softwaretechnik-Trends* 34.
- Roth, S., M. Zec, and F. Matthes (2020). *Enterprise Architecture Visualization Tool Survey 2014*. Tech. rep. Technical University Munich. URL: <https://www.matthes.in.tum.de/pages/o790x7rho1te/EAVTS-2014-Final-Report> (visited on 02/14/2020).
- Runeson, P. and M. Höst (2008). "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2, p. 131.
- Sadalage, P. and M. Fowler (2013). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Always learning. Addison-Wesley.
- Salah, T., M. Jamal Zemerly, Chan Yeob Yeun, M. Al-Qutayri, and Y. Al-Hammadi (2016). "The evolution of distributed systems towards microservices architecture". In: *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pp. 318–325.

- Santana, A., K. Fischbach, and H. Moura (2016). "Enterprise Architecture Analysis and Network Thinking: A Literature Review". In: *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pp. 4566–4575.
- Schaefer, R. (2016). *From Monolith to Microservices, Zalando's Journey*. URL: <https://www.infoq.com/news/2016/02/Monolith-Microservices-Zalando/> (visited on 02/14/2020).
- Schäfer, P. (2017). "Eine prototypische Implementierung zur Erkennung von Architekturänderungen eines verteilten Systems basierend auf unterschiedlichen Monitoring Datenquellen". Master's Thesis. Munich, Germany.
- Schermann, G., J. Cito, and P. Leitner (2015). "All the Services Large and Micro: Revisiting Industrial Practice in Services Computing". In: *ICSOC Workshops*.
- Schreiner, M., T. Hess, and A. Benlian (2015). *Gestaltungsorientierter Kern oder Tendenz zur Empirie? Zur neueren methodischen Entwicklung der Wirtschaftsinformatik*. Tech. rep. 1/2015. München.
- Schwaber, K. (2004). *Agile Project Management with Scrum*. Developer Best Practices. Pearson Education.
- Schwaber, K. and M. Beedle (2002). *Agile Software Development with Scrum*. Agile Software Development. Prentice Hall.
- Shah, H. and M. El Kourdi (2007). "Frameworks for Enterprise Architecture". In: *IT Professional 9.5*, pp. 36–41.
- Shahin, M., M. A. Babar, and L. Zhu (2017). "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access 5*, pp. 3909–3943.
- Sharma, S. and B. Coyne (2015). *DevOps*. John Wiley Sons.
- Shull, F., J. Carver, and G. H. Travassos (2001). "An Empirical Methodology for Introducing Software Processes". In: *SIGSOFT Software Engineering Notes 26.5*, pp. 288–296.
- Sigelman, B. H., L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag (2010). *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc.
- Silva, N. M., P. Sousa, and M. M. da Silva (2020). "Maintenance of enterprise architecture models as tools to support enterprise transformation: A systematic literature review". In: *Business Information Systems Engineering (BUIS)*.
- Smeds, J., K. Nybom, and I. Porres (2015). "DevOps: A Definition and Perceived Adoption Impediments". In: *Proceedings of the International Conference on Agile Processes in Software Engineering and Extreme Programming (XP 2015)*. Springer International Publishing, pp. 166–177.
- Smith, H. J., T. Dinev, and H. Xu (2011). "Information Privacy Research: An Interdisciplinary Review". In: *MIS Quarterly 35.4*, pp. 989–1016.
- Society, I. C. (2000). "IEEE Recommended Practice for Architectural Description for Software-Intensive Systems". In: *IEEE Std 1471-2000*, pp. 1–30.
- Sousa, P., R. Gabriel, G. Tadao, R. Carvalho, P. M. Sousa, and A. Sampaio (2011). "Enterprise Transformation: The Serasa Experian Case". In: *Practice-Driven Research on Enterprise*

- Transformation*. Ed. by F. Harmsen, K. Grahlmann, and E. Proper. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 134–145.
- Sousa, T. B., H. S. Ferreira, F. F. Correia, and A. Aguiar (2017). “Engineering Software for the Cloud: Messaging Systems and Logging”. In: *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. EuroPLOP '17. Irsee, Germany: Association for Computing Machinery.
- Stachowiak, H. (1973). *Allgemeine Modelltheorie*. Wien, New York: Springer Verlag.
- Ståhl, D. and J. Bosch (2014). “Modeling Continuous Integration Practice Differences in Industry Software Development”. In: *Journal of System Software* 87, pp. 48–59.
- Stapleton, J. (1997). *DSDM, Dynamic Systems Development Method: The Method in Practice*. Addison-Wesley.
- Strauss, A. (1987). *Qualitative Analysis for Social Scientists*. Cambridge University Press.
- Tao, F., J. Cheng, Q. Qi, M. Zhang, H. Zhang, and F. Sui (2018). “Digital twin-driven product design, manufacturing and service with big data”. In: *The International Journal of Advanced Manufacturing Technology* 94.
- Taylor, R., N. Medvidovic, and E. Dashofy (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley.
- Templeton, G. F., C.-P. Lee, and C. A. Snyder (2006). “Validation of a Content Analysis System Using an Iterative Prototyping Approach to Action Research”. In: *CAIS* 17, p. 24.
- The Open Group (2016). *ArchiMate 3.1 Specification*. URL: <https://pubs.opengroup.org/architecture/archimate3-doc/> (visited on 11/18/2019).
- The Open Group (2019). *The Open Group IT4IT Reference Architecture, Version 2.0*. URL: <https://publications.opengroup.org/c155> (visited on 11/18/2019).
- Thomas, O. (2005). *Das Modellverständnis in der Wirtschaftsinformatik: Historie, Literaturanalyse und Begriffsexplikation*. Tech. rep. Saarbrücken.
- Tufte, E. (2001). *The Visual Display of Quantitative Information*. Graphics Press.
- Ulich, D. (1985). *Psychologie der Krisenbewältigung: eine Längsschnittuntersuchung mit arbeitsslosen Lehrern*. Beltz.
- Vaarandi, R. (2003). “A data clustering algorithm for mining patterns from event logs”. In: *Proceedings of the 3rd IEEE Workshop on IP Operations Management (IPOM 2003)*, pp. 119–126.
- Vaarandi, R. and M. Pihelgas (2015). “LogCluster - A data clustering and pattern mining algorithm for event logs”. In: *2015 11th International Conference on Network and Service Management (CNSM)*, pp. 1–7.
- Välja, M., R. Lagerström, M. Ekstedt, and M. Korman (2015). “A Requirements Based Approach for Automating Enterprise IT Architecture Modeling Using Multiple Data Sources”. In: *2015 IEEE 19th International Enterprise Distributed Object Computing Workshop*, pp. 79–87.
- Venkatesh, V., S. Brown, and H. Bala (2013). “Bridging the Qualitative-Quantitative Divide: Guidelines for Conducting Mixed Methods Research in Information Systems”. In: *MIS Quarterly: Management Information Systems* 37, pp. 21–54.

- Völter, M., T. Stahl, J. Bettin, A. Haase, S. Helsen, K. Czarnecki, and B. von Stockfleth (2013). *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. Wiley.
- Wehrens, O. (2017). *Service Discovery for Machines and Humans*. URL: <http://pivio.io/> (visited on 02/14/2020).
- Weske, M. (2007). *Business Process Management: Concepts, Languages, Architectures*. Berlin, Heidelberg: Springer.
- Wiese, R., M. Eiglsperger, and M. Kaufmann (2001). "yFiles: Visualization and Automatic Layout of Graphs". In: vol. 2265.
- Wilson, P., L. Dell, and G. Anderson (1993). *Root Cause Analysis: A Tool for Total Quality Management Workbook*. ASQC Quality Press.
- Winter, K., S. Buckl, F. Matthes, and C. M. Schweda (2010). "Investigating the State-of-the-Art in Enterprise Architecture Management Methods in literature and Practice." In: *MCIS 90*.
- Winter, R. and R. Fischer (2007). "Essential Layers, Artifacts, and Dependencies of Enterprise Architecture". In: *Journal of Enterprise Architecture* 3.2, pp. 7–18.
- Wise, M. J. and T. McDermott (1993). "Message-brokers: a novel interprocess communications primitive". In: *Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences*. Vol. 2, pp. 184–193.
- Wittenburg, A. (2007). "Softwarekartographie: Modelle und Methoden zur systematischen Visualisierung von Anwendungslandschaften". Dissertation. Munich, Germany.
- Wohlin, C., P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.
- Xiao, Z., I. Wijegunaratne, and X. Qiang (2016). "Reflections on SOA and Microservices". In: *2016 4th International Conference on Enterprise Systems (ES)*, pp. 60–67.
- Yin, R. and SAGE. (2003). *Case Study Research: Design and Methods*. Applied Social Research Methods. SAGE Publications.
- Young, R. and S. Poon (2013). "Top management support—almost always necessary and sometimes sufficient for success: Findings from a fuzzy set analysis". In: *International Journal of Project Management* 31.7, pp. 943–957.
- Yu, Y., H. Silveira, and M. Sundaram (2016). "A microservice based reference architecture model in the context of enterprise architecture". In: *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pp. 1856–1860.
- Zachman, J. A. (1987). "A framework for information systems architecture". In: *IBM Systems Journal* 26.3, pp. 276–292.
- Zajicek, M. (2007). "Web 2.0: Hype or Happiness?" In: *Proceedings of the 2007 International Cross-Disciplinary Conference on Web Accessibility (W4A)*. New York, NY, USA: Association for Computing Machinery, pp. 35–39.
- Zhou, J., Z. Chen, H. Mi, and J. Wang (2014). "MTracer: A Trace-Oriented Monitoring Framework for Medium-Scale Distributed Systems". In: *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pp. 266–271.