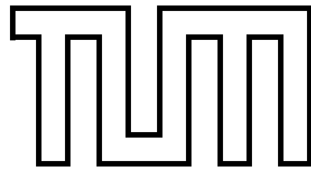


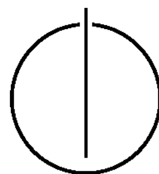
FAKULTÄT FÜR INFORMATIK

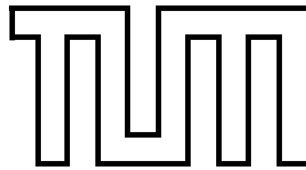
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Information Systems

Enhancing enterprise architecture models using application performance monitoring data

Christopher Janietz





FAKULTÄT FÜR INFORMATIK

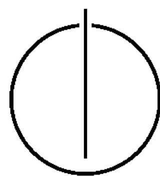
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Information Systems

Enhancing enterprise architecture models using application performance monitoring data

Erweiterung von Enterprise Architekturmodellen um Daten aus dem Applikations Performance Monitoring

Author:	Christopher Janietz
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	M.Sc. Martin Kleehaus
Date:	October 15, 2018



Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 23. Oktober 2018 Christopher Janietz

Acknowledgments

In the creation of this thesis a lot of people were involved whom I would like to thank very much. First of my advisor Martin Kleehaus with whom I was able to exchange lots of different ideas on the matter and helped me in conceptualizing them. Further I would like to thank my colleagues and other supportive people in the company in which a lot of the content of this thesis was created in. Finally I would like to thank my family and friends who encouraged me to continue the journey, even if sometimes times were tough. Especially I would like to thank namely Andreas Geroe, Emanuel Kechter, Dominik Pusch, Ralf Hecktor, Josua Nuernberger and Maximilian Scheler.

Abstract

With the advent of microservices, more and more parts of the overall architecture are not designed upfront, but rather product of continuous evolution of the application landscape. For this reason especially for the practice of enterprise architecture it becomes increasingly difficult to cope with the amount of changes and reflect them accordingly in architecture model designs. This thesis presents an automated approach to help integrate and update these respective models from data that can be extracted from the runtime architecture. More specifically the application performance management (APM) system and its capabilities of runtime service instrumentation are used in an approach to build an integration layer and thereby enhance information which is maintained in existing enterprise architecture tooling. This integration layer allows to explore information from both worlds on the basis of a linked graph, enabling a new field of use cases and analysis capabilities to explore.

Contents

Acknowledgements vii

Abstract ix

Outline of the Thesis xiii

I. Introduction and Theory 1

1. Introduction 3
- 1.1. Motivation . . . 3
  - 1.1.1. Terms and definitions . . . 4
  - 1.1.2. Research goals . . . 5
  - 1.1.3. Related work . . . 6
- 1.2. Foundations . . . 8
  - 1.2.1. APM . . . 8
  - 1.2.2. EAM . . . 12

II. Conceptualization 15

2. Conceptualization 17
- 2.1. Requirements . . . 17
  - 2.1.1. General requirements . . . 17
  - 2.1.2. Stakeholder requirements . . . 17
- 2.2. Prerequisites . . . 18
- 2.3. Product organization model . . . 20
- 2.4. Data integration automation . . . 22
- 2.5. Data integration workflow . . . 26
  - 2.5.1. Creation workflow . . . 26
  - 2.5.2. Read workflow . . . 26
  - 2.5.3. Update workflow . . . 27
  - 2.5.4. Deletion workflow . . . 28
  - 2.5.5. Synchronization . . . 28
- 2.6. Enterprise graph . . . 28

III. Implementation 31

3. Implementation 33
3.1. Architecture overview 33
3.2. Technologies 35
3.3. Unified data model 36
3.4. Provider model 37
3.5. Inferencing model 40
3.6. Enterprise graph implementation 42
3.6.1. Interface implementation 42
3.6.2. Interface usage 43
3.7. Synchronization 45

IV. Evaluation, Limitations and Outlook 47

4. Evaluation 49
4.1. Case study: Multichannel retailer 49
4.2. Application of APEAM 50
4.2.1. Naming convention 50
4.2.2. EAM integration workflow 51
4.3. Interviews 51
4.3.1. Questions and descriptions 51
4.3.2. Results 54

5. Limitations and Outlook 55
5.1. Limitations 55
5.2. Outlook 55

6. Conclusion 57

Appendix 61

A. Interview results 61
A.1. Interview: Backend Developer - Checkout 62
A.2. Interview: Backend Developer - Search 64
A.3. Interview: DevOps/SRE 66
A.4. Interview: Product Owner 68
A.5. Interview: Enterprise Architect 69

Bibliography 71

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION

This chapter presents an the motivation and overall goals for this thesis. It also lays the foundations in terms of existing research and theory in the area of interest and the tools which were utilized for the realization.

Part II: Conceptualization

CHAPTER 2: CONCEPTUALIZATION

This chapter describes different concepts and approaches to solving the identified problems. It lays the groundwork for the implementation of a prototype.

Part III: Implementation

CHAPTER 3: IMPLEMENTATION

This chapter shows how the prototype APEAM was realized as a result of the foundations and in the context of a concrete application field.

Part IV: Evaluation, Limitations and Outlook

CHAPTER 4: EVALUATION

This chapter describes the context the research was conducted in and how the prototype was applied and evaluated.

CHAPTER 5: LIMITATIONS AND OUTLOOK

This chapter shows the limits of the presented concepts and implementation and what potential improvements could be made to the presented tool and what further research options are open.

CHAPTER 6: CONCLUSION

This chapter concludes the thesis and gives a short overview on the results.

Part I.

Introduction and Theory

1. Introduction

1.1. Motivation

EAM (Enterprise Architecture Management) is a practice seen in large to mid-sized organizations which aims to document and manage the complexity of the IT landscape in relation to the business requirements[6]. As there has been a continuous decrease of time to market in the IT-industry in recent years, this discipline faces a multitude of new challenges. New software development practices such as microservices, as well as delegation of architectural decisions to teams, makes architecture designs suffer more than ever from the problem of capturing the full essence of an ever-changing landscape. Furthermore agile practices support an increasing deviation from architecture designs due to changing business goals or technical implementation differences. The result of this is either missing or outdated documentation [15].

To cope with the general complexities of enterprise architecture documentation, organizations use EA (Enterprise Architecture) tool support [16]. EA tools range from very simple diagram software, such as Microsoft Visio, to complete EAM software suites such as iteraplan or planningIT. While these EAM tools help with the capturing of model entities, visualization and analysis of the architecture as a whole, they still require manual input and therefore need to be kept up to date with changes occurring in the architecture, which were not originally envisioned.

Even though this type of tool support exists and leaving the challenges of agile practices out of the picture, the practice of EA has already been seeing the prescribed types of challenges in the past. A study in 2010, while agile and microservice architectures were not yet popular, has found similar issues across the industry [19]. In the course of the evaluation (4.3) in this research these type of issues could be validated up to this day and have lately become even more complex with the advent of microservices.

In contrast to the developments in the EA and EAM tool domain, APM (Application Performance Management) software has become more powerful and automated. Tools such as dynatrace can discover whole runtime architectures just by the instrumentation of the respective running software components. By simply comparing runtime architecture as seen and discovered by the APM software to the one defined by the EAM software, possibly differences can be identified.

At exactly this point, enterprise architecture documentation can then be augmented with information from runtime instrumentation systems, to further gain access to information, such as Domain association, which otherwise is not available from the runtime alone. In this research ways to link these two types of system are explored. Both of these type of information sources have inherently different goals when it comes to consistency and liveliness. For this purpose, a model is established which allows to map from low level monitoring information derived from service communication to high level architecture descriptions, such as business domains. In a prototypical implementation, APEAM

(Application Performance Enterprise Architecture Management) is presented, which symbolizes the synthesis of both the APM and EAM world. Using the notion of an "unified service", APEAM connects the description of services from both the EAM and APM perspective and maintains this type of reference. On the basis of these references, this research further explores the notion of an "enterprise graph", which allows to query and traverse resources owned by both the APM and EAM system.

By automating the process of identifying differences between both worlds and synchronizing newly discovered services into the EAM repository, it further is possible to draw conclusions in the differences between current and planned architecture. This research defines workflows so that exactly this is possible.

For validating the proposed approach, the implementation is done and evaluated in the context of a large European retailer, which is utilizing a microservice based architecture in the context of a product based organization model. As exemplary data sources, dynatrace is used as the APM and planningIT as the EAM software.

1.1.1. Terms and definitions

In this research terms will be used which could be interpreted in different ways and are also used inconsistently by some of the involved software components. Due to this in the following paragraphs definitions will be established which represent a consistent interpretation, as used for this thesis, unless denoted otherwise. Whenever the definition of a term is dependent on the context of the current section and therefore not the following definitions it will be denoted in *italics*.

Runtime architecture. The runtime architecture is the architecture the way it is implicitly given by the running software artifacts and their respective backings (e.g. Server, Database, ...). It therefore will be considered the *actual* architecture. The term will usually be used when referring to the architecture as discovered by the APM system.

Design architecture. In contrast the design architecture will be the architecture as originally envisioned by architects either before or after the implementation. This term is a mixture of current and planned architecture but in relation to the architecture design in the EAM tool.

Business Architecture. The business architecture is the high level architecture and thereby describes the relationship between different business entities, such as Domains, and their respective realization such as processes. Technical components in this level of abstraction are at best described in the notion of a service or application.

Technical Architecture The technical architecture is the implementation or realization of services. In this type of architecture, services are described in their relationship to certain infrastructure components and other realization details.

Service. Service will be used as the umbrella term for a deployed software artifact, independent of the amount of instances. In the particular application scenario of this thesis

it will can be summarized as a microservice.

Service discovery. The term ”Service discovery” will be used to refer to the existence discovery of a service. Thereby aspects such as addressing the service for reaching its interfaces is left out of the picture.

Application. An application will be considered as the aggregation of multiple services to a certain ”application”. For example the application ”Webshop” will be the combination of a ”Product Display”, ”Checkout” and ”Payment” service. Different services might occur in the context of different types of applications, which essentially is the idea of reuse in microservice architectures.

API. API is very general term, which by itself only means that an interface exists for programmatic interaction between different software components. In this research the term is narrowed down to HTTP interfaces which build on the concept of REST (Representational state transfer) using JSON payloads.

1.1.2. Research goals

This research establishes three questions as result of the problems described in 1.1.

Q1. Due to an inherent disconnect between design of architectures and development of the concrete software artifacts, the relationship between the actual deployed software and the original architecture model element is immediately not apparent. As an example consider the element of a ”Payment Service”. While this naming could be used for the model element, deployed software artifacts hardly have any name at all other than for example ”application.jar”. In order to thereby define an automated process of integrating runtime artifacts to the EAM repositories, the following question is raised. *How to extract and map coherent service and infrastructure topologies from enterprise architecture management and monitoring systems?*

Q2. Monitoring data is by nature non-permanent, as it is an expression of many changes in the runtime architecture and therefore in a stark contrast to EAM information, which is rather static and expected to be stable. For this reason it becomes important to properly identify how and in which part of the software lifecycle monitoring data is most complete, resulting in the following question. *How can this new feedback channel be sufficiently integrated into the software and architecture lifecycle?*

Q3. The integration of automation into EAM processes is a precedence and therefore requires further some definitions on how to handle these new types of data. Thereby the following question is raised. *How can architecture differences be sufficiently presented to gain knowledge on differences and react accordingly?*

Research	Goal	Discovery approaches
[20]	Cloud migration	Proprietary monitoring solution with plugins and manual input
[5]	Enhancing the EAM documentation	Instrumentation of clour resource
[4]	Generation of runtime from models	Discovery by design
[1]	Generation of runtime from models	Discovery by design
[2]	Establish microservice architecture metamodel	Code analysis
[3]	Discovery of runtime architecture	Plugin architecture with different reporters
[12]	Discovery of microservice runtime architecture	Runtime instrumentation
[8]	Discovery of microservice runtime architecture	Network monitoring,
[10]	Discovery microservice runtime architecture	Tracking of service calls

Figure 1.1.: Literature overview

1.1.3. Related work

To understand the state of the art in the field of service discovery in relation to the practices of EAM, a literature research has been conducted. The terms ”service discovery”, ”automated enterprise architecture management”, ”enterprise architecture discovery”, ”real time enterprise architecture management”, ”microservice architecture discovery” and ”enterprise architecture monitoring tools” were used in combination with the ”Google Scholar” search engine. Overall this search resulted in 22 researches which bear resemblance to the goals of this research. As to the definition of service discovery in 1.1.1, literature which was related to more complex service discovery than simple existence has been neglected. The amount of research specifically dealing with pure discovery of service existence could therefore be narrowed down further to the papers described in the following. All papers will be described by the primary goal of the discovery, their respective approaches for the discovery and their difference in relation to this research. 1.1 gives an overview on all the researches which were reviewed for describing the state of the art in architecture discovery and enterprise architecture augmentation.

One of the goals for service discovery was cloud migration. As it would be desirable for cloud migrations not to happen with a ”big bang”, but rather iterative, it thereby becomes important to identify clusters of services as candidates for migration with minimal or no dependencies. In the situation of an abundance of a well defined architecture documenta-

tion, which also includes dependency relationships, this becomes a rather tedious process. For identifying such candidates [20] describe a system which based on collected monitoring metrics groups servers by their similarity. Comparable to this thesis monitoring data is utilized but on a lower level, such as CPU, memory usage and process names. The identification of services and their relationships is not extracted from actual runtime calls but rather verified manually and left to the user.

[5] approach the topic of cloud differently by using it as a source to identifying existing resources and similar to this research using it as basis for enhancing the data in an EAM software. It is done on the basis of metadata which is used to enhance descriptions of resources hosted in the cloud. The main difference to this research is the manual efforts which have to applied in initially capturing this metadata and keeping it up to date.

A complete different approach is taken by [4] and similarly by [1]. Their primary goal is similar to the goal of this research, which is the synchronization between runtime and design architecture. The approach taken however is different. As explained in the 1.1, architecture in the world assumption of this thesis is result of many implicit decisions on different levels of the software development lifecycle. In contrast the mentioned researches consider the design to be done upfront and manifest itself in nothing but architecture models. Then all running artifacts are the result of generation from these models and changes to the system architecture are always reflected in changes to these respective models. The nature of generating code from models is framed by the term ”living models”. While this idealistic approach is sensible in theory, it requires a lot design work and coordination upfront and might result in models of complexity levels which are hard to cope with. With the lessons that can be drawn from [19] it remains to be verified whether this can work in practice, as no larger scale evaluations are presented.

In [3] the authors describe the notion of an ”enterprise topology graph”. An ”enterprise topology graph” is thereby the technical documentation or otherwise technical architecture of all components in the landscape. For the automation of keeping this graph up to date, they describe a ”plugin” architecture with well defined interfaces for different data sources to provide the respective data. While they consider different data sources to build a whole picture of the technical architecture, in this research this task is completely delegated to the APM software. In a somewhat similar manner this thesis also talks about the idea of a ”enterprise graph”, there are however some major differences. The ”enterprise topology graph” is only the technical architecture, while the enterprise graph in this research also includes the business architecture. Further it also defines a structured API interface and a possible transport to realize the graph.

The first descriptions of the term microservice architecture have only been seen as recent as 2014 when coined by Martin Fowler and James Lewis[7]. It comes to no surprise, counting in the time it takes for such practices to become popular, that only few research has been done on the topic in relation to discovering architecture descriptions. When referring to the architecture discovery in this context the term ”microservice architecture reverse engineering” is used. [10], [8] and [12] are the first researches specifically dealing with this topic.

Most of the papers reviewed do not concern themselves with documenting the architec-

ture in any existing tool, but rather building their own type of documentation in contrast to using existing EAM software[12][3]. For the part of identifying the services either custom integration has to be built into the software [10][12] or requires manual efforts[5]. Overall this research is the first to concern itself specifically with the synthesis of APM and EAM.

1.2. Foundations

1.2.1. APM

"Application performance management" (APM) or also "application performance monitoring" is the practice of collecting, evaluating and interpreting performance of applications at runtime. Runtime describes the instrumentation point in the SDL (Software Development Lifecycle), implicating applications are already built into runnable artifacts and deployed to some kind of infrastructure. APM is a part of the larger overall field of monitoring. It separates itself from the realm of "infrastructure monitoring" or "traditional monitoring" by the fact that it tries infer monitoring on a higher level of abstraction than just raw metrics. Data like endpoint response errors, request amount and method call instrumentation is utilized by APM to gain insights.

There are different approaches to the collection of performance data, they can be divided into active and passive collection from the perspective of the monitored application. In the active collection process an application provides the data either by itself at different inflection points or is being instrumented by a third-party system, which hooks into such inflection points. This is the approach taken by many APM tools and respective agents. The passive scenario describes the situation where an application offers monitoring data by itself which then is collected and evaluated outside of the boundaries of the source application. This approach is often seen by tools who are based on scripts like Nagios.

The implementation of passive monitoring mechanisms into applications by developers is usually a secondary concern, as this information serves no value to the goal of implementing a particular functionality. This is the reason the need increased for monitoring tools which are more and more sophisticated to the point of what APM tools offer today. Looking at applications developed in the enterprise area, patterns in frameworks, programming languages and runtimes reoccur. In the Java world Stacks such as Java EE, Spring Boot or other Frameworks are used and allow developers to use harmonized APIs (Application Programming Interfaces) to handle platform concerns such as database access, network calls or serving web requests. The similarity in theses lower level frameworks is how APM monitoring solutions can hook into these known interfaces and collect relevant instrumentation data without being specifically aware about implementation details of the higher-level business logic. This intrinsic monitoring data again allows to also find details on level of business processes as these are represented partially or completely by transactions or endpoint calls application wise. To this end however gaining these insights how a process is related to the lower level technical details is only known to the developers.

Service discovery

Service discovery is a concern of application development in which one application wants to talk to another application but is initially not aware of the route to address it. Such a route can be either as simple as a hostname or as complex as a specific URI (Unique Resource Identifier). The problem itself stems from the dynamic modern day application environments such as seen with service distribution strategies in virtualized environments. In this case there is a disconnect between the deployment of services and their connection to each other. Service discovery is becoming even more important with the recent advancements in the domain of SOA (service-oriented architectures) and especially microservice architectures which expect developers to design smaller application parts instead of larger monolithic ones. In practice service discovery in large organizations is based on more than one single solution. Development frameworks such as Eureka[1] or ZooKeeper[2] aid developers in this process. This however poses a problem in software solutions for which the code behind can not be customized or extended such as with standard software. Further many companies have legacy applications in their portfolio which are either no longer maintained or do not map ideally to the service discovery concepts and therefore are neglected. For the aforementioned reasons the service discovery and registration mechanisms applied in practice are expected to be either incomplete or the information is distributed across different tools.

As APM software strives to provide a complete picture of every service and its dependencies, it needs to address similar concerns as service discovery in general does. Combining the fact that standard software as well self developed software are built on top known software stacks, the APM monitoring agent is one of the few components present across the software stack of a company. APM software is therefore a potential place with a wholesome picture of the runtime architecture, which is up-to-date.

Alongside the discovery of existence of different software components different APM solutions also allow to understand the communication patterns between them. The assumption here is that these APM solutions are instrumenting instances on both ends of a communication channel, e.g. a HTTP call or a JMS message. For a HTTP call in practice it means, extra HTTP headers are added on the sending and evaluated on the receiving end. A HTTP header then includes a unique identifier which is tracked across the chain of calls between different software components and allows APM software thereby to identify bottlenecks or errors. The methodology applied here is similar to the idea of distributed tracing [17] for which the open source solution Zipkin [3] operates similarly.

Dynatrace

Dynatrace[4] is one of leaders in the space of APM according to Gartner ([9]). As described in 4.1 Dynatrace is also used as the APM solution in the company of this research's case

[1]https://github.com/Netflix/eureka
[2]https://zookeeper.apache.org/
[3]https://zipkin.io/
[4]https://www.dynatrace.com/

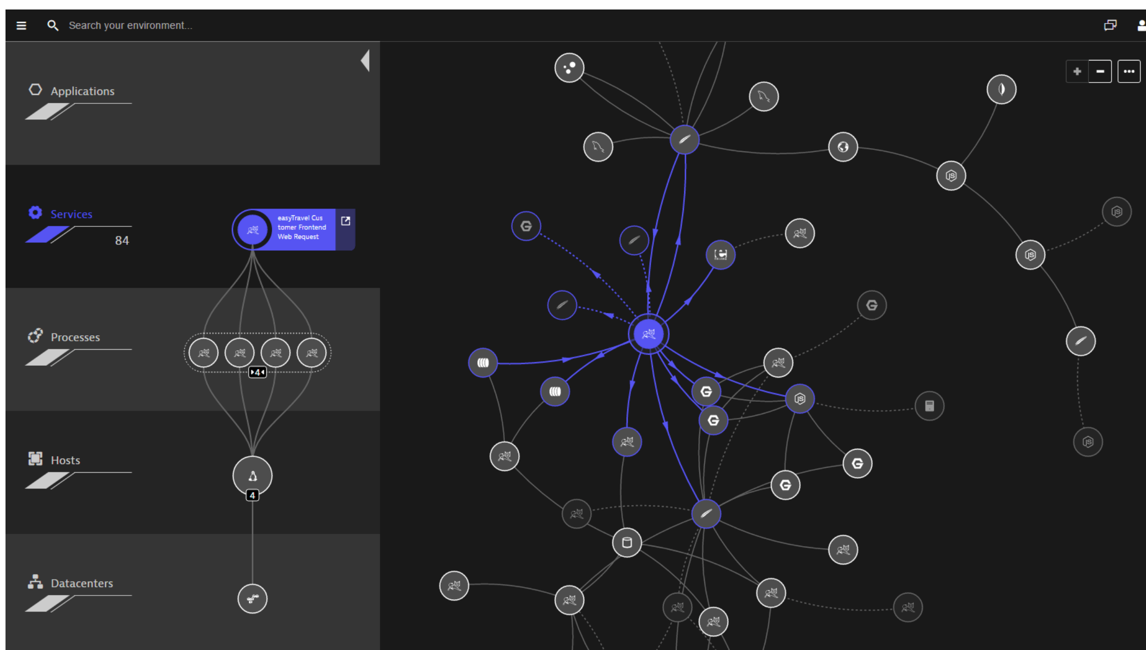


Figure 1.2.: Dynatrace Smartscape

study and is therefore the basis of the reference implementation in APEAM. Instead of the solution known as "dynatrace APM" this research specifically refers to "Dynatrace" the current solution, formerly known as "Ruxit". A few features about Dynatrace are standing out which at least today are distinguishable features when compared to other solutions on the market such as AppDynamics[5] or NewRelic[6]. The features relevant to this research will be presented in the following paragraphs.

OneAgent. The dynatrace "OneAgent" is a software component which can be installed on any type of server. Due to the way it is designed to work, processes located on these servers do not need to be configured for integration with or reporting to the agent. Instead the agent injects itself in processes of known technology patterns such as NodeJS, Java, Python and others. While it can be controlled if some types of technologies or deployments are ignored, dynatrace by default generates a complete environment picture by using this mechanism. This is different to other APM solutions, e.g. AppDynamics, as they require a manual integration which is configured on the application level.

Service backtrace. The service backtrace in dynatrace is the materialization of the distributed tracing concept. It allows to traverse relationships between different services. In case of a problem with one service it can be traced back to an originating service based on this mechanism.

Smartscape. The dynatrace "Smartscape" is a visual representation of the discovered

[5]https://www.appdynamics.com/
[6]https://newrelic.com/

components and their relationships towards each other. As such it can be seen as a visualization of the running technical architecture of a company, in case every infrastructure component is monitored with the "OneAgent". It is based on the "Service backtrace" information as well as the different components discovered by the "OneAgent". The "Smartscape" by itself makes a great basis on which to build or connect the business architecture. As the "Smartscape" by itself is generated from purely technical aspects, it becomes hard to understand how to interprete it in practice. In 1.2 it can be seen that the layers included are Applications, Services, Processes, Hosts and Datacenters. As the naming conventions used in dynatrace are relevant to the rest of this research, definitions of these terms are provided here. The definitions are only for reference and are not applied in this research unless specifically noted.

Application: Application is a wholesome picture of something a end user can use. It therefore is typically seen as a web frontend with a an URL and instrumented with a Javascript agent which is injected into the HTML payload of services who serve these resources. Thereby the application is the manifestation of the consumption of different services to offer a functionality, similar to the definition in 1.1.1.

Services: The term of services in dynatrace differs from the initial definition in 1.1.1. Dynatrace considers an API interface controller to be a "service". The differentiation is sensible from the perspective that an *Application* might consume more than one interface of the same *process group*. What is framed as *Service* in this research therefore might have multiple "services" from the perspective of dynatrace.

Processes: Processes are the running instance of the same deployment artifact. The same running artifacts build a *Process Group* in dynatrace and thereby allow to track the instances of what this research frames as a service or what otherwise would be simply the microservice.

Hosts: Hosts are instances of an operating system which runs these processes independent of the fact whether there is a Docker layer in between or not.

Datacenters: Datacenters are used differently depending on the context dynatrace operates in. For on premise deployments of virtual machines it for example is based on VMware clusters, while for cloud applications it can be a deployment region.

Docker monitoring. As Docker has become a popular choice among the deployments of applications, dynatrace offers the capability to also separate *Process Groups* on the basis of which Docker Container and the respective Image they are deployed in.

User interface. All information which is discovered and interpreted by dynatrace is available from the user interface. There visualizations such as the "Smartscape" and the "Service backtrace" can be viewed and also used in the context of discovering and interpreting incidents.

API. Dynatrace offers an API[7] which allows to explore parts of the data which has been collected and interpreted by dynatrace itself. This includes information about running services and their topology, events in the monitoring system and metrics in general. The API is however not complete when compared to the information available from the user

[7] https://www.dynatrace.com/support/help/dynatrace-api/

interface. The user interface also uses an API of its own, which is different to the aforementioned general API.

1.2.2. EAM

"Enterprise architecture" (EA) or "Enterprice architecture management" (EAM) is an approach to information systems management in relation to the business [14]. It therefore strives to document, plan and design repositories of model elements which are expressed in different types of views on an architecture.

The processes in the EAM practice are based on collaboration and documentation which can be aided by help of tool support. While such documentation can be practiced with known information capture tools such as Wikis or Microsoft Excel and visualized using diagramming tools such as Visio, EAM tools more specifically deal with known practices in the area of EAM. As seen in [11] a lot of tools are available which offer special capabilities tailored to the EAM practice.

Capabilities of this type of software include but are not limited to the following aspects. The mentioned aspects are the relevant ones for this research, as they are expected in order to support a few of the assumptions made.

Capturing of domain model. A domain model [6] is the expression of the business capabilities of a company. Especially in the practice of "domain-driven-design" which is applied in the context of microservices[7] this type of model becomes even more relevant. EAM tools therefore allow to design a domain model with different types of hierarchies.

Separation of architectures. As the process of EAM is about planning and capturing of the existing, tools are able to separate between the design of a current or "present" architecture and a "planned" architecture. This assumption is important for this research as it enables the "present" architecture to be aided by automation while planning can be done separately.

Approval model. In the course of the conception of architecture elements inside the EAM tool there will be different working states of a model element. This is the result of the process of acceptance of an architecture change in which different parties can review and approve this process.

Workflows. Workflows allow to further control the aforementioned process. If a new architecture element is created, the process of reviewing the change might include the requirement to add missing data to enable the correct integration into different architecture views. For example a new application might require the definition whether it is a self developed or standard solution.

Versioning. In the course of the development of the company architecture the associated data of an architectural element is changing. Therefore in order to also be able to visualize

and manage these type of changes such model elements can be versioned.

Metamodel customization. The way companies are structured is changing, for example with the change from a business unit based company to a product based organization. Ideally an EAM tool allows the metamodel to easily adapt to these type of structural changes or differences, otherwise different workarounds will be required to bend the old to the new.

PlanningIT

The ”Alfabet IT Portfolio Management Suite” [8] also known as ”planningIT”, developed by the company Software AG, is one of the popular choices among different tools specifically built for practicing enterprise architecture management[11]. Its core capabilities include managing different types of architectures of a company such as the technical and business architecture. PlanningIT offers all of the aforementioned capabilities except for bigger changes to the metamodel. Alongside these capabilities planningIT offers the following which are relevant to this research.

Communication dependencies. In planningIT it is possible to capture the communication dependencies between different *applications*. The data which can be captured is as fine granular as to the extent of the name of the interface and the applied transport protocol.

API. planningIT offers an API which allows to access the captured data and also modify it. This API is HTTP-based and uses JSON for the transport encoding. Essentially it is a managed interface which allows to execute SQL on behalf of planningIT on the backing database.

As defined in 1.1.1 and also seen with the APM tool dynatrace1.2.1, planningIT also has its own type of definitions for some of the terms applied in this research. The ones with a resemblance to this research are defined in the following.

Domain: A domain is part of the highest level of enterprise architecture in planningIT. It is linked to defined business processes and applications.

Application: An application in planningIT is the only representation of any kind of software, whether it is deployed as a tool on a client computer or a service in running on a server. It therefore separates itself from the idea of an application in dynatrace. An application offers a version descriptor which in turn requires every version change to be captured as a ”new” application object.

ICT object: While *applications* are more of a raw resource in planningIT, the ICT (Information Communication Technology) object is used to manage complexities of differing versions of an application and therefore allows to link other relevant information, such as associated processes, on a higher level.

Component: A component is more fine granular than an *application*. It is linked to an *ICT object* and used to capture aspects such as frameworks or technologies which are applied for the realization of a concrete *application*.

[8] https://www.softwareag.com/de/products/aris_alfabet/it_portfolio/default.html

Device: A device in planningIT represents by default either a client or server device. It can be used either in the context of an actual physical device or also to capture the idea of a server including a virtual machine.

Part II.

Conceptualization

2. Conceptualization

2.1. Requirements

Before the concept for ”APEAM” and its prototype were built requirements were gathered to define the boundaries of the implementation. The following sections define these requirements.

2.1.1. General requirements

The general requirements were gathered by the analysis of the problem at hand and do not necessarily relate to requirements of the stakeholders which are defined in the next section. Rather these requirements relate to the research questions and the overall quality expectations.

References only. When building an integration layer between two systems it seems plausible to collect data initially from the source system and later copy it to the the target system. As inevitable result the integrating system becomes the owner of this data in transit. In this research however the goal is to keep the data in the respective systems and build only links between both of them. Therefore when later building views for both linked systems in the integration layer, data is always ”live” and never copied.

Referential integrity. In this research existing information from the APM system is utilized to insert data into the EAM system. As therefore data to some extent has to be copied different kinds of responsibilities arise for the copying system, in this case APEAM. This kind of referential integrity has to be maintained in relation to all aspects of CRUD (Create Read Update Delete) from the source system.

Abstraction layer. This research was built in the context of the case study mentioned in 4.1 and two concrete tools in the space of APM and EAM. It however tries to generalize across the capabilities expected from both of these tooling worlds, so that many of the concepts introduced in this research could be applied with other types of tools, especially in the realm of APM and EAM.

2.1.2. Stakeholder requirements

The implementation of this research was further driven by the requirements of different types of stakeholders. The following paragraphs describes different stakeholders in terms of their role inside an organization and how this research relates to them. Most requirements were gathered initially while some were added as a follow up from different stakeholder interviews. Wherever this is the case it is explicitly noted.

Enterprise architect. Due to the fact that one of the main goals of this research is the enhancement of enterprise architecture models with live data, it should be apparent that enterprise architects can therefore be considered the main stake holder and driver for requirements. As evaluated by [3] and also from an interview in A.5 it becomes clear that while enterprise architects might be interested in "live" data, it is more important to them that consistency is maintained and information communicated by different architectural views does not become polluted by changing runtime information. It thereby becomes a challenge to balance between correctness and liveliness. This results in the requirement that changes to the EAM system should be transparent and also verifiable.

Software developer. The software developer, while not directly benefiting from enhancing the enterprise architecture models, is however the main source for changes to the architecture and therefore the driver for the data which can be gather from the APM system. Especially as explained in 1.1, when considering that a lot of decisions are driven by the teams themselves rather than the enterprise architects. From interviews in A.3 and A.1 it became clear that the APM tool is used for some of the purposes enterprise architecture management tools were originally meant for. Developers therefore do not concern themselves directly with such tool and keep the efforts spent there down to a minimum. For this reason this research strives to keep the footprint or the tasks of developers down to a minimum.

Operations. Operations is the practice of maintaining and running deployed software artifacts. Nowadays this term often combined with the practices of DevOps or SRE (Site reliability engineering). While the just mentioned two types of stakeholders are considered the primary types of stakeholders of APEAM being both source and target of the data synchronization process, during the interview in A.3 it became apparent that the information contained in APEAM is also relevant to some of the doings of operations. Operations drives the way software is deployed to production and can therefore also define the rules how software has to be "prepared", such as it would be the case for enforcing name conventions. This is relevant to this research as it can to some extent control how data in the APM can be gathered.

2.2. Prerequisites

For the implementation of the proposed software solution "APEAM" as well as applying the corresponding workflows certain organizational requirements exist. While these might be considered limitations of the implementation itself, these prerequisites are rather an explicit expression of the environment the proposed solution applies to.

Microservice architecture. As described in the motivation, the main goal of this work is related to the discovery of microservice architectures, as these pose a problem in large organizations who try to apply enterprise architecture management practices. While many of the applied techniques used here theoretically could also be used with typical application based architectures, some of the conventions such as the proposed automation techniques might not apply.

Organizational model. More than the idea of building microservices themselves, who are just a technical expression of slicing a bigger monolith into smaller parts this research also applies to a organizational model which operates and builds on the basis of microservices and domain driven design. The conventions used therefore apply to product based organization types, which is explained further in 2.3. As microservices are built in product teams that work in a domain context, this allows some of the inferencing techniques applied here to infer the correct domain in terms of the business architecture. While this is no strict requirement, some more manual work will be required when initially assigning services to their respective domain.

EAM and APM software. The proposed software solution is very much dependent on the existence of an enterprise architecture management software such as planningIT or iteraplan. The reason being APEAM is mainly functioning as an integration layer between the worlds of APM and EAM. APEAM is neither persisting nor orchestrating information it does not own itself. It only allows to traverse data in the EAM and APM domain in order to allow the representation of an "enterprise graph".

Application programming interface. In this work both the APM software (dynatrace) and EAM software (planningIT) provided an external facing API interface, which to some extent and given later described restrictions, allowed to retrieve and also manipulate (in terms of the EAM software) respective data in these systems. While this might be considered self explanatory it usually is not, especially in the realm of EAM tooling. Tools in the realm of EAM consider themselves the absolute owner of information with import and export mechanisms being rather file based than driven by an API [11].

Organization of model instance objects. To some extent the way data is organized in both the APM and EAM system has to adhere to certain requirements of the proposed metamodel in this research. In general, the metamodel of the proposed solution was abstracted from the concrete meta models in the respective software solutions whom were described in 1.2.1 and 1.2.2. However, the availability of certain types of objects is required in order to fulfill the respective slot in the proposed metamodel, this is described in more detail in 2.4. Different types of workarounds in theory could however be used to hide this fact. For example when "Product Teams" can not be expressed in the realm of the EAM software they could instead be realized on the basis of authorization model and groups.

Referential consistency. The artifact recognition in this research is left to the APM software, to prevent the breakage of model integrity across different deployments. This type of consistency can only be maintained by the APM software as it is the only part which can track the artifacts when operating at the edge where software is actually deployed.

Synchronization. One of the goals of APEAM is to synchronize or enhance the data that is persisted inside the EAM software. For this process to work properly and consistently the APM software has to adhere to two qualities. The first quality is the availability of inferring changes in the architecture, the other one was mentioned before and is related to the consistency of this data. For the first quality a software such as dynatrace offers an event feed for whom changes in the architecture can be detected easily by interpreting

certain types of events such as the addition of a new deployed artifact or the establishment of a new connection between two services. For the synchronization to not take irreversible actions in the EAM software, the EAM software should allow the versioning of data as well as applying some kind of acceptance state model.

Workflow. The in the previous paragraph described qualities of the EAM software should materialize itself in a proper workflow when working with possibly inconsistent or further "unacceptable" changes to the architecture models. Therefore a "workflow approval model" should be offered by the EAM software itself. APEAM does not offer this kind of quality and delegates it to the EAM system.

Completeness. One of the main requirements to both the EAM and APM software is completeness. Completeness hereby describes the availability of information that maps from both systems to each other. For example, when mapping a service from the instance of APM software to the instance of the EAM software, the EAM software should be aware of the domain the service should be correctly mapped to. When however, the information of the domain model is spread across different instances of EAM tools or other sources, this would require APEAM to identify the correct target. The same applies to the APM software, where data about the environment could be spread across different types of APM or other monitoring tools.

Liveliness. Liveliness has to some extent been described in terms of how data is made available by the respective API interfaces already. The enterprise graph in APEAM is designed to allow queries in both the worlds of EAM and APM based on the assumption that the data is always up-to-date. This also applies whenever the synchronization will be run. For an APM software this is an interesting problem as data typically is presented in a time range based manner, therefore some data might only be visible for certain time ranges. To avoid the problem of missing or loose data, APEAM therefore requires either the APM software to offer a time range independent runtime architecture model or otherwise a large enough time range so that consistency can be maintained. As an example the pure existence of a service in the architecture should not solely be dependent on the fact that a call to one of its interfaces has been made recently.

2.3. Product organization model

In the course of the development of APEAM it became apparent that the terminology used in the APM and EAM system differs to some extent. This chapter gives a proper definition of an organizational model in which these terms apply.

A product based organization tries build around the idea of a product which is the result of domain driven development. In 2.1 such a product organization model can be seen. Products are materializing themselves in the form of teams who build or maintain software in the boundaries of a certain domain. Therefore the the term product and team can be used interchangeable. As an example consider the team "Basket" which covers part of the domain "Checkout".

The responsibility of such a product team is then one or more microservices up to the

Figure 2.1.: Product Organization Model

decision of that team. Therefore the team ”Basket” might build a ”calculation” service, for calculating a basket, as well as an ”aggregation” service, for collecting products in a basket. As can be seen by the concept of ”Basket” the microservices belong to their assigned domain boundary. The term basket can therefore also be seen as an abbreviation term for the domain ”Checkout” in which it is used exclusively.

From observations made in the course of this research these teams often also maintain ”OOD” (Out of Domain) services. These services concern boundaries of a different domain and are built for several reasons. As an example consider a payment processing service which in this case is not be part of the domain ”Checkout”, but ”Debitor Management” instead. The service however is related to the business process of checkout as a whole. Due to the fact that the aim of product teams is to generalize functionality offered as much as possible this payment service might lack capabilities which are tailored to a special consumer interface such as a webshop. In this case the previously mentioned ”Basket” team might build a payment adapter service to either optimize the API interfaces or the whole payment orchestration process itself in the business process occurring as part of an online purchase. Therefore, a ”payment gateway” or orchestration service might be built which sits in between the responsibilities of both domains, but in this case would be part of and maintained in the ”Checkout” domain.

Parts of the software of an enterprise is not built by the companies themselves but bought as standard software, especially when this software is used for processes that do not present any kind of advantages over competitors. In a strictly product based organization these types of software are also handled by product teams.

All services are characterized by the notion of an API interface. This enables products to

exchange the software behind these interfaces whether it is standard software or self developed microservices. Standard software or legacy software is therefore also put behind an API interface.

2.4. Data integration automation

Data integration describes the flow in which APEAM uses data originating from the APM system and persists it automatically in a target system, in this case the EAM system. This section defines how this mapping can be done in practice and what different options were evaluated.

One of the main complexities is the proper linkage in each respective system and maintaining these references. As the APM system has its own kind of view on a system architecture no further changes are required, as everything can be inferred from the communication architecture of the services and the infrastructure they are deployed on. As this data is result of different types of instrumentation and therefore automated it is read only and can not be changed. Therefore the APM is seen as read-only and therefore not candidate for maintaining the references towards the EAM system.

The EAM system is the owner of the business architecture which is realized by different services to be found in the APM system. The service essentially is the intersection between both worlds and the way a link can be maintained. Therefore the service becomes the desired object for updating and inserting into the EAM system. Looking at the data which is linked to a service in the realm of the EAM system, it should be linked with the respective business domains to maintain it properly. Afterwards a service can then be maintained by other links such as a owning team or business processes it is used in, to name a few. These links shall be added to enable architects to understand the context a service operates in, without further consultation of the origin a service stems from. As also derived from the requirements of enterprise architects, APEAM should able to infer respective domains and/or teams from information already available without further annotation.

The problem now becomes where information such as team or domain can be inferred from, as in software development it is not made explicit by known means. From the perspective of the APM system where service data is originating from, there is a limited amount of data available which could be used for inferring this information. In the following paragraphs different approaches to solving this issue will be explained and rated in terms of applicability. 2.2 describes possible data points to be used for this purpose and should be available from APM systems, not requiring any specific implementations or maintenance in the respective developed services.

Another source of data next to the data from the APM system could be any type of information which is produced in the course of the software development lifecycle and could possibly linked to data in the APM system. While these different tools can be great sources of information, when starting from the APM system as source of data, deriving this link can pose a problem. Examples for this data are shown in 2.3.

Data element	Example
Link/communication architecture with other services	Service Basket talks to Service Order
Link/communication architecture with known backings	Service Basket talks to MongoDB X1
Deployment artifact name	bt-basket-calculation-service
Interface name (REST endpoints)	POST on /basket/createBasket
Infrastructure	Kubernetes Cluster 01
Hostname/Domain name	bt-basket.development .environment.com
Technology	Java, Spring Boot, ...

Figure 2.2.: Service data retrieved from an APM system

Tool	Phase	Possible source
Requirements tracking tool	Requirements gathering/Development planning	JIRA
Code maintenance	Development	Git
CI/CD Tool	Build / Deployment	Jenkins
Artifact runtime	Run	Docker

Figure 2.3.: Service data retrieved from other tools

In the following paragraphs different strategies will be explained as a synthesis of different data sources and possible applications.

Artifact description. As a manual method, which requires effort by developers of every service, a file can be established which describes certain metadata of the service itself. Such metadata could therefore also include information of respective domain association and the maintaining team. An exemplary format or way of realizing this is the software Pivio[1]. The problem with this approach is, to maintain an acceptable source of data, the existence of such file has to be enforced by some part of the software development pipeline such as a quality gate, otherwise it can possibly be neglected by developers. When not used by the developers due to an intrinsic need, there could possibly still be discrepancies due to a lack of updates, even with the existence of such file. Since the involved tools for this approach would be the Code maintenance and the CI/CD tool, there is a lack of involvement of the APM tool in this regard. So this data will not be available from the APM system as it is not scraped by a monitoring agent. To make use of this data APEAM would therefore need some publishing of the collected information. Solutions as the mentioned Pivio publish this data via a REST interface and could be seen as an alternative solution for discovering services next to the APM system.

[1] http://pivio.io/

Name convention. When deploying services to production, teams need some kind of way to reidentify their deployed services. As by default the data known to an APM system as described in 2.2 includes at best names of a deployment artifact this becomes a challenging task, when names such as "application.jar" were used in the build process. APM systems such as AppDynamics or Dynatrace therefore allow to rename identified services in the user interface manually by users for proper reidentification. Also, for manually configured monitoring agents, as it is the case for AppDynamics, it is possible to specify the name beforehand in a configuration file. For some types of technology such as the Java JAR file or the application name in the spring configuration, the name can also be derived from the runtime. This approach however does not work for applications written in NodeJS as the artifact typically is called "index.js" and is spread across multiple files. In this case the "package.json" file could be used, however it is very often removed during the course of packaging or deployment as it serves no further use later on. Leaving other technologies such as .NET, Go or Ruby out of the picture, it should be apparent that the name retrieval of an artifact is inconsistent across technologies and therefore also hard to enforce.

There however is a type of runtime environment which has gained a lot of momentum in the world of microservice development, which is Docker[2]. Docker is the only type of runtime environment which works for all technologies and programming languages at it is similar to a virtual machines by providing an operating system platform and is supported for both Windows and Linux. It also enforces the idea of a single application per image/container, which is not applied for virtual machines, who have to host an operating system and its services as well. Further Docker maintains a versioning and artifact naming system on the basis of a Docker image. A typical Docker image name has the following structure:

<Registry>/<Artifact Name>:<Version>
test.registry.com/bt-basket-service:1

Images are a zipped up representation of the operating system configuration, all respective libraries and the application of interest as entry point. All running instances of such an image are called container and can always be referred back to the original image name.

All of the reviewed APM systems (Dynatrace / AppDynamics / NewRelic) support Docker integration, according to their web presence, and therefore allow to link running instances of a service to a respective container and its image. When now being able to infer the artifact name consistently for any type of application, there still remains one problem, which is inferring the respective domain or team from this name.

From observing the names used by a lot of the teams of the company this research was applied to in the case study 4.1 an implicit convention was inferred. Even before any established or enforced conventions were present, a lot of similarities could be discovered that were chosen by these teams to more easily maintain and identify their services. While it might be possible they were inspired by already running services with this naming structure, it also seems to be intuitive by the way it is structured. As a result the following naming convention was established for the artifact name:

[2]https://www.docker.com/

<Team Name>-<Domain/Abbreviation>-<Function>?-<Type>?
bt-basket-calculation-service

All analyzed teams and services maintained some kind of team name in their artifact name for reasons of re identification of deployed services. When leaving out reoccurring and non relevant filterable terms, such as the company name, all teams maintained the team name as the first word in the artifact name. As explained in 2.3 that team and product go hand in hand the respective product can also be inferred from this convention. In a well maintained business domain model where products/teams are associated with their respective domains this would already allow to infer the domain a service belongs to and associate, neglecting the rest of the convention. However as one team rarely maintains only one service there is an interest to further split the naming. The first split here is the domain abbreviation, which typically describes a word or function of a service in the respective domain. For the domain "Checkout" such a word would for example be "basket" and therefore be used as an abbreviation for this domain. In the context of a domain such abbreviation allows to explain concepts or parts of the realization of the domain function in the overall business context. As the name of a domain rarely is in a technically maintainable and compact manner such as Checkout, but instead complex and long such as "Customer Attraction" including empty spaces, these abbreviations enable better suitable representation in terms of the artifact name. The other two components "Function" and "Type" allow for further extension and separation but are not necessarily required for the described goal of domain identification. The function could be used to further understand the participation of a service in a certain business process. The service type allows to separate artifacts with no special meaning to the domain context. For example deployments of aiding tools such as a configuration database can be named "utility" instead of "service" and then be filtered out. Due to the aforementioned existence of this convention, this is the chosen approach for APEAM as it is already intrinsically motivated for software developers due to the fact that they are interested in finding the relevant monitoring data for their services and it is already documented in the APM system.

Communication architecture. Looking at the communication architecture of a service, as inferred by an APM system, some claims can be made. *Claim 1:* Services in the same domain context often talk to each other more than they talk to services outside of their context, when leaving out often re-utilized master data services, such as product data or prices. *Claim 2:* Communication architecture of services of the same domain is very similar in nature. This means for example that a basket service and a order creation service both might talk to the customer service as well as the payment service.

If these claims could be proven true, it would mean that from observing the communication architecture of services as done by an APM, some heuristic could try to infer the domain from existing services. Unfortunately, due to a scarcity of data these claims could not be verified and require further research. Therefore this strategy was not integrated into APEAM and left as an option to explore further.

Other APM data. The remaining data of the APM which was not mentioned in the paragraphs before was identified not to be helping in the process of domain or team identification. The hostname very often is arbitrary or legacy due to the replacement of an

older system with a newer service as an adapter in place. Endpoints, used backings, deployment infrastructure and applied technologies are too generic and often the same for most services.

Other software development lifecycle data. While all the other mentioned systems in the lifecycle or even systems outside the boundaries of software development could be used to identify the domain relationship of a service, they were not considered further as the linking to the primary data source of this work, the APM, was not further explored.

2.5. Data integration workflow

After retrieving services from the APM system and identifying their association as explained in 2.4 the service still remains to be integrated into the EAM system properly. As explained in the requirements2.1 their are different goals and restrictions to be considered. This includes maintaining the CRUD lifecycle for the inserted service in the EAM system as well as giving enterprise architects the possibility to control when or how this data is actually integrated into effective architecture designs.

2.5.1. Creation workflow

In 2.4 the process of how a service is created is shown. After the retrieval and discovery of a service from a source system, APEAM first has to check whether a reference towards the EAM system has been already maintained, therefore it needs host its own representation of a service in which this reference can be maintained. It builds on the foundation mentioned in 2.2 that the source system, in this case the APM system is able to maintain consistency for a deployed artifact except for the version. Looking at the way Docker image names are built this restriction can also be fulfilled just by the image name convention. When registering a service for the first time a timestamp is maintained which will be used by the deletion flow in 2.5.4. If the service has not yet been inserted into the EAM system and the inference association workflow was successful resulting in the extraction of the product and/or domain name, it will be inserted as a draft service. The name for this service then is only the extracted naming convention, not the actual technical name of the docker image. The concept of ”draft” is a special state in respect to the requirement of maintainability by enterprise architects and has to be maintained by the EAM system. From this point on the EAM tool is in charge of all further proceedings, which includes applying changes to possibly wrong domain associations and adding further information to a service which can not be inferred from the runtime alone. APEAM will keep a technical reference which allows it to be independent of any kind of change done inside the EAM system, making also renaming possible.

2.5.2. Read workflow

The workflow of reading from either the EAM or APM system is not relevant to the intents of data integration. It is however used in the context of the enterprise graph explained in 2.6.

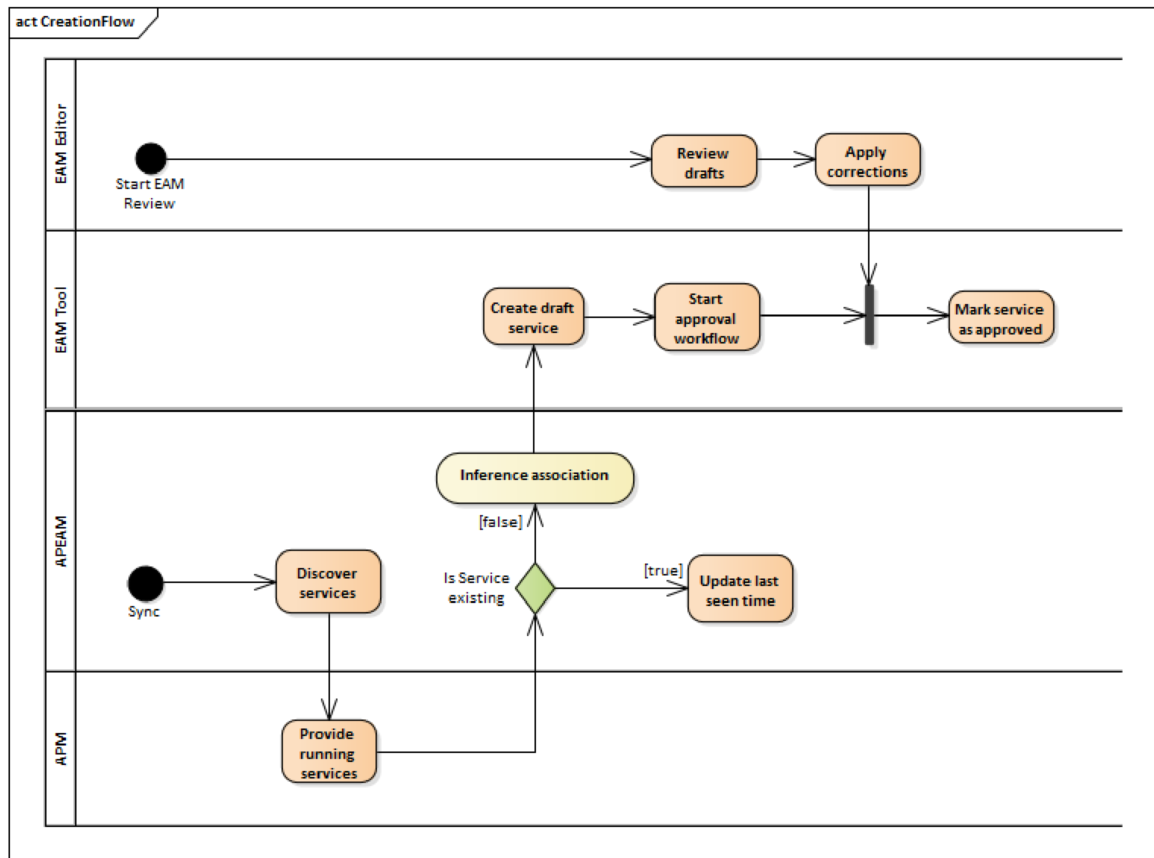


Figure 2.4.: Service creation flow

2.5.3. Update workflow

When inserting a service to the EAM system minimal data is inserted and therefore has to be kept updated. At best the inserted data includes the name of the service, the product team and possibly a version. In case of changes to the main name of a service, e.g. the Docker image, which also includes the inferred product team, it will inevitably be considered as a new service and therefore not require any types of update. Instead the creation flow for a new service will be triggered and at some point the deletion flow will be occurring for the old service which was renamed. The version of a service which can possibly inferred also from the APM system when used with Docker images could be maintained in the EAM system as well, which would result in retiring the old version and creating a new one. As the implications of this are individual to the capabilities of the respective EAM system it was chosen to be left out of the picture as it requires further research in terms of feasibility.

2.5.4. Deletion workflow

After the initial insertion of a service as explained in 2.5.1, APEAM is in charge of maintaining the existence of the service. It does so by regularly checking the last seen timestamp in relation to the APM system and comparing it to an availability threshold which can be defined individually to the needs of different environments. When encountering a service whose last seen timestamp is beyond this threshold APEAM will mark the service as to be retired in the EAM system. This in turn allows other workflows for retirement of such service to be run in the EAM system boundaries. If the created and referenced service from APEAM has been removed in the EAM system it will be reinserted and APEAM will replace the existing reference. In order to prevent further insertion of such service it is to be marked with a state of "Disapproved", so that reinsertion will not occur and the service will not become part of the overall architecture.

2.5.5. Synchronization

The process of synchronization essentially is the rerunning of the flow shown in 2.4 for every service discovered. The process of synchronization shall be independent of a point in time it is being run as defined by the prerequisites2.2. Essentially it becomes a balancing act between the removal threshold of the deletion workflow2.5.4 and the time range applied for discovering services from the APM system.

2.6. Enterprise graph

In [3] the "Enterprise topology graph" was introduced. The idea is, as explained in 1.1.3 to represent the technical architecture of the whole company. This is similar to what the dynatrace Smartscape does as explained in 1.2.1. In contrast to this, this thesis aims to provide a wholesome picture on all enterprise resources, starting with the APM and EAM source systems. Therefore this research defines the concept of an enterprise graph.

The basis is realized by connecting the respective data owning systems as presented in the previous chapters. By owning the references into different systems all that is left to realizing the enterprise graph is finding a way to transport and expose the respective entities which are owned and exposed by the connected systems. As the presented system does not own any other resource than the reference, all data exposed shall always be "live" and retrieved via the respective interfaces on traversal of the graph.

To the end of building such a graph it becomes necessary to define a generalized meta-model on whose basis entities independent of the software behind can be integrated. Alongside of mapping properties from the entities it also becomes necessary to enable the traversal of entities which are not related to each other from the owning application perspective but from the perspective of certain use cases or interests.

For example consider a domain entity such as "Checkout" which is maintained and owned by the EAM system. At runtime after the synchronization via the mechanisms explained in 2.5 APEAM will be aware which services represent this domain inside the APM system. Therefore a traversal of the graph would allow to explore starting from the domain "Checkout" traversing to the services linked to it in the EAM system and further on the representation of these in the APM system. When now reaching the context of

the APM system a vast amount of data such as the hosts these services are running on can be explored. When this research was conducted different options on data relevant to be synced to the EAM system were considered. This included also information such as exposed interfaces and thereby communication between services. However it was decided that data at best remains with the system which it was originally created in and thus the enterprise graph was born.

With the choice of the GraphQL technology in **??** and the examples in 3.6 this research presents a concrete transport to realize the traversal of the graph.

Part III.

Implementation

3. Implementation

3.1. Architecture overview

The architecture of APEAM is built on 4 main components, the APM system, the EAM system, the respective monitored servers, their hosted services and the APEAM software itself, as shown in 3.1. In the following paragraphs each of these components will be described by its role in the whole system. This architecture is built around two concrete tools, dynatrace as the APM and planningIT as the EAM tool.

APM system. The APM system hereby describes the source of the "live" technical architecture data. It is represented by the application dynatrace. In the diagram there are two types of APIs, the first one is the official REST API, which still is in active development as the time of this writing. There also is an API that is solely meant to be used by the dynatrace user interface, which in the visualization is distinguished by the term "Dynatrace Frontend API". As there is a discrepancy between the features and available data in the official REST and the Frontend API, it was decided in order to build the proposed solution, to make use of the frontend API as well. The APM server is the actually processing unit receiving data from one or more monitoring agents which are installed on respective monitored servers. The APM server itself has a lot more integrations than shown in this diagram, but the presented ones are sufficient to this cause.

Monitored servers. Monitored servers are infrastructure components on which the services are deployed in the form of artifacts (e.g. a Docker Image or a JAR file). Dynatrace "OneAgent" describes a software which is installed once on every server and injects itself into all running applications, whether it is in a Docker container or on the host itself. It could be considered "invasive" as it does not have to be explicitly configured, for example as an agent library for the Java runtime. In the diagram, services are represented by a runtime, this is meant to abstract the technical implementation details which dynatrace uses to understand different types of software. A runtime therefore is a known technology such as Java or a NodeJS application for whom dynatrace has specific integrations. Services do not necessarily need to be installed on the same server as might be concluded from the diagram. They can be instrumented across the boundaries of multiple servers.

EAM system. The EAM system is the target system of the whole architecture. It is the one, where data is not only retrieved but also inserted into. In this research it is represented by the software planningIT. PlanningIT has a server side component which is built in .NET and therefore running on top of an IIS server. It exposes a "REST-like" interface which allows to retrieve and manipulate objects. The REST interface component is licensed separately and therefore has to be bought as it is not part of the regular application package. Furthermore, there is a client library for Java server which allows to make calls to the API. Unfortunately, the accompanying documentation was not sufficient as it advised a

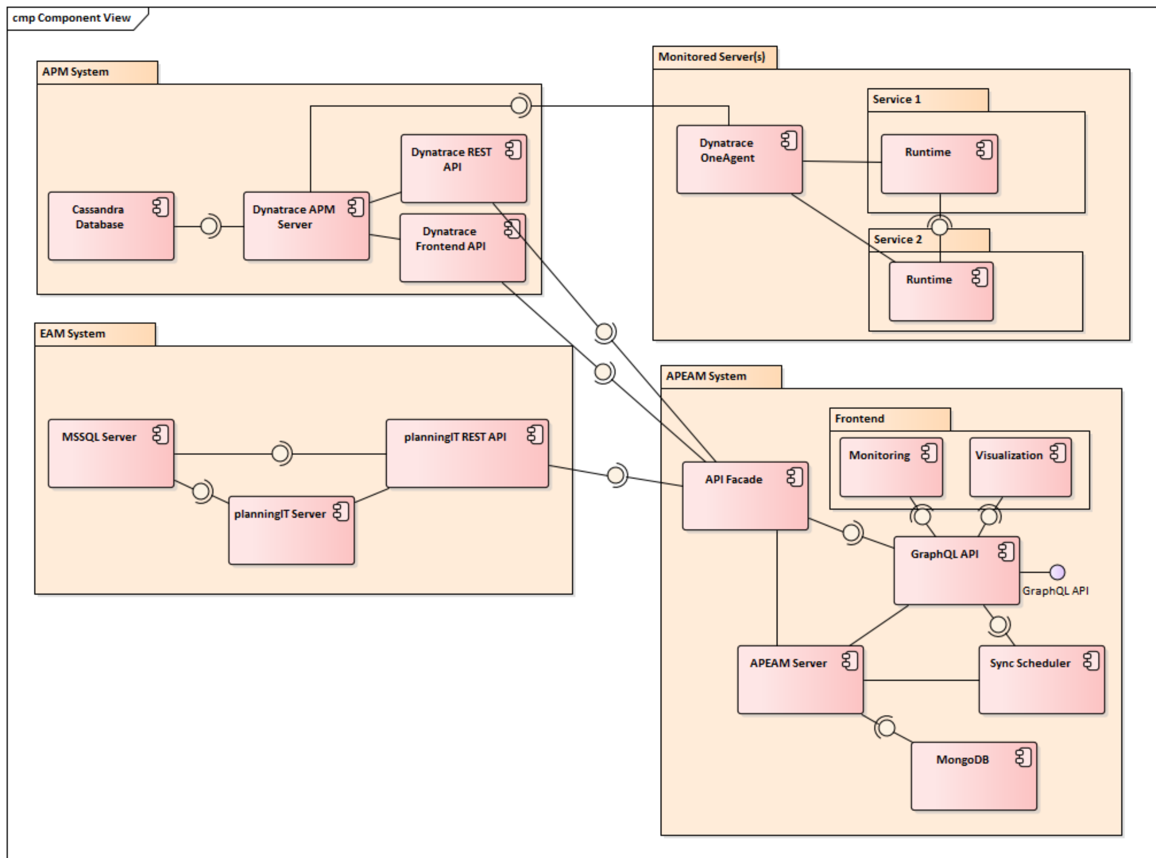


Figure 3.1.: APEAM Architecture Overview

newer version of the API which was not available during this implementation. Therefore, the database behind planningIT which is based on MSSQL is of interest, as the "V1" (version 1) API is not a lot different to making SQL queries on the database itself, but wrapping it into the object model of planningIT and therefore guaranteeing some types of validation.

APEAM system. While all previously defined components have been in existence in the company of the case study, APEAM and all of its inner components are the result of this research. The core component is the APEAM server, who is the orchestrating unit and owner of all exposed interfaces.

While it tries to keep the data it owns and therefore persists to a minimum, it still maintains some link and update data in its own MongoDB based database. Further it exposes an API interface on the basis of GraphQL. This interface more specifically represents the enterprise graph explained further in 3.6. This graph API could further be used for a frontend of APEAM, which allowing to do data maintenance and visualizations of the enterprise graph. In this prototype the frontend to the graph was represented by existing tools mentioned in 3.2.

The API facade is an abstraction layer for talking to a generic APM or EAM system and

therefore maintains the generalization of the proposed approach explained in 3.4.

The sync scheduler takes care of how and when to update data. The naming might suggest it is time based, this is however only true to some extent as different kinds of triggers are explored in 3.7

3.2. Technologies

The prototype APEAM server was implemented on the NodeJS[1] runtime platform. Due to the complexities of translating properties from different source and target systems, as well as exposing them in a metamodel of its own, the choice was made to make use of proper type enforcement. For this reason the code was implemented in Typescript[2] which similar to the programming language Java can enforce strict typing rules at transpilation time to Javascript.

The backing database was chosen to be based on the idea of "NoSQL" and the most popular choice[3] in this area MongoDB[4]. The rationale was to keep the complexity of joining down to a minimum. Overall the entities required for the implementation could be persisted in a single collection. For using MongoDB in the prototype the library "Typegoose"[5] was used, an abstraction of the popular "Mongoose"[6] library, allowing to apply the capabilities of Typescript in the context of constructing the persistence model. While possibly counter intuitive with the "enterprise graph" concept, the database was not chosen to be graph based. The data which is actually exposed for the graph is not persisted at all by APEAM and therefore a graph based database was of little help. In case there might be extra metadata persisted by APEAM in the future the choice might be different.

The only API APEAM exposes is based on the GraphQL[7] technology and is a transport realization of the concept presented in 2.6. While there are multiple bindings for different programming languages to use GraphQL in, it is native to the NodeJS ecosystem, which further supported the initial platform choice. GraphQL itself is a query language only and does not define any type of transport. For exposing the GraphQL API the "Apollo Server"[8] was used which exposed the interface on the default endpoint "/graphql". Alongside the capabilities of querying a graph, GraphQL is self documenting. It allows to view the graph metamodel which can be enhanced with descriptions by the implementer. To make use of Typescript's capabilities while constructing the GraphQL model, it was built on top of a library called "type-graphql"[9]. The GraphQL API is used not only for the purpose of traversing and reading data, but also allows to control the syncing and persistence capabilities of APEAM. For the prototype no user interface was built. Instead two tools of the GraphQL ecosystem were used to allow understanding and querying the graph. The

[1] https://nodejs.org/en/
[2] https://www.typescriptlang.org/
[3] https://db-engines.com/de/ranking
[4] https://www.mongodb.com
[5] https://github.com/szokodiakos/typegoose
[6] https://mongoosejs.com/
[7] https://graphql.org/
[8] https://github.com/apollographql/apollo-server
[9] https://github.com/19majkel94/type-graphql

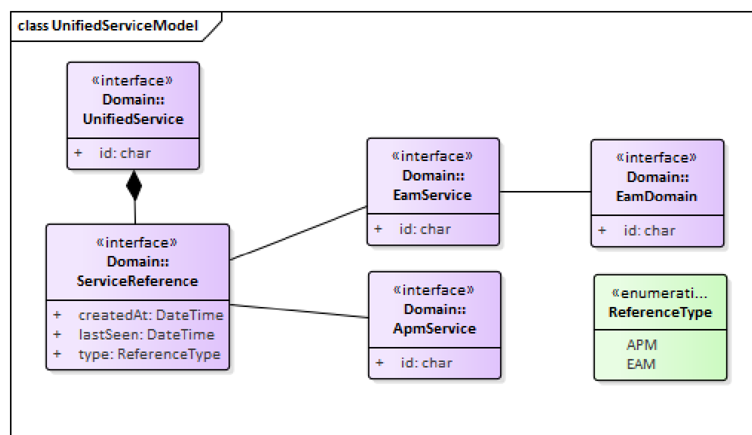


Figure 3.2.: Unified data model

first tool is "GraphQL Voyager"[10] which allows to explore the metamodel exposed by a GraphQL API in a graph like relationship diagram. This tools aids in understanding how the graph is constructed and how different entities and properties can be reached from a query perspective. The second tool is the "GraphQL Playground"[11] which allows to actually query the graph and also supports the user in the construction of a query with possible choices by auto completion.

3.3. Unified data model

For the purpose of abstraction, a unified data model was built that is generalizing the entities of both the APM and EAM system. These entities can later be joined by reference under the umbrella of a "UnifiedService".

The "UnifiedService" as shown in 3.2 interface defines the holistic view on a service in the landscape and is owned by the APEAM system itself. An example for such service would be the "bt-basket-calculation-service". Important to note here is that for instances of the unified service the naming convention is enforced inside of APEAM as the id property. Therefore the creation of a unified service cannot happen unless such a name is provided or inferred. The "UnifiedService" holds multiple references, most notably one to the APM service and one to the EAM service. A "ServiceReference" can only associate with one such service, however there can be multiple instances of "ServiceReference" in a "UnifiedService". This allows to connect an arbitrary amount of references to other systems where an information about this particular service might reside. The type of a "ServiceReference" is not expressed by inheritance, but by an enumerable called "ReferenceType". This was done so that persistence avoids the complexity of maintaining types, as the types are expressed by a property instead by the type system itself.

The "ServiceReference" maintains two types of timestamps. This helps processes such as

[10]https://github.com/APIs-guru/graphql-voyager
[11]https://github.com/prisma/graphql-playground

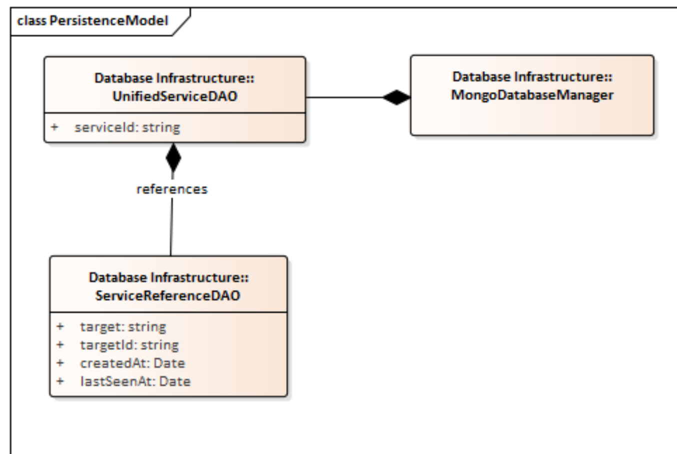


Figure 3.3.: Persistence data model

the synchronization to validate whether a given service might still exist or not, as explained in 2.5.4.

Any instance of a service in the "ServiceReference" has to maintain an "id" which allows to exactly address the instance in the respective source system independent of any changes made to it such as renaming. For example, in the APM service for this scenario it would be the Docker image id such as "eu.gcr.io/bt-basket-calculation-service". In this case the id neglects the version of the Docker image to allow tracking the service across different versions or deployments. As explained in 2.5.3, the version was neglected as a concept for this prototype. The interface "EamDomain" in this diagram is only presented to show the possibility of traversing any arbitrary data which is connected to the instances of a linked service.

APEAM tries to keep the data it manages down to a minimum to avoid any type sync 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. APEAM realizes these constraints by only saving the data of the "UnifiedService" in the form of the "UnifiedServiceDAO". The persisted entity maintains an unique index based on the "serviceId" attribute and thereby preventing reinsertion of the same source service. All instances of "ServiceReference" are maintained as a list inside this object. Due to the fact that the persistence is based on NoSQL, all data is maintained in the same entity and no joining or other types of relationships are required. When extending the reference model new targets with respective ids can simply be added to this array. 3.4 shows an example how such service would be persisted in the database.

3.4. Provider model

The heart of the prototype is building the connector model for both the EAM and APM system. In general, the abstraction model here defines providers which similar to 3.2, ab-

```
{"_id": {"$oid": "5bc05fd7466fd767883fa395"},
  "serviceId": "bt-basket-service",
  "references": [{
      "target": "APM",
      "targetId": "eu.gcr.io/dev/bt-basket-service",
      "createdAt": {"$date": "2018-10-12T08:48:23.492+0000"},
      "lastSeenAt": {"$date": "2018-10-12T08:48:23.492+0000"}
    },{
      "target": "EAM",
      "targetId": "APP-1543",
      "createdAt": {"$date": "2018-10-12T08:48:24.001+0000"},
      "lastSeenAt": {"$date": "2018-10-12T08:48:24.001+0000"}
    }]}
```

Figure 3.4.: Persisted unified service

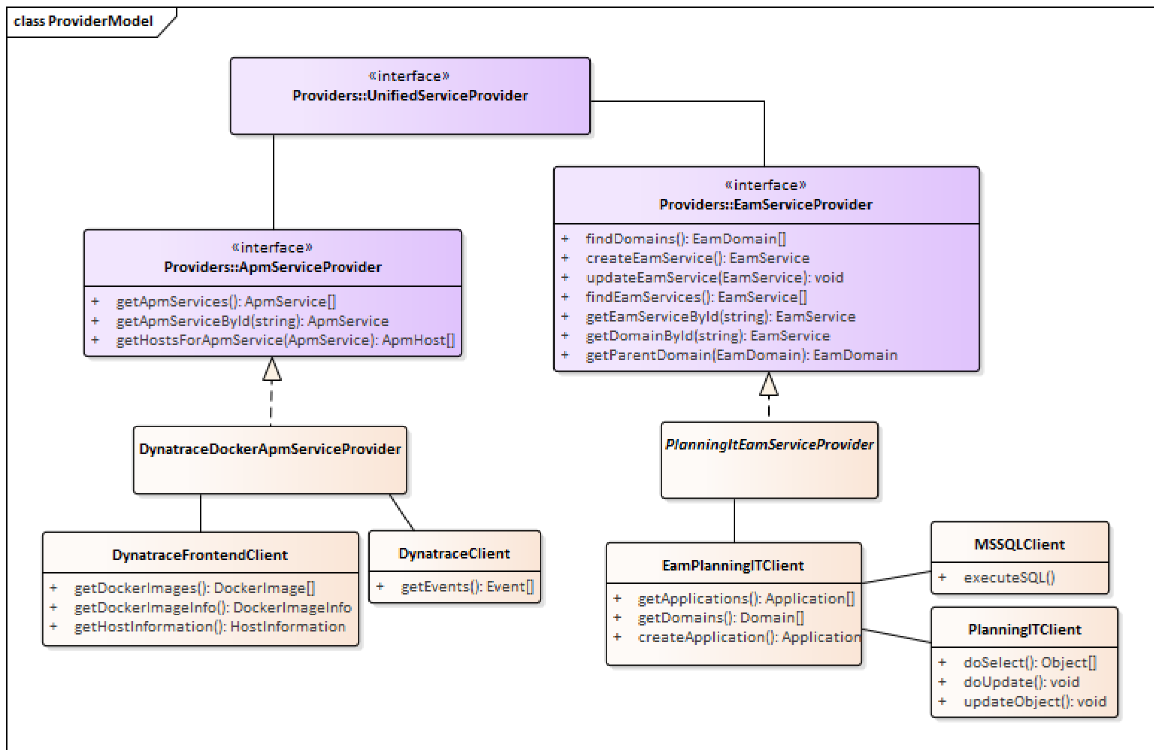


Figure 3.5.: Provider model overview

stract the concrete instances and clients of the respective systems. The overall architecture is shown in 3.5 and represents the goal of constructing an API facade as shown by the overall architecture in 3.1. The diagram is not complete in relation to all arguments or methods, but illustrates the necessary parts to understand how the provider model is realized.

The "UnifiedServiceProvider" is effectively an access facade to the database layer. This abstraction would allow to exchange the way unified services are persisted or retrieved. It is the core interface utilized by many processes and especially the GraphQL API in 3.6.

The "APMServiceProvider" defines the abstraction to retrieve instances of services which reside in the APM system. In this case the concrete implementation is the "Dynatrace-DockerApmServiceProvider", whose name is related to the fact that it makes specifically use of the Docker data in Dynatrace. In order retrieve data it utilizes two types of clients. Most of the relevant data however is currently retrieved using the "DynatraceFrontend-Client". The "DynatraceFrontendClient" utilizes APIs which are yet only available to the user interface of Dynatrace and therefore imitates the user interface in this regard. All APIs utilized from the frontend client are JSON-based and use the HTTP transport. The only special function in this client is the login process which based on a cookie and user login data, generates a session that allows to query the respective APIs. The "Dynatrace-Client" utilizes the official API which allows to retrieve general data on services and their relationships and can be used in combination with the data from the "DynatraceFrontend-Client" as the exposed entities are built on a shared model. The general Dynatrace API as of this writing is not yet able to get data on Docker images, so the provider combines data from both clients using the consistent "SERVICE-<ID>" reference. Dynatrace generates a unique service id for every service and manages to keep this id consistent across deployments.

The "EAMServiceProvider" abstracts functionality that shall be provided by an EAM system. For the relevant use cases this includes the possibility to enumerate all domains as well as the services which are already described in the EAM system. For some of the core functions of APEAM it must be possible to create and update a service and associate respective data using this process. These capabilties are represent by "createEamService" and "updateEamService". For APEAM and planningIT the capabilities are realized by the "PlanningItEamServiceProvider" which in the background utiltizes the "EamPlanningIT-Client", an adapter for the "PlanningItClient". The "PlanningItClient" exposes the basic methods of the planningIT API, which is based on calls that execute SQL statements on the MSSQL database and wraps them in the context of the planningIT Server and object model. As these types of calls are by nature rather not concrete in terms of the model managed by planningIT (e.g. POST /getobject "SELECT FROM APPLICATION WHERE X"), the "EamPlanningITClient" encapsulates these queries to more meaningful methods such as "getApplications" or "createApplication". When building the prototype the planningIT API unfortunately was not able to associate a domain with an application, for which a workaround using direct database calls via the "MSSQLClient" was realized.

These provider models build the foundation of all the processes that later are orchestrated via the API. An implementer looking to utilize other software than planningIT or Dynatrace must at least fulfill the provider interfaces.

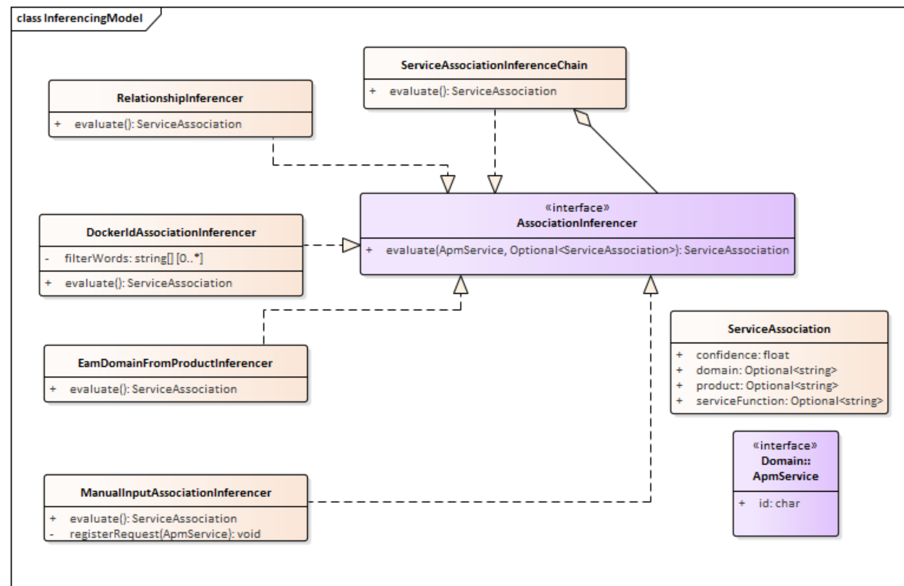


Figure 3.6.: Inferencing model

3.5. Inferencing model

The inferencing model was built to automate the association of a service as retrieved from the APM system with its respective product, domain and function, as described in 2.4. The inferencing is also run as part of the insertion of an EAM service to the EAM tool2.5.1. In the actual implementation no unified service is created unless the inferencing chain was run successfully, which also allows the ”serviceId” for the unified service to always be consistent. The main goal is to identify these respective attributes based on different potential information sources about that particular service. The implementation of the inferencing model reflects the ability to integrate multiple sources by being based on the concept of a chain. In this chain multiple sources can provide parts of the final result using the input from previous elements which were run in this chain. This helps with problems such as the scattering of information in an enterprise across multiple tools.

The general interface ”AssociationInferencer” represents the basis on which evaluation is done. When an unknown service is discovered from the APM system which is represented by the ”ApmService” interface, the inferencer uses this object instance to retrieve association data for the particular service in the ”evaluate” method.

In 3.6 the primary object of interest is the ”ServiceAssociation” as it represents the desired result of the inferencing task. The three relevant attributes for fulfilling the name convention is the name of the domain, represented by ”domain”, the name of the product represented by ”product” and the ”serviceFunction” which represents the use case of a particular service. An example for such as attributes would be, ”bt” as a short abbreviation for the product ”Basket”, ”Checkout” for the domain and ”Calculation” for its respective

function. While the ”ServiceAssociation” is the result, it also is the intermediate state object which is run through the chain. For this reason the attributes are represented as optional, which allows different types of inferencers to only add parts of the attributes. Further the confidence is used to allow different types of heuristics to add information which may have lower confidentiality that can be overruled later in the chain. The executor of the inferencer is responsible to handle or discard partial results.

ServiceAssociationInferenceChain. The main implementation is located in the ”ServiceAssociationInferenceChain” utilizing multiple ”AssociationInferencer” instances in an algorithm, which runs until a complete result with a 1.0 confidence has been retrieved or the ”evaluate” method on all included inferencers has been executed. The arrangement and types of inferencers provided to this chain allow for different priorities as composition patterns. For example when chaining the ”ManualAssociationInferencer” before the ”DockerIdAssociationInferencer” would prevent the second one to run in case the former is already successful.

DockerIdAssociationInferencer. The ”DockerIdAssociationInferencer” is the main data supplier which uses the naming conventions defined in 2.4 and the respective Docker image of the APM service by trying to extract the relevant terms from the Docker image names. Based on the APM service id the Docker image name is retrieved and cleaned. For the cleanup filter words can be provided, which are removed from the image name. As an example consider the image name ”eu.gcr.io/dev/corp/bt-basket-calculation-service”. In the first stage the repository is removed, which is always based on an address, in this case ”eu.gcr.io”. This is a regular Docker pattern and therefore applies on any self hosted Docker image. Further based on the filter words, ”dev” and ”corp” are removed. These type of words are often seen to provide different logical hierarchies to the storage of docker images, but are not relevant to the cause of the actual image. This leaves ”bt-basket-calculation-service” which is then handled based on the proposed naming convention. Using splitting based on the separators ”-” and the arrangement, the terms ”bt”, ”basket” and ”calculation” are extracted and used for building a ”ServiceAssociation”.

EamDomainFromProductInferencer. The ”EamDomainFromProductInferencer” is an inferencer which can enhance but not fulfill the ”ServiceAssociation” object. If a ”ServiceAssociation” has been fulfilled to the extent that the product name is known, however the respective business domain could not be determined, the ”EamDomainFromProductInferencer” is able to utilize the EAM tool or any other source which manages associations of product teams to respective domains. For example when for a service the product ”Basket” is known, a lookup in which domain the product operates, can add the domain ”Checkout” for this particular case. As it can not fulfill the whole ”ServiceAssociation” on its own the term ”Association” was deliberately left out of the class name.

ManualInputAssociationInferencer. The ”ManualInputAssociationInferencer” is typically used as last element in the chain and allows an external party such as a user interface to provide a definition for a respective ”ApmService”. For that reason the ”ManualInputAssociationInferencer” retrieves data from some type of storage, which in this implementation is the MongoDB. If no data for a particular ”ApmService” could be found and

the ”ServiceAssociation” attributes are not fulfilled, a call to the internal method ”registerRequest” is used, which creates a request object. This request object can then be fulfilled at a later point in time with the proper association. The result of this mechanism is that no service enters the management of APEAM unless either manually a association was provided or the chain has already been successfully in extracting it before.

RelationshipAssociationInferencer. Finally the ”RelationshipInferencer” is based on the ideas presented in 2.4 that relationships of interface calls, as they can be extracted from the APM system, could potentially help in identifying the association of a service. This idea remains to be explored further, but is another potential extension of the chain for whom the heuristic confidence value could be applied for.

For the prototype which was applied in the case study the chain was sufficient by using the ”DockerIdAssociationInferencer” and ”ManualInputAssociationInferencer”. While initially it was considerable to also make use of the ”EamDomainFromProductInferencer”, their however was no proper way to infer this type of association properly as no data source was available linking teams to domains yet.

3.6. Enterprise graph implementation

3.6.1. Interface implementation

In 2.6 the enterprise graph was conceptually described. The enterprise graph is the API model which APEAM was built on and further enable benefits from being at the intersection of two systems. On the basis of a GraphQL schema it is possible to describe different meta model elements, on whom later queries can be created.

The metamodel which was built for the enterprise graph API is independent of any of the systems which were used to build the prototype. It is backed by the provider model defined in 3.4. Based on the connections maintained by APEAM it is possible to use different aspects as starting points for a query and traverse the graph into the respective other system. For the prototype on the APM side it is possible to traverse into the infrastructure hosts services are being run on and on the EAM side parent domains of domains can be queried.

In order to allow data to be queried, resolvers have to defined and implemented on the root level. Resolvers such as ”domain” or ”apmService” allow to query for an object or collections of objects and work in a similar manner to Remote Proucedure Calls, as they can also be parameterized with multiple arguments whom the implementation can use to filter the amount of objects returned. All further traversal of the graph is done on behalf of the implemented object types and selections on their properties, which can also again be methods similar to the resolvers described before.

Alongside the concept of querying a graph, it is also possible to define ”Mutation” operations, which are, while being built on the very same query technique, are allowed to have side effects. In general this differentiation is purely semantic and has to be applied by the implementer correctly.

```
1  query {
2      domain(name: "Checkout") {
3          name
4          services {
5              name
6              domain {
7                  name
8              }
9              infrastructure {
10                 hosts {
11                     hostname
12                     cpu(timerange: {from: "2018-10-12", to:
                           "2018-10-13"}) {
13                         high
14                         mean
15                         low
16                     }
17                 }
18             }
19         }
20     }
21 }
```

Figure 3.7.: GraphQL Query for the domain model

3.6.2. Interface usage

In 3.7 an example for a simplified query can be seen. A query always starts with the term ”query” as defined by the GraphQL standard. All other parts of this query are individual to this particular implementation. On the root level of a query different resolvers can be defined, as the term ”domain” in this case. Similar to method calls, this resolver can be used with different types of arguments which are predefined when initially designing the schema. In this example the argument name was used to find a particular object with the name ”Checkout”. Different types of properties on the original metamodel element can then be selected if that object is found. In this case the property ”name” selects a field of type ”String”, while ”services” selects an Array of the type ”Service” and thereby again allows to select fields beneath this object. Using this kind of mechanism allows to traverse the enterprise graph with a theoretically arbitrary depth. In practice however query complexity, should be verified by the server. The result of the query is shown in 3.8 and follows the pattern how the initial query was structured, thereby allowing a consumer to expect predictable result structuring.

Any extra data to be included in the enterprise graph has to defined in terms metamodel representation. As an example when retrieving a service representation from the EAM system, other entities such as ”Cost center” can not be retrieved by default. To this end the enterprise graph is static and can only handle known metamodel entities. While in theory

```
{"data": {
        "domain": {
            "name": "Checkout",
            "services": [{
                    "name": "bt-basket-calculation-service",
                    "domain": {"name": "Checkout"},
                    "infrastructure": {
                        "hosts": [{
                            "hostname": "app01.docker.host.com",
                            "cpu": {
                                "high": 7.5,
                                "mean": 3.4,
                                "low": 2.1
                            }
                        }, ...]
                    }
}, ...]}}}
```

Figure 3.8.: GraphQL Result for the domain model

these representations could be generated by retrieving the metamodel from planningIT for example, this approach was not taken due to a complexity in filtering relevant metamodel attributes and leaving out technical ones such as the "alfa_instid". Also the enterprise graph shall serve as an abstraction from the concrete tool rather than just exposing every property available.

In 3.9 a mutation type query can be seen. In this case an "apmService" is initially selected similar to a normal query, however on the selection of the service property, the server will in the background try to create the unified service object by running the inferencing process3.5. If the creation was successful the operation "createEamService" is run on that

```
mutation {
    apmService(id: "eu.gcr.io/dev/corp/bt-checkout-calculation-
       service") {
        service {
            createEamService {
                id
            }
        }
    }
}
```

Figure 3.9.: GraphQL Mutation for creating an EAM service

object and inserts the service information to the EAM system. Similar to a regular query then properties can be selected to view the result of this creation process, in this case the ″id″ property of the service. The implementation then prevents any reinsertion on the first successful run. Using this mechanism any type of consumer can run the inferencing and service creation process. Alongside of creating an EAM service, it is also possible to link an existing service maintained in the EAM system by using the method ″linkEamService″ on the service object.

3.7. Synchronization

The synchronization process is simple and was designed around orchestration already existing interfaces which were created for the enterprise graph in 3.6 as parts of the mutation. The main goal of the synchronization process is to identify an APM service for whom no unified service is existing already, try to create it and also its corresponding EAM service in the process. It does so by initially querying the ″apmServices″ list on the GraphQL root model and then using the mutation API to create the unified service implicitly by a call to ″service″ and then running ″createEamService″, as seen in 3.9. In case the creation fails a error will occur and the sync process can retry on the next trigger. If a problem occurred in the inference association process, a user can help the process later on based on the ″ManualInferenceAssociationInferencer″ as described in 3.5.

The whole orchestration is run in the boundaries of the application server and therefore is independent of the HTTP transport. The synchronization process 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. The identification of architecture changes still is interesting to be run it at different points in time. For this reason and based on the interviews in A different types of triggers were conceptualized. When the trigger is aware of changes immediately when they occur, this could potentially give birth to the concept of ″real-time″ EAM as well. 3.10 gives an overview on different trigger options. Most notable is the variant of event based triggers by either the EAM or APM system.

Trigger type	**Source**	**Description**
Scheduled trigger	Interval timer	Default trigger running on a schedule
Manual trigger	GraphQL API	Allows external tools such as a CI/CD tool or a user to trigger the scheduler
APM Event trigger	APM system	Whenever changes in the infrastructure are detected, an event is raised which in turn is being subscribed to by this trigger
EAM Event trigger	EAM system	Whenever changes to the metamodel such as adding domains in the EAM system are made

Figure 3.10.: Triggers for the synchronization process

Part IV.

Evaluation, Limitations and Outlook

4. Evaluation

4.1. Case study: Multichannel retailer

This section introduces the boundaries and the application space in which this research has been conducted. The company is a large European retailer for electronics with an Omnichannel presence. Omnichannel thereby describes the way of offering customers the ability to buy products online or in brick and mortar stores as well as in a combination of both channels. The company structure separates the IT organization as a separate juridical organization which acts as an IT service provider for the main company and all other country branches. Lately this IT branch changed the organizational model towards a "product-based" organization (2.3), where IT services are no longer delivered for business units but rather as products in different domain contexts. Alongside the actual work in product teams for the delivery part, all employees are organized in a "chapter structure"**??** which defines core capabilities from the perspective of the IT organization. For example an employee might be working on the product "Basket" but is hierarchical assigned to the backend development chapter, which thereby defines the hierarchical structure in which a employee is managed. This model originating from the company Spotify has seen adoption also among other German companies, such as the Deutsche Telekom[1].

Before the restructuring actually was started domain boundaries were defined by the now former enterprise architecture unit of the company. The domain model itself is primarily driven by customers, as it is the main source of revenue for retail companies. The domain boundaries served as the basis for splitting the core applications into smaller units (products).

Products are built by teams, organizing themselves usually in the SCRUM model, built on a mixture of backend and frontend developers, agile support functions (SCRUM Master/Agile coach) and the product owner. Due to the delegation of responsibilities a lot of architectural decisions are done on a team level in terms of lower level technical architecture. This includes choice of technologies, microservice splits and others. Teams are therefore free to act in the boundaries of their product, while still some of the programming practices might be driven by the community in different chapters. This model was introduced to enable the ability to innovate individually on the basis of different technology approaches. When this research was done the new organizational model has however been only been in place for about 3 months.

The former enterprise architecture unit now is organized as a chapter similar to the aforementioned "backend chapter". In terms of overall architectural design decisions this chapter kept the main responsibility. Enterprise architecture is further on seen rather as a service that can be consulted by the different teams, especially when it comes to decisions in relation to the domain boundaries and functional placements.

[1]https://www.welove.ai/de/blog/post/spotify-modell-im-einsatz-bei-telekom.html

As the former described product team structure does not properly apply to all types former applications, responsibilities or teams, there are exceptions in the path of the ongoing transition. For example this means that central functions such as IT infrastructure management remain to be defined in terms of a product model. Also not all teams maintain self developed software, but standard software. The expectation is however to hide such facts behind API interfaces as shown in the product model described in 2.3.

The company uses ”dynatrace” as an APM solution for its core applications covering about 40% of the whole application landscape, but especially the part of self-developed software. The APM software is primarily managed by the DevOps division of the company, while access is open to other stakeholders and especially all product teams as well. These product teams make use of it for problem analysis and overall improvement, as shown by interviews 4.3. The APM solution is currently completely disconnected from any higher-level context and reports problems on an application or resource level.

planningIT is used as an EAM solution, which is primarily managed by the architectural division of the company. In the old business unit model of the company, ”Application owners” were responsible for keeping details of the documented applications up to date. This responsibility changed in the product-domain-model structure towards the product owner and the respective team. In general what can be seen from this description is that in the old as well as the new organizational model, many lower level changes to application architecture were delegated to people outside of the EAM division, which as seen in the interviews resulted in outdated, missing and wrong documentation.

4.2. Application of APEAM

The APEAM prototype was developed and applied in the environment of the company described in the case study. This section explores the core capabilities in relation to the established naming convention and the insertion process to the EAM system.

4.2.1. Naming convention

The naming convention introduced in 2.4 was the result of analyzing existing service data. 247 Docker images residing in the pre-production environment were considered for this analysis. These Docker images are a mixture of utility tools and actual microservices. The microservices again are a mixture of legacy applications and services created or migrated into the context of the product based organization model. By maintaining filters for certain types of utility images, 40 docker images can be ignored based on their repository or other reoccurring patterns, such as ”google_containers”. This pattern for example is used in utility software of Kubernetes clusters. By applying the presented convention, 153 running services can be identified with at least the team term leading in the name. Some are however duplicates of the same microservice. These duplicates are the result ”feature-branch” deployments and also can be filtered out automatically due to a name pattern in the image name. Overall 52 services can not be associated properly of whom all are legacy applications without any proper team association.

All services of teams operating under the domain umbrella were at least using the convention of a leading team or product name, as it was added by the CI (Continuous Inte-

gration)/CD (Continuous Deployment) pipeline and can not be controlled by the teams themselves. When thereby maintaining proper domain association of teams in the EAM tool all services can be associated automatically.

Docker is not the only way applications are deployed at the company, but however applied across all microservice deployments, therefore giving a wholesome picture on all applications developed as part of the product organization model.

4.2.2. EAM integration workflow

The automatic insertion of services to the planningIT tool and the associated workflows were verified in the course of the interview(A.5) with one of the enterprise architects at the company. Acceptance workflows for the addition of new applications to planningIT were already in place and therefore allowed known processes to be applied to inserted services. In practice however not all the details required in this process were known to the architect. Therefore it was further the goal to enable teams in maintaining their services as well. APEAM further associated authorization access to services for respective teams, when already maintained as user groups in planningIT. The ability to change any property about these services as well as updating the domain association was possible and still enabled APEAM to maintain the reference and thereby keep the enterprise graph in tact.

The proposed model for service retirement in 2.5.4 was evaluated in terms of finding a proper threshold for service retirement. In the time range this research was conducted in no services were removed or disappearing in the APM system. As this type of consistency was expected as defined in 2.2 and no services were actually removed an ideal value has yet to be defined. Interviews were also not able to determine a proper value, as there was no general consensus of how often or when services are retired.

4.3. Interviews

The proposed approach for the integration of an APM system with an EAM system was evaluated at the company it was developed in. For this purpose, structured interviews with multiple stakeholders were conducted after the implementation of the first prototype. In the course of the interviews five people were asked in personal interviews for their background in the relevant disciplines to this solution, their first impression of the solution itself, proposals for improvements and extensions as well possible applications of it. The interviews were done in person due to the complexity of the questions, for the possibility of a deeper understanding of the respective answers and because the amount of interviewees was manageable. A few unstructured interviews were already conducted at the beginning of the research and resulted in some of the requirements mentioned in 2.1.2.

4.3.1. Questions and descriptions

The questions addressed to the participants were framed, so that on the one hand a basis could be established to understand the background of each individual as it could be affecting some of the answers in relationship to APEAM. All answers of the participants

were initially noted down and later put into written form, which was provided to every participant and accepted. The questions were designed so they could potentially be applied to other companies than the ones used in this case study. This section presents an overview of the questions and the rationale behind them. All interviews in the appendix use shorthand convention to link answers to the questions such as **GQ**.

GQ1: *Please describe your current role in the company and the context you are working in.*
This question is to understand whether for the role at hand information of APEAM might be considered non relevant. Further it allows to establish different perspectives on APEAM and how potential extensions serve different needs.

GQ2: *Have you worked with/developed microservices in the past?*
This question allows to understand whether further question on microservices are relevant, which are marked with **MS**. The question is also used because people who worked with these type of services understand some of the problems at hand going along with these type of architectures.

GQ3: *Have you worked with APM software in the past and if so which APM software did you use?*
The data APEAM uses stems primarily from the APM system. For that reason it is relevant which types of APM systems are used in practice and to which extent they are being used. Similar to **MS**, **APM** is used to mark questions as a followup to this one.

GQ4: *Have you used enterprise architecture tooling in the past and if so which tool did you use?*
The other main data connector of APEAM is the EAM software. In the realm of EAM tooling lots of very different types of tools are used. Knowing how exactly which tools are utilized can help extending APEAM for certain usage patterns in the practice of EA. Further questions to this topic are marked with **EAM**.

MS1: *Do you practice domain driven design?*
While microservices as presented by [**?**] strive for Domain-Driven-Design, this is not necessarily practiced by engineers. This question can be interesting in relation to the relevancy of the business architecture as presented with APEAM.

MS2: *How often are software increments deployed to production?*
This question helps in understanding how dynamic data in the APM system is and therefore helps in determining ideal refresh scenarios for the EAM data.

MS3:*Is the deployment process automated and by which means?*
When considering how to integrate APEAM into the software development and deployment lifecycle, this question helps in finding potential insertion points.

MS4: *Are you aware of the external and internal dependencies of your services and their implications?*
External dependencies hereby mean services outside of the own domain context, while in contrast internal dependencies are inside the own domain context. The assumption here is

that determining responsibilities outside of ones own domain context can be complicated and there might be a lack of information.

MS5: *Are conventions applied when naming a service and where are they used?*
This question allows to understand where the data used for the inference of the naming convention of APEAM could be extracted from. This question helps to understand whether the Docker assumption used in APEAM holds true as a source for the naming convention. Further it allows to find new sources for defining the naming required for creating the unified service entity.

MS6: *Do you think the microservice architecture at your company reflects domain driven design?*
If there already is an inherent trust issue in the current architecture designs it bears the potential for applying APEAM to fail. The conventions of APEAMs automated components build on the basis of defined conventions and the proper implementation of the product based organization model.

APM1: *How and do you trace back incidents to affected business units?*
One of the potential use cases of APEAM is understanding the relationship of runtime components to their respective overall business impact. APM software by itself cannot fulfill this task completely as it lacks this specific information. It then is interesting to understand how this problem is handled in practice or whether it even can be considered problem.

APM2: *What information or data do you typically use in the APM software?*
From understanding how the APM software is used in practice, potential integration points for APEAM could be deducted, so that it more ideally integrates into the respective workflow.

APM3: *Do you think that your APM solution ideally supports your daily routines?*
Similar to the aforementioned question, this question can help in understanding whether APEAM and its enterprise graph have a broader understanding about relevant information required to fulfill certain tasks.

EAM1: *How and by whom are architecture models kept up to date at your company?*
This question helps in identifying different stakeholders of APEAM as well as processes in which EA models are updated.

EAM2: *Do you think the data in your APM tool is complete and if not what do you think is missing?*
One of the main goals of APEAM is to increase the trust in the data used in the EAM software. For this to be relevant there to some extent has to be an inherent trust issue with the existing solution, which this question validates.

EAM3: *How is the information from your EAM models typically used?*
This question helps in identifying which other types of use cases APEAM could poten-

tially fulfill if not done completely by the EAM tool itself already.

EAM4: *What granularity level of the technical architecture do the model elements in your EAM tool reflect?*
While APEAM was primarily designed with the scope of connecting the entity of services, a lot of other low level information could potentially be synchronized from the APM to the EAM models. This question help in identifying further relevant entities.

APEAM1: *How do you think APEAM could support some of your tasks and if so to what extent?*
With this question a general idea of the practicalities of APEAM are achieved. In general this question should be answered after a longer period of making use of APEAM, which unfortunately could not be done in the scope of this research.

APEAM2: *What do you see as potential usages of the Enterprise graph API?*
The enterprise graph API bears a lot of potential use cases which can be identified from different stakeholders using this question. Further this question help also in identifying other potential data sources for APEAM.

APEAM3: *Where do you think APEAM could be integrated into the current development processes?*
Different perspectives on where APEAM fits in the software development lifecycle not only help potential points at which to synchronize the data, but also help in understanding the different states APM data has from the perspective of the users.

4.3.2. Results

In general it can be seen that the feedback for the proposed solution has been positive. Some of the necessary assumptions, such as the proposed naming conventions, were considered as feasible, especially due to the fact that they have been in place for some teams already. The enterprise graph and its API was considered as one of the main features as many of the interviewees were already proposing a lot of different use cases by attaching further metadata and systems. Even for participants who were not directly interested in the data residing in planningIT itself, the data exposed via the API was relevant for applications such as monitoring.

5. Limitations and Outlook

5.1. Limitations

In the course of the development of APEAM a few assumptions were made or some restrictions deliberately applied in order to build a working prototype. These type of limitations to the current implementation are explained in the following paragraphs.

Naming convention. The naming conventions which were established in 2.4 were only validated in the context of the company the prototype was implemented in. Therefore further validation is required, if the conventions make sense in other companies. Alongside the option of other sources for inference of domain association could be evaluated as present in the mentioned section.

Applicability across different tools. APEAM was designed to be independent of the choice of APM or EAM tool, which is reflected by the provider model design in 3.4. To this end however a few of the capabilities from the APM software dynatrace and the EAM software planningIT were seen as given and not validated across other tools of choice in the industry. While the provider model allows this data to originate also from outside the boundaries of these respective types of tools there still remains to be proven whether all of these capabilities can be fulfilled by other tools of the trade.

Long term application. APEAM was built as a prototype with multiple iterations and changes in the time range this research was conducted in. To this end some of the concepts as explained in 4.2.2 should be tested for a longer time especially when it comes to the proper inclusion in the workflows of teams and architects.

5.2. Outlook

On the basis of the current capabilities of APEAM, the evaluation presented in **??** and other matching open research questions some potential extensions of the current prototype are summarized in the following paragraphs.

Extending the enterprise graph. As part of the evaluation extensions points for the enterprise graph were identified. In contrast to the topic of **Enterprise graph generation**, it is very well possible to add more data source systems to the GraphQL layer. Overall many of the following use cases are related to extending the enterprise graph in some specific way.

Interface documentation. The APM system is capable of identifying exposed and called interfaces of respective deployed services, as long as they are being called. Some EAM tools, as e.g. planningIT, offer the ability to document technical interfaces between services. To this end there is further sync potential. As there has been debate about where and how to document interfaces for APIs [13][18], the enterprise architecture model repositories can be one of the richest places due to the a better understanding in which domain contexts and connections certain interfaces operate. APEAM could in this regard add further sync mechanisms or help exposing this information in the context of the enterprise graph.

Incident management. From the interviews it became apparent that especially the operational aspects of services and their domain relationship in turn is interesting in practice. For that reason APEAM could utilize incident data as provided by the APM system and traverse the graph for relating it to affected business domains.

Business process integration. Based on the link created by APEAM services can be connected. On a more granular level however these services and their interfaces represent different parts of an overall business process. Therefore based on the existing graph deeper links could help in understanding these type of relationships to further analyse business impact from any type of operational problem.

Technical architecture sync. Some EAM tools, as it is the case for planningIT, are capable of representing different types of architectures, including the technical architecture. Therefore items such as infrastructure components can be documented as well. To this end when enriching the EAM system with further real time information about these components, different type of planning capabilities could be used.

6. Conclusion

The trend of developing larger applications in the form of microservices as well as the accompanying practice of domain driven design pose new challenges to the practices of enterprise architects. To provide a better tool support and drive the trend for automation also in the EA domain, APEAM was proposed, a solution built on the foundations of existing tools in companies. By combining automatically discovered data from APM systems and interpreting it to enhance information in an EAM tool, the presented problem could be addressed by unifying the concept of a "service" of both worlds and maintaining the relationship. To enable fully automated data integration, naming conventions and workflows were introduced so that manual efforts could be kept to a minimum. These conventions were verified and tested in the context of a large company. With the integration layer between both worlds in place, this research further introduced the notion of an "enterprise graph". The "enterprise graph" introduced the idea providing views on source systems in a way, which allowed data to be integrated across entities originating from these source systems and joining them for a single representation and thereby allowing new types of use cases and analysis capabilities to be explored.

Appendix

A. Interview results

A.1. Interview: Backend Developer - Checkout

GQ1: Backend developer working in the domain "Checkout" on different services such as the "Basket" for online and offline.

GQ2: Working with microservices for about 2 years

GQ3: Currently using primarily dynatrace, also used AppDynamics in the past.

GQ4: Not worked with any type of EAM tool, but designed different software architecture aspects in diagram tools

MS1: While when starting developing with the technical concept of microservices for separation of concerns, domain driven design has only be applied recently with the introduction of a domain based product organization.

MS2: Right now software increments are deployed with every commit that is merged to the "master" branch of the respective repository. However there lately have been ideas to introduce an error budget for limiting teams with higher frequency of runtime breakage, so this could potentially change.

MS3: The deployment process is currently automated by the build tool of the used cloud provider. In the past build automation was done with the CI/CD tool "Jenkins". Both of these tools can still be seen across teams in the company.

MS4: In general it can not be said that there is awareness for every interface due to the fact that different aspects were developed by different team members. In case of a problem in production however dynatrace helps in discovering the type of relationships.

MS5: Right now no naming conventions are applied in the software itself. In the past naming conventions were used to find deployed services based on Docker image names. However the deployment automation now uses naming conventions and applies them on the basis of the product organization.

MS6: I would say yes, however there are different maturity levels

APM1: This is not yet responsibility of the product teams, but will likely change soon. Therefore right now the product owner knows at best about the affected units and application operations.

APM2: While we would like to use the APM software for identifying improvements, right now it is mostly used for error analysis. For this we mostly look at CPU, memory, and requests to other respective services.

APM3: APM software by itself is great, but we currently are exploring capabilities of runtime debugging in production, as offered by tools such as OverOps.

APEAM1: I could see APEAM help us in designing APIs which adhere better to domain driven design. Right now there is general availability of good API collaboration tools other than just working on the same documents, most of the tools we used did not allow to explore how APIs are in context to each other.

APEAM2: No specific use case for the enterprise graph

APEAM3: The most stable architecture can probably only be instrumented in the production environment. For this reason I would recommend running it there, as there also synthetic monitoring right now triggers most processes automatically.

A.2. Interview: Backend Developer - Search

GQ1: Backend developer working in the domain ”Discovery” on services related to searching and recommending products.

GQ2: Started working with microservices just recently and at this company for the first time

GQ3: Started using dynatrace just recently

GQ4: Not worked with any type of EAM tool ever

MS1: To some extent we do, however there hardly is a lot of domain specific logic and entities required in our domain context

MS2: Deployment is running on every push to the master branch, but only based on accepted pull requests on sprint ending. However we use feature branch deployments which deploy on every pushed commit.

MS3: Deployment process is automated by an internal orchestration of the a proprietary cloud provider tool

MS4: We hardly know about many of our users due to the fact that the search API is used across many different services in the company. For the most part we only consume product data which is the only dependency as of now.

MS5: Based on our product name, naming conventions are applied on Docker Image build

MS6: I can not really tell, as I hardly have access to this information

APM1: Our product owner is contact with stakeholders affected by outages or other problems. However there might be services which use our service without our knowledge.

APM2: We just started using the APM software more, due to the nature of the criticality of our service in terms of speed, we looking more into metrics such as response times for different resources.

APM3: We have too little of history in using the APM tool yet to be able to answer this question.

APEAM1: APEAM could help us understanding where many of the consumers of our API are coming from and in which contexts the data is applied.

APEAM2: The enterprise graph could help us by maybe discovering the team members of some of our consumers and inform the about changes automatically, such as deploy-

ments or temporary downtimes.

APEAM3: Our service is constellations are most stable in the production environment. However in our integration test pipelines all of our dependencies could potentially be discovered as well.

A.3. Interview: DevOps/SRE

GQ1: Developer working in the company for about 8 years, now for the platform product team for site reliability

GQ2: Not directly working in the context of microservice development, but however running the base platform many microservices are running on

GQ3: Responsible for managing dynatrace and AppDynamics in the past

GQ4: Worked a lot with planningIT in the past

APM1: This has been and still is one of the main tasks that our product team fulfills. We made use of a service responsibility registry when determining the relevant stakeholders. However this registry is often lacking the latest information on new services or changes in the responsibilities.

APM2: We utilize APM data to identify and solve all kinds of errors in the running environments, as we are still the maintainers of quite a few non product based legacy applications. Further on we want to use APM data to define error boundaries (”error budgets”) for product teams, so that there are consequences for

APM3: We are very happy with our current APM solution. However we often still lack information on some of the services which are automatically captured and reporting data to the APM software, as we are not directly involved in the plans of the different product teams.

EAM1: In the past the data was supposed to be kept up to date by application owners, however usually different individuals of enterprise architecture unit tried to keep it up to date based information they gathered with lots of effort.

EAM2: From the fact that I myself was not really consistent on keeping it up to date, I would rather say no.

EAM3: The data from the model entities we were supposed keep up to date, was used for capacity and budget planning. However as it has proven not to be reliable in the past it was a rather tedious process.

EAM4: In the past there have been different levels of abstraction in planningIT. While the business architecture as defined was rather stable, the technical architecture was usually drifting apart from reality.

APEAM1: APEAM bears a lot of potential when it comes to linking our findings to the relevant teams/products.

APEAM2: As we do not utilize the API of planningIT in particular, APEAM could serve

as the basis for retrieving the product team and other organizational data, such as affected domains of certain outages.

APEAM3: As we are seeing lots of different services pop up in testing and development environments already, we would like to see it running as early as possible in the development lifecycle. We currently utilize the dynatrace event log to identify changes to artifacts, which could also serve as the basis for running APEAM sync processes as well.

A.4. Interview: Product Owner

GQ1: Product owner in the domain Checkout, responsible for the online and offline journey

GQ2: Working in a team which utilizes microservice architecture style

GQ3: We lately started utilizing the APM software, in particular dynatrace

GQ4: Not yet, but planningIT is currently being introduced as parts of the duties of a product owner

APM1: As a product team in a business domain context we are aware of our sponsors, however we also have operational responsibility for integrations to our dependencies we lack this kinds of transparency right now. Usually a develop of the team can infer it from erroneous network calls, but unfortunately this knowledge is usually a silo.

APM2: Personally I just use the dashboards to get an overview of the operational performance of our services.

APM3: For the most part the APM software is not relevant to my daily routines, however the team is using it a lot and therefore I would say it supports at least our overall goals.

APEAM1: I can see the linkage of architectural changes to the respective user stories as relevant. As user stories are also captured in the context of enabling some part of a business process this could possibly enhance the business process documentation as well. With the current implementation it should allow me to prevent bothering the team to help me understand which changes were done recently.

APEAM2: The enterprise graph could be linked to JIRA and Confluence so that for certain deployed software components documentation can be more easily discovered. When maybe also introducing some kind of versioning, we can add the deployment state and JIRA tickets which aided the current artifact, as we already capture GIT commits in relationship to JIRA tickets.

APEAM3: When looking at the point the data is best captured, I would say as early as our development environments so that other product teams could possibly be informed about new consumers as this is not always communicated well.

A.5. Interview: Enterprise Architect

GQ1: Architect who worked as developer in the past. Now working in a direct CTO reporting function as part of the EA team.

GQ2: Not worked with the actual development of microservices but worked on the conception to some extent supporting different teams

GQ3: Not used this software, however considered when we were made aware of the capabilities in seeing communication relationships between different services, due to the fact that we are lacking transparency on this level.

GQ4: We use different tools for documentation, primarily a mixture of Confluence, PowerPoint and planningIT right now.

EAM1: In the past we tried to delegate responsibilities for keeping data up to date to the different teams, as it simply was not possible to cope with the amount of changes happening. In the current organizational model we try to establish this as a responsibility of the product owner role.

EAM2: We consider it complete to the limits of what we are actually aware. But probably the data in the technical architecture is incomplete and probably also partially wrong.

EAM3: We utilized the data in different scenarios for our own planning scenarios, e.g. when evaluating standard and individual software, migration planning and other types of refactoring and restructuring projects. We however also supported other endeavors from our stakeholders , such as team planning, budget planning or infrastructure planning. With the product organizational model we also support the functional placement in the different domain contexts.

EAM4: The metamodel elements in our architecture reflect a lot of different abstraction levels, from interfaces of applications, up to country level business support structures, depending on the different visualizations we build. This however is very much dependent on the use case.

APEAM1: We see APEAM as a potential solution enhancing the representation of our runtime architecture. Lately API management has become a very important topic and we see potential extension points for APEAM here.

APEAM2: The enterprise graph is a worthwhile idea and we could think about building visualization on the basis of it.

APEAM3: We work primarily on the more stable data from production environments, which is therefore the ideal point in time and source for APEAM.

Bibliography

[1] Rama Akkiraju, Tilak Mitra, and Usha Thulasiram. Reverse Engineering Platform Independent Models from Business Software Applications. 2012.

[2] N. Alshuqayran, N. Ali, and R. Evans. Towards Micro Service Architecture Recovery: An Empirical Study. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 47–4709, April 2018.

[3] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. Automated Discovery and Maintenance of Enterprise Topology Graphs. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 126–134, December 2013.

[4] Ruth Breu, Berthold Agreiter, Matthias Farwick, Michael Felderer, Michael Hafner, and Frank Innerhofer-Oberperfler. Living Models - Ten Principles for Change-Driven Software Engineering. *Int. J. Software and Informatics*, 5:267–290, 2011.

[5] M. Farwick, B. Agreiter, R. Breu, M. Häring, K. Voges, and I. Hanschke. Towards Living Landscape Models: Automated Integration of Infrastructure Cloud in Enterprise Architecture Management. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 35–42, July 2010.

[6] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, Boston, 1 edition edition, November 2002.

[7] Martin Fowler. Microservices, March 2014. https://martinfowler.com/articles/microservices.html accessed on 09/30/2018.

[8] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle. MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 298–302, April 2017.

[9] Gartner Inc. Application Performance Monitoring (APM) Suites Reviews.

[10] Martin Kleehaus, Ömer Uludag, Patrick Schäfer, and Florian Matthes. MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-Based Environments. pages 148–162. June 2018.

[11] Florian Matthes, Sabine Buckl, Jana Leitel, and Christian M. Schweda. *Enterprise architecture management tool survey 2008*. Techn. Univ. München, 2008.

[12] B. Mayer and R. Weinreich. An Approach to Extract the Architecture of Microservice-Based Software Systems. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 21–30, March 2018.

[13] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, December 2011.

[14] Jeanne Ross, Peter Weill, and David C. Robertson. *Enterprise Architecture as Strategy — Creating a Foundation for Business Execution*. May 2006.

[15] Dominik Rost, Matthias Naab, Crescencio Lima, and Christina von Flach Garcia Chavez. Software Architecture Documentation for Developers: A Survey. In Khalil Drira, editor, *Software Architecture*, Lecture Notes in Computer Science, pages 72–88. Springer Berlin Heidelberg, 2013.

[16] Sascha Roth. *Enterprise Architecture Visualization Tool Survey 2014*. epubli, Berlin, 1 edition, April 2014.

[17] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.

[18] S. M. Sohan, C. Anslow, and F. Maurer. SpyREST: Automated RESTful API Documentation Using an HTTP Proxy Server (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 271–276, November 2015.

[19] Katharina Winter, Sabine Buckl, Florian Matthes, and Christian Schweda. INVESTIGATING THE STATE-OF-THE-ART IN ENTERPRISE ARCHITECTURE MANAGEMENT METHODS IN LITERATURE AND PRACTICE. *MCIS 2010 Proceedings*, September 2010.

[20] Dannver Wu, Jinho Hwang, Maja Vukovic, and Nikos Anerousis. BlueSight: Automated Discovery Service for Cloud Migration of Enterprises. In Khalil Drira, Hongbing Wang, Qi Yu, Yan Wang, Yuhong Yan, François Charoy, Jan Mendling, Mohamed Mohamed, Zhongjie Wang, and Sami Bhiri, editors, *Service-Oriented Computing – ICSOC 2016 Workshops*, Lecture Notes in Computer Science, pages 211–215. Springer International Publishing, 2017.