TECHNISCHE UNIVERSITÄT
MÜNCHEN

DEPARTMENT OF INFORMATICS

Master's Thesis in Information Systems

Enhancing Business Process Mining with Distributed Tracing Data in a Microservice Architecture

Jochen Graeff

TECHNISCHE UNIVERSITÄT
MÜNCHEN

DEPARTMENT OF INFORMATICS

Master's Thesis in Information Systems

Enhancing Business Process Mining with Distributed Tracing Data in a Microservice Architecture

Eine Erweiterung von Business Process Mining mit Hilfe von Distributed Tracing Daten in einer Microservice-Architektur

Author:	Jochen Graeff
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Martin Kleehaus, M.Sc.
Submission Date:	15.08.2017

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.08.2017 Jochen Graeff

Abstract

A main goal of Enterprise Architecture Management (EAM) is to align business needs with IT capabilities in order to understand and visualise interdependencies between the layers of an Enterprise Architecture (EA) and therefore gain a holistic view.

This thesis aims to provide the required visibility between the business and application layer through a monitoring and analysis approach. The approach enhances traditional process mining with performance indicators obtained from the application layer. With the developed prototype, correlations between user behaviour from the business layer and system performance occurring in the application layer can be detected.

This is achieved by applying process mining discovery techniques on distributed tracing data originating from an instrumented microservice architecture. Through an activity log generation component, low level events of the distributed tracing data are transformed into a high level activity log comprising of two hierarchies: user and system activities.

It is shown, that the developed prototype can be implemented with little effort and provides a cost-efficient bottom-up approach to discover business processes in microservice architectures in near real-time. The approach comes with little instrumentation effort and is agnostic to programming frameworks or architectural styles.

Limitations especially emerge through possible performance losses caused by high sampling rates in the instrumented software component.

Contents

List of Tables v

List of Figures vii

Listings ix

Glossary xi

1. Introduction 1

1.1. Motivation and problem statement . . . 1
1.2. Research methodology . . . 3
1.3. Thesis structure . . . 4

2. Foundations of microservice architectures, distributed tracing and process mining 7

2.1. Microservice architectures . . . 7
2.1.1. Characteristics, benefits and challenges of microservice architectures . . . 7
2.1.2. Microservice composition styles . . . 9
2.2. Distributed tracing . . . 12
2.2.1. General concept and raison d'être . . . 12
2.2.2. Principle function, terminology and tracing architecture . . . 13
2.3. Business process mining . . . 18
2.3.1. Use cases of business process mining . . . 18
2.3.2. Types of process mining . . . 18
2.3.3. Description of the α-Algorithm . . . 19
2.3.4. Structure of an event log . . . 21
2.3.5. Strategies of data acquisition . . . 22
2.3.6. Transforming the multi-dimensional reality into a flat event log . 23
2.3.7. Challenges in extracting data and generating event logs . . . 23

3. Description of the system under survey 27

3.1. General system architecture . . . 27
3.2. Description of the business functionalities . . . 31

3.3. Description of the software architecture 37
3.3.1. Service communication architecture 37
3.3.2. Infrastructure microservices 38
3.3.3. Exemplary setup of a business microservice 39

4. Description of the automated process discovery prototype **43**
4.1. Extended system under survey architecture 43
4.2. Distributed tracing and process mining tool selection 45
4.3. Instrumentation of the system under survey 46
4.4. Data architecture 47
4.4.1. Tracing data persistence data 48
4.4.2. General table structure 48
4.5. Event log generation 53
4.5.1. Scoping the event data 53
4.5.2. Transformation of low level events to activities and binding to process instances 54
4.6. Process configuration and analysis creation in the process mining tool . 60
4.6.1. Process mining workflow 60
4.6.2. Data model and activity table configuration 61
4.6.3. Process visualisation 63
4.6.4. Description of created analyses 67

5. Discussion **81**
5.1. Benefits of the proposed solution 81
5.1.1. Cross-domain analysis 81
5.1.2. Resource efficient data source for generating event logs 82
5.1.3. Portability 82
5.1.4. Ubiquity 83
5.1.5. Flexibility on process perspectives 83
5.1.6. Foundation for real-time process mining 84
5.1.7. Bottom-up process discovery in legacy systems 84
5.2. Limitations of the proposed solution 85
5.2.1. System under survey 85
5.2.2. Suitability of applied process visualisations 86
5.2.3. Performance overhead through high sampling rates 89
5.2.4. Real-time event handling, event log generation and process discovery 90
5.3. Related work 91

6. Summary and outlook **97**

Bibliography **101**

A. Appendix **109**
A.1. Additional figures and tables 109
A.2. Technical documentation of analyses in the PMT 114
A.3. Installation guide for prototype setup 123

List of Tables

2.1. Sample event log 21

3.1. Dependency matrix between user activities and microservice involvement 30
3.2. *Webui service* endpoints and their usage in user activities 34
3.3. *Maps service* endpoints and their usage in user activities 35
3.4. Accounting service endpoints and their usage in user activities 35
3.5. *Notifications service* endpoints and their usage in user activities 36
3.6. *Payments service* endpoints and their usage in user activities 36
3.7. *User service* endpoints and their usage in user activities 36
3.8. *Cars service* endpoints and their usage in user activities 37

4.1. Database tables usage classification 48
4.2. *zipkin_spans* table definition 49
4.3. *zipkin_annotations* table definition 50
4.4. Description of the *activites* table 51
4.5. *Activities table* configuration in the Process Mining Tool (PMT) 62
4.6. Structure of analysis description 68
4.7. Key Performance Indicators (KPIs) of header component in the Business Analysis (BA) analysis 69
4.8. KPIs of header component in the Application Analysis (AA) analysis . 72
4.9. KPIs of header component in the Cross-domain Analysis (CDA) analysis 75
4.10. KPIs of header component in the Single User Activity Analysis (SUAA) analysis 78

5.1. Description of the classification framework for related work 92
5.2. Classification of related work 95

List of Figures

1.1. Research process . . . 3

2.1. Service orchestration, adapted from [46] . . . 11
2.2. Service choreography, adapted from [46] . . . 12
2.3. Example trace of a request from the front-end processed by multiple back-end services, adapted from [64] . . . 14
2.4. Waterfall diagram showing the four spans and their ids from a temporal perspective . . . 14
2.5. Temporal occurrence of timestamps during a span's life for a request and response between Service A and B, adapted from [8] . . . 15
2.6. Distributed tracing architecture [18] . . . 16
2.7. Exemplary trace generation explained in sequence diagram [10] . . . 17

3.1. Architectural overview of System under Survey (SUS) . . . 28
3.2. Call flow for */endRental* request at the front-end (i.e. *End car rental* user activity) . . . 31
3.3. Web front-end rendered by the *webui service* . . . 33

4.1. Overview of the prototype development process . . . 43
4.2. Architectural setup for the data processing . . . 44
4.3. End-to-end process mining workflow . . . 61
4.4. Data model configuration in the PMT . . . 62
4.5. User click path visualisation in the PMT . . . 64
4.6. Model excerpt showing the *case frequency* KPI . . . 65
4.7. Model excerpt showing the *durations* KPI . . . 66
4.8. Model excerpt showing the *conversion rate* KPI . . . 67
4.9. Business Analysis in PMT . . . 71
4.10. Application Analysis in PMT . . . 74
4.11. Cross-Domain Analysis in PMT . . . 77
4.12. Single User Activity Analysis in PMT . . . 80

5.1. Ambiguous placement of system activities . . . 87
5.2. Waterfall-like visualisation of a trace in *Zipkin*, triggered through the /bookPackage request from the front-end . . . 87

5.3. Process visualisation of a the *Book package* user activity and the called system activities in the PMT 88

A.1. *List available cars* activity, triggered through /getCars request 109
A.2. *Reserve car* activity, triggered through /reserveCar request 109
A.3. *Book car* activity, triggered through /bookCar request 109
A.4. *Unlock car* activity, triggered through /openCar request 109
A.5. *End car rental* activity, triggered through /endRental request 110
A.6. *Show balance* activity, triggered through /showBalance request 110
A.7. *Book package* activity, triggered through /bookPackage request 110
A.8. *Show driving history* activity, triggered through /showHistory request . 110
A.9. *Report issue* activity, triggered through /reportIssue request 110
A.10. *Find route* activity, triggered through /findRoute request 110
A.11. Complete data model in UML notation 113

Listings

3.1. The application's local bootstrap.properties 39
3.2. Class definition for the accounting application 40
3.3. Endpoints definition in the accounting controller 41

4.1. Class definition for ZipkinApplication.java 46
4.2. Appending the *session_id* to the span 47
4.3. Creation of healthy user activites 55
4.4. Activity name alteration 57
4.5. Filter alteration on join 57
4.6. Creation of healthy user activites 58
4.7. First filter alteration on join for failed system activities 60
4.8. Second filter alteration on join for failed system activities 60
4.9. Definition of edge durations 65
4.10. Definition of activity conversion rate 66
4.11. Definition of edge conversion rate 67

Glossary

A

AA Application Analysis
AOP Aspect-oriented Programming
API Application Programming Interface
APM Application Performance Management

B

BA Business Analysis
BI Business Intelligence
BPEL Business Process Execution Language
BPMN Business Process Model and Notation

C

CDA Cross-domain Analysis
CRM Customer Relationship Management

E

EA Enterprise Architecture
EAM Enterprise Architecture Management
ERP Enterprise Resource Planning
ESB Enterprise Service Bus
ESP Event Stream Processing

H

HTTP Hypertext Transfer Protocol

K

KPI Key Performance Indicator

L

LLCM Living Lab Connected Mobility

O

OLAP Online Analytical Processing

P

PAIS Process-Aware Information Systems
PMT Process Mining Tool
POM Project Object Model
PQL Process Query Language

R

RDBMS Relational Database Management System
REST Representational State Transfer
RPC Remote Procedure Call

S

SOA Service Oriented Architecture
SQL Structured Query Language
SUAA Single User Activity Analysis
SUS System under Survey

U

UI User Interface
UML Unified Modeling Language
URI Uniform Resource Identifier
URL Uniform Resource Locator
UX User Experience

W

WFM Workflow Management System

1. Introduction

1.1. Motivation and problem statement

One of the main challenges in handling EAs is reacting to new environments, characterised through ever-evolving technologies and rapidly changing business requirements [22]. The discipline of Enterprise Architecture Management (EAM) tries to address these challenges and produced several frameworks in the past, that help to align business needs and IT capabilities from a holistic perspective through models and processes [9, 57]. EA models capture connections and dependencies between the layers of an enterprise [13]. The division of an EA into layers, as applied in most frameworks, aims to enable the management of entities and the reduction of complexity [60]. The definition of layers varies in the many frameworks developed while the contents introduced by a layer are mostly resembling each other [54]. In this work, a three layer perspective will be applied, which can often be found in practice and consists of an infrastructure layer at the bottom, focusing on technological aspects such as computing and communication infrastructure, followed by the application layer above that entails aspects of the software that is deployed on the IT infrastructure and the business layer on top, that is comprised of the organisation's business processes [40].

The transparency, that holistic EA models aim to provide, is endangered trough rapidly changing environments affecting all layers of an EA. Maintaining these models to continuously support the EAM function requires on-going effort [26].

A coexisting approach for providing transparency in an EA is monitoring. In comparison to EA models, that capture static representation at build time, monitoring provides visibility in form of real-time behaviour at run-time. For each of the before described layers, various approaches were developed that try to achieve transparency [39]. These monitoring approaches are all driven by a set of analysis questions, that arise most often from of a single domain. Therefore, they focus on monitoring and analysing an EA from a local perspective and provide transparency for a single layer only.

One of theses approaches, that provides visibility for the business layer, is *process mining*. Process mining is a relatively young research discipline that makes use of the growing amount of available event data that IT systems produce. This data is used to generate process models and discover process weaknesses.

An emergent technique that provides visibility for the application layer is *distributed*

tracing. Distributed tracing found a lot of application recently due to the spread of microservice architectures, which especially require visibility through their distributed nature. Distributed tracing enables latency optimisation and the discovery and analysis of back-end errors.

However, analysis questions are not restricted to a single layer only. They often emerge in-between the layers and are not addressed through the existent approaches. An example, that illustrates the correlation between system and business performance, is the case of Amazon, that found out that 100 ms in surplus latency drops sales by 1% [62]. Moreover Google stated, that they receive 20% less traffic, when their web sites have an additional latency of 500 ms [43].

Finding exactly these correlations and dependencies, which lay in-between the layers or become apparent through combining metrics from multiple layers, promise a vast potential for generating insights. So far, no approach has been described that enables analysis questions between the business and the application layer which can be implemented with little effort on the one hand and contains potential for deep drill downs through a rich availability of data on the other hand (see section 5.3). This thesis presents such an approach.

One objective of this work is to evaluate distributed tracing data as an alternative input source for generating high level event logs for traditional process mining. Since distributed tracing comes with almost negligible instrumentation effort, it can serve as an alternative to custom business logging of existing IT systems. A second objective is to demonstrate, that the technical origin of that data can enhance the traditional business view on a process by technical performance data and thereby give system planners and process managers better insights into the dependencies between user behaviour, business and application performance.

To achieve the objectives, the following research questions were defined in order to govern the work of the thesis.

RQ1. How can a relationship between business activities and a distributed application architecture be established?

RQ2. What data has to be extracted and how has it to be mapped to enable and store the relationship knowledge?

RQ3. How can business process mining be extended with technical aspects in order to uncover

1. user and system throughput times for business activity executions and
2. correlations between business process performance and system behaviour?

The approach was tested through instrumenting a sample microservice architecture that mocks a car sharing service. Through user interaction with the system, logs were

automatically generated through a distributed tracing system which was connected to the existing architecture. The application context is related to the use case of the TUM Living Lab Connected Mobility (LLCM)[1] project, that tries to support the digital transformation in the area of smart mobility and smart city. One of the sub-projects is concerned with the integrated monitoring of a platform that provides various mobility services. This thesis relates to that sub-project.

The approach presented in this thesis contributes to two research disciplines through combining techniques of distributed tracing and process mining.

Firstly, it contributes to the research field of process mining, by presenting a novel approach for generating high level event logs using tracing data of low level and high granularity as an data input. Secondly, it contributes to EAM, to which it presents a novel technique for analysing and monitoring the business and application layer combined and provides thereby a more holistic view on an EA.

1.2. Research methodology

The research methodology used for the thesis is inspired by the design science approach described by Hevner et al. [34].

The research process (see Figure 1.1) started with an extensive literature review in the fields of monitoring both business and application layer combined. After identifying a research gap and the lack of suitable approaches for the described problem, the identified research questions were formulated.

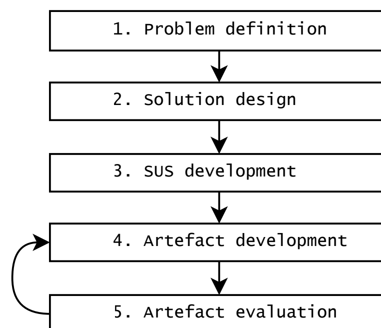


Figure 1.1.: Research process

In the next stage, a solution scenario was sketched and first hypothesises were formulated in order to close this gap. This included the selection of *distributed tracing* and *process mining* as essential technologies of the later proof-of-concept artefact as well as an outline of a possible architecture.

[1] http://tum-llcm.de/

Third, a sample architecture was built, that serves as a SUS. The architecture was built after common patterns that were identified through sample applications available on the internet and literature research.

In the fourth step, the sample architecture was instrumented with a distributed tracing library. From there on, tracing data could be generated through simulating user clicks in the SUS. Moreover, the actual prototype was developed, which entails the event log generation algorithm, implemented in an own system component. Additionally, multiple analyses were developed in the PMT to evaluate and showcase the applicability for different use cases.

Following the design science approach, an evaluation was performed as a fifth step by using the data created from the sample system in order to verify the correctness of data and application of the solution in general. During this phase, benefits and limitations were identified and validated. Moreover, the presented approach was evaluated against related work.

Stages four and five embody the integral parts of the work and were passed through in multiple iterations during the research process.

1.3. Thesis structure

The thesis is structured as follows: Chapter two conveys basic knowledge in the domains of microservice architectures, distributed tracing and process mining. This is necessary, since the work will use a microservice architecture as a SUS and distributed tracing as well as process mining will present integral parts of the proposed solution. Thereby, the main characteristics together with benefits and challenges of microservices will be described first. Moreover, two main prevalent service composition styles, namely service orchestration and service choreography will be described together with benefits and limitations of each. Afterwards, the concept of distributed tracing is described along with the specifics of its data model and logging architecture. This provides an understanding of the data that is needed for developing the event log algorithm. The third part of chapter two gives an introduction into process mining including use cases and typical analysis questions, types of process mining, an exemplary description of a process mining algorithm as well as typical challenges in extracting data and generating event logs.

Chapter three documents the prerequisite architecture that was built in order to serve as a SUS. It provides a general overview of the system, a description of business functionalities that the system entails as well as a description of the software architecture applied and components used in the SUS.

Chapter four entails the main part of the work: the approach applied for implementing the proof-of-concept prototype. This includes first a tool selection process followed by

a description of the instrumentation applied to the SUS. Next, the architectural setup for generating event logs in near real-time is described, followed by a description of the developed table structure. Thereafter, the central event log generation algorithm, that translates low level events from the distributed tracing instrumentation to high level events in a structure that is applicable for the process mining algorithm, is described. In the next section, the created event log is, together with further tables, configured in the data model of the PMT. Subsequently, it is described how the two layers of the EA are visualised in one process model, followed by a description of the four analyses build, that demonstrate the applicability of the proposed approach.

Chapter five discusses benefits, limitations of the presented poof-of-concept prototype. Moreover, the proposed solution is evaluated against related work.

The final chapter six summarises the thesis and provides an outlook over research gaps that need to be addressed in the future.

2. Foundations of microservice architectures, distributed tracing and process mining

This chapter introduces the foundations of microservice architectures, distributed tracing and process mining.

The first part of the chapter is an introduction into microservices, which is of importance, since this thesis tries to discover processes from architectures build with this pattern. Therefore, the characteristics of microservice architectures need to be understood first.

The second and third part describe the basic concepts of each process mining and distributed tracing, which are the two techniques that will applied together in the presented approach. Both techniques are employed in the proof-of-concept in order to expand transparency from a business-only perspective to a holistic view, that also integrates the application layer.

2.1. Microservice architectures

In the following section, first a short introduction about the characteristics of microservice architectures is given. Moreover, the benefits that make them a widely used pattern in implementing enterprise software architectures are being described. The second part elaborates on how a microservice architectures handles business processes and how in general inter-service communication can be realised. Therefore, two distinctive patterns were identified, that will be presented along with their benefits and drawbacks.

2.1.1. Characteristics, benefits and challenges of microservice architectures

In recent years, developing enterprise systems using a microservice pattern have become en vogue [22]. Many leading technology companies like Netflix, Google, Twitter, and Amazon, that in the past have paved the way for technological advance, adopted the architectural style and migrated their systems successfully from a monolithic to a microservice architecture [15]. Martin Fowler and James Lewis, were among the first

providing a working definition [42]:

> "In short, the microservice architectural style is an approach to developing a single application as **a suite of small services**, each running in its **own process** and **communicating with lightweight mechanisms**, often an HTTP resource API. These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery. There is a **bare minimum of centralised management** of these services, which may be written in different programming languages and use different data storage technologies."

The reasons for more and more enterprises and practitioners using this architectural pattern are diverse. Most causes are connected to the downsides that developing applications in a monolithic style bring and that appear together with external drivers like organisational agility, shorter time-to-market as well as the need for selective scalability [69, 46]. Microservices aim to minimise the costs for change, create more operational efficiency and encourage generalisation, replaceability and reuse [22, 46]. In the following the main characteristics of microservices are explained by mapping out how they deal with the pitfalls that monolithic architectures bring.

Technological pluralism In a monolithic system, organizations tend to standardize their platform with growing complexity according to a centralized governance [70, 42]. This contains restrictions for using a predefined set of technologies (programming languages, frameworks and tools) that are not necessarily the best suited for the in hand problem. By splitting a monolith into services, the developing teams freely choose the technology that fits best for the problem [28]. Advances in technology can also be adopted easier in a microservice architecture since for example using new frameworks doesn't require a complete rewrite of the whole application.

Polyglott persistence Enterprises tend to use single databases across the organization which often has its motivation in limited know-how in maintainability, database license costs that convene in the aim of standardisation [42]. Polyglott persistence allows every application or service to choose the best suited database technology and data model for the problem at hand. The different types of databases are part of the platform's core design. Polyglott persistence is something that may also exist in monolithic systems but is more likely applied in microservice architectures [42].

Deployment With shorter development cycles, where incremental change is desired, building and starting huge single applications takes a lot of time and slows down the development process. In a microservice environment, a developer typically works on a single service, that by lines of code is much smaller compared to a whole application. Therefore start up times for applications is much faster [46].

Especially in a continuous development setting, where changes are often pushed into production multiple times per day, it is difficult to do it with a monolithic architecture, since the complete application would have to be deployed [42].

Scalability In a microservice architecture it is possible to only replicate single services according to the need instead of horizontally scaling the whole monolith to multiple instances and placing it behind a load-balancer. Therefore, resources can be scaled selectively, only for points in the system where bottlenecks occur [56].

Together with the advantages of microservices, of course there are also downsides. Some are characteristic for distributed systems as for instance lower performance and reliability for communication through remote calls compared to in-process function calls [27]. Another issue, that microservice architectures implicate, is eventual consistency, which occurs through decentralised data management and data replication by multiple microservices [56]. A third challenge is increased operational complexity. Instead of a manageable amount of applications, with microservices, a high number of services needs to be operated and deployed regularly [68, 27]. Proponents of microservices moreover state, that through smaller application size complexity decreases likewise. This is often not true because the complexity remains and is now just occurring *between* the services [63]. This complexity somehow needs be monitored. Gaining visibility into systems and debugging behaviour that spans over multiple services is therefore a main challenge [8, 63]. To address the challenge, distributed tracing systems were developed that provide the required visibility and add minimum performance overhead. Insights into operational performance as for instance finding root causes for high user request latency is feasible with distributed tracing [32]. A description of its functioning and what further drivers exist that make distributed tracing a crucial tool for developing and maintaining microservice architectures can be found in section 2.2.

2.1.2. Microservice composition styles

An aspect about microservices that is especially relevant for this work is how architectures apply different patterns for inter-service communication and the execution of business logic. This is of special interest, since a sequence of inter-service communication can be interpreted as a process. This work aims to detect and visualise such processes. Therefore, a dependency arises from how the system generally integrates various microservices and provides microservice composition to how the resulting tracing data is affected by the architecture and can be used for the generation of events logs as input for process mining. A microservice composition style is defined as "how multiple microservices are connected in a flow to deliver what a user requested" [50]. Every composition style deals differently with the execution of cross-service processes in a microservice architecture.

This subsection mainly deals with architectural patterns of managing business processes

that span across multiple services in a *microservice architecture*. Nevertheless, it makes sense to first understand how predecessor architectural styles, namely Service Oriented Architectures (SOAs), dealt with the problem of routing and applying business logic across services, in order comprehend the differences.

In a SOA, the central unit that is steering process instances is the Enterprise Service Bus (ESB) [61]. Through the ESB, the communication between all services is handled centrally using messages. A message completes two tasks: First, it transports the actual data that is shared between the services and second, it controls the sequence of calls between the different services [16]. In the ESB, business rules are stored and it is defined how messages between services are routed, transformed, and orchestrated [36].

A microservices architecture shares common goals compared to a SOA as for instance removing tight dependencies between components. Some argue that the microservices pattern is a variant of a SOA [70], or microservices are "SOA done tight" [11, 46]. What can be generally said about a microservice architecture, is that a ESB, as described above, is eliminated and generally regarded as an anti-pattern [36], since it contradicts the objective of loose coupling. The central smartness of the ESB is moved to the services itself [42]. Each microservice carries its own business logic and can trigger various of other services in a synchronous or asynchronous fashion [42].

If requests between the services are ad-hoc and uncoupled, the question arises on how those cross-service request are routed through the system? Sam Newman [46] poses the questions as follows:

> "As we start to model more and more complex logic, we have to deal with the problem of managing business processes that stretch across the boundary of individual services."

Two opposing patterns exist, that address the problem of a inter-service communication: service orchestration and service choreography.

Service orchestration

In service *orchestration* some authority like a *conductor* in an orchestra guides the overall service interactions [46]. In Figure 2.1 a *car rental* is finished by the user who triggers a request at the *cars service*. The *cars service*, who is the conductor, subsequently calls the *maps*, *accounting* and *billing service* in a defined order, and waits for a response of each before calling the other.

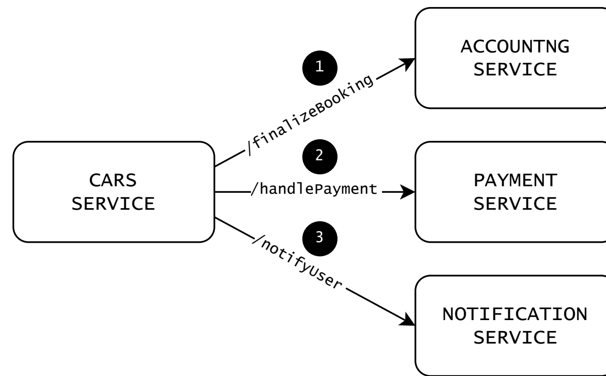


Figure 2.1.: Service orchestration, adapted from [46]

The orchestration unit typically communicates with its downstream services in a request/response manner [46]. The process sequence is directly implemented in the code of the cars service.

With this pattern, the end-to-end flow can be seen both at design as well as at run-time [12]. It is well suited for centralising the control flow using synchronous communication, where the state can be easily tracked by the conductor. That also comes with drawbacks [36]. For one, it creates dependencies between the services which lead to high coupling [46]. If for instance the accounting service is down, the car service will wait for its response before calling the next service. Another form of dependency appears when changing the process or adding new services [12]. Every calling service needs to know its downstream services. Moreover, the end-to-end processing time, which is the sum of each service processing time can become relatively high compared to asynchronous request without blocking [12].

Service choreography

In service *choreography*, governance of interaction between the services is decentralised [46]. Services can emit events to inform other services about 'things that happened'. They also subscribe to events or channels of interest and react accordingly to those without a higher supervision. Events are typically send asynchronously to an event bus, which is a dumb pipe where no logic is implemented [36]. Therefore an emitting service does not expect any response from the downstream service [35, 14]. The events sent do not comprise any commands as used in the CRUD approach but are defined as something remarkable that takes places inside or outside a business domain [45]. A service can also act as an event aggregator, that listens to multiple events, combining information and creating new events [20].

See below in Figure 2.2 how the same business workflow as above would be handled in a choreography styled architecture. The cars service would emit an *End car rental* event and the notifications, accounting and payment service would react to it and apply

specific business logic responsive to the event. They could also emit own events.

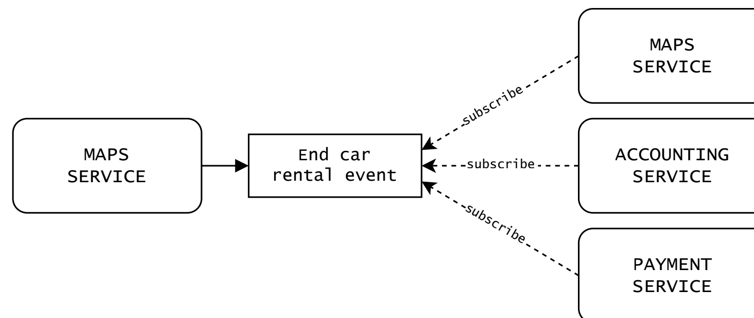


Figure 2.2.: Service choreography, adapted from [46]

The choreography style is highly decoupled, since an emitting service does not need to know its consumers. When a new service is added, it only has to be connected to the event stream and configured to subscribe to channels of interest [46]. Maybe a new event would have to be written but already implemented logic won't break due to that [68]. Moreover, the asynchronity applied leads to a faster end-to-end processing [12]. Due to the decentralised nature, no single services gains too much importance which means that a single point of failure does not evolve easily over time [12]. In case a service fails, and other services where still produce events stored in the event stream, the restarting services will be able to replay the events and catch up [12].

The downsides are, that the process flow is not explicit at design time. Additional components need to be configured to track if the execution was successful, since it is not per default apparent if an error occurred or the process is stuck [46]. Moreover, this approach requires a mind shift for developers, that have to have all possible situations in mind, that could appear during operations and are not explicit at design time [12].

2.2. Distributed tracing

2.2.1. General concept and raison d'être

With the conversion from monoliths to SOAs and the recently increasing adoption of microservices, the complexity of architectures increases: In distributed systems, multiple services run in various processes, possibly written in different programming languages being maintained by multiple teams [64]. This increases the need for more advanced monitoring and debugging techniques [63, 8].

Monitoring performance and debugging system malfunction in a monolithic system can be established through analysing log files originating from a single system. However, considering performance of microservices from an isolated perspective is not feasible

[64]. Transactions, where an initial user request might invoke a plethora of downstream services, need to be analysed as a whole, for instance to find out what request is the cause for abnormal latency. Moreover, the root cause for latency often lies in-between the services.

Without distributed tracing it would be necessary to look into every single log of each service that has been invoked through the request. Knowledge, about which services are involved in the request needs to be at hand. Furthermore, an investigating engineer needs to know about the implementation specifics of each service in order to understand the logged data. In a second step, transactions logged in each service somehow would need to be correlated *manually* to the initial request in order to derive insights about latency bottlenecks. To overcome this problem, workflow tracing systems were developed that centralise logging and correlate transactions spanning over different processes through a trace id [64, 31]. Those tools provide transparency in the application layer and establish a relationship between a user request in top of the stack and its complex processing in the distributed system [37].

Google was among the first who developed a tracing infrastructure, called Dapper [64], that aids to gain visibility in distributed systems. The design goals of Dapper were low instrumentation overhead, application-level transparency and ubiquitous deployment. Since then, a plethora of both open source as well es commercial distributed tracing systems were developed. Many of them implement the approach that was initially presented by Google in 2010.

Distributed tracing can be seen as a part of Application Performance Management (APM), which aims to provide techniques that help to achieve a sufficient level of performance during operation of a system, having in mind that performance is critical to business success [62].

In the following, the basic concepts of distributed tracing, including the terminology, the data model as well as the tracing architecture will be explained independent of a specific implementation.

2.2.2. Principle function, terminology and tracing architecture

Figure 2.3 shows how an exemplary user request in a distributed system is represented in a tree-like structure. The system contains four services (one *front-end service* and three business services *A*, *B* and *C*) that communicate among themselves via Remote Procedure Calls (RPCs). The initial user request *requestX* arrives at the front-end service. From there, a RPC is sent to *service A*, who next sends two RPCs to *service B* and *C*. Both *B* and *C* further process the request before they reply and send a response back to *service A*. *Service A*, who waited temporarily for the two downstream services, now sends its own reply back to the *fronted-service*. From there a response to the initial request is sent back to the user.

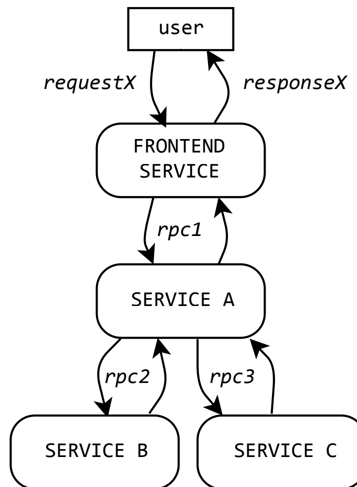


Figure 2.3.: Example trace of a request from the front-end processed by multiple back-end services, adapted from [64]

The described trace is recorded by a distributed tracing system using an annotation based schema, where each application tags all requests with a global identifier in order to link the message back to the originating request [64].

Following the Dapper's terminology [64], the tree nodes from the example above are different *spans*. Each span represents a logical operation as the basic unit of work. Such a casual relationship is manifested in an edge, that materialises between a span and its parent span. Every span has a probabilistically unique 64 bit integer called *span id*.

In Figure 2.4, the above described four spans are presented in a waterfall diagram that depicts the relation between their *trace id*, *span id* and *parent span id* from a temporal perspective.

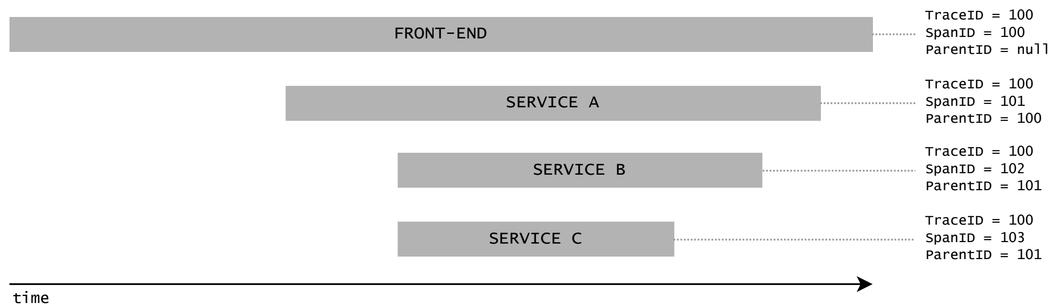


Figure 2.4.: Waterfall diagram showing the four spans and their ids from a temporal perspective

The four spans together build a trace, correlated through the *trace id*. The trace id is determined through the initial span, which is called the root span. Its *trace id* and

span id are equal. Moreover, every span except the parent span, has a parent id, that indicates its calling service. Furthermore, a span has a *span name*, an *start timestamp*, a *duration* and a set of *annotations*, which are timestamped events such as a server send or a server received, key-value annotations (tags) or process ids [18]. Key-value annotations are customisable and can be added by the developer to attach arbitrary content to a span that helps during debugging and provides additional information as for example a server id.

Timestamps received for one span usually originate from two hosts [64]. This fact can be depicted in the sequence diagram of Figure 2.5, that shows the temporal occurrence of the four event timestamps that are generated for span 1 in a request from service A (client) to service B (server).

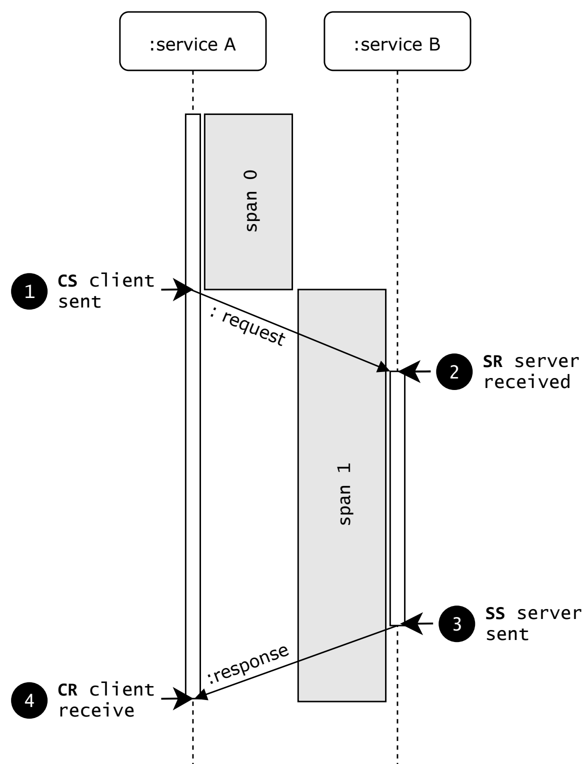


Figure 2.5.: Temporal occurrence of timestamps during a span's life for a request and response between Service A and B, adapted from [8]

See below described the four core annotations for timestamped events [18]:

Client Sent (cs) Indicates the start of a span when the client has initiated a request.

Server Received (sr) The server received the request and starts processing it.
sr - cs = [Network latency + clock jitter]

Server Sent (ss) The time when the server's request is completed and a response is sent back to the client.
ss - sr = [Time the server side needs for processing the request]

Client Received (cr) Time when the client successfully receives the respond from the server. Depicts the end of the span.
cr - cs = [Overall time needed till client receives the request]

The core components for distributed tracing include the instrumented applications, that generate new spans on the one hand, and the distributed tracing system, that collects the data on the other hand [10]. Figure 2.6 depicts this connection between a set of instrumented applications (client and server in this example) and the distributed tracing system.

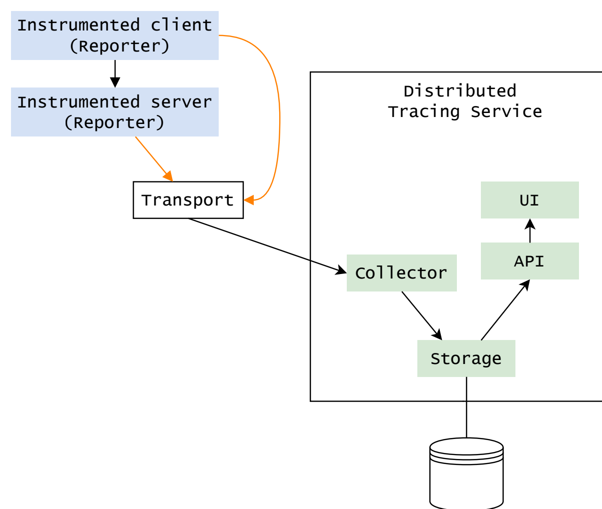


Figure 2.6.: Distributed tracing architecture [18]

Every instrumented application creates a `Tracer` that lives inside the service and records timestamps and other meta-data of occurring operations [18]. See in Figure 2.7 an architecture consisting of two instrumented services, where one is the client and the other a server. The spans are generated in the instrumented client and server and exported via a transport to the distributed tracing system. The distributed tracing system receives the spans via its collector component. It moreover entails a storage, Application Programming Interface (API) and User Interface (UI) component to store, query and visualise the data.

Figure 2.7 depicts a sequence diagram which explains the process of how a HTTP request, arriving at a instrumented service, is recorded. After the request arrives at the service, the `Tracer` first records incoming tags, generates a new *span_id* and stores it to a possibly already existent *trace_id* in-band to the HTTP header, to assure little

overhead. After the request is forwarded and processed, the duration is recorded by the `Tracer` and sent (out-of-band) along with more detailed metadata to the distributed tracing collector component asynchronously within a transport which could be for instance handled via HTTP or buffered via Kafka[1]. The collector component then reads the transports and puts them into a data store (storage component).

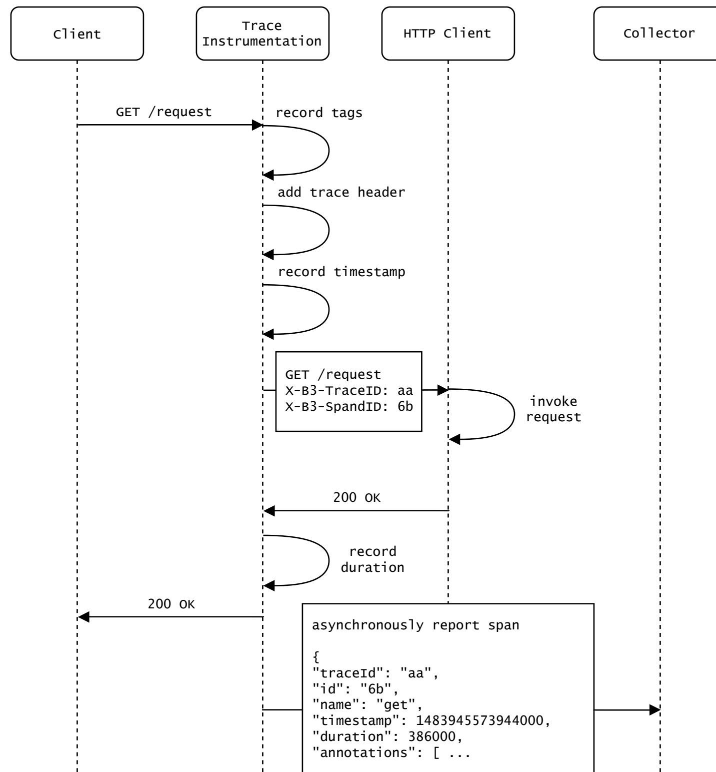


Figure 2.7.: Exemplary trace generation explained in sequence diagram [10]

The specifics of the data generation (data format, types of annotations) are dependent on the the instrumenting tracing library [31]. To overcome this problem, the OpenTracing[2] initiative aims to establish an open standard for application level instrumentations. The community provides vendor neutral APIs for various programming languages and frameworks. The standard is adopted by many distributed tracing tools and makes it therefore possible to uncouple the system from a particular instrumentation. By that, the distributed tracing tools can be switched easily. Furthermore, software artefacts written in different programming languages can continue traces through a common "lingua franca". With that, no knowledge about the underlying language-specific instrumentation library is needed to describe and propagate trace information.

[1]https://kafka.apache.org
[2]https://opentracing.io

2.3. Business process mining

A technique that provides visibility for the business layer is process mining, which is a still very young but already well-established research discipline. Its development in recent years, amplified through the industries' huge interest, yielded in multiple process mining techniques and tools. Process mining can be located between computational intelligence and data mining on the technical side and process modelling and analysis with the origin from business on the other side [1]. It can be described as the interface between traditional business process management (BPM) and data mining. Therefore, process mining should not be seen as a type of data mining, but more like an extension to it [1]. Business process mining is also often used as a term but it only restricts the areas of interest to processes out of a business context. In the remainder of the work, the term process mining will be used, although the business context is apparent through the whole work.

2.3.1. Use cases of business process mining

The applications of process mining are manifold [3]. Its main usage is to automatically discover processes of an organisation without applying any prior modelling in order to capture the *real* occurring system behaviour [1]. Moreover, process mining is generally applied to find process weaknesses and their root causes. Process weaknesses can be classified into process inefficiencies as for instance bottlenecks, rework and changes and into deviations from a to-be process like for example compliance violations. Process mining is used to continuously measure outcomes of process improvement initiatives like fundamental re-engineering as well as incremental improvement [5]. Mature PMTs recommend actions for strategic and operational decisions and can predict costs, risks and delays.

2.3.2. Types of process mining

Van der Aalst [3] distinguishes between four types of process mining that can also be understood as stages since they partly build upon each other:

Discovery *Process discovery* constructs a process model without any previous knowledge and is solely built on the three basic attributes of an event log (*case id*, *activity* and *sorting*) as an input. It is the most used technique in process mining and has several advantages over classical techniques for discovering an organisation's process. It is superior to other process discovery techniques such as interviewing in terms of accuracy because it generalises a model from real-life data and is therefore an abstraction of the reality instead of a desired to-be model that was used during design time.

Conformance The second type of process mining is *conformance checking* where a normative or descriptive to-be model is checked against the as-is reality. The goal is to find deviations between the process model and the event log that might indicate inefficiencies or compliance violations. A distinction is made between global conformance measures and local diagnostics. The first ones interpret the overall conformance of a process (e.g. 38% of all cases are conform to the given model) while the latter one gives detailed insights where a violation occurs (e.g. activity X is followed by activity Y although the model forbids this sequence). Deviations from the to-be model cannot per se be seen as something undesired. If for instance additional activities are being performed to retain a customer, this single process instance might have violations that are desirable. Therefore, the viewpoint of interpreting violations is of importance. In the described scenario, the model could be possibly adapted to achieve a higher fit to reality.

Enhancement As in *conformance checking*, *enhancement* also makes use of a a priori model. The aim of the third type of process mining is to either extend or repair the existent model through data coming from the event log. In *repair*, the process model is altered so that it reflects the reality better then the a priori model, e.g. a sequential order is modelled in parallel since that is how the case is being processed in reality. *Extension* on the other hand extends the model by a new aspect or perspective. An example could be performance indicators such as throughput times that are being calculated and shown in the model.

Operational support The fourth type, *operational support* seeks to not analyse process instances offline but online (or in near real-time). Through historical data from the event log, predictions on how a case will be processed in the future can be made. With these insights the future processing of a case can be influenced.

2.3.3. Description of the α-Algorithm

The α-algorithm is one of the first algorithms that was used to discover *control-flows* while being able to handle concurrency [4]. Therefore, it can be classified to the aforementioned process *discovery* techniques. It comes with limitations such as livelocks and deadlocks [6, 2] but is in general very suitable to understand the core concepts of process mining, since various more advanced algorithms incorporate concepts of the alpha miner [2].

The aim of the α-algorithm is to automatically learn a petri net model from a simple event log L. A simple event log L consist of a multi-set of traces, while a trace is sequence of ordered activity names [4]. Find an example of a simple event log comprising of six cases in total with three unique sequences below:

$$L_1 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^2, \langle a, e, d\rangle]$$

Depart from the event log described in Table 2.1, no timestamps are included in the data of L_1. Only ordering relations, that lead to the different sets of sequence, are given. The indices indicate how often a unique trace is found. Moreover, a *case id* also does not exist explicitly but is implicitly given through having sets of activity traces.

The algorithm scans [4, 5] the event log for four basic ordering relations that occur between two activities.

Direct succession $A >_W B$ holds true iff there is at least one ordering relation where B is directly followed by A. But A and B do not necessarily have to have a dependency relation since A and B can also have a parallel relation.

Causality $A \rightarrow_W B$ holds true iff $A >_W B$ and $B \not>_W A$.

Parallelism $A \parallel_W B$ holds true iff $A >_W B$ and $B >_W A$.

Choice $A \#_W B$ holds true iff $A \not>_W B$ and $B \not>_W A$.

These described relations are used to learn patterns in the event log. So called *footprints* of a Log L_1 can be created that map the relationship of two activities in a matrix. The activities tuples all have one of the four described relationship.

The actual algorithm is performed in eight steps:

1. The event log L is checked on activities that appear in an event log. The activities will correspond to *transitions* in the workflow petri net and will be saved to a set T.
2. Starting activities, meaning those who appear first in a trace are saved in set T_1.
3. Final activities, meaning those who appear as last in a trace are saved in set T_0.
4. The algorithm determines the positions of activities and their connections in the graph. Pairs (A,B) of sets of activities are found, where
   a) every element $a \in A$ and every element $b \in B$ are causally related meaning (i.e. $a \rightarrow_L b$) and,
   b) two arbitrary activities from A (or B) never follow each other or one activity never follows itself i.e. $a_1 \#_L a_2$ and $b_1 \#_L b_2$.
5. From the sum of all possible places, the ones are removed that also appear in other relations. So only maximum pairs of (A, B) remain in the set.
6. The remaining places are now being stored in P_L and a unique source and sink place is added.
7. Now the connections *(arcs)* between the activities are defined. A connection is made between each place of $P(A, B)$ and the elements $a \in A$ of source transitions and each elements $b \in B$ of target transitions. Moreover, arcs are drawn from the

source place to each start transition and from the sink to each end transition.

8. Finally, the petri net is returned with places P_L, transitions T_L and arcs F_L.

2.3.4. Structure of an event log

The basic element needed for process mining is the event log [5]. It is filled with all events that took place during a specified time frame and consists of three minimum attributes: *activity*, *case id* and *sorting* (usually a *timestamp*) [5]. An *event* is a well defined activity of a process that shares the same characteristics in the whole system and refers to a single process instance (*case id*), also described as a case. The third basic attribute that every row of the event log possesses is some sort of *sorting* indicator that gives the events of a case an order in a process sequence. Often timestamps as for example the start or the end of an activity are included in the event log and one of the two is then used to indicate an order. Table 2.1 shows an example of a basic event log.

Table 2.1.: Sample event log

Case id	Activity	Timestamp
273826	Register request	15.08.2017 13:30:43
273826	Examine thoroughly	15.08.2017 14:23:24
273826	Check ticket	17.08.2017 18:12:45
273826	Decide	17.08.2017 23:19:03
273826	Reject request	19.08.2017 17:57:12
192893	Register request	03.08.2017 08:17:37
192893	Check ticket	04.08.2017 06:27:24
192893	Examine thoroughly	04.08.2017 15:18:58
192893	Decide	05.08.2017 19:42:25
192893	Examine casually	09.08.2017 23:38:42
192893	Pay compensation	10.08.2017 01:37:31
283902	Register request	05.06.2017 17:59:44
283902	Check ticket	06.06.2017 14:45:23

Event logs can be further extended by an arbitrary amount of attributes [2]. Often a resource column is added, that indicates which person or what user type has conducted the activity. Through the ratio of activities performed by an automated user (e.g. batch user) compared to the amount of all activities an automation rate per process activity can be calculated easily.

In situations were no case table is linked to the event log, other additional attributes typically contain case specific data such as volumes, costs, product or customer specific information. In terms of redundancy, this is not a desired approach for modelling the data but sometimes applied to keep the data model compact.

2.3.5. Strategies of data acquisition

Data acquisition in business process mining is one of the most challenging tasks [3, 2]. The minimum requirements for generating a process visualisation, as stated before, are a *case identifier*, an *activity name* and an *ordering attribute*. The aforementioned techniques of process mining (see 2.3.2) are based on an event log, where each activity refers to exactly one case. The three attributes are often only available jointly in so called process Process-Aware Information Systems (PAIS) such as Enterprise Resource Planning (ERP), Customer Relationship Management (CRM) or Workflow Management System (WFM) where audit trails are often produced [5]. In such systems, the event log can be directly employed as an input for process mining. In the *Process Mining Manifesto* [1], van der Aalst et. al describe how event data has to be regarded as "first-class citizens" meaning that information systems should be able to write event logs of high quality, since the quality of the process mining results heavily depends on the quality of the input data.

However, in many of the systems events have to be correlated manually from traditional databases [3]. The concept of a case and corresponding activities in such systems "only exists implicitly" [3]. Therefore, we focus on those settings in the following, where no event logging functionality is implemented natively and real process awareness does not exist. In such cases, the input for the event log can be dispersed in different information system, possibly spanning across organisations, being present in various unstructured formats in complex data models [66]. If data warehouses exist, they might be a promising source for process mining as they include data from different operational sources and transform them into a consistent format [5]. Unfortunately, they often lack relevant information that is required for an end-to-end view of a process. First, due to the warehouse's focus on a certain scope, e.g. vendor information, which excludes possibly necessary information on customers, for instance, in a Order-to-Cash process. Second, data warehouses, with their focus on Online Analytical Processing (OLAP) of multi-dimensional data models, do not necessarily include the appropriate data that relates to process instances and their execution [5]. For instance, meta data such as timestamps, which are crucial for the activity generation, are often not included in such systems. Furthermore only current states are recorded and no historical data about when a certain entry was created, altered or deleted [3]. However, this type information would be useful to understand the history of a case and to generate corresponding change activities.

2.3.6. Transforming the multi-dimensional reality into a flat event log

The techniques of process mining require a 'flat' event log (described in the previous subsection) as an input, meaning that every activity refers to a single case [3, 1]. This assumption is also the underlying concept in common process modelling languages such as Business Process Model and Notation (BPMN), Business Process Execution Language (BPEL) or activity diagrams in Unified Modeling Language (UML).

In reality, a process would not be flat: For instance information for the event creation in an order process could come from multiple tables. Imagine an *order* table acts as a header document with a unique *order id* while possibly multiple *order lines* per *order* exist that are stored together with an unique *orderline id* in the *orderline* table [3] . Furthermore, each *order* can have 0..1 *deliveries* and each *delivery* could take multiple (0..*) *attempts* to be delivered. Now, the perspective from which the order process should be analysed needs to be defined, either on the *order* (header) or *orderline* (item) level [3]. Furthermore, one could also analyse *deliveries*, and define the *delivery id* as the *case id*. However, timestamps from different process activities could come from different tables. E.g. the payment date of an invoice would sit in the *order* table while the backorder date of an item would come from the *orderline* table.

To create the event log, this multidimensional data model needs to be flatten into a single file. Focusing on one level, either *order* or *orderline* would cause the loss of timestamps coming from the table that is not seen as a case and therefore the basis for activity creation [1]. In the described scenario, a solution could be to concatenate the order number with the item number and thus create a global *case id* with the possibility to join the tables to receive the required timestamps. The perspective would be set on a order item level. The availability of consistent links between different tables and levels need to be taken into consideration when choosing the perspective and creating the event log for process mining. The present analysis questions should indicate what process lifecycle and type of *case id* needs to be chosen.

2.3.7. Challenges in extracting data and generating event logs

Van der Aalst [5] maps out multiple challenges that occur during the data extracting and event log generation which are discussed below.

Correlation An important task that has to be conducted in order to link different activities to a single process instance (i.e. cases) is the event correlation. This correlation is applied through a shared identifier, often called *case id*, which could be for instance a document number of a purchase order in a purchasing process or the order number in a sales process of a standard ERP system. This id must span through all source systems from which activities are generated. In a scenario where certain parts of the processes

are handled through a different system, e.g. a sub-process in a supply chain process, a case has to be correlateable through the same id or a concatenation that results in a system-wide, cross-system or even cross-enterprises id. Ferreira and Gillblad [25] present a probabilistic approach to find process models in event data where a *case id* is missing. Through an iterative expectation–maximization algorithm they were able to derive a general process model and also correlate unlabelled event logs to a certain process instance.

Timestamps To sort events in their order of occurrence either a sorting attribute or timestamp is necessary. When multiple systems are used as a source for the event log generation, timestamps might not be accurate since clocks are probably not synchronised. This can become a problem if accuracy is of high significance and execution times of events appear within a very short time. Unsynchronized clocks could then lead to a wrong order of events. It also might occur that some events are executed concurrently. In that case a *sorting* can be applied that acts as a 'tie breaker' in case that two or more events share the same timestamp. This *sorting* is a guessed order of events, following a to-be sequence, which needs to defined with the required domain knowledge. Another issue timestamp accuracy entails, is that some systems only log dates and not times. In that case one has to work with partial ordering or again predefine a sorting with the required knowledge.

Snapshots A further challenge or characteristic that one has to address is that every data extract from a source system is only a snapshot and captures process instances that 1) might have started before the extraction period and/or 2) are not finished at the time of the extraction. In the first scenario, activities, that belong to a case where a starting activity is missing could just be completely deleted from the event log. In the latter case, one could delete the activities but it could also present a desired modelling, for instance if open cases should be analysed e.g. open payments in a accounts receivable process.

Scoping Process start and end is something that has to be defined and that depends on the perspective and underlying questions. Again domain knowledge is needed to choose the appropriate data and scope the process according to the needs. Depending on the analysis question, only certain parts of a process for instance (e.g. supply chain sub-process) would be in model.

Granularity Next to the scoping, that defines the process start and end also granularity, meaning the level of detail to which activities are generated, plays an important role during the shaping of the activity log. Furthermore, different layers of granularity in event logging systems strengthen the need to limit activities to a level that is still

relevant for a user. Too many activities that represent the process in a granular way might distract the user and lower the chance of generating insights from the process [30].

3. Description of the system under survey

The sample architecture (i.e. SUS) was built using common patterns and best practices to simulate a most realistic setting that could be found in the real world and especially in the context of the TUM LLCM project. This chapter describes the sample architecture and elaborates upon which design aspects it was built on and why certain technologies and frameworks were chosen.

The first part of the chapter gives an overview of the system architecture including its system components. Moreover, the available user activities are listed and it is exemplarily described, how a user activity is processed by the system. Moreover, a definition for *user* and *system activities* is given, that will be used throughout this work.

In the second section, the implemented business functionalities are described in detailed. Therein, the functional scope of each business microservice is presented, which includes the service's broader role in the system together with its provided endpoints.

The third section provides a description of the software architecture. This includes the service communication architecture, a description of the implemented infrastructure microservices as well as an exemplary setup of one business microservice.

3.1. General system architecture

Within the course of the LLCM project, a mobility platform was developed as a reference architecture. Since this thesis leans on the project, a similar functional context as well as an architectural approach was chosen to develop the prerequisite SUS for this work.

The SUS mocks a car sharing platform that provides typical functionalists as for instance searching and booking of available cars, calculating routes, booking packages, reporting issues and more.

To resemble the reference architecture of the LLCM project also from a technical perspective, the same framework, namely Spring Cloud[1], was employed. Spring Cloud is often used for building microservice architectures, since it gives the system engineers a brought tool set for building distributed applications relatively fast. These tools help to tackle the challenges of developing applications in distributed systems (as described in the *8 Fallacies of Distributed Computing* [58]). Moreover, multiple libraries exist for

[1] http://projects.spring.io/spring-cloud/

Spring Cloud, many of them from the Netflix OSS stack[2], that provide implementations for modern patterns of microservice architectures like configurations management, gateways, load balancing, service discovery and many more, that are also applied in the SUS.

The architecture, depicted in Figure 3.1 consists of three types of services, that are clustered according to the following taxonomy [55]: First, a front-end service, that renders the UI for performing user requests. Second, six business services, that provide the actual business functionality and support business operations. And third, three infrastructure microservices, that provide non-functional tasks and help to apply architectural patterns and best practices that are typically found in a microservice architecture.

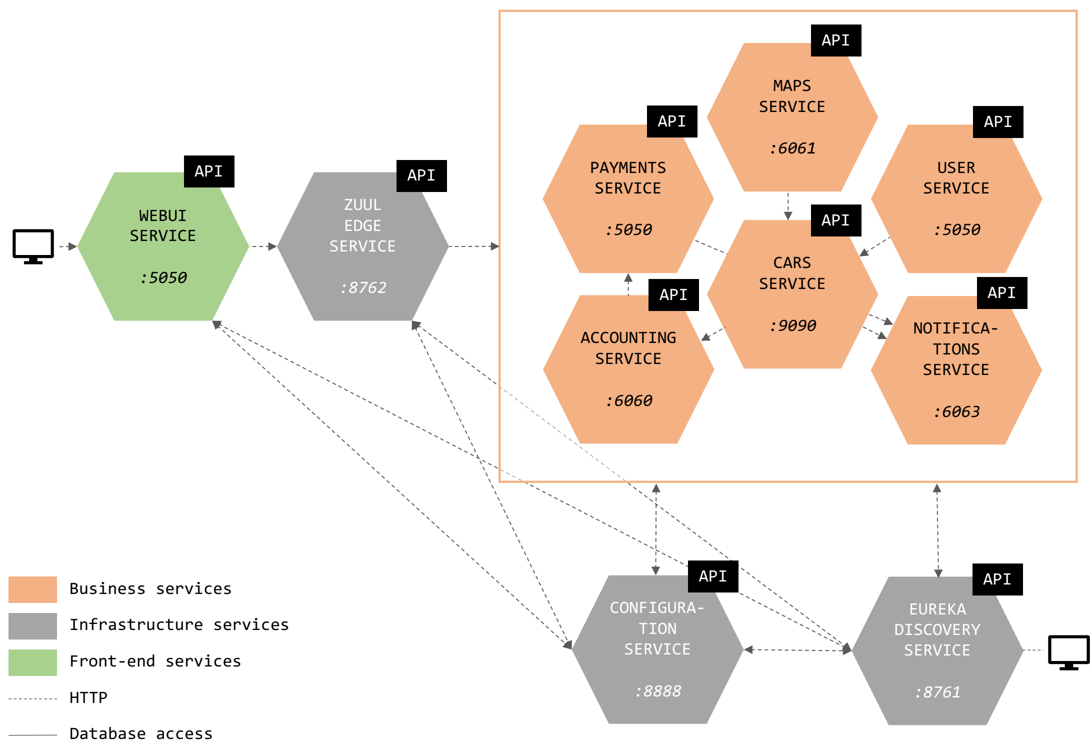


Figure 3.1.: Architectural overview of SUS

Business functionality is exposed via an *edge service*, that provides an external API to multiple clients and hides the business services' complexity behind the proxy. In the described architecture, only one client, namely the web front-end (rendered through the *webui service*), is accessing resources provided by the *edge service*. After an incoming user request, the *edge service* initially calls one of the six business services. From there, further

[2]https://cloud.spring.io/spring-cloud-netflix/

interaction between the business services is maintained via a point-to-point interaction style. The approach of *"dumb pipes and smart endpoints"* [28] is applied, where every service entails its own business logic, can request resources from downstream services if required and processes received requests at the controller according to defined business logic in the service.

Moreover, all services are connected with the *configurations service*, from where each service requests its configurations during startup. A third technical service is the *Eureka discovery service*, to which all services register initially and afterwards constantly send status information. Through *Eureka*, service addresses do not have to be hard-coded in each application, since it takes care of managing the service's physical locations.

Trough an arbitrary front-end, the user can perform multiple *user activities* like for instance the *search for an available car* or a *car booking*. A set of *user activities* represents a user's click path in the front-end and is defined as the *business process*. Each user activity is followed by a composition of synchronously invoked inter-service requests, where each is defined as *system activity*. A sequence of system activities, is defined as a *business transaction*.

In the implemented web front-end, a user can perform ten activities in total, that correspond to the ten endpoints the *edge service* provides. A dependencies matrix (see Table 3.1) lists all user activities together with the in the request involved business services. The *End car rental* activity for instance invokes the *cars*, *accounting*, *payment* as well as *notification* service in order to deliver the user the requested functionality.

Table 3.1.: Dependency matrix between user activities and microservice involvement

	Maps service	**Accounting service**	**Notifications service**	**Payments service**	**User service**	**Cars service**
List available cars	x					x
Reserve car			x			x
Book car		x				x
Unlock car						x
End car rental		x	x	x		x
Show balance		x				
Book package		x	x	x		
Show driving history	x				x	x
Report issue			x			x
Find route	x					

An example of how a front-end request (i.e. user activity) is routed using synchronous communication between the services of the architecture can be found in Figure 3.2 below, where a business transaction triggered by the *End car rental* activity.

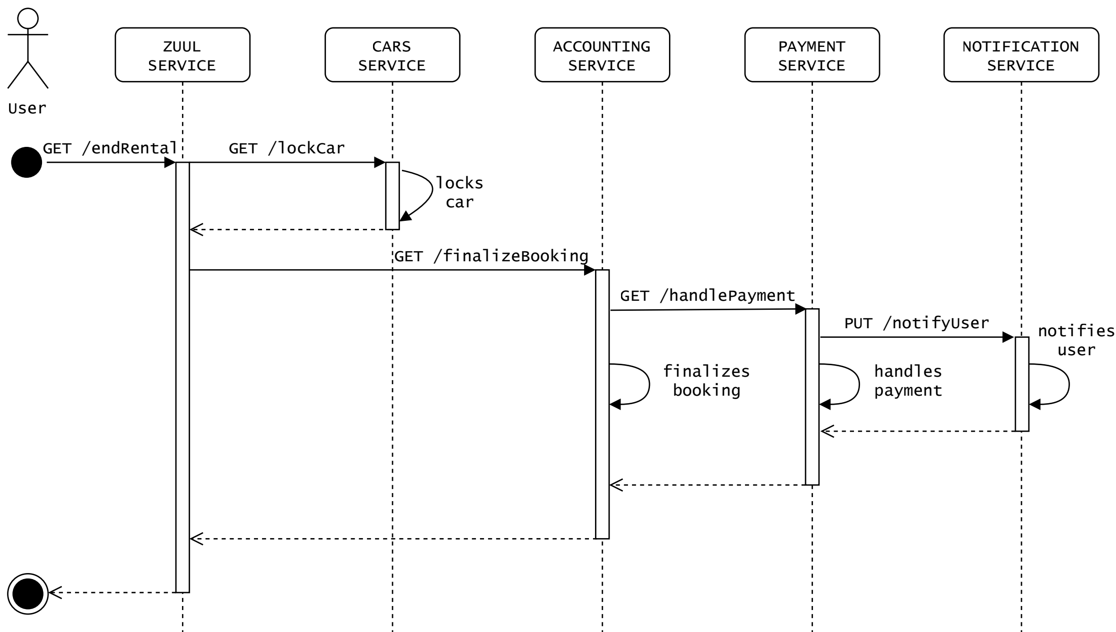


Figure 3.2.: Call flow for */endRental* request at the front-end (i.e. *End car rental* user activity)

The `/endRental` request arrives at the *zuul service* from where the *cars service* is called via its `/lockCar` endpoint. The service processes the request which leads in further consequence to the actual locking of the car followed by a response message. Parallel to that, the *accounting service* is invoked via `GET /finalizeBooking` in order to end the car booking, calculate a fee and further process it from an accounting perspective. From the accounting service in turn, a payment process is triggered via `GET /handlePayment` to the *payment service*. After the payment has been processed successfully, a `GET /notifyUser` request is sent to the *notification service* from where the user is eventually notified about the status of the transaction via possibly multiple channels. Call flows for the remaining nine user activities can be found in the appendix (see section A.1).

3.2. Description of the business functionalities

In the following, the business functionalities of the SUS are described by sketching the roles of the business microservices including their provided endpoints.

As described above, the system provides functionality in form of ten user activities, that can be performed in an arbitrary sequence. In a real application, not every sequence of activities would be executable (e.g. an *End car rental* would not be allowed before a *Book car* activity was performed). A sequence of performed activities in an user session yields in a *user click path*, which is defined as the *business process*. In general, the presented approach is not limited to analysing user focused processes. See subsection 5.1.5 for a

more extensive discussion on process perspectives.

Webui service

The *webui service* renders a web front-end (see Figure 3.3) where all the implemented user activities as for instance a *Search for available cars*, a *Book package* or *End rental* can be performed. With that, one can simulate user clicks that would, in an operational setting, originate from real users either via a real web application, mobile app or any other front-end. The resulting click paths will be analysed later with the techniques of process mining by correlating the distributed tracing data.

All provided user activities can be triggered through the different buttons (1) of the user interface that is depicted in Figure 3.3. After performing a click on a user activity, a console output (2) logs, what services are being called after triggering a certain request. The output is not generated through any instrumentation but represents a composition of return attributes of each called endpoint. Since every endpoint returns its service name, a sequence of called services is being displayed.

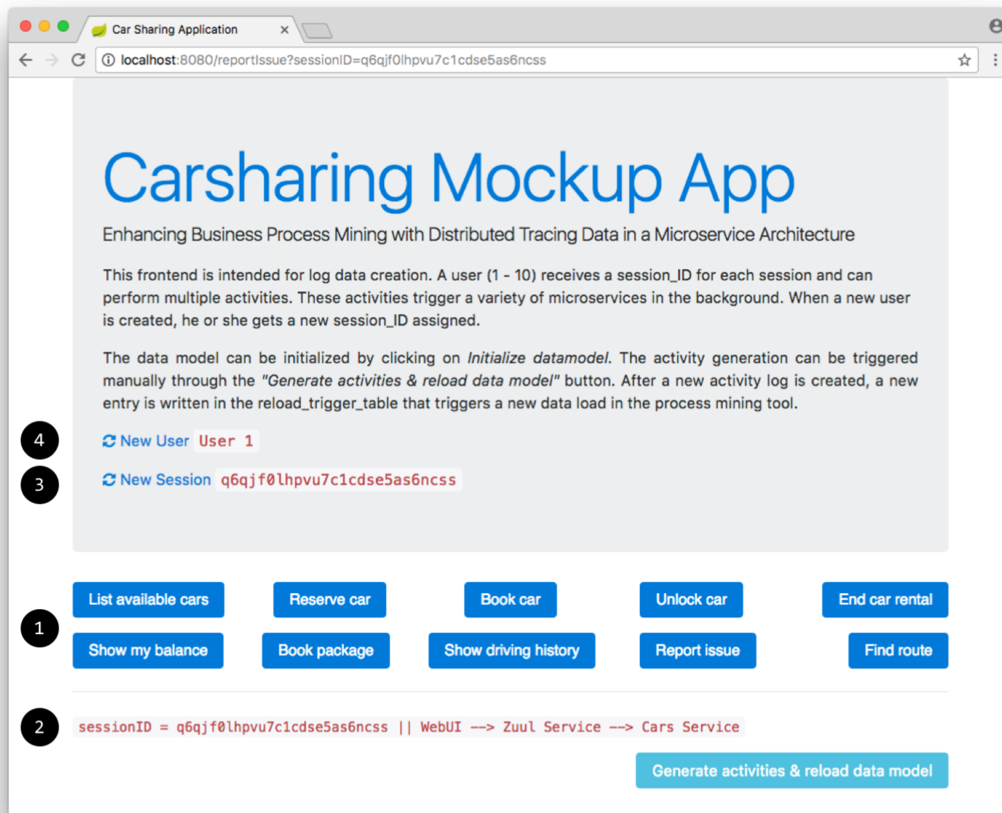


Figure 3.3.: Web front-end rendered by the *webui service*

User clicks and therefore user activities occur during a user session in the front-end. A session is usually generated by the web service in form of a session cookie that is stored in the user's browser to recognise a user between page visits. A session is unique and entails an id.

The thesis aims to discover user click paths from the SUSs front-end. This is achieved through transforming tracing data that is recorded by the distributed tracing instrumentation. With that, user activities, consisting of an activity name and a timestamp, can be generated. In order to analyse click paths of an unique user and session, moreover, some sort of *case id* is required, that correlates the activities to a single process instance and gives the activities a shared context. The before mentioned availability of a session id in most web applications and the use case of analysing user sessions lead to choosing the *session id* as the *case id*. Alternatives to the *session id* would change the context which will be described more extensively in subsection 5.1.5.

A session's durability would be in general defined individually by the service. In

the SUS one can generate a new probabilistic unique *session id* manually, through a Representational State Transfer (REST) call at the *webui service* (3). Furthermore, a random user (4) can be linked to a unique session. By requesting one, the *webui service* generates a random *user id* (User 1 - 10) and persists it together with the *session id* as well as the device type in a *sessions* database table. Creating a new user automatically creates a new session which does not hold true vice versa. The relationship between a user and a session is a one-to-many relationship.

Alternatively to a random user generation, the feature could have been implemented with a regular user login where for a predefined group of users, unique sessions would have been generated automatically after login. For the purpose of generating click paths for users within a reasonable amount of time, a random user generation linked to a virtual session generation was preferred over a login functionality.

Table 3.2 shows all available endpoints, including their description in which user activity the endpoint is involved. For the *webui service*, endpoints can be translated one-to-one to user activities. The ten endpoints listed below also present the functional scope of the SUS.

Table 3.2.: *Webui service* endpoints and their usage in user activities

Endpoint	**Description**	**User activity involvement**
GET /getCars	Return all available cars and displays them on a map.	*List available cars*
GET /reserveCar	Lets the user reserve a car.	*Reserve car*
GET /bookCar	Lets the user book a car.	*Book car*
GET /openCar	Lets the user unlock a car.	*Unlock car*
GET /endRental	Lets the user end a car rental.	*End car rental*
GET /showBalance	Shows the user's balance.	*Show balance*
GET /bookPackage	Books a driving or parking package.	*Book package*
GET /showHistory	Shows the history of the last routes.	*Show driving history*
GET /reportIssue	Lets the user report a malfunction.	*Report issue*
GET /findRoute	Lets the user find a route.	*Find route*

Maps service

The *maps service* is mainly responsible for rendering a map in different contexts and displaying additional information e.g on layers on the map. Moreover, route calculations, address searches and display available cars within a defined service area are further functionalities that the *map service* entails. The service provides three endpoints (see Table 3.3) that are used for the *List available cars*, *Show driving history* and *Find route* user activities.

Table 3.3.: *Maps service* endpoints and their usage in user activities

Endpoint	Description	User activity involvement
GET /generateMap	Renders a map of available cars.	*List available cars*
GET /getHistoricalRoutes	Displays historical routes on a map.	*Show driving history*
GET /getRoutes	Calculates available routes on a map.	*Find route*

Accounting service

The *accounting service* provides all functionality that is needed for billing services offered by the platform. It provides endpoints for initialising and finalising a booking, querying the user's balance and handling a package booking (see Table 3.4).

Table 3.4.: Accounting service endpoints and their usage in user activities

Endpoint	Description	User activity involvement
GET /initilizeBooking	Prepares and initialises a car booking.	*Book car*
GET /finalizeBooking	Finishes a car booking.	*End car rental*
GET /getBalance	Queries the database for the user's current balance.	*Show balance*
GET /newPackage	Initiates the booking of a package.	*Book package*

Notifications service

The *notifications service* provides the user with different types of notifications. These include push notifications for smartphones, inbox messages for various front-ends as well as e-mail and SMS notifications. The *notification service* only provides one single

endpoint `/notifyUser` that is called by the activities *Reserve car*, *End car rental*, *Report issue*, and *Book package* and is typically called as the last endpoint (see Table 3.5).

Table 3.5.: *Notifications service* endpoints and their usage in user activities

Endpoint	Description	User activity involvement
GET /notifyUser	Notifies user via different channels.	*Reserve Car* *End car rental* *Report issue* *Book package*

Payments service

The *payments service* provides functionality for processing payments internally and passing transactions to external payment providers. Both activities *End car rental* as well as *Book package* use the service's single `/handlePayment` endpoint (see Table 3.6).

Table 3.6.: *Payments service* endpoints and their usage in user activities

Endpoint	Description	User activity involvement
GET /handlePayment	Processes a payment internally or externally.	*End car rental* *Book package*

User service

The *user service* handles user identities through providing login functionality and manages both master as well transition user data. In the SUS, it only provides a `/getUserHistory` endpoint that is called to retrieve a history of the user's past routes (see Table 3.7).

Table 3.7.: *User service* endpoints and their usage in user activities

Endpoint	Description	User activity involvement
GET /getUserHistory	Query historical trips that are attached to the user.	*Show driving history*

Cars service

The *cars service* keeps track of the different states of a car like location, availability, reservation or maintenance. It also provides access to master data as id, car name, car type and more. It is a substantial service providing endpoints to seven user activities in total (see Table 3.8).

Table 3.8.: *Cars service* endpoints and their usage in user activities

Endpoint	Description	User activity involvement
GET /getAvailableCars	Query a list of all available cars.	*List available cars*
GET /allocateCar	Reserve a car.	*Reserve car*
GET /handleCarBooking	Start booking of a car.	*Book car*
GET /unlockCar	Unlock a car.	*Unlock car*
GET /lockCar	Lock a car.	*End car rental*
GET /getCarHistory	Query driving history of a specific car.	*Show driving history*
GET /createIssue	Create an issue and attach it to a car.	*Report issue*

3.3. Description of the software architecture

In the following, a general description of the system's architecture is given. First, the communication between the services is characterised. Thereafter, the role of the three infrastructure microservices is described. Finally, an exemplary setup of one business microservice is given, that shows in what manner the application and the controller is built for all business microservices.

3.3.1. Service communication architecture

Subsection 2.1.2 has already given an overview of patterns present for microservice composition and the execution of business logic across services. Thereby, the two main distinctive patterns, orchestration and choreography, have been discussed including the benefits and drawbacks of each. The service composition architecture applied in the SUS conforms to a point-to-point pattern. In point-to-point, every service contains its own orchestration logic in the endpoints and services communicate directly with each other [20, 65].

The in section 3.1 demonstrated call flow for a user's `/endCar` request shows the

synchronous request/response inter-service communication. In the SUS, every service has its own logic implemented and decides what additional resources need to be requested in order to return a response back to the its upstream service. The pattern is applied for all user requests and inter-service call sequences appearing in the SUS.

The following rationale led to the implementation of a point-to-point communication architecture in the SUS: First, the point-to-point is a often applied and popular pattern, which is due to its intuitive applicability [20, 67]. However, the pattern comes with a lot of drawbacks that create complexity. This emerging complexity makes the architectural style a suitable object of investigation for applying techniques of distributed tracing and process mining on it. Second, one of the goals of this work is to discover process flows from information systems that are per default process-unaware. The presented SUS, which uses elements of orchestration, does not have a notion of process awareness implemented. Architectures with choreography patterns also do not fall into the definition of a process aware system, but they do create events. These events can give insights about the state of a process and could theoretically be correlated to user or business activities. While this could be a further promising source of activity generation for process mining, this work focuses on architectures written in an orchestration pattern that do not write any events. The only source for the generation of high level event logs is distributed tracing data from an instrumented service.

3.3.2. Infrastructure microservices

Three infrastructure microservices are implemented in the SUS. First, a *configuration service*, that provides configurations in a consistent and maintainable manner to all other microservices. Second, a *discovery service*, to which all other microservices register and that keeps track of locations and different states. And last, an *edge service*, that works like a micro-proxy which forwards requests coming from different clients to the corresponding services.

Configuration service Microservices require a much higher demand for configuration management compared to monolithic-styled architectures [38, 46]. In such, there is also a need for pushing new configurations across multiple environments for different versions during runtime. A new configuration can be pulled by a service through its `/refresh` endpoint. In the sample architecture, a *Spring Cloud Config*[4] service was implemented that centralises configuration files in a local repository for each microservice and provides them in a maintainable and consistent manner.

Discovery service A *discovery service* keeps track of the service's different network locations and states. Such a component is needed in environments where the locations of services change and are dynamically assigned e.g. in cloud settings where replications of instances are dynamically booted [55]. The used component for the discovery service is *Eureka*[3] which is also part of the Netflix OSS stack. It

implements a client-side discovery pattern. Within this pattern, clients register during startup to the discovery service. During runtime they periodically send their status through heartbeats till they are deregistered after termination. Before requesting a service, the client queries the discovery service to obtain the locations of a service or its replications. In this pattern the client also chooses between the different instances that are replicated through Netflix *Ribbon* by applying a load balancing algorithm.

Edge service An *edge service* can be described as the "front door" [17] for all devices that are accessing services of a system's back-end. Thereby, a unified interface for consumers is created, that proxies request to multiple back-end services. In the sample architecture, Netlix's *Zuul*[3] is used as an edge service.

One force for using an *edge service* is, that the APIs of services often provide high granularity and clients would have to retrieve data from multiple services for receiving a desired single user request [46]. Another challenge is that the content of a response might vary from consumer to consumer [49]. A mobile client might expect a different response than a web application. He might for instance receive a different composition compared to a web application. This is accomplished through rules that can be defined in form of filters in the *edge service*. By applying routing logic individually for each type of client various APIs are exposed. Lastly, the partitioning of services might change over time while interfaces to the front-end should remain stable [47]. Complexity in general should be hidden from the client [46].

3.3.3. Exemplary setup of a business microservice

The six business microservices are all built in a similar manner. Therefore, only one, namely the *accounting service*, will be extensively described by its core elements. It is suitable for an exemplary description since it is used as an intermediate as well as an last called service of a user request.

All services are build as Spring Boot applications using different modules from Spring Cloud in its Dalston Service Release 1 (`Dalston.SR1`). The dependencies `xml`spring-boot-starter-web and `spring-boot-starter-test` are thereby used by each application. The main application is implemented as a regular spring boot application, indicated through the `@SpringBootApplication` annotation (Listing 3.2, l. 2).

The aforementioned *configuration service* is used by every application for receiving its own configurations. Therefore, in the application's local configurations only the own application name for the look-up at the server and the URI of the *configurations service* are defined (see Listing 3.1).

[3]https://cloud.spring.io/spring-cloud-netflix/

Listing 3.1.: The application's local bootstrap.properties

```
spring.application.name=accounting-service
spring.cloud.config.uri=http://localhost:8888
```

The actual configurations are stored in the *configurations service* and pulled by the respective service on start-up. To enable a central configuration management, the dependencies `spring-cloud-starter-config` and `spring-boot-starter-actuator` have to be added in the Project Object Model (POM).

In lines 5 - 9 (Listing 3.2), a default REST template method is defined, that will be used by the controller.

Listing 3.2.: Class definition for the accounting application

```
@EnableDiscoveryClient
@SpringBootApplication
public class AccountingApplication {

    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

    public static void main(String[] args) {
        SpringApplication.run(AccountingApplication.class, args);
    }

}
```

The architecture also makes use of the service registry and discovery component Netflix *Eureka* (as described in subsection 3.3.2), which is implemented as an own microservice. To communicate with the discovery service, all business services are configured to act as a discovery clients, indicated through the `@EnableDiscoveryClient` annotation.

All services further apply Netflix *Ribbon* for client side load balancing. An integration with Eureka is provided in the library. To configure a load balanced REST template, the bean is annotated with a `@LoadBalanced` qualifier (see Listing 3.2, l. 6).

Listing 3.3.: Endpoints definition in the accounting controller

```
@RestController
public class AccountingController {

    private final RestTemplate restTemplate;
    private static final Logger log =
     ↪  LoggerFactory.getLogger(AccountingApplication.class.getName());
    private static final String app_name = "Accounting Service";

    public AccountingController(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

     @RequestMapping("/getBalance")
    public String getBalance() throws InterruptedException {
        String msg = app_name;
        // query database for user's balance
        Thread.sleep(300);
        log.info(msg);
        return msg;
    }

    @RequestMapping("/finalizeBooking")
    public String finalizeBooking() throws InterruptedException {
        String msg = app_name;
        // process booking internally
        Thread.sleep(200);
        msg += " --> " +
         ↪  restTemplate.getForObject("http://payments-service/handlePayment",
         ↪  String.class);
        log.info(msg);
        return msg;
    }

    // further endpoint definitions ...

}
```

Listing 3.3 shows an excerpt of the *accounting service* controller, that handles incoming HTTP requests. The controller is established through a `@RestController` annotation (l. 1). The class contains three variable definitions. The first is a `RestTemplate` that is

necessary for synchronous client-side http requests (l. 4), the second is a `Logger` object for printing logs in the console (l. 5) and the last one is a `String` object for returning the application name (l. 6). Furthermore, a constructor is defined that references an instance of the RestTemplate.

The controller also defines the two endpoints `/getBalance` and `/finalizeBooking`. Web requests are mapped to a method through the `@RequestMapping` annotation. The `/getBalance` endpoint would only query its database to provide the requested balance for a user. In the sample architecture this functionality is not implemented. However, a `Thread.sleep()` is implemented that pauses the thread in order to simulate a database's request latency. The method and therefore the request only returns the name of the application. The `/finalizeBooking` endpoint by contrast triggers further inter-process communication by calling its downstream payments-service for initialising a transaction. This synchronous communication is performed through the above defined RestTemplate with the method `getForObject` that stands for a HTTP GET request. The received object (if any) is rendered to a String class and appended to the *msg* String defined above. The endpoint returns the application name as a message.

4. Description of the automated process discovery prototype

The development of the prototype was carried out in six steps (see Figure 4.1). First, an initial architecture design had to be developed that identifies how the existent SUS needs to be extended by additional components. In a second step, one tool for collecting distributed tracing data and another for the process mining part was selected. As a third step, the existent SUS was instrumented with a distributed tracing library. In a followed fourth step, a persistence strategy was developed for the tracing data. Moreover, a data platform including a data model was established on which the log generation algorithm could be executed. With theses prerequisites, the actual log generation algorithm was developed and implemented in an own microservice in the fifths step. Finally, the PMT was configured to load the previous created tables. The relations and roles of the tables were setup in a data model. On top of the data model, four different analysis were created. Each answers unique analysis questions and thereby proofs the applicability of the presented approach.

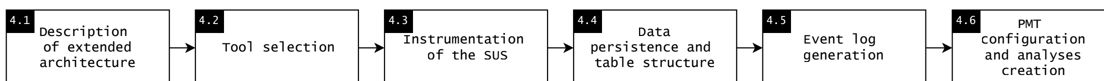


Figure 4.1.: Overview of the prototype development process

4.1. Extended system under survey architecture

In the first step, the six business services of the prerequisite architecture were instrumented with a distributed tracing library (see section 4.3), that creates the trace data. Afterwards, the SUS was extended by four additional components. See Figure 4.2 that depicts the extended sample architecture.

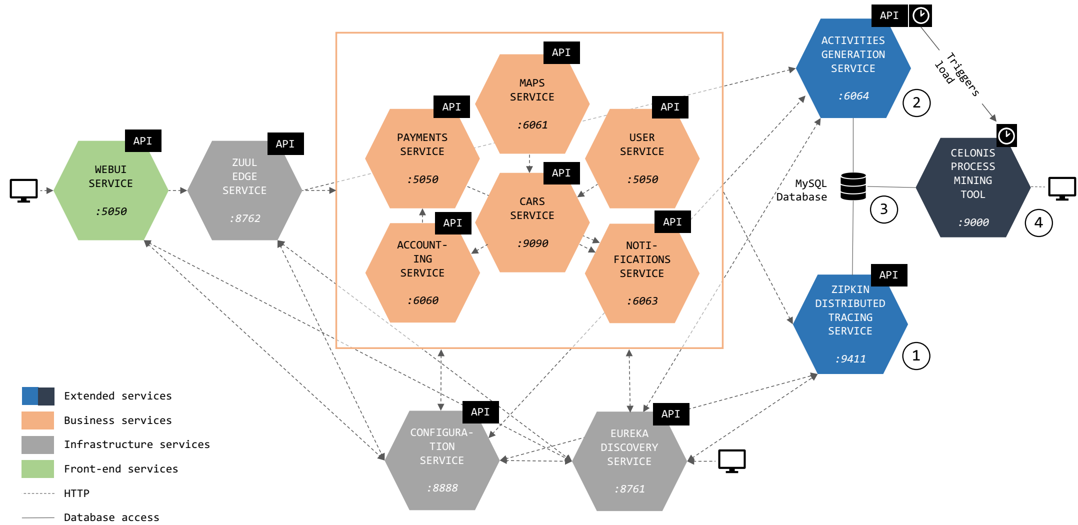


Figure 4.2.: Architectural setup for the data processing

The first component added is the *Zipkin distributed tracing service* (1), that collects the trace data created at and sent from each instrumented business service. Besides the collector, the distributed tracing service also entails a storage and API element and moreover provides an UI for visualising the traces. The typical architecture of the distributed tracing system has already been depicted in Figure 2.6.

For the event log generation, an additional *activities generation service* (2) was build that transforms the raw low level events recorded through *Zipkin* to a high level event log existing of system and user activities (see section 4.5). The component entails the activity generation algorithm in form of Structured Query Language (SQL) scripts and manages its execution.

The actual correlation is performed on a *MySQL* database, that was added as a third component (3). The data base is the persistence layer for two described components above: It stores the original trace data from *Zipkin* and moreover serves as an execution and persistence platform for the *activities generation service*.

Consequently, also the fourth component, the PMT (4), has been connected to the database. The PMT, which is deployed as a web application, calculates the process model and provides further functionality, similar to a Business Intelligence (BI) tool for creating process-related analysis. The PMT uses the database as a primary data source for importing tables that are necessary for the process mining. In the tool, the relation between the imported tables has to be mapped in a data model. The data model is the foundation of every analysis and will be further described in subsection 4.6.2.

Through the web front-end, it is possible to trigger the whole workflow for generating the event log as well as for reloading the PMT's data model in order to update the analysis. This workflow, which is also configured to run automated, will be more thoroughly described in subsection 4.6.1.

4.2. Distributed tracing and process mining tool selection

Choosing distributed tracing and process mining as core techniques for the proposed approach, a tool for each had to be selected first.

For the distributed tracing part, *Zipkin* was chosen as a tool for collecting and visualising trace data. *Zipkin* is a java-based distributed tracing system that is based on the architecture described by Google [64]. It was initially developed by Twitter in 2010 and open sourced in 2012 with the official name *OpenZipkin*. However, the name *Zipkin* will be used to refer to the *OpenZipkin* project in the remainder of this work. Since then, the project matured and is supported by an active developer community. *Zipkin* was chosen over other tools for various reasons. First, it is the biggest open-source project for distributed tracing in terms of active users and developers, therefore free to use, well documented and supported through various channels. Second, it is widely adopted by technologically leading companies [44], which manifests its applicability for the usage in a production environment. Third, it implements the *OpenTracing* standard, already described in subsection 2.2.2. The log generation, that will be described in section 4.5, is dependent on the implementation regards to the data format in which the tracing data is stored. By choosing a well-adopted standard, the scripts developed for this proof-of-concept remain valid and are agnostic to the instrumentation used. Lastly, and most importantly, by choosing *Zipkin* together with the *OpenTracing* standard, an existent architecture can be instrumented with minimal code change which will be shown in the next section.

For the process mining part, the software *Celonis Process Mining* was chosen. *Celonis* is a web-based enterprise solution that provides real-time process discovery techniques, connects to operational source systems (e.g ERP, CRM) and is capable of handling large volumes of data. Since distributed tracing creates high data volumes, this was an important aspect for choosing *Celonis* over others tools. Moreover, the tool does not only provide means for generation process models, it combines it with OLAP functionalities, known from traditional BI. Thus, it is possible to add different component types to an analysis and enhance the process visualisation with charts, tables and selection elements [21]. *Celonis* provides an own Process Query Language (PQL) to define complex KPIs in the dashboards and stores process data along with additional data in a process cube. The described functionalities are required for the *proof-of-concept* prototype. Besides the functional fit, *Celonis* is widely adopted in industry and can be used within an academic license for free.

4.3. Instrumentation of the system under survey

This section first describes how a *Zipkin* microservice, that collects, persists and visualises traces was created. In the second part, it is explained how the services of the sample architecture were instrumented using Spring Cloud Sleuth[1], which is an adaptation of the *Zipkin's* instrumentation library for Spring Cloud applications. In the third part it is explained, what additional modifications were applied for the *webui service*, which generates the front-end and is thus the starting point for all process activities.

To establish a *Zipkin* server, a standard Spring Boot application was built first, that was extended with the dependencies `zipkin-server` and `zipkin-autoconfigure-ui`. For persisting the data in a *MySQL* database, three further dependencies `zipkin-autoconfigure-storage-mysql`, `mysql-connector-java` and `mysql-connector-java` had to be added. Configurations for the database connection are applied in the applications configurations file of the *configurations service*. The *Zipkin* server is finally established through the `@EnableZipkinServer` annotation (see listing 4.1).

Listing 4.1.: Class definition for ZipkinApplication.java

```java
@SpringBootApplication
@EnableZipkinServer
public class ZipkinApplication {
	public static void main(String[] args) {
		SpringApplication.run(ZipkinApplication.class, args);
	}
}
```

Zipkin provides a collector, a storage component, an UI and an API. For performing distributed tracing through all services, a *Zipkin* client needs to be installed on each spring boot application that is supposed to be instrumented. The *Zipkin* client is added as a dependency for each business service by appending `spring-cloud-starter-zipkin` to the classpath. The sampling rate can be adjusted in the application's configuration file. The rate is independent from the actual generation of spans. That means by adding Spring Cloud Sleuth, spans are generated in the application per default but only attached to the HTTP request according to the sampling rate. Therefore, the rate only defines the ratio of traces attached and exported. A higher rate subsequently leads to a higher application overhead.

For the proof-of-concept architecture, the sampling rate was set to 100% for all instrumented services, which samples every request. Since only little amounts of data are produced in the proof-of-concept, performance issues that potentially emerge through

[1] https://cloud.spring.io/spring-cloud-sleuth

a high sampling rate, can be neglected at first stage. The aspects of overhead and sampling rate will be discussed more thoroughly in subsection 5.2.3.

Besides the just described configurations that must be undertaken for instrumenting a microservice additional modifications of the system only need to be made in the endpoints of the *webui service*. As described in subsection 2.3.7, an integral part of the modelling in process mining is the `case_id`, that relates activities to a single process instance. Since the process as a whole is defined as user clicks that emerge from the front-end and the temporal context is defined as one session, a `session_id` was chosen to serve as the `case_id`. The `session_id` is part of all user requests from the front-end as they are passed along as a parameter to the mapped controller method. From there the `session_id` needs to be stored to the span.

In addition to the default annotations that are stored per span it is possible to append *custom annotations*, so called *tags* to a span. To achieve this, a `Tracer` object needs to be initialised in the controller first (see listing 4.2, l. 1). Afterwards, the *session id*, received through the parameter in the endpoint, is added to the tracer (see l. 5). With that, an additional annotation is written to the span.

Listing 4.2.: Appending the *session_id* to the span

```
1  @Autowired Tracer tracer;
2
3  @RequestMapping("/bookCar")
4  public String bookCar(@RequestParam(value="sessionID", required=false,
   ↪  defaultValue="null") String sessionID, Model model) {
5      tracer.addTag("sessionID", sessionID);
6      String msg = "sessionID = " + sessionID + " || WebUI ";
7      msg += " --> " + restTemplate.getForObject("http://cars-service/handleCarBooking",
       ↪  String.class);
8      model.addAttribute("msg", msg);
9      return "index";
10 }
```

4.4. Data architecture

In the following, an overview of the applied data architecture is given. This includes a persistence strategy for the tracing data as well an general description of the created tables and their roles in the data model. A description of the full data model, which shows the table's relationships in an UML diagram, can be found in the appendix (Figure A.11).

4.4.1. Tracing data persistence data

Zipkin per default does not persist the tracing data. The UI queries the API to access the data which is stored in its storage component in-memory. To persist traces, a *Zipkin* server module can be utilised that supports the three storage types *Cassandra*, *MySQL* and *Elasticsearch* natively. *Elasticsearch* is a full-text indexing/search engine that helps querying data but is not best suitable as a data store due to the limited support for data manipulation. From the remaining two default data stores, *MySQL* was chosen over *Cassandra* for two reasons: First, the used PMT requires a relational databases for a continuous data loading. Second and subsequent to the first, the event logs need to be in a flat table structure (see subsection 2.3.6). Since the data has to be pre-processed before it can be imported by the PMT, a platform is needed on which the event log can be generated (see section 4.1). Moreover, the correlation of low level distributed tracing log files in order to create a high-level event log can be easily accomplished through joins, which are standard constructs of SQL and thus relational databases in general. At the same time, drawbacks emerge when choosing *MySQL* as a data storage. For instance a lower write speed compared to *Cassandra* would affect the prototype in a real-world setting with millions of spans stored per day. Limitations of the prototype, emerging of the requirement of real-time are further discussed in subsection 5.1.6.

4.4.2. General table structure

This section classifies and describes the database tables used in the prototype. In the following, the structure of the *spans*, *annotations* and *activities* table is explained extensively, since they are the central tables used in the whole prototype, while the remaining tables (see Table 4.1) will only be described by their role.

Table 4.1.: Database tables usage classification

	Distributed tracing	**Activity generation**	**Data model in PMT**	**Data loading**
zipkin_spans	x	x	x	
zipkin_annotations	x	x	x	
activities		x	x	
technical_activities		x	x	
activity_mappings		x		
trace_span			x	
sessions			x	
reload_trigger				x

spans and *annotations* tables

The *spans* and *annotations* tables structure and relationship to each other is predefined through *Zipkins* data model, where they are used to store the tracing data. They are furthermore used as the two main source tables in the activity generation for the process mining. Third, they will be imported in the PMT to extend the data model and provide additional information for the analysis. Next to the two, a third table is created automatically by *Zipkin* which is the *zipkin_dependencies* table, that stores parent child relationships between the spans that are used for a dependency visualisation in the *Zipkin* UI but will not be used further in this work.

In the *zipkin_spans* table (see Table 4.2), each row provides information about exactly one span whereas in the *zipkin_annotations* table multiple annotations, providing additional information through multiple key value pairs, are stored per span. Thus, a one-to-many relationship between *zipkin_spans* and *zipkin_annotations* exists.

Table 4.2.: *zipkin_spans* table definition

Column	Data type	Key
trace_id_high	bigint(20)	x
trace_id	bigint(20)	x
id	bigint(20)	x
name	varchar(255)	
parent_id	bigint(20)	
debug	bit(1)	
start_ts	bigint(20)	
duration	bigint(20)	

The *zipkin_spans* table has four attributes as a primary key which are the *trace_id_high*, the *trace_id*, the *id*, and the *name*. For the log generation, the *trace_id* and *id* will be used for joining with the *zipkin_annotations* table. The attributes *name*, *parent_id*, *start_ts*, and *duration* will serve to filter and fill the event log attributes like activity name, timestamp and duration.

Primary keys in the *zipkin_annotations* table are the *trace_id_high*, the *trace_id*, the *span_id*, the *a_key*, and the *a_timestamp*. Further attributes of interest are the *a_value*, which stores the actual value of a key, for example the message of an error annotation or the *endpoint_service_name*, which indicates from which service the information of each row originates from.

Table 4.3.: *zipkin_annotations* table definition

Column	Data type	Key
trace_id_high	bigint(20)	x
trace_id	bigint(20)	x
span_id	bigint(20)	x
a_key	varchar(255)	x
a_value	blob	
a_type	int(11)	
a_timestamp	bigint(20)	x
endpoint_ipv4	int(11)	
endpoint_ipv6	binary(16)	
endpoint_port	smallint(6)	
endpoint_service_name	varchar(255)	

activities table

As already described in subsection 2.3.4, the three minimum attributes that are required for process mining are the *case_id*, the *activity name* and some sort of ordering which is typically a *timestamp*. As described in section 4.3, for the proof-of-concept the *session_id* is used as a *case_id*. The *activity table*, that will be used as the main input for the process mining is extended by additional attributes, like the *activity_type*, that indicates whether an activity is a user or system activity or a *failure* attribute, that indicates if an activity has been executed successfully. More attributes are part of the table that are described along with the define data type in table 4.4.

Table 4.4.: Description of the *activites* table

Attribute	Datatype	Description
session_id	*varchar(30)*	Determines the level and scope of a process. Correlates activities to a process instance.
activity	*varchar(50)*	Name of the activity.
start_ts	*datetime(3)*	Determines the start of an activity.
end_ts	*datetime(3)*	Determines the end of an activity.
start_ts_edges	*datetime(3)*	Same as start_ts but used for the calculation of durations between user activities.
end_ts_edges	*datetime(3)*	Same as end_ts but used for the calculation of durations between user activities.
duration	*bigint(20)*	Indicates the duration of an activity.
trace_id	*bigint(20)*	Foreign key to the *annotations* table.
trace_id_hex	*varchar(30)*	Foreign key to the trace of an activity. Hexadecimal used for external link to the Zipkin UI.
span_id	*bigint(30)*	Foreign key to *spans* table.
activity_type	*varchar(20)*	Type of activity: user or system activity
device_type	*varchar(20)*	Device type: indicates the origin of a front-end request
service_name	*varchar(20)*	Indicates which service is called.
failure	*tinyint(1)*	Indicates whether an activity is faulty or not.
sorting	*int(11)*	Indicates an order when two activity share an identical timestamp.

***technical_activities* table**

This table is a sub-set of the before described full *activities* table and serves for displaying system activities that belong to a distinct user activity. The duplicate table, comprising of system activities only, had to be created due to circumstance of modelling both activity types equally in the same event log. To display technical events that relate to a user activity in a second component, the *technical activities* table was created.

***activity_mappings* table**

The *activity_mappings* table is solely used during the activity generation and is not part of the data model in the PMT. In the table, *'pretty names'* are defined for the span names, that imply an inter-service request. E.g. the technical span name `http:/getcars` is a request to the *webui service* and therefore modelled as an user activity. It would be mapped to *List available cars* in order to provide human-friendly activity names in the process visualisation. Moreover, a `is_activity` flag is set, that indicates if an activity should be created and be part of the process. The table's last column `calls_service` indicates to which service the request is directed to.

The columns *pretty_name*, *type* and *is_activity* would have to be configured once by an domain expert that is able to map service calls to process activities.

***trace_span* table**

The *trace_span* table is solely used in the PMT's data model to dissolve the many-to-many relationship that occurs between the *activities* and the *spans* table.

***sessions* table**

The sessions table is filled by the *webui service* and logs sessions (`session_id`) to users (`user_id`) together with a timestamp (`session_start`) device type (`device`) of a session. Through connecting the sessions in the PMT's data model, analysis cannot only be made on a session but also on a user and device level.

***reload_trigger* table**

The last table utilised is the *reload_trigger* table, that triggers a complete reload of the PMT's data model. For that, an entry consisting of the concerned data model (`data_model_name`) and date of request (`reload_request_time`) is written after every new event log generation by the event log generation service. During the data model load, the PMT itself writes a `reload_start_time` and possibly a `reload_success_time` and `reload_message` in the columns of the request. The structure is determined by the PMT's requirements.

4.5. Event log generation

For creating the event log, a two-step approach containing of

1. scoping the event data, and
2. binding the events to instances,

was applied based on [3]. The original approach includes three steps with an original third phase, where the process instances are classified. The authors propose that in that phase, the process log should be split into smaller groups of process instances, to analyse them separately. This step is not conducted since the used PMT is capable of slicing the process cube which enables using a single activity log.

The event log generation as a whole was performed in an iterative process to understand the necessary level of granularity and abstraction [2]. A traces-centric, technical view on a low level was applied first, where the state of a process instance is reflected by REST calls between the microservices, manifested through spans. Coming from that detailed layer with rich information at hand, abstractions were made to derive business relevant activities that represent the business layer. In the SUS, regards to the use case of analysing user click paths in a car sharing system, a *session id* was chosen as a *case id*. Through that, one can analyse all activities respectively clicks a user performed during one session. As described in section 3.2, such a session could for example expire after 30 minutes of inactivity. This definition gives the process a natural scope that is being described in the next subsection.

4.5.1. Scoping the event data

By scoping the event data, one first needs to decide which events are of importance and need to be correlated and transformed to activities of relevance for answering the analysis questions [5]. This activity is typically performed by an domain expert [7].

In the context of distributed tracing data, every span is a type of a low level event. A span stands for a service invocation or procedure call and entails a timestamp. Moreover, multiple annotations are written per span that provide more fine grained information. Since each annotation, as for example a client receive, entails a timestamp, an annotation also embodies an event, but on higher level of detail. The amount of annotations written per span varies between the type of span (method call versus service invocations) but is also dependent on the actual implementation of the architecture. In the SUS up to 14 annotations per spans are written and therefore only a small subset of these evets will serve as an information basis for the creation of an activity. In the following, it will be described how these low level events of the *spans* and *annotation* table, both from different hierarchy levels, are being transformed to events of higher level, namely system and user activities.

The process' functional scope lies on analysing user click streams, therefore all events that stand for a user click in the front-end will be merged into a user activity, since it reflects user behaviour. The event that constitutes a user activity can be found in the *spans* table and is defined as the initials span of a trace. This first span (parent span), is typically followed by a plethora of child spans. Child spans that represent a inter-service request, are defined as *system activities*, as they reflect system behaviour. The mapping table, in which every possible user or system activity is listed, is used to filter activities that are not relevant for the analysis and therefore should not be contained in the event log. The two types of activities, originating from the two layers, can both be viewed independently or conjunct in the process graph of the PMT. For making the event log appropriate for a PMT to work with, both event types were modelled as activities, so the PMT does not distinguish between the types of activities during the generation of the process model through the algorithm. To be able to separate and filter the two layers in the PMT, an extra column was added that indicates the activity type. For a better distinction of these in the process, user activities were translated to human-readable activity names, using the mapping table. System activities were not altered and keep the original span's name to indicate their technical origin.

User as well as system activities can both fail. A failed invocation between two services, e.g. a timeout, is indicated through an error tag in both the corresponding span as well as in the parent spans of a trace. The creation of failed user and system activities is described below.

4.5.2. Transformation of low level events to activities and binding to process instances

In this step the transformation of low level events into activities together with the correlation to a single case (i.e. process instance) is performed. The so far empty activities table, will be subsequently filled with the activities of the four types. The activity types include

1. healthy user activities,
2. failed user activities,
3. healthy system activities, and
4. failed system layer activities.

For each type, the event log algorithm provides a distinctive SQL script, that scans the *spans* and *annotations* tables for events that match the respective activity definition. The following description, that defines the activity types, is supported by sample tracing data of the SUS that was recorded from a random `/bookCar` request. The sample data from the *spans* and *annotation* table can be found in the appendix (see Table A.1 and Table A.2). Due to simplification, the 64 bit `trace_id`, `span_id` and `parent_id` from both tables have been altered to single character IDs. Furthermore not the complete

tables but only attributes of relevance (cf. 4.4.2) are shown.

Healthy user layer activities

As described in the data structure section (subsection 2.2.2), a request in the front-end (e.g. a `/bookCar` request) may yield to multiple service invocations in the back-end, that all share the same `trace_id`. For every of these downstream service calls, a new span is written. Additionally, also method calls can trigger a new span.

The existence of a user activity is defined by the first span of a trace. To verify that only user activities are recorded where no technical failure occurred, an inner join between the spans and annotations table on the `trace_id` and `span_id` (see ll. 17 - 25) is executed.

Listing 4.3.: Creation of healthy user activites

```
1  INSERT INTO activities (
2    session_id, activity, start_ts, end_ts, start_ts_edges, end_ts_edges, duration,
     ↪ trace_id, trace_id_hex, activity_type, failure, sorting
3  )
4    SELECT DISTINCT
5      a1.a_value                                        AS session_ID,
6      m1.pretty_name                                    AS activity,
7      FROM_UNIXTIME((s1.start_ts * 0.000001))           AS start_ts,
8      FROM_UNIXTIME((s1.start_ts + s1.duration) * 0.000001) AS end_ts,
9      FROM_UNIXTIME((s1.start_ts * 0.000001))           AS start_ts_edges,
10     FROM_UNIXTIME((s1.start_ts + s1.duration) * 0.000001) AS end_ts_edges,
11     s1.duration * 0.001                               AS duration,
12     s1.trace_id                                       AS trace_id,
13     LOWER(HEX(s1.trace_id))                           AS trace_id_hex,
14     'user'                                            AS activity_type,
15     FALSE                                             AS failure,
16     1                                                 AS sorting
17   FROM zipkin_spans AS s1
18     INNER JOIN zipkin_annotations AS a1 ON
19                                 s1.trace_id = a1.trace_id
20                                 AND s1.id = a1.span_id
21                                 AND s1.parent_id IS NULL
22                                 AND a1.a_key = 'sessionID'
23                                 AND a1.trace_id NOT IN (SELECT trace_id
24                                                         FROM zipkin_annotations
25                                                         WHERE a_key = 'error')
26     INNER JOIN activity_mappings AS m1 ON s1.name = m1.technical_activity AND
       ↪ m1.is_activity = TRUE;
```

Only the initial span of every trace is joined, namely all entries that do not have a parent_id (l. 21). To only generate activities that are processed successfully within a trace, a second filter (ll. 23 - 25) is applied. Only such traces remain that don't have a error tag in any of span's annotations. The third filter that is applied on this join is the look up of the sessionID key from the annotations table (l. 22). The session_id is passed along as a *tag* in the initial request and is stored as a custom tag in the annotations of the first span, which is created from the *edge service*. With this join between the spans and annotations table, all distinct user activities are selected.

The session_ID, that is the first attribute for which a value needs to be allocated, can be extracted from the above described result set via the a_value attribute. Furthermore, information for creating all relevant timestamps are part of the result. The 16 digit microsecond start_ts timestamp from the *spans* table is multiplied by 0.000001 to receive a value in datetime(6) format that is suitable for further processing in the PMT. The end_ts is defined through adding the spans duration to the start_ts. Through this definition from a single span perspective, one can make sure that effects such as clock skew do not appear. Effects like these would materialise by using timestamps from the *annotations* tables, since they origin from multiple services with possibly synchronised clocks. Therefore, it is recommended to work with local measurements as the start_ts and duration from the spans table to calculate an end_ts.

The start_ts_edges and end_ts_edges timestamps are created in the same manner as the just described ones and only serve for defining durations on the edges between activities in the PMT.

The trace_id and trace_id_hex can also be found in the result set of the applied join.

Lastly, a third join (l. 26) is applied with the *activity_mappings* table on spans name in order to receive a more user friendly 'pretty name' for the activity name that can be understood without possible required domain knowledge. Furthermore, only activities that have a is_activity flag are created. With that, one can define relevant activities in one single repository.

Three attributes remain that are inserted statically. With activity_type = 'user' it is indicated from which layer the activity originates from. It is mainly implemented to later filter the event log during the process mining. The failure attribute indicates if a activity is healthy or not and is set to FALSE for this activity. Lastly, the sorting attribute is inserted which is is implemented as a 'tie breaker' for the process mining algorithm in case two activities share the same timestamp. This behaviour occurs between the user activity and the first system activity, since it reflects the users activity execution on a technical level. Since the user activity should stand chronologically before the system activity, a lower sorting is applied for the user activity.

Failed user activities

Failed business layer activities are created in the same manner as their healthy counterparts that were just described above except two alterations:

First, in line 6 from the original script (see Listing 4.3), the activity name is appended by a "failed" annotation (see Listing 4.4) to indicate the failed state.

Listing 4.4.: Activity name alteration

```
CONCAT(m1.pretty_name, ' failed')  AS activity,
```

Second, the filter described before is now applied on the opposite, so that all traces are selected that have at least one error key for a span in the *annotations* table. These activities are labelled as *failed* user activities. See below the difference from the original code in line 23 - 25.

Listing 4.5.: Filter alteration on join

```
 AND a1.trace_id IN (SELECT trace_id FROM zipkin_annotations WHERE a_key = 'error')
```

In the following, the generation of system activities from the application layer will be described.

Healthy system activities

System layer activities in the SUS reflect inter-service communication. In comparison to user activities, each one is not manually executed but represents downstream calls that are initially triggered by manual user requests from the front-end. The data foundation for the creation of a system activity is also found in the *spans* table. In comparison to an user activity, the creation of a system activity is more complex due to the different types of errors that lead to various forms of tags written in the annotations table.

As for the user activities, system activities stem from the *spans* table. During the first self join (Listing 4.6, ll. 15 - 29) all those entries are filtered, that are inter-service request, indicated by a `http:/` prefix in the span's name attribute. Additionally two filters (ll. 16 - 22 and 23 - 29) are applied that filter the spans so that no failed requests are part of the results set.

Two types of error occur in the SUS. In the first one, a service tries to request a resource from a non-active downstream service. Here, in the last span written in the trace, the attempt of calling the downstream service is recorded. In the annotations of this last span a client sent (cs) to the potential server is written but no response is received. As

a consequence, no client receive (cr) for this span is recorded in the *annotations* table. In the code, a self join of the annotations table is applied to find all spans that share the aforementioned attributes (ll. 16 - 22). This set of `span_ids` is then used in the join attribute (l. 16) for the above described self join of the *spans* tables to filter on spans that are not part of the annotations join result set, which entails the faulty spans of the first error type.

The second error type that the system possibly produces is a 500 server failure. For this type of failure, error tags are written not only in the concerned span but to all upstream spans. To only filter the actual source span of the failure, all spans that have the `a_key = error` tag and at the same time do not have a server send (`ss`) annotation are selected (ll. 26 - 29). Since all upstream spans record the error, the filter on spans without the server send annotation indicates that it is not a first or intermediate but the last span of a trace. As for error type one, the resulting `span_id`'s of the described annotations table self join are also filtered on the `span_id` of the one hierarchy higher lying span self join (ll. 15).

Listing 4.6.: Creation of healthy user activites

```
1   INSERT INTO activities (
2     session_id, activity, start_ts, end_ts, duration, span_id, activity_type,
      ↪  service_name, failure, sorting
3   SELECT
4       a5.a_value                                          AS session_id,
5       s1.name                                             AS activity,
6       FROM_UNIXTIME(s1.start_ts * 0.000001)               AS start_ts,
7       FROM_UNIXTIME((s1.start_ts + s1.duration) * 0.000001) AS end_ts,
8       s1.duration * 0.001                                 AS duration,
9       s1.id                                               AS span_id,
10      'system'                                            AS activity_type,
11      am.calls_service                                    AS service_name,
12      FALSE                                               AS failure,
13      2                                                   AS sorting
14    FROM zipkin_spans s1
15      INNER JOIN zipkin_spans s2 ON s1.id = s2.id AND s1.name LIKE 'http:/%'
16              AND (s1.id NOT IN (SELECT DISTINCT a1.span_id
17                                 FROM zipkin_annotations a1
18                                   JOIN zipkin_annotations a2
19                                     ON a1.a_key = 'cs' AND a1.span_id = a2.span_id AND
20                                        a2.span_id NOT IN (SELECT span_id
21                                                           FROM zipkin_annotations
22                                                           WHERE a_key = 'cr'))
23                   AND s1.id NOT IN (SELECT DISTINCT a3.span_id
24                                     FROM zipkin_annotations a3
```

```
25                                          JOIN zipkin_annotations a4
26                                            ON a3.a_key = 'error' AND a3.span_id =
                                             ↪  a4.span_id AND
27                                               a4.span_id NOT IN (SELECT span_id
28                                                                  FROM zipkin_annotations
29                                                                  WHERE a_key = 'ss')))
30     INNER JOIN zipkin_spans s3 ON s3.trace_id = s1.trace_id AND s3.parent_id IS NULL
31     INNER JOIN zipkin_annotations a5 ON a5.span_id = s3.id AND a5.a_key = 'sessionID'
32     INNER JOIN activity_mappings am ON s2.name = technical_activity AND is_activity =
      ↪  TRUE;
```

The so far described result set is joined with a second spans table (l. 30) on the `trace_id` in order to receive the initial span of the trace, from where the `session_id` tag can be obtained. Here, the so called log correlation is applied, since events that are linked via the `trace_id` are correlated to the `session_id` which can be found in the initial span. This attribute is located in the annotations of the initial span and is attained through a join on the `span_id` with the `session_ID` key.

The described joins (ll. 15 - 30) lay the foundation for an activity existence. With the result set, the attribute's `activity`, `start_ts`, `end_ts`, `duration`, `span_id` can be inserted. To derive the `session_id`, an additional join has to be applied (l. 31).

To only receive system activities of interest in the result set and in order to derive the `service_name` attribute, a last join with the mapping table is applied via the span's `name` attribute.

Failed system activities

As for the failed user activities before, only little changes in the script have to be applied to generate failed system activities. First of all, the activity name has to be appended via a `CONCAT(m1.pretty_name, ' failed')` suffix. Moreover, the applied filters have to be reversed. In essence, both filters that capture the two distinct error types remain the same but the resulting set of `span_ids` is inverted in its lookup with the *spans* table.

For failures of the first type, see the alteration in the original code (Listing 4.6, l. 16) from a `NOT IN` to to a `IN`.

Listing 4.7.: First filter alteration on join for failed system activities

```
AND (s1.id IN (SELECT DISTINCT a1.span_id
```

The same applies for the second filter's IDs (see original code Listing 4.6, l. 23). The lookup is inverted and the `AND` is changed to a `OR` in order to record a failure for any of the two cases.

Listing 4.8.: Second filter alteration on join for failed system activities

```
OR s1.id IN (SELECT DISTINCT a3.span_id
```

4.6. Process configuration and analysis creation in the process mining tool

4.6.1. Process mining workflow

One goal of the proof-of-concept was to design a prototype that allows the monitoring of business processes with distributed tracing data in near real-time. To achieve this goal, the process of log generation and data loading was automated. As depicted in Figure 4.3, the workflow starts with an user that performs activities in the front-end. From the front-end it is possible to trigger the whole process manually. Alternatively, the log generation service is scheduled to start the process every five minutes. The interval is adjustable. After the activity generation algorithm is started, which is stored in the log generation service, the data base executes the algorithm and writes data to several tables. It is important to note, that for the *activities*, *technical_activities* and *trace_span* table a complete re-write is performed. For the *reload_trigger* table, a new entry is inserted for every execution. The PMT is configured to watch the *reload_trigger* table and to check for new entries every 5 minutes. If a new entry exists that has no value for the `reload_start_time`, which indicates if a reload started, a new data load of the complete data model is performed.

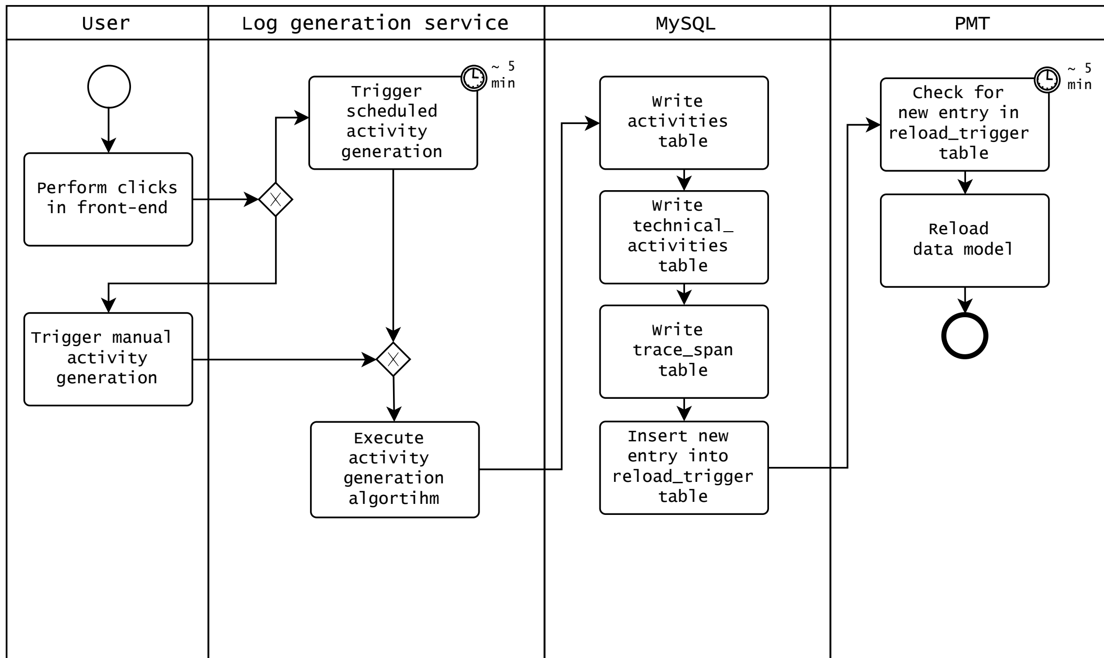


Figure 4.3.: End-to-end process mining workflow

4.6.2. Data model and activity table configuration

Every analysis in the PMT must be connected to one *data model*, that defines the relations of imported tables to each other. The connected tables are later accessible from the PMT to define dimensions, build KPIs and apply filters. One data model thereby can serve multiple analysis but one analysis can only make use of one data model at a time.

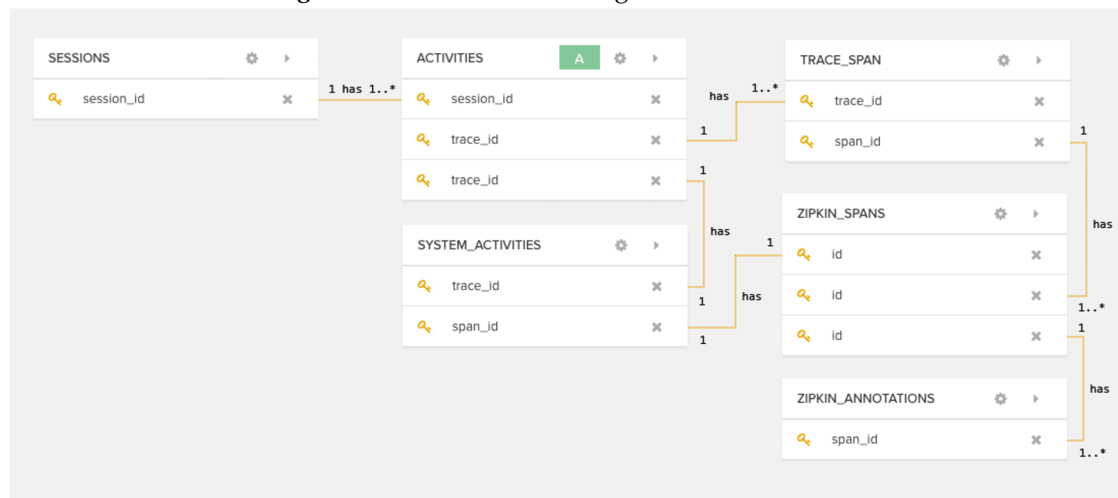
As described in subsection 2.3.3, the central and minimum required table for generating a process model is the *activities* table. The table is configured by assigning it the role of the activity table. To interpret the table by the process mining algorithm, the three basic required attributes *case_id*, *activity* and *timestamp* have to be mapped to the columns of the source table (see Table 4.5). Further process configurations, as for example a sorting or resource columns, are also defined here and can be found in Table 4.5.

Table 4.5.: *Activities table* configuration in the PMT

Attribute	Value
Case ID column	*session_id*
Activity column	*activity*
Timestamp column	*start_ts*
End Timestamp column	*none*
Show resource column	*false*
Respect Sorting Column	*true*
Sorting column	*sorting*

Besides the *activities* table, the data model for the *proof-of-concept* consists of four more tables, that are connected via several foreign key - primary key relationships. The *activities* table builds the centre of the data model and is directly connected with the *sessions*, *trace_span* and *system_activities* table and indirectly with the *zipkin_spans* and *zipkin_annotations* table.

Figure 4.4.: Data model configuration in the PMT



In order to use span and annotation data in the analysis, the *trace_span* helper table had to be setup and put between the *activities* and *spans* table. It provides a mapping for the multiple spans that exist per traces. The *activities* table and *trace_span* table have a one-to-many relationship and are connected via the *trace_id*. Since only user activities are modelled with a trace_id, a join is only realised for this type of activities. The *trace_span* table is connected with the *zipkin_spans* table via the *span_id*. They also

share a one-to-many relationship.

One span again has multiple annotations. Therefore, the *zipkin_annotations* table is connected with the *zipkin_spans* table via the span_id having a one-to-many relationship that builds another second hierarchy.

A second *system_activities* table is connected to the *activities* table, that only contains technical activities but besides resembles the *activities* table. This is necessary to establish a relationship between a calling user activity and its called system activities. However, the process visualisation only builds on the regular *activities* table.

The last table connected is the *sessions* table, that has a one-to-many relationship to the *activities* table, since one session can contain multiple user and system activities. The tables are connected via the *session_id*.

4.6.3. Process visualisation

The PMT applies a *fuzzy mining* algorithm for generating the process model. *Fuzzy models* are well suited for expressing complexity, that arises for example through a high number of distinct activities and edges [29]. In the context of mining user click paths, complexity appears through the high amount of edges, since a user can perform almost any activity at every step of the process. This leads to a potentially high number of edges (i.e. connections) in the process model. Early process mining algorithms were not able to control this complexity, since they assumed a more structured and controlled process [30]. To handle this complexity, fuzzy mining techniques simplify complex process typologies by using aggregation and abstraction mechanisms, as applied for geographic maps [4]. The complexity of the model can be influenced with two parameters that increase or decrease the amount of activities and edges shown in the model, which is called the *process coverage*. Starting with a low coverage, one would see a high level view of the process with the most common paths and activities. By increasing the process coverage, one could step-by-step analyse less frequently occurring patterns of a process.

Figure 4.5 shows a fuzzy model in which activities are displayed as nodes and transitions between activities as edges. The coverage is not set to 100%, so only the most important activities and edges are displayed in the visualisation. Moreover, it is possible to hide certain activities in the process visualisation. In this view, only *user activites* are shown.

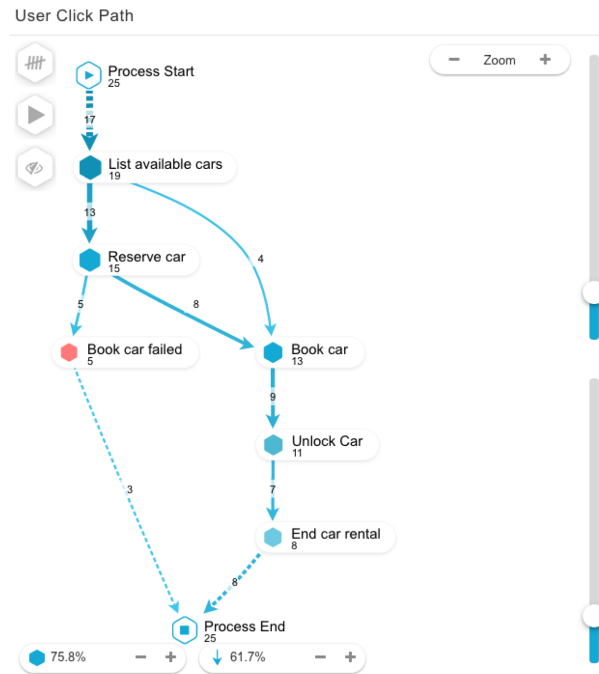


Figure 4.5.: User click path visualisation in the PMT

The prototype provides two perspectives on a process with two types of activities that originate from two domains: The first is a user-centric click path visualisation, that displays *user activities* only. The second is a technical application layer perspective, in which *system activities* define the process model. For the visualisation it is possible to include both activity types in the model, in order to analyse the process from a cross-domain perspective.

Besides the two perspectives, the analysis provides four different KPIs that enhance (see subsection 2.3.2) the process model by adding them on the activities and edges of the model. The *Case Frequency* and *Activity Frequency* KPIs are provided by default by the PMT. The *Duration* and *Conversion Rate* KPIs were implemented individually for the specifics of the data model and the analysis questions that the work imposes. The meaning of all four is being described in the following.

Case Frequency The Case Frequency KPI represents the total amount of cases that pass a certain activity or connection between two activities. In Figure 4.6, one can see that out of 25 sessions in total (source), in 19 cases the user clicked on *List available cars* at least once. A followed car reservation (*Reserve car*) was performed directly after the *List available cars* in exactly 13 cases, but is in total part of 15 cases. This discrepancy is due to the aforementioned *process coverage*, that is not set to 100% for the model excerpt. For two cases, *Reserve car* has a different direct predecessor activity then list available cars. This could for example be another

activity that is not part of the process model yet.

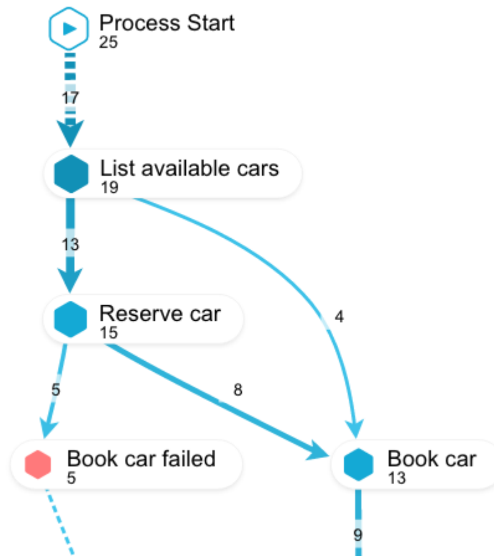


Figure 4.6.: Model excerpt showing the *case frequency* KPI

Activity Frequency In contrast, the Activity Frequency KPI describes how often a certain activity was executed in total and not per case. The perspective is switched from a case to an activity perspective. If the user for instance searches for available cars more than once during one session, the case frequency would only count it once while the activity frequency would note two executions and sum it up to derive the total amount of called activities.

Duration The Duration KPI enhances the process model with two types of durations. The first is tied to each *activity* and represents the average time the system needs to process an activity. The formula implemented for the activity calculates the average value for all equal activities and is defined in PQL as `AVG("activities"."duration")`.

The second duration is laid over the *edges* and represents the time between two activities. The duration between an *activity a* that is directly followed by *activity b* is calculated as the date difference between the start timestamp of *activity a* and the end timestamp of *activity b*, which is implemented as follows:

Listing 4.9.: Definition of edge durations

```
AVG(1.0*DATEDIFF(ms, SOURCE("activities"."end_ts_edges"),
↪   TARGET("activities"."start_ts_edges")))*0.001
```

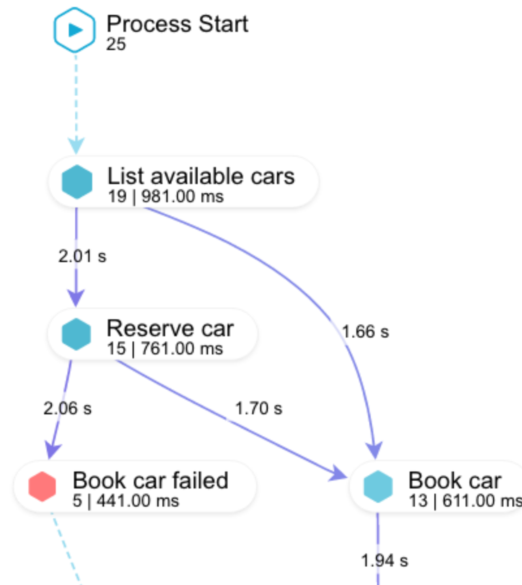


Figure 4.7.: Model excerpt showing the *durations* KPI

The durations on edges are only implemented between activities of the user type. In this case, they for example represent the time a user needs for navigating in an app or the time between a car reservation and a booking. Durations between technical activities are not implemented which is due to the fact that they are calling each other and an end timestamp of *activity a*, followed by *activity b*, is higher then the end start timestamp of *activity b*, which follows *activity a*. This and further issues of visualisation of two hierarchies in one model will be further discussed in subsection 5.2.2.

Conversion Rate The Conversion Rate KPI calculates a probability of how likely it is, that performing a concerned activity ultimately leads to a conversion during a session. This probability is calculated from historic sessions, where it is checked how often the concerned activity led to a conversion. With that information one could estimate how critical an activity is for business success. The *Conversion Rate* for the activities is defined in Listing 4.10.

Listing 4.10.: Definition of activity conversion rate

```
SUM(CASE WHEN process equals 'Book car' THEN 1.0 ELSE 0.0
 ↪  END)/COUNT_TABLE("activities_CASES")
```

The conversion rate for edges is define in Listing 4.11.

Listing 4.11.: Definition of edge conversion rate

```
AVG(
    CASE
        WHEN TARGET(PU_COUNT("activities_cases", "activities"."activity",
         ↪  "activities"."activity" = 'Book car')) > 0 THEN 1.0
        ELSE 0.0
    END
)
```

See below Figure 4.8 a model excerpt, where additionally colour mappings where applied to support the visualisation of the conversion rate.

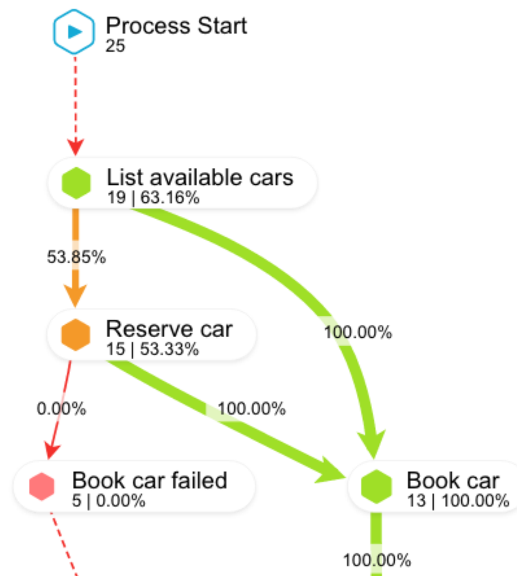


Figure 4.8.: Model excerpt showing the *conversion rate* KPI

4.6.4. Description of created analyses

In the following, four different analyses are presented and described. All of them address unique analysis questions that arise from the business, system operations and user experience domains. They serve to proof the two central themes that the approach presented in this work proposes:

1. Event logs generated from distributed tracing data are suitable for analysing business processes of a microservice architecture, and

2. distributed tracing data can be employed to enhance traditional business process mining with technical performance data.

While the Business Analysis and Application Analysis both only possess a single domain perspective on the process, the Cross-domain Analysis and Single User Activity Analysis provide a multi-perspective view on the process that enable answering analysis questions that span over multiple layers.

The following description of the analysis is realised using the following structure:

Table 4.6.: Structure of analysis description

Structure element	Explanation
Aim	*Describes the objective of the analyis. Which questions does it try to answer?*
Perspective	*Describes the perspective taken on the process.*
Components	*Describes the components used to reach the objective of the analysis. Components specifics (applied filters, defined dimensions and KPIs) are documented in the appendix.*
Value add	*Describes how value is created beyond traditional business process mining.*

Every analysis consist of multiple components. A component can be comprised of a set of sub-components. A technical documentation of the implemented (sub-)components, including defined dimensions, KPIs and filters, can be found in the appendix (section A.1). Every component and sub-component is numbered according the following syntax:

$$\{analysis\ abbreviation\}_\{component\ number\}.\{subcomponent\ number\}?$$

Analysis 1 - Business Analysis (BA)

Aim The analysis aims to investigate the user’s click paths in order to understand important activities that lead to a conversion (booking of a car or booking of a package) along with basic business KPIs, that measure the business success of the car sharing platform, as for example the total amount of sessions or bookings. Moreover, the analysis provides means to investigate time-related aspects like the durations between two user activities.

Perspective The perspective applied for the analysis focuses on user activities only.

Components The header (1) consists of seven components, that are business-relevant single KPIs, described in Table 4.7.

Table 4.7.: KPIs of header component in the BA analysis

KPI Name	Description	Token
Conversion Rate	Indicates the ratio between amount of session with at least one conversion (*Book car* or *Book package* activity) compared to the total amount sessions.	*BA_1.1*
# Package Bookings	Total amount of packages booked.	BA_1.2
# Car Bookings	Total amount of cars booked.	*BA_1.3*
Reservation + Car Booking	Ratio of sessions where a car was reserved and subsequently booked compared to all sessions.	*BA_1.4*
Ratio Car Reservations Only	Ratio of sessions where only a car reservation but no booking was conducted compared to all sessions.	*BA_1.5*
AVG Session Length	Average length of a session from the (Duration from first to last activity).	*BA_1.6*
AVG Rental Length	Average length of a car rental (Duration from *Unlock car* to *End car rental*).	*BA_1.7*

The click path component (2) is the actual process visualisation that was described earlier (see subsection 4.6.3). For this analysis, one can choose between displaying the durations and conversion rates KPIs for the activities and edges. The durations can be interpreted for multiple purposes like e.g. as the time the user needs to navigate in the application (Activity *Search for available cars* to *Reserve car*), that would help to optimise the UI from an User Experience (UX) point of view. Other durations, e.g. between the start of a booking (*Book car*) and the locking of the car (*Lock car*) lead to typical business related KPIs, like an average car rental time. Moreover the conversion rate KPI can be displayed, which has already been described above.

The chart component (3) shows time series data for the total amount of bookings and total amount of sessions per day. Moreover, the conversion rate is displayed on a secondary axes. Through that, one can see how business success has developed over time.

The table component (4) lists all available activities and their impact for a conversion. This again gives an indication of importance per activity of the process.

Component (5), which is also a table component, breaks down the amount of sessions and bookings of cars and packages on either a user or device dimension. This is achieved by using the dropdown field. Through the user dimension, one can conduct user segmentations and compare how for instance 'heavy' users navigate through the application compared to users that only use the service rarely. Multiple types of segmentations are possible e.g. one could distinguish between the total amount of bookings, the total time spent on the platform or the duration of bookings as well as combinations as for instance users who spend a lot of time on the platform without actually booking. Furthermore, one can compare statistics for different device types (e.g. mobile vs. web) and again filter, to spot deviations in the usage of the service (e.g. less clicks till conversion in a mobile app compared to the web application).

The last component of the analysis is a chart that displays the time between two user activities in predefined buckets (6). The start and end activities can be thereby chosen freely. In the diagram, the amount of sessions that fall into a specific duration are displayed as a column. This gives an indication about the distribution of durations between activities. One could for instance filter on all sessions, where the time between a reservation and booking was above average.

Value add For this analysis, no additional value add beyond traditional business process mining is shown. Instead, the created analysis proofs that the derived data from the distributed tracing system is suitable for answering typical analysis question that arise from the business domain.

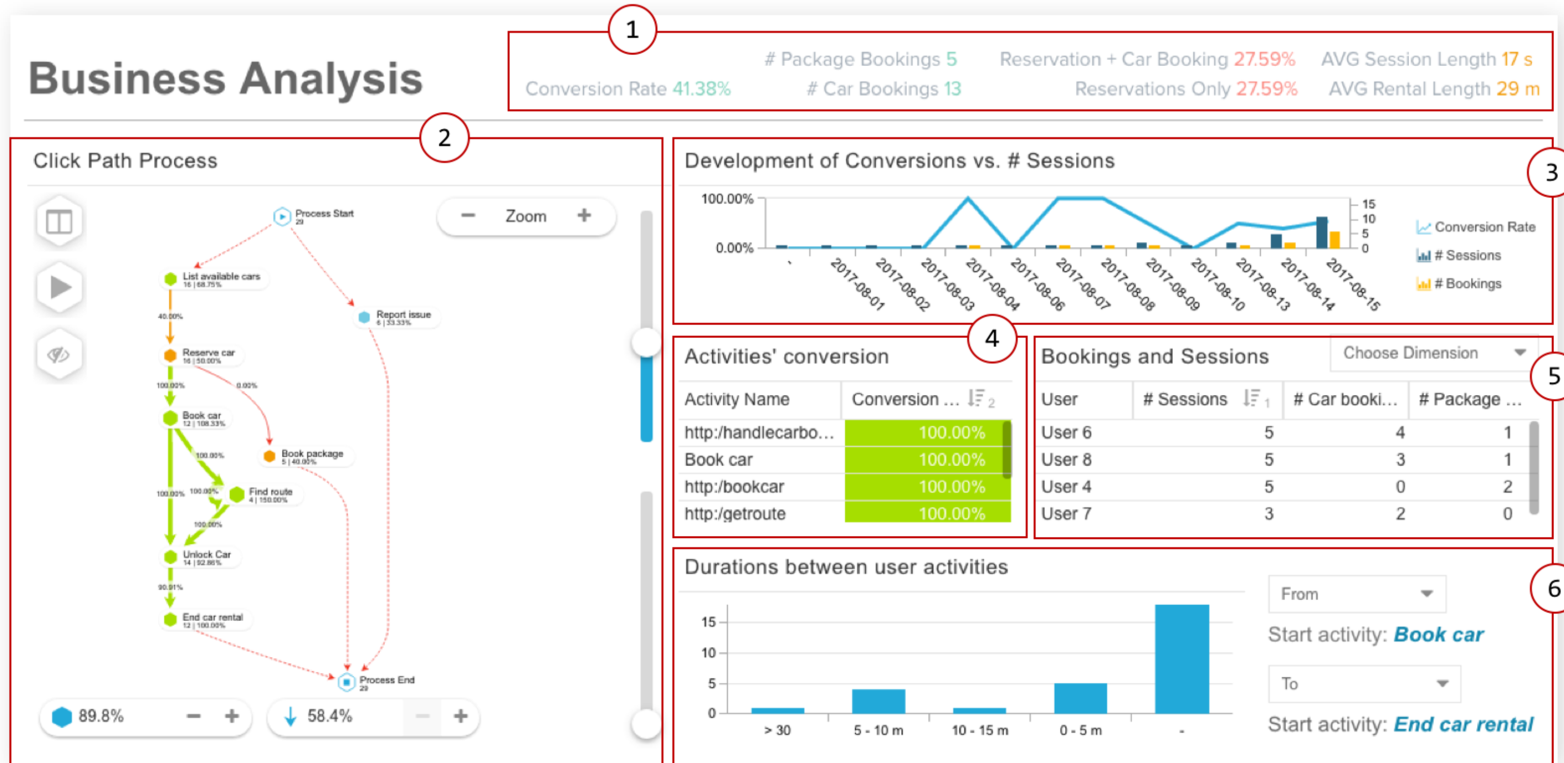


Figure 4.9.: Business Analysis in PMT

Analysis 2 - Application Analysis (AA)

Aim The Application Analysis aims to give an overview of relevant KPIs for monitoring the performance of the application layer. Besides the performance, also the criticality of a service, meaning how involved it is in the service composition for a business process, is a subject of interest.

Perspective The perspective applied is focusing on the application layer of the system. The performance of user activities is analysed as well; but not from an order perspective as in click path for the user activities, but rather by comparing the durations for processing the user requests in the back-end. Performance is measured for three objects: a microservice, a user activity (service compositions of multiple request) and a system activity (single intra-service request).

Components The header (1) gives a quick overview of relevant performance indicators and consists of three single KPI components, described below (Table 4.8).

Table 4.8.: KPIs of header component in the AA analysis

KPI Name	Description	Token
AVG Duration of User Activity	Average duration it takes in total to process a user request (i.e. user activity) by the back-end.	*AA_1.1*
User Activity Success Rate	Ratio between user activities that are processed without an error compared to total amount of user activities.	*AA_1.2*
System Activity Success Rate	Ratio between system activities that are processed without an error compared to total amount of system activities.	*AA_1.3*

The bar chart of component (2) lists all user activities and compares their duration in percentage to the mean of durations for all activities.

Component (3), in comparison focuses on system activities. The bar chart shows the amount of requests arriving at a microservice and the amount of these requests that cannot be processed successfully through the service.

Component (4) is a line chart that displays historical performance data for each microservice. It represents the successful system activities arriving at the service compared to all request and is calculated as *AA_1.3*.

Table component (5) lists all system activities together with the total amount of how often the activity was called and the average execution time for the request. Since

technical activities might call downstream services, their duration can span across multiple requests respectively multiple system activities. This has to be taken into consideration when analysing and comparing performance.

Component (6) condenses information from component (3) and (4) in a table view.

The last component (7) shows different error types that occurring and a count on the amount of cases. Again it can be filtered on all cases with the specific failure type.

Value add This analysis only serves as a dashboard and and does not include any visualisation of a process model. Therefore it does not address the typical analysis questions of process mining.

Nevertheless, it proves that the data derived from the activity log can be used with the techniques of process mining, to illustrate relevant information for monitoring performance on the application layer.

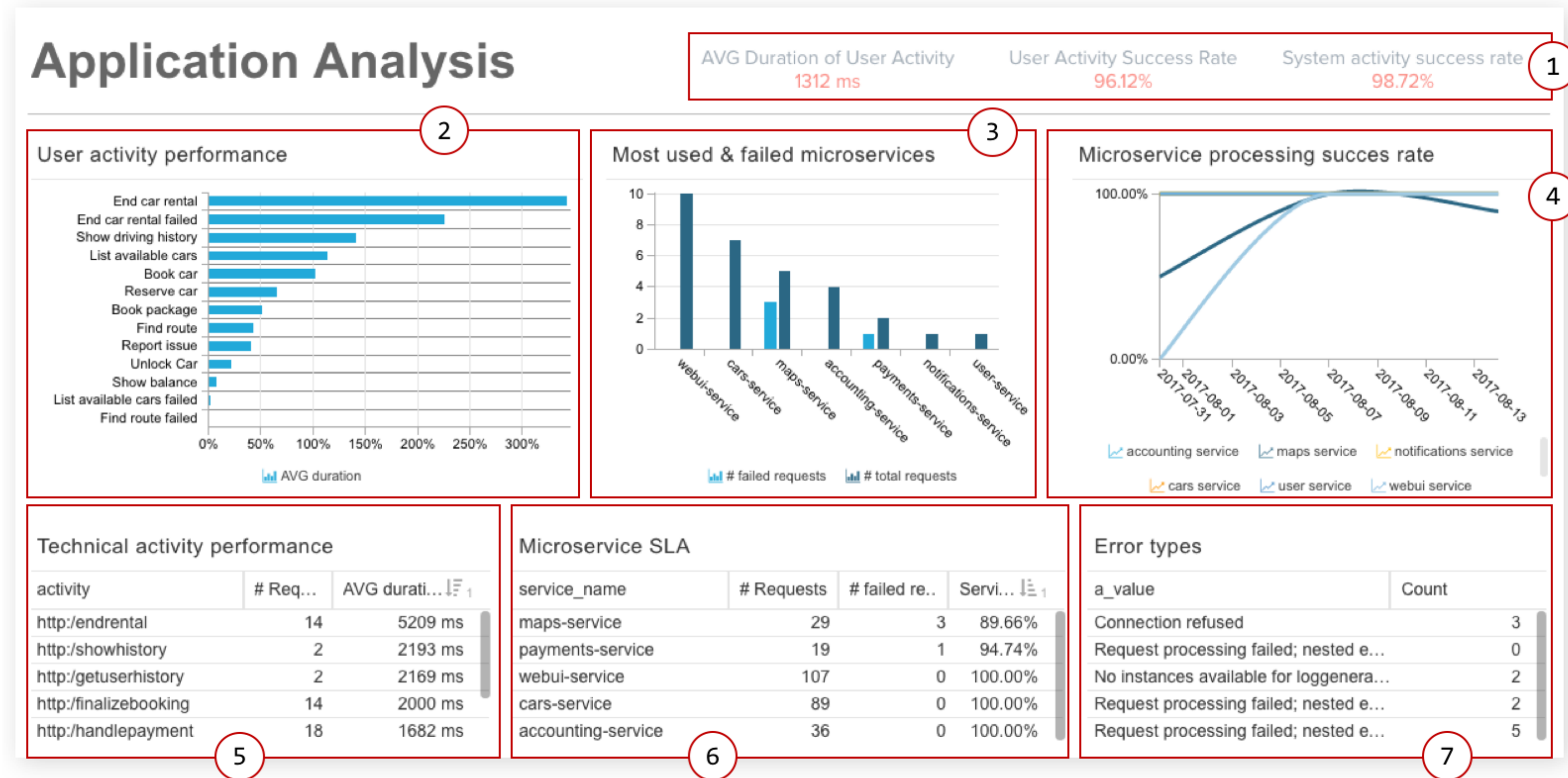


Figure 4.10.: Application Analysis in PMT

Analyis 3 - Cross-Domain Analysis (CDA)

Aim The aim of this analysis is to find correlations between business and application performance. One exemplary analysis question could be, if the root causes for decreased business performance (e.g. less bookings) are due to technical performance issues like delays or microservice failure in the application layer. Moreover, it is possible to deduce a criticality per microservice within a business process. This is accomplished through first analysing most critical user activities through calculating their impact for a conversion (see subsection 4.6.3). In a second step, involved system activities, that are triggered through the user activity and each invoke a microservice endpoint are totalled. The aim is to get an overview in which user requests a microservice is involved and how often its resources are requested. With both information, a business criticality can be derived.

Perspective The *Cross-Domain Analysis* aims to combine the business and application layer perspective in one analysis.

Components The analysis also provides a header component (1) that displays the most relevant KPIs but from both an application as well as business perspective. They are described in Table 4.9.

Table 4.9.: KPIs of header component in the CDA analysis

KPI Name	Description	Token
Conversion rate	same as BA_1.1	see *BA_1.1*
User Activity Failure Rate	Shows the ratio of failed user activities compared to all user activities.	*CDA_1.2*
AVG Clicks to Conversion	Depicts how many clicks in the front-end are needed on average till a conversion, meaning a car booking, is accomplished.	*CDA_1.3*
Session Duration w/ c	Indicates the average time of a session (duration from first to last activity) for all cases where a conversion did occur.	*CDA_1.4*
Session Duration w/o c	Indicates the average time of a session (duration from first to last activity) for all cases where no conversion did occur.	*CDA_1.5*

Component (2) is the user click path. Like in the Business Analysis, one can choose between the durations and conversion rate KPIs, that enrich the process model. Again, only user activities are displayed for this analysis.

Component (3) shows a line chart that compares the historical development of the conversion rate with the user activity failure rate. This component is used to find correlations between the two indicators.

Component (4) is a bar chart that compares the duration from start of a session to conversion for different device types. With that, differences in UX but also system performance become apparent. This component can be used to filter on specific device types and analyse the device dimensions using all other components of the analysis.

The bar chart of component (5) displays microservices by the amount of requests that arrive at their endpoints.

The table component (6) shows all user activities, their total number of invocations, their average durations and the actual impact for a successful conversion. For both duration and conversion rate, colour mappings are applied to indicate a context for the performance.

By filtering on the activities with highest conversion impact and activity count, component (5) adapts automatically and presents the most critical microservices as formulated in the exemplary analysis question.

Value add The value add to business process mining is manifested through the described integration of performance indicators from the application layer that help to find root causes for anomalies in business performance.

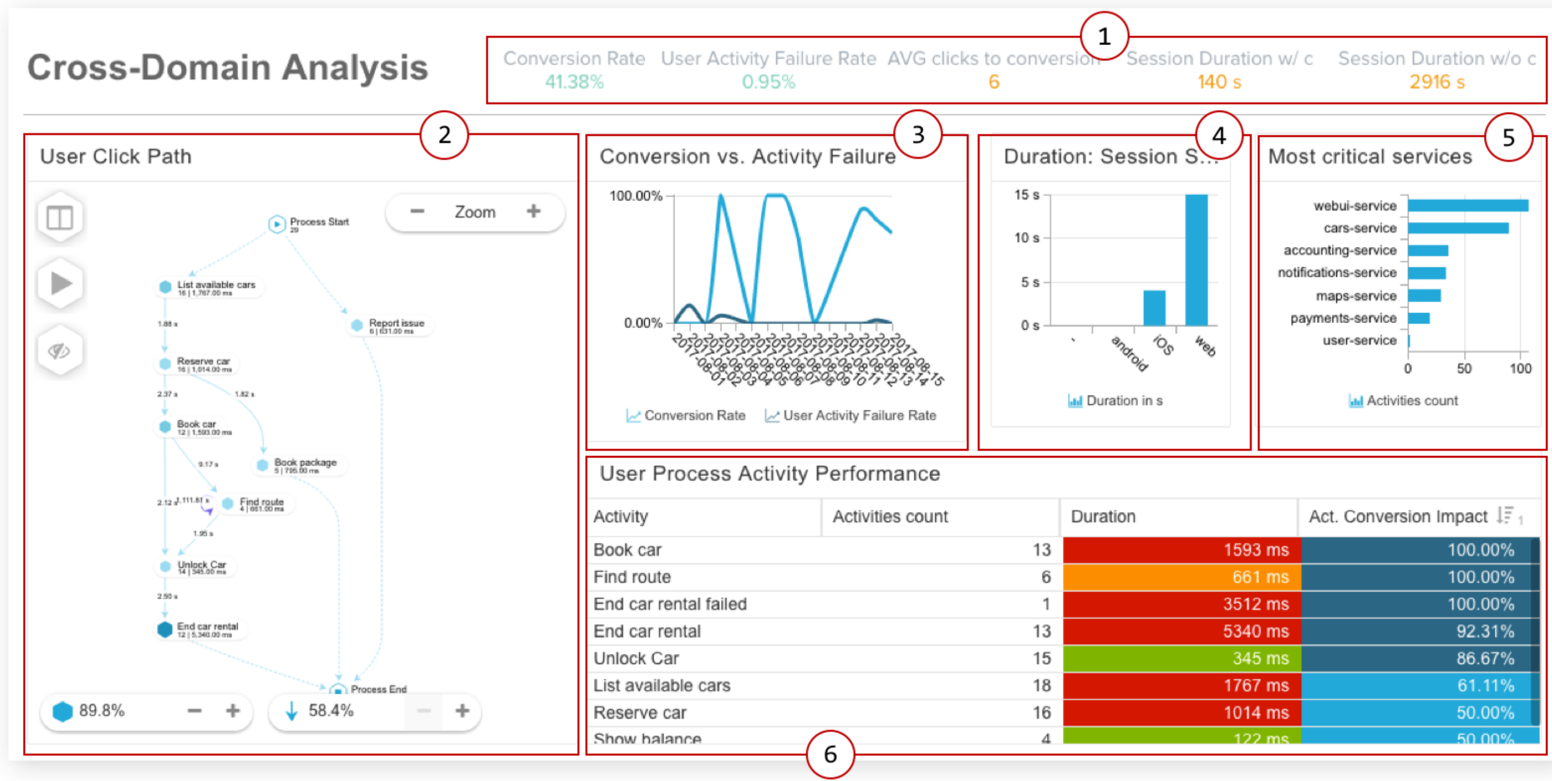


Figure 4.11.: Cross-Domain Analysis in PMT

Analysis 3 - Single User Activity Analysis (SUAA)

Aim The aim of this analysis is to twofold. First, technical performance for unique user activity requests can be compared to the mean of all user activities of that type (e.g. *Reserve car*) to identify long-running activities and select them for further analysis. Second, the relation between a user activity and its followed system activities can be analysed, to for instance find out what system activity led to a failed user activity.

Perspective For the analysis, not only user activities but also system activities are displayed in the graph. Therefore a business as well as technical view is applied in the process.

Components In the *Single User Activity Analysis*, the header component (1) contains three KPIs that are described more detailed below (see Table 4.10).

Table 4.10.: KPIs of header component in the SUAA analysis

KPI Name	Description	Token
Variance of Duration for Selected User Activity	The variance of a user activity, that can be selected within component (6), gives a hint, how the durations for all of the selected user activities deviate from their mean.	*SUAA_1.1*
AVG Duration for Selected User Activity	Shows the average duration of a type of user activity, that can be selected within component (6).	*SUAA_1.2*
AVG User Activity Duration	Depicts the average duration over all user activities.	*SUAA_1.3*

Moreover, component (2) shows the user click path including user and technical activities. As a KPI for the activity and edges, the durations KPI is chosen. As described in subsection 4.6.3, the durations for the edges of system activities are not calculated as it is the case for the user activities.

Components (3) and (4) can be described together since they serve the aim to represent the relationship between user and system activities. Table (3) lists all available user activities together with a trace id. By selecting one of them, table (4) displays the corresponding technical activities together with a count that belongs to the user activity in general or to the specific trace in particular. With that, it can be analysed for instance, what (failed) system activities are typically involved in a failed user activity.

Component (5) is equal to component (6) from the Cross-Domain Analysis but extended with a variance column that indicates how the durations for user activities deviate from the mean. User activities with high variance are a starting point for further investigation in component (6), that displays user activities for single trace ids, their duration and a percentage that shows the duration compared to the mean. Moreover, a dropdown menu is implemented, that filters the component on different types of activities (e.g. *Reserve car*). Through applying appropriate filters, one can derive a set of traces where performance differs significantly from the mean. These traces can be now further analysed in the distributed tracing tool, to find root causes for the long running user activities.

After identifying patterns poor system performance and filtering on set of concerned traces, root causes can be analysed with *Zipkin*. This is possible by clicking on the trace id, that opens a new tab in the browser with the visualisation of the trace in *Zipkin*.

Value add The value added to business process mining lies in the connection between a criticality of a user activity for business success (conversion) and its analysis of a technical performance (duration). Since those two measures are in general correlated [51], a performance analysis can be conducted on a single trace level to find root causes for long running activities.

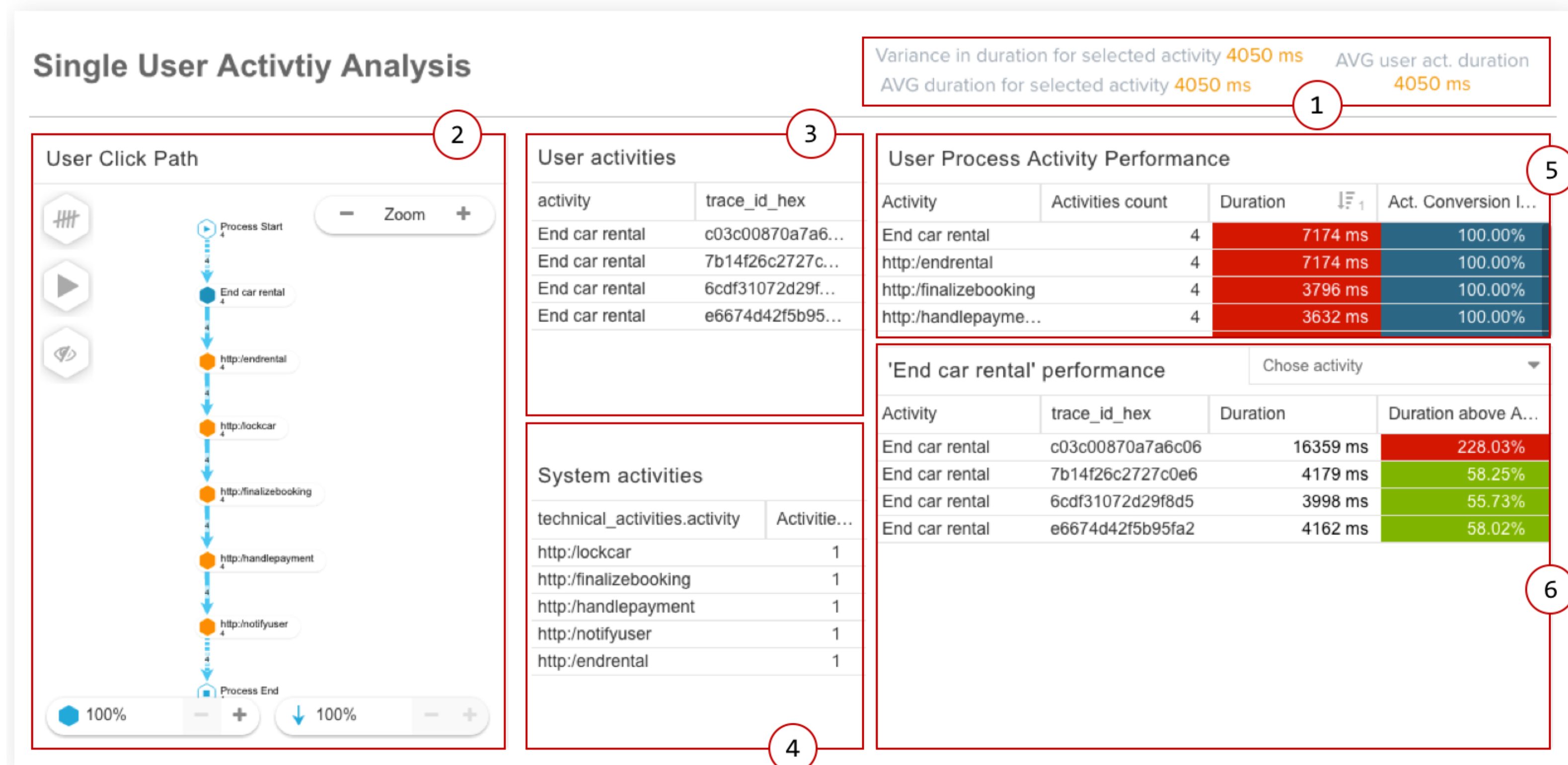


Figure 4.12.: Single User Activity Analysis in PMT

5. Discussion

Following the design science approach, this chapter evaluates the developed artefact and the general approach presented. Thereby, benefits and limitations are discussed that became apparent during the creation of the prototype. Moreover, the work is compared and set in the context of existing research.

5.1. Benefits of the proposed solution

The upcoming section points out the benefits that the proposed solution entails. They can be clustered into monetary advantages (5.1.2), facilitating advantages (5.1.3, 5.1.4, 5.1.5) and advantages that extend the functional capabilities of existent solutions (5.1.1, 5.1.6, 5.1.7).

5.1.1. Cross-domain analysis

One of the main benefits the described proof-of-concept offers, is a tool for answering cross-domain analysis questions. Through providing a technical as well as business view on a process, root causes for poor performance can be analysed inter-disciplinary. Through that, potential correlations between business performance and system performance can be detected.

This is made possible through utilising distributed tracing data, that records fine grained events occurring in a system. The proof-of-concept tool tries to enable a holistic process view that cuts down information silos between the layers of an enterprise architecture. While the focus of traditional business process mining lies in finding inefficiencies or compliance violations (see subsection 2.3.1), *extended* process mining with distributed tracing data provides the means for integrating service availability and service performance together with business performance like for example successful conversions.

The approach is supported by three practices. First, by generating an event log that includes two activity types. *User activities* on the one hand, that represent requests a user manually performs in an arbitrary front-end and *system activities* on the other hand, that represent inter-service communication and provide a detailed understanding of how the system processes business transactions (e.g. *How does the business transaction*

for a failed user activity look like?). The event log was used to discover a process model with the the techniques of process mining. As a second practise, the event log was extended with performance data from the distributed tracing instrumentation (e.g. durations for user requests), to analyse the application performance directly referring to activities. The process model is extended by throughput times between two user activities and durations of activities. Hereby, one can distinguish between the amount of time it takes for the back-end to process a user request, and the the time that passes between two completed user activities. Through that a clear segregation of throughput times that describe either system ore user behaviour is provided. As a third practise, the data model was extended in the PMT with the original distributed tracing data sources (*spans* and *annotations* table), in order to calculate complex KPIs, define analysis dimensions and filter the analysis on the basis of attributes from the distributed tracing data (e.g. filter all cases where an error 500 occurred).

5.1.2. Resource efficient data source for generating event logs

Another central benefit the approach constitutes is the utilisation of distributed tracing data as a resource efficient input source for process mining. The suitability of this novel type of data input was proven within the the proof-of-concept, where different process mining techniques were tested on an event log that utilises distributed tracing data.

Compared to other approaches and inputs for event log generation (see subsection 2.3.5) the proposed method comes with relatively little implementation effort - both for the instrumentation of a SUS (see section 4.3) as well as for the event log generation (see section 4.5).

5.1.3. Portability

The aforementioned limited implementation effort is furthermore extended by the portability of the proposed solution that relates to the question of *How much effort it takes to implement the same approach for another system?* An answer to the question is split into the steps *instrumentation*, *event log generation process configuration* and *analysis creation in the PMT*.

Instrumention First, the system would need to be instrumented with a tracing library. The effort for instrumenting a new component is minimal but scales naturally with the total amount of components in a system. For each component, an instrumentation library has to be chosen according to the component's programming language and framework used. OpenTracing already provides ready instrumentations for the most prominent programming languages (therof Go, JavaScript, Java, Python, Ruby, Objective-C, C++, C#, PHP). SUSs written in different programming languages using different frameworks produce tracing data in an equal structure.

Event log generation Most customisations have to be applied for the log generation algorithm in order to adhere to the specifics of the system. Some customisations only refer to the contents of logged attributes (e.g. how is the content of an error tag defined) and can be customised relatively easy. The overall data structure (spans, annotations) would also remain the same through the OpenTracing standard. However, the architectural style and inter-process communication (see subsection 5.2.1) substantially determine the way activities are defined. Therefore one needs to redefine what constitutes a healthy or unhealthy user or system activity.

Process configuration and analysis creation in the PMT When using the same PMT with a different architecture, various other Relational Database Management Systems (RDBMSs) beside the used MySQL could be connected to the tool. It is possible to export the data model and the analysis and reuse them for any other system. Changes possibly need to be applied to the data model (e.g. change table names, import additional tables) but the core tables (activites, spans, and annotation table) and their relations to each other in the data model would remain the same. Same applies for the analysis where possibly table and attribute names would need to be customised but the defined components would remain the same. It is also possible to use the generated event log with other process mining tools than the selected. The event log created is in a general form so that it can be used by any PMT.

5.1.4. Ubiquity

The in section 2.1 described characteristics of microservice architectures make distributed tracing becoming a standard tool for gaining visibility during the development and operation of microservice architectures [63]. With this source data for an event log generation already at hand, process mining techniques for instrumented systems could be applied relatively fast. Moreover, costly initiatives to instrument a system with custom business logging functionality would become obsolete.

5.1.5. Flexibility on process perspectives

The *case id*, as already discussed in subsection 2.3.5, defines the perspective from how business activities are viewed and correlated into one sequence, which determines *the process*. For the sake of analysing user paths of a mocked car sharing application, a *session id* was chosen as the *case id* in the proof-of-concept SUS, which constitutes a distinct session that expires within a defined time interval. Using a *user id* instead, would create life-long running processes that span across one session for instance.

The distributed tracing instrumentation per default records inter-process communic-

ation for instrumented services. An initial request is correlated with its downstream request to a unique trace, while its source is often a user activity. However, the presented approach is not limited on analysing user paths and subsequent business transactions.

Imagine for instance a credit application scenario where a customer's application is followed by a semi-automated workflow that includes manual as well as system activities that together constitute a process. An appropriate *case id* for this scenario would be for instance an *application id*. Let us assume that an user activity is somehow related to a manual request that arrives at any front-end. In order to correlate this activity now to a sequence of other activities that belong to the distinct credit application process, the initial request needs to be annotated with the *application id*, like it was the case in the instrumented SUS. A domain expert, could now decide if subsequent requests between services should also appear in the process or not. This is initially defined in the mapping table. The log generation algorithm could also be customised, so that only a set of inter-service requests define what a business relevant activity is and if it will be part of the model. For correlating different traces, it is sufficient that only one span of the trace is annotated with a case id, since all spans are already correlated through the *trace id*. As a consequence, the *case id* does not necessarily need to be annotated for the initial request at the front-end, and could be added at any later point where its availability is assured.

Scheduled activities or such that start through any other trigger than a user, can be instrumented in the same way. Instead of distinguishing between user and system activities, as it has been done done in the prototype, one could create activities of one type only or differentiate between automated and manual activities. It is moreover possible to manually define spans at a method level, which makes it possible to define activities with a even higher level of granularity.

5.1.6. Foundation for real-time process mining

The approach presented in this work, and thereby especially the event log algorithm, can serve as a foundational strategy to generate event logs in order to discover process models from microservice architectures in real-time. For that, feasible adjustments, that will be described below (subsection 5.2.4), would have to be made in order to realise real-time business process discovery.

5.1.7. Bottom-up process discovery in legacy systems

The presented approach is not limited to *greenfield* implementations. Distributed systems that do not provide business event log functionalities can be instrumented with a tracing library retrospectively. Therefore, business process models from legacy systems, where documentation is often missing, could be derived with the presented approach.

5.2. Limitations of the proposed solution

In the following, limitations of the proposed solution are discussed. One domain of limitations arise through the employed SUS. Other limitations can be attributed to the areas of visualisation, sampling rates and real-time usage.

5.2.1. System under survey

A general limitation that appears is evaluating the approach with one SUS only. Therefore not every scenario that would appear in a real world system can be captured. For instance, only failures that are producible in the SUS (e.g. through shutting down services) could be captured. This does not necessarily cover the full scope of possible system failure. The following domains contribute to a general bias that emerges through using a laboratory SUS.

Software architecture The content of the tracing data is dependent on the architecture of the instrumented system [41]. The log generation algorithm is customised to generate activities that are specific to the architecture (e.g. what departs a healthy from a failed activity). To overcome this problem, common patterns and best practises in developing microservice architectures, as for example a centralised configuration management, service discovery, client-side load balancing and tunnelling user requests through an API gateway have been applied, to provide an architecture that is as close to a real world setting as possible. During the different phases of developing the system, the event log algorithm was customised multiple times to adhere to the changes of the system. During theses iterations it became clear that the approach is still valid if architectural alterations appear, even if little changes have to be applied to the original algorithm.

Data volume Limitation through using a laboratory SUS also appear in terms of data volume produced. Since user requests were performed manually, the amounts of tracing and event log data were comparably low to a real-world setting, where possibly millions of requests per day occur and huge amounts of tracing data is generated. The system could therefore not be tested under heavy loads and high volumes. Since architectural bottlenecks of the proposed solution are apparent, a solution that adheres to high data volumes is presented in subsection 5.2.4.

Data contents The tracing data was created through simulating user clicks in a front-end that was designed for the purpose of generating data. Therefore, the user activities created do not display any real user interaction (e.g. cycle times between the begin of a car rental (*Unlock car*) end the end (*End car rental*) do not represent realistic time frames). Since the objective of the work is not on analysing click streams from a real data set in order to discover real insight but in presenting a tool that enables these analysis questions, this limitation is less important.

5.2.2. Suitability of applied process visualisations

One of the objectives of this work was to find correlations between business and system behaviour. Therefore, one strategy applied (see subsection 4.6.3), was to model business transactions (system activities) together with the triggering user activity in the process model. Nested business transactions (i.e. system activities), where interpreted as a sub-processes of a user activity and translated into fuzzy model visualisation.

The applied fuzzy model visualisation, is in general very suitable in the contexts of user click path mining [30]. Nevertheless, it does not provide sufficient means for visualising technical sub-processes, as it will be shown below. This visualisation approach comes with the following limitations.

Ambiguous visualisation of distinct system activities Petri nets display distinct activities (identifiable through an distinct activity name) only once in a process model. System activities, as for example a */notifyUser* activity, is used for multiple user activities (see section 3.2), typically at the end of a business transaction. This leads to confusion in the process graph, since it is not clear where the activity is placed. Moreover, the model becomes complex through the fact that a system activity has many connections to predecessor and successor activities. This issue can be seen in Figure 5.1, where the */notifyUser* activity is placed in the sequence of a *Reserve car* user activity. This is only due to the fact, that the *Reserve car* activity in total was called more often then for example the *Book car* or *End car rental* activity, that also makes use of *notifyUser*. Therefore, the algorithm puts the */notifyUser* close to the *Reserve car* activity.

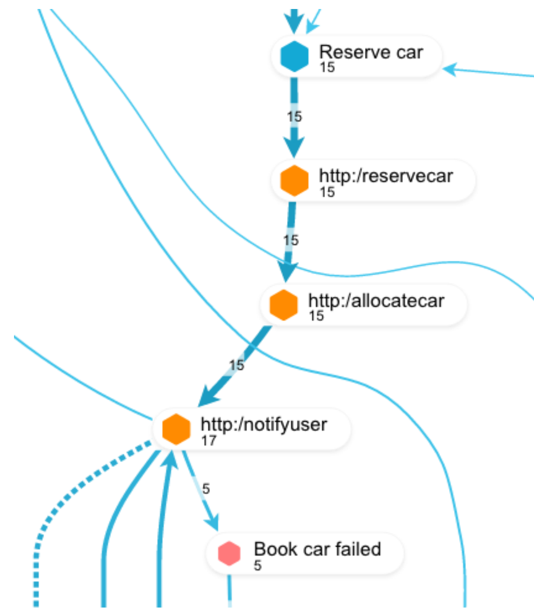


Figure 5.1.: Ambiguous placement of system activities

To overcome this problem, system activities could have been made distinct, e.g. through giving them a prefix from its calling user activity. A downside of that approach is, that system activities would not be comparable among themselves.

Visualisation of hierachies A general issue that occurs through modelling business transaction as system activities is the fact, that representing nested traces in a petri net-like process model leads to ambiguity. In the example (Figure 5.2) see a trace of a /bookPackage request from the front-end. In this synchronous request/response mechanism, the calling service blocks and waits for an answer of its downstream services.

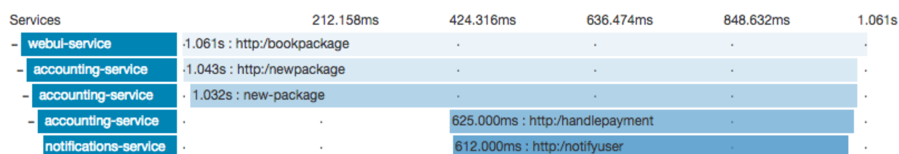


Figure 5.2.: Waterfall-like visualisation of a trace in *Zipkin*, triggered through the /bookPackage request from the front-end

Now when visualising the trace in a petri net-like process model questions like *In which order should activities be modelled?* or *What do durations between two system activities stand for?*. See Figure 5.3, where system activities are displayed as a process in the consecutively order of how they are called.

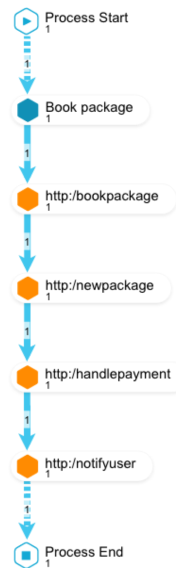


Figure 5.3.: Process visualisation of a the *Book package* user activity and the called system activities in the PMT

Through comparing the two visualisations with the same underlying system behaviour, one can conclude that waterfall diagrams provide a more precise and content-rich presentation of system behaviour that are moreover scaling better. Hierarchies, parallelism and aspects of timing are easier to convey in a waterfall representation compared to the process diagram. Especially for finding perpetrators of latency and a a critical path waterfall diagrams are more suitable.

During the creation of the event log algorithm, multiple alternative visualisations were tested in the PMT.

Splitting one activity into a start and end activity One alternative way of modelling would be to generate separate start end end activities instead of one activity per request as it is the case in the proof-of-concept. This would increase the accuracy especially for system activities, since the nesting of inter-service request would become observable. This approach comes also with disadvantages. It would for instance double the amount of activities which leads to undesired complexity in the process model. Moreover, duration and performance could not be displayed per activity anymore, since the activities would only represent a timestamp without a duration.

Grouping activities One feature the PMT provides is defining groups of activities. Every user activity could be grouped together with its system activities. By clicking on the group, all activities would be expanded. A problem that arises

again, is that some system activities are used in more than one sub-process. Nevertheless one could fix this, by generating distinct system activities that only belong to one user activity which comes with the before described drawbacks.

Cropping parts of the process Another option for analysing user activities and their followed system activities conjunct is to crop parts of the process. What fuzzy models do, is to generalise an event log and generate a model that gives insights into the most common paths a process contains, with the option to zoom to derive a detailed view. Cropping the process starting with the to be analysed user activity to the last system activity, one could analyse the typical path of a business transaction. This makes especially sense for failed activities (e.g. to analyse which failed technical activity is the root cause for a failed user activity) or for architectures where a user activity is followed by diverse sequence of system activities, and not a static sequence as it is the case in the SUS.

Waterfall component and process visualisation A further visualisation alternative would be, to introduce a new component that is able to represent traces in a hierarchical waterfall model as in *Zipkin*. It would respond through clicking on a user activity in the click path model and show a generalised view for all traces that exist for the activity. A *typical* trace with a generalised view would give insights beyond a single trace analysis. The waterfall component would need to be adopted to display average durations together with lower and upper bounds. For user activities that are followed by varying business transactions, a visualisation type is required that is capable of displaying different variants of a trace.

5.2.3. Performance overhead through high sampling rates

On of the main design goals for distributed tracing was to provide minimal instrumentation overhead [64]. Sampling all traces from a system of a certain size would create enormous traffic. For limiting the amount of sampled traces, instrumentations provide different sampling strategies.

Spring Cloud Sleuth uses a probabilistic approach that samples only fixed fractions of spans. This does not mean that span data is not created, but rather makes sure that no tags and events are being attached and exported to the collector, which creates most of the overhead [18].

Instead of tracing a fixed amount of traces, Uber's jaeger[1] instrumentation for example provides more sophisticated sampling strategies, that use a rate limiting approach, where a fixed amount of traces is sampled per time unit [63]. Therefore the rate is dynamic and adheres to the changing traffic of the service.

[1]https://github.com/uber/jaeger

In the proof-of-concept prototype, the sampling rate is set to 100% for all instrumented service. When decreasing the amount of sampled traces, user click flows that represent a single session would miss certain user activities completely, since only a fraction of traces (and therefore user activities) is sampled. For using the approach in a productive setting, a sample rate of 100% would lead to latency issues and huge amounts of sampling data that needs to be processed. Therefore, the sampling strategy would need to be adapted so that a Tracer recognises if a trace belongs to a spanning case and could therefore recognise if it needs to be recorded or not. With this customisation, it would be possible to not sample on a trace but on a case level. Depending on the analysis questions to be answered, analysing a fraction of cases only could be sufficient for identifying patterns.

5.2.4. Real-time event handling, event log generation and process discovery

In the proof-of-concept, span data is sent via REST to the *Zipkin* collector that persists the traces in a *MySQL* database. The log generation service uses the trace data to create and write a whole new event log every 5 minutes. The service can be configured to execute the scripts flexibly but is restricted to the time it takes for executing the scripts and persisting the results in the *MySQL* database. Moreover, a constraint exists through the PMT, that can only be configured to reload the data model at most every 5 minutes.

Oliveira [48] describes an approach for discovering business processes in real-time, that could be applied to the proof-of-concept architecture of this work, too. Especially the two first phases *event production* and *event processing* are areas of influence, since the PMT itself is a third-party implementation.

Event production Alternatively to collecting traces data via REST and persisting them in a *MySQL* database, span data could also be published via a stream binder like *RabbitMQ* or *Apache Kafka* using *Spring Cloud Stream*.

Event processing As a consumer of the events stream, a so called Event Stream Processing (ESP) engine, like the open source tool *Esper*[2], could be implemented in the log generation service of the proof-of-concept architecture to query the data and continuously produce events. An ESP can be used to store static query expressions in an application. Compared to database querying with SQL, ESPs work with data flows that replace tables. Instead of tuples, events present the basic unit of data [19]. Static query expressions, that have similar language constructs as in SQL, can be stored in the application, while continuously querying the data and generating activities of the event log. The output again is published to an event stream. Since ESP works with similar language constructs as SQL, the presented log generation algorithm could be adapted and used for generating

[2] http://www.espertech.com/products/esper.php

business process activities. A restriction appears due to limited RAM, since for the generation of activities, it is necessary to store the events in-memory. Strategies that address the challenge are discussed in [4].

Process discovery A PMT would need the capabilities to subscribes to events streams that are created by the ESP component in order to generate process models in real-time. In [23, 33, 24] various scalable approaches to discover processes from streams of events in big data settings are presented, that apply existent process mining algorithms in map-reduce implementations.

As of today, no open-source or commercial tool exists that is capable of generating event models from event streams in real-time.

5.3. Related work

In this section, the proof-of-concept prototype is evaluated against related work. Although an extensive literature review was conducted, that manifested the need for a multi-layer approach [13, 9], only little work was found that applies the techniques of process mining on tracing data in order to provide a multi-layer view on a process.

In the following, the main characteristics of five research contributions (see Table 5.2) are classified and described. The analysed literature fulfils at least two of the three criteria stated below.

1. The research contribution generates run-time process models through process discovery techniques from process mining.
2. The research contribution uses system logs as an input source for the event log generation.
3. The research contribution captures user or business as well as system behaviour in the analysis.

For structuring and making the related work comparable, a classification framework was developed that is described in Table 5.1.

Table 5.1.: Description of the classification framework for related work

Classifier	Description
Log origin	*Describes the technical log origin type that is used as an input for generating the event log.*
Activity types	*Describes what types of activities are used in the process.*
Captured behaviour	*Describes what type(s) of behaviour is analysed by the approach.*
Type of work	*Describes the applied methodology of the work.*
Evaluation environment	*Describes the evaluation environment of the approach.*
System architecture	*Describes for which system architectures the approach can be applied.*
Language independency	*Describes if the approach is restricted to a certain programming language.*
(Near) real-time	*Describes if the approach can process data in real- or near real-time.*

Poggi et al. [52] use web logs to generate a user click paths. In their work they discuss challenges and benefits for using process mining to analyse user behaviour. The authors provide a methodology for classifying and transforming URLs into business activities. They also evaluate different process mining algorithms using real-life data from a e-commerce web shop. Thereby they apply a single-layer perspective where the only focus is on analysing user behaviour with neglecting system performance and the underlying business transactions involved.

Abe and Kudo [7] present a monitoring framework that also discovers business processes from real-life web logs. In their framework, the *viewpoint* of the visualised workflows can be changed dynamically. Contrary to the definition of *viewpoint* applied in this work, which is defined as the layer perspective on a system, the authors interpret it as for instance a temporal aspect (e.g. process performance on weekday vs. weekend). As it is the case for [52], no cross-layer metrics or process visualisations are provided. The captured behaviour only focuses on the business layer.

Brückmann et al. [13] describe an approach that enables the real-time monitoring and controlling of an EA. The authors adapt the "control centre" concept, known from designing power plant control centres, as an approach for designing the system in order to provide a real-time view of business process instances together with performance indicators of involved software systems and services. The proposed solution supports IT operators in controlling the loads of software services in order to control the flow of instances of business services. The authors describe several functional components that

the solution should contain and propose a concept that describes how an architecture with the presented functionality should look like, without providing a prototypical implementation. Moreover, the authors approach departs from the approach presented in this work since it is focusing on controlling and monitoring an EA, while this works provides means of finding correlations between performance of the business and application layer, without providing functionality to control process execution or application performance actively.

Other authors propose solutions that try to establish a cross-domain monitoring without providing a process visualisation. These approaches rather focus on single KPIs and local decision making [13]. Therefore, a gap appears between the identification of a process weakness, which might be achievable with existing tools, and finding a root cause for the undesired behaviour, since it is not possible to analyse and visualise processes from an end-to-end perspective [66].

Leemands and van der Aalst [41] also present a *bottom-up* or *reverse engineering approach* to discover operational processes from event logs of a distributed system that shares a lot of similarities with the approach presented in this work. The formal models generated through the approach provide a "integrated view, across system components, and across perspectives (performance, end-to-end control flow, etc)" as it is also the case for the approach presented in this work. The author's work also provides an implementation as well as an instrumentation strategy, where the latter one is based on the joinpoint-pointcut model from Aspect-oriented Programming (AOP). The strategy aims to add as little implementation effort and instrumentation impact on the system as possible. Correlation of system events is not accomplished through a trace id but through a shared communication channel between two nodes. The level of granularity achieved from the instrumentation is on method call level and can be specified to provide a even more detailed context information through specifying interface pointcuts. One of the main differences to this work is, that their content of analysis lies in business transactions, defined as the communication *between* and *in* components of a system which are triggered to a specific user request. Using the vocabulary from this work, only the application layer processes with system activities are analysed. The higher level of interplay between user activities on level higher are not part of the models generated. Therefore no correlations between layer of an EA can be detected.

Ruben et al. [59] present an approach, that uses techniques of process mining for creating feedback loops during agile software development. Their bottom-up approach takes low level events from a software system in order to analyse runtime behaviour of the system and user behaviour, for improving the design and usability of the system together with performance aspects like latency. After each iteration in the development cycle, the system is tested by users. The tests are analysed in retrospective with the techniques of process mining to visualise user click paths and system latency in a process graph. Thus, a new feedback channel is created, that allows early user feedback.

As in the approach of this work, user behaviour as well as system performance are analysed combined. To obtain the detailed data, an existing instrumentation has been extended by an inbound adaptor, to capture user as well as system runtime behaviour on a low level.

In summary it can be stated that only Ruben et al. [59] provide a solution that combines a multi-layer view on a process together with low level system logs as data input. Other approaches either do not capture behaviour form multiple layers [53, 7, 41], or do not use the techniques of process mining [13]. Moreover only [41] provides a prototypical implementation including an instrumentation strategy that is independent from the systems programming language(s) and applicable in distributed systems.

Table 5.2.: Classification of related work

	Log origin	**Activity types**	**Captured behaviour**	**Type of work**	**Evaluation environ-ment**	**System architecture**	**Language independ-ence**	**(Near) real-time**
Poggi et al. [53]	Web logs	User Activities	Business	Algorithm evaluation	Real-life event logs	No	n/a	No
Abe & Kudo [7]	Web logs	User activities	Business	Framework	Real-life event logs	n/a	n/a	No
Bruckmann et al. [13]	n/a	User Activities, system activities	Business and system	Architecture proposal	n/a	n/a	n/a	Yes
Leemans & van der Aalst [41]	Joinpoint-pointcut model instru-mentation	User activities	System	Instrumentation strategy, implementation	Real-life event logs	Yes	Yes	No
Rubin et al. [59]	Custom instru-mentation	User activities, system activities	User and system	Experimental	Real-life event logs	No	n/a	No
Proof-of-concept prototype of this work	Distributed tracing instru-mentation	User activities, system activities	Business, user and system	Instrumentation strategy, architecture description, implementation	Simulated user requests on testing system	Yes	Yes	Yes

6. Summary and outlook

This last chapter summarises the thesis, highlights potential for solution extensions and provides open research questions that have emerged throughout the work.

The objectives addressed by the approach are twofold. First, it has been demonstrated that distributed tracing data from an instrumented microservice architecture generates suitable low level event logs. These were then correlated to a high level event log which is used to discover a user-centric business process. This finding contributes to the area of process mining, in which alternative sources for an event log generation are constantly evaluated. Furthermore, the presented approach provides a strategy for correlating activities in systems that do not have the notion of a process natively implemented. Process discovery in practice is often applied to information systems that make use of already implemented audit trails (e.g. in WFM). Another strategy is to create activities by analysing databases that store information about cases along with *change tables*, that log any data manipulation related to a single case (e.g. in SAP ERP). In the presented approach, such capabilities are not present for the SUS. The notion of a case is introduced through annotating user requests with an appropriate *case id*. In the context of analysing user click paths in a front-end, the *session id* was chosen as a *chase id*. The approach thereby provides a strategy for systems in which no audit logs are written or *case ids* can be found in the databases of the system. On top of that, this strategy comes with limited implementation effort.

As a second main objective it has been shown, how the fine grained distributed tracing data enables the extension of process mining towards a more holistic view of the application and business layer of an EA. This has been accomplished through two strategies: First, by utilising technical performance data, expressed in form of the duration required for processing a user activity in the back-end. This leads to a clear distinction of throughput times needed for the user to navigate through a process and the time the system needs to handle a request. Second, it has been accomplished with the introduction of user and system activities, that enable the combined view of how a user interacts with the front-end and how the system processes the requests through its services in the back-end.

The two objectives were achieved through the following steps that are embedded in a design science research methodology.

In the first chapter, the problem has been motivated and formulated in the form of three

research questions. Moreover, the context of this work and its relation to the overarching TUM LLCM project was introduced. In addition, the design science approach was introduced as the applied research methodology, which includes the implementation and evaluation of a prototype as guiding elements for answering the stated research questions.

In chapter two, the foundations for microservices, distributed tracing and process mining were laid. A microservice architecture, as an architectural style that is adopted by more and more organisations, presents the technical setting in which the work tries to reach its objectives. Subsequently, an understanding of the techniques and specifics of both distributed tracing and process mining was given, since they serve as core elements in the approach.

In the third chapter, a prerequisite SUS was built, that encompasses state-of-the-art patterns for building microservice architectures.

This SUS has been instrumented and extended as described in chapter four. The setup includes the instrumented system and three extended components: a distributed tracing service, a log generation service as well as the configured process mining tool. Moreover, a central database including a data model has been deployed to which all extended services connect. The chapter also describes the workflow of how user activities are performed, captured, transformed to system and user activities and finally loaded in the PMT. Finally, the four created analysis in the PMT are described. Each answers a set of unique analysis questions. They serve to proof the applicability of the approach and correctness of process models generated as well as enable the discovery of shortcomings of the proposed solution.

In the fifth chapter, the presented approach is evaluated. Thereby, benefits and limitations that became apparent during the development process are presented. Main benefits include the shown capabilities of analysing a process from a cross-domain perspective as well as the resource-efficient strategy of employing distributed tracing as a source for the event log generation. Moreover, the portability, meaning how flexible the approach can be transferred to other architectures as well as the ubiquity of the solution are discussed. Furthermore it is reasoned, how perspectives on a process can be easily changed by altering the *case id*. Multiple limitations arise, that are naturally present through applying the approach to one SUS only. Moreover, possible alternative visualisations are discussed together with their drawbacks. It is stated, that further visualisations need to be developed in order to better display the relationship between user and system activities. The third limitation is concerned with performance overhead issues that appear through sampling every request in the system. For capturing every trace of a process, new sampling strategies need to be developed. The last limitation covers the issues of real-time event generation and bottlenecks that the presented prototype presents. These issues restrict the application from real-time to *near* real-time. The third part of chapter 5 relates the presented approach to work from other authors.

A classification scheme has been applied to make related work comparable. It has been found out that little approaches exists that provide a working prototype of similar scope and functionality as presented in this work.

Resulting from the above described limitations and challenges that have been addressed in section 5.2, further research needs to be conducted regarding to visualising business and user activities in a holistic view. Moreover, performance issues, that appear due to a high sampling rate present a major obstacle to the approach. A sampling strategy needs to be developed, that is able to sample traces on a case level. Additionally, the approach needs to be applied to more architectures in order to verify its general applicability in greater depth.

Furthermore, the solution could be extended towards a real-time process discovery, which has not been the focus of this work. Other authors [48, 23, 33, 24] have already described approaches that try to address this challenge.

Bibliography

[1] W. van der Aalst, A. Adriansyah, A. K. A. de Medeiros, F. Arcieri, T. Baier, T. Blickle, J. C. Bose, P. van den Brand, R. Brandtjen, J. Buijs, A. Burattin, J. Carmona, M. Castellanos, J. Claes, J. Cook, N. Costantini, F. Curbera, E. Damiani, M. de Leoni, P. Delias, B. F. van Dongen, M. Dumas, S. Dustdar, D. Fahland, D. R. Ferreira, W. Gaaloul, F. van Geffen, S. Goel, C. Günther, A. Guzzo, P. Harmon, A. ter Hofstede, J. Hoogland, J. E. Ingvaldsen, K. Kato, R. Kuhn, A. Kumar, M. La Rosa, F. Maggi, D. Malerba, R. S. Mans, A. Manuel, M. McCreesh, P. Mello, J. Mendling, M. Montali, H. R. Motahari-Nezhad, M. zur Muehlen, J. Munoz-Gama, L. Pontieri, J. Ribeiro, A. Rozinat, H. Seguel Pérez, R. Seguel Pérez, M. Sepúlveda, J. Sinur, P. Soffer, M. Song, A. Sperduti, G. Stilo, C. Stoel, K. Swenson, M. Talamo, W. Tan, C. Turner, J. Vanthienen, G. Varvaressos, E. Verbeek, M. Verdonk, R. Vigo, J. Wang, B. Weber, M. Weidlich, T. Weijters, L. Wen, M. Westergaard and M. Wynn. 'Process Mining Manifesto'. In: *Business Process Management Workshops*. Ed. by S. Rinderle-Ma, S. Sadiq and F. Leymann. Vol. 43. Lecture Notes in Business Information Processing August. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 169–194. DOI: 10.1007/978-3-642-28108-2_19.

[2] W. M. P. van der Aalst. 'Do petri nets provide the right representational bias for process mining?' In: *CEUR Workshop Proceedings* 725 (2011), pp. 85–94. ISSN: 16130073.

[3] W. M. P. van der Aalst. 'Extracting Event Data from Databases to Unleash Process Mining'. In: *BPM - Driving Innovation in a Digital World SE - 8* (2015), pp. 105–128. DOI: 10.1007/978-3-319-14430-6_8. URL: http://dx.doi.org/10.1007/978-3-319-14430-6{_}8.

[4] W. M. P. van der Aalst. *Process Mining - Data Science in Action*. Vol. 5. 2016, pp. 301–317. ISBN: 9783642193453. DOI: 10.1007/978-3-642-19345-3. arXiv: arXiv:1011.1669v3.

[5] W. M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer Berlin Heidelberg, 2011. ISBN: 9783642193453.

[6] W. M. P. van der Aalst, T. Weijters and L Maruster. 'Workflow mining: discovering process models from event logs'. In: *IEEE Transactions on Knowledge and Data Engineering* 16.9 (2004), pp. 1128–1142. ISSN: 1041-4347. DOI: 10.1109/TKDE.2004.47.

[7] M. Abe and M. Kudo. 'Business Monitoring Framework for Process Discovery with Real-Life Logs'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8659 LNCS. 2014, pp. 416–423. ISBN: 9783319101712. DOI: 10.1007/978-3-319-10172-9_30.

[8] P. Agarwal. *Distributed tracing at Yelp*. 2016. URL: https://engineeringblog.yelp.com/amp/2016/04/distributed-tracing-at-yelp.html (visited on 01/08/2017).

[9] F. Ahlemann. *Strategic enterprise architecture management : challenges, best practices, and future developments*. Springer, 2012, p. 298. ISBN: 3642242235.

[10] *Architecture · OpenZipkin*. 2017. URL: http://zipkin.io/pages/architecture.html (visited on 13/07/2017).

[11] J. Bloomberg. *Are Microservices 'SOA Done Right'? - Intellyx*. 2015. URL: https://intellyx.com/2015/07/20/are-microservices-soa-done-right/ (visited on 02/08/2017).

[12] A. Bonham. *Microservices — When to React Vs. Orchestrate – Capital One DevExchange – Medium*. 2017. URL: https://medium.com/capital-one-developers/microservices-when-to-react-vs-orchestrate-c6b18308a14c (visited on 03/08/2017).

[13] T. Brückmann, V. Gruhn, M. Pfeiffer, T. Bruckmann, V. Gruhn and M. Pfeiffer. 'Towards real-time monitoring and controlling of enterprise architectures using business software control centers'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6903 LNCS (2011), pp. 287–294. ISSN: 03029743.

[14] R. Bruns and J. Dunkel. 'Event-Driven Architecture und Complex Event Processing im Überblick'. In: 2010, pp. 47–82. DOI: 10.1007/978-3-642-02439-9_3.

[15] C Carneiro and T Schmelmer. *Microservices From Day One: Build robust and scalable software from the start*. 2016. ISBN: 9781484219379.

[16] D. Chappell. *Enterprise Service Bus*. O'Reilly Series. O'Reilly Media, Incorporated, 2004. ISBN: 9780596006754.

[17] M. Cohen and M. Hawthorne. *Announcing Zuul: Edge Service in the Cloud – Netflix TechBlog – Medium*. 2013. URL: https://medium.com/netflix-techblog/announcing-zuul-edge-service-in-the-cloud-ab3af5be08ee (visited on 01/08/2017).

[18] A. Cole, S. Gibb, M. Grzejszczak and D. Syer. *Spring Cloud Sleuth*. URL: http://cloud.spring.io/spring-cloud-static/spring-cloud-sleuth/1.1.1.RELEASE/ (visited on 01/08/2017).

[19] J Cordeiro, S Hammoudi, L Maciaszek, O Camp and J Filipe. *Enterprise Information Systems: 16th International Conference, ICEIS 2014, Lisbon, Portugal, April 27-30, 2014, Revised Selected Papers*. Lecture Notes in Business Information Processing. Springer International Publishing, 2015. ISBN: 9783319223483.

[20] D'Amore JR. *Scaling Microservices with an Event Stream | ThoughtWorks*. 2015. URL: https://www.thoughtworks.com/de/insights/blog/scaling-microservices-event-stream (visited on 09/08/2017).

[21] A. T. Devi. 'An Informative and Comparative Study of Process Mining Tools'. In: *International Journal of Scientific & Engineering Research* 8.5 (2017), pp. 8–10. ISSN: 2229-5518.

[22] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin and L. Safina. 'Microservices: yesterday, today, and tomorrow'. In: March (2016), pp. 1–17. DOI: 10.13140/RG.2.1.3257.4961. arXiv: 1606.04036.

[23] J Evermann. 'Scalable Process Discovery Using Map-Reduce'. In: *IEEE Transactions on Services Computing* 9.3 (2016), pp. 469–481. DOI: 10.1109/TSC.2014.2367525.

[24] J Evermann, J.-R. Rehse and P Fettke. 'Process discovery from event stream data in the cloud - A scalable, distributed implementation of the flexible heuristics miner on the amazon kinesis cloud infrastructure'. In: *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*. 2017, pp. 645–652. DOI: 10.1109/CloudCom.2016.0111.

[25] D. R. Ferreira and D. Gillblad. 'Discovering process models from unlabelled event logs'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 5701 LNCS. 2009, pp. 143–158. ISBN: 3642038476. DOI: 10.1007/978-3-642-03848-8_11.

[26] R. Fischer, S. Aier and R. Winter. 'A Federated Approach to Enterprise Architecture Model Maintenance'. In: *Enterprise Modelling and Information Systems Architectures* 2.2 (2015), pp. 14–22. ISSN: 1866-3621. DOI: 10.18417/emisa.2.2.2.

[27] M. Fowler. *Microservice Trade-Offs - Martin Fowler*. 2015. URL: https://martinfowler.com/articles/microservice-trade-offs.html{\#}ops (visited on 02/08/2017).

[28] Fowler Martin and Lewis James. *Microservices - a definition of this new architectural term*. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 18/05/2017).

[29] C Günther, A Rozinat, W. van der Aalst and K. van Uden. 'Monitoring deployed application usage with process mining'. In: *BPM Center Report* (2008), pp. 1–8.

[30] C. W. Günther and W. M. P. van der Aalst. 'Fuzzy Mining – Adaptive Process Simplification Based on Multi-perspective Metrics'. In: *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*. Brisbane, Australia, 2007, pp. 328–343. DOI: 10.1007/978-3-540-75183-0_24.

[31] S. Hall. *OpenTracing Aims for a Clearer View of Processes in Distributed Systems - The New Stack*. 2016. URL: https://thenewstack.io/opentracing-aims-clearer-view-processes-distributed-systems/ (visited on 02/08/2017).

[32] C. Heger, A. van Hoorn, M. Mann and D. Okanović. 'Application Performance Management'. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering - ICPE '17*. 2017, pp. 429–432. ISBN: 9781450344043. DOI: 10.1145/3030207.3053674.

[33] S Hernández, J Ezpeleta, S. J. Van Zelst and W. M. P. Van Der Aalst. 'Assessing Process Discovery Scalability in Data Intensive Environments'. In: *Proceedings - 2015 2nd IEEE/ACM International Symposium on Big Data Computing, BDC 2015*. 2016, pp. 99–104. DOI: 10.1109/BDC.2015.31.

[34] A. R. Hevner, S. T. March, J. Park and S. Ram. 'Design Science in Information Systems Research'. In: *MIS Quarterly* 28.1 (2004), pp. 75–105. ISSN: 02767783. DOI: 10.2307/25148625. arXiv: /dl.acm.org/citation.cfm?id=2017212.2017217 [http:].

[35] G. Hohpe and B. Woolf. *Enterprise integration patterns : designing, building, and deploying messaging solutions*. Addison-Wesley, 2004, p. 683. ISBN: 0321200683.

[36] K. Indrasiri. *Microservices in Practice - Key Achitectural Concepts of an MSA*. 2016. URL: http://wso2.com/whitepapers/microservices-in-practice-key-architectural-concepts-of-an-msa/{\#}10 (visited on 02/08/2017).

[37] J. Jackson. *Meet Zipkin: A Tracer for Debugging Microservices - The New Stack*. 2016. URL: https://thenewstack.io/meet-zipkin-tracer-debugging-microservices/ (visited on 13/07/2017).

[38] S. Janser. 'Konfigurationsmanagement für Microservices mit Spring Cloud Config'. In: *heise Developer* (2016). URL: https://www.heise.de/developer/artikel/Konfigurationsmanagement-fuer-Microservices-mit-Spring-Cloud-Config-3200235.html.

[39] M. Kleehaus, Ö. Uludag and F. Matthes. 'Towards a Multi-Layer IT Infrastructure Monitoring Approach based on Enterprise Architecture Information'. In: *2nd Workshop on Continuous Software Engineering co-located with SE 2017*. Vol. i. 2017, pp. 12–17.

[40] M. Lange and J. Mendling. 'An Experts' Perspective on Enterprise Architecture Goals, Framework Adoption and Benefit Assessment'. In: *2011 IEEE 15th International Enterprise Distributed Object Computing Conference Workshops*. IEEE, 2011, pp. 304–313. ISBN: 978-1-4577-0869-5. DOI: 10.1109/EDOCW.2011.41.

[41] M Leemans and W. van der Aalst. 'Process Mining in Software Systems: Discovering Real-Life Business Transactions and Process Models from Distributed Systems'. In: *MODELS 2015*. Ottawa, ON, Canada, 2015, pp. 44–53. ISBN: 9781467369084.

[42] J. Lewis and M. Fowler. *Microservices - Martin Fowler*. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 02/08/2017).

[43] G. Linden. *Geeking with Greg: Marissa Mayer at Web 2.0*. 2006. URL: https://glinden.blogspot.de/2006/11/marissa-mayer-at-web-20.html (visited on 08/08/2017).

[44] J. Mace. *End-to-End Tracing: Adoption and Use Cases*. Tech. rep. Chicago: Brown University, 2017.

[45] B. Michelson. *Event-Driven Architecture Overview*. Tech. rep. Boston, Massachusetts: Patricia Seybold Group, 2006. DOI: 10.1571/bda2-2-06cc.

[46] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015, p. 280. ISBN: 978-1-491-95035-7. DOI: 10.1109/MS.2016.64. arXiv: 1606.04036.

[47] C. Nissen. *[Guide] Implementing API Gateway pattern with Netflix Zuul and Spring Cloud*. 2016. URL: http://kubecloud.io/apigatewaypattern/ (visited on 14/08/2017).

[48] J. J. Oliveira. 'Business Process Discovery in Real Time'.

[49] Pelka Carsten and M. Plöd. *Microservices à la Netflix — Die Bestandteile von Spring Cloud Netflix*. 2016. URL: https://www.innoq.com/de/articles/2016/12/microservices-a-la-netflix/ (visited on 14/08/2017).

[50] S. Perera. *Walking the Microservices Path towards Loose coupling? Look out for these Pitfalls | My views of the World and Systems*. 2016. URL: https://iwringer.wordpress.com/2016/03/11/walking-the-microservices-path-towards-loose-coupling-look-out-for-these-pitfalls/ (visited on 03/08/2017).

[51] N. Poggi, D. Carrera, R. Gavaldà, E. Ayguadé and J. Torres. 'A methodology for the evaluation of high response time on E-commerce users and sales'. In: *Information Systems Frontiers* 16.5 (2014), pp. 867–885. ISSN: 13873326. DOI: 10.1007/s10796-012-9387-4.

[52] N. Poggi, V. Muthusamy, D. Carrera and R. Khalaf. 'Business Process Mining from E-Commerce Web Logs'. In: *BPM'13 Proceedings of the 11th international conference on Business Process Management*. Beijing, China, 2013, pp. 65–80. DOI: 10.1007/978-3-642-40176-3_7. URL: http://link.springer.com/10.1007/978-3-642-40176-3{_}7.

[53] N. Poggi, V. Muthusamy, D. Carrera and R. Khalaf. *Business Process Mining from E-Commerce Web Logs*. 2013. DOI: 10.1007/978-3-642-40176-3.

[54] M. Postina, J. Trefke and U. Steffens. 'An EA-approach to Develop SOA View-points'. In: *2010 14th IEEE International Enterprise Distributed Object Computing Conference*. IEEE, 2010, pp. 37–46. ISBN: 978-1-4244-7966-5. DOI: 10.1109/EDOC.2010.25.

[55] M. Richards. *Microservices vs. service-oriented architecture - O'Reilly Media*. 2016. URL: https://www.oreilly.com/learning/microservices-vs-service-oriented-architecture (visited on 11/08/2017).

[56] Richardson Chris. *Introduction to Microservices*. 2015. URL: https://www.nginx.com/blog/introduction-to-microservices/?utm{_}source=building-microservices-inter-process-communication{\&}utm{_}medium=blog{\&}utm{_}campaign=Microservices (visited on 22/05/2017).

[57] J. W. Ross, P Weill and D Robertson. *Enterprise Architecture as Strategy: Creating a Foundation for Business Execution*. Harvard Business Publishing. Harvard Business School Press, 2006. ISBN: 9781591398394.

[58] A. Rotem-Gal-Oz. 'Fallacies of distributed computing explained'. In: (2006), p. 11.

[59] V. Rubin, I. Lomazova and W. M. P. van der Aalst. 'Agile development with software process mining'. In: *Proceedings of the 2014 International Conference on Software and System Process - ICSSP 2014*. February 2015. 2014, pp. 70–74. ISBN: 9781450327541. DOI: 10.1145/2600821.2600842.

[60] J. Schekkerman and Jaap. *How to survive in the jungle of enterprise architecture frameworks : creating or choosing an enterprise architecture framework*. Trafford, 2004, p. 222. ISBN: 141201607X.

[61] M.-T. Schmidt, B. Hutchison, P. Lambros and R. Phippen. 'The Enterprise Service Bus: Making service-oriented architecture real'. In: *IBM Systems Journal* 44.4 (2005), pp. 781–797. ISSN: 0018-8670. DOI: 10.1147/sj.444.0781.

[62] Shalom Nati. *Amazon found every 100ms of latency cost them 1% in sales. | GigaSpaces Blog*. 2008. URL: http://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/ (visited on 20/06/2017).

[63] Y. Shkur. *Evolving Distributed Tracing at Uber Engineering - Uber Engineering Blog*. 2017. URL: https://eng.uber.com/distributed-tracing/ (visited on 05/08/2017).

[64] B. H. Sigelman, L. Andr, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan and C. Shanbhag. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. April. 2010, p. 14. DOI: dapper-2010-1.

[65] P. Sylvester. *Two Mistakes You Need to Avoid When Integrating Services*. 2016. URL: https://www.infoq.com/articles/integration-mistakes (visited on 09/08/2017).

[66] A. Vera-baquero, R. Colomo-Palacios, O. Molloy, R. Colomo-Palacios and O. Molloy. 'Real-time business activity monitoring and analysis of process performance on big-data domains'. In: *Telematics and Informatics* 33.3 (2016), pp. 793–807. ISSN: 07365853. DOI: 10.1016/j.tele.2015.12.005.

[67] C. Williams. *Is REST Best in a Microservices Architecture? | Capgemini Engineering*. 2015. URL: https://capgemini.github.io/architecture/is-rest-best-microservices/ (visited on 14/08/2017).

[68] O. Wolf. *Benefits of Microservices - Choreography over Orchestration, Low Coupling and High Cohesion*. 2016. URL: https://specify.io/concepts/microservices (visited on 02/08/2017).

[69] Yale Yu, H. Silveira and M. Sundaram. 'A microservice based reference architecture model in the context of enterprise architecture'. In: *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)* (2016), pp. 1856–1860. DOI: 10.1109/IMCEC.2016.7867539.

[70] O. Zimmermann. 'Microservices tenets: Agile approach to service development and deployment'. In: *Computer Science - Research and Development* 32.3-4 (2017), pp. 301–310. ISSN: 18652042. DOI: 10.1007/s00450-016-0337-0.

A. Appendix

A.1. Additional figures and tables

Figure A.1.: *List available cars* activity, triggered through /getCars request

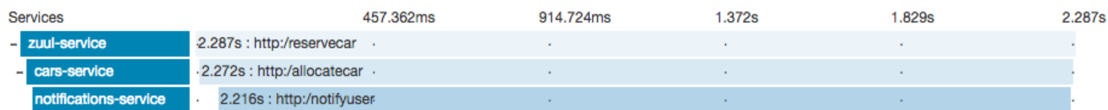


Figure A.2.: *Reserve car* activity, triggered through /reserveCar request

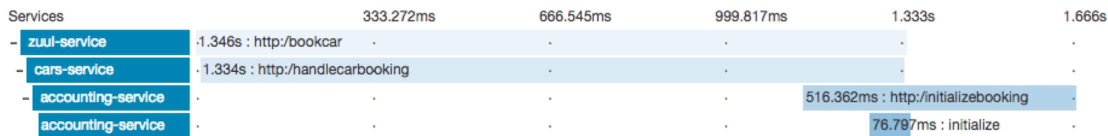


Figure A.3.: *Book car* activity, triggered through /bookCar request

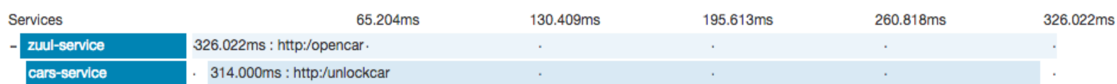


Figure A.4.: *Unlock car* activity, triggered through /openCar request

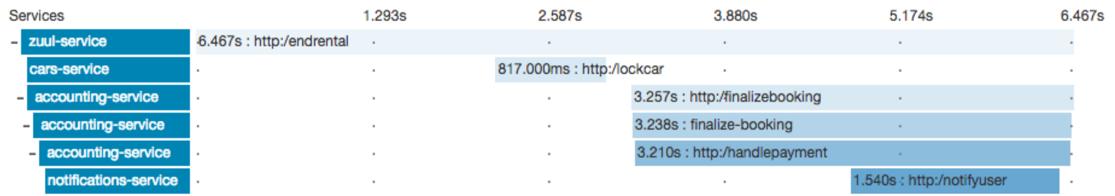


Figure A.5.: *End car rental* activity, triggered through /endRental request

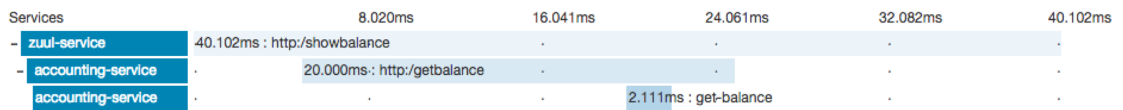


Figure A.6.: *Show balance* activity, triggered through /showBalance request

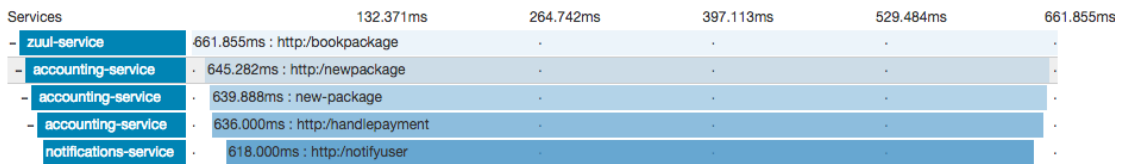


Figure A.7.: *Book package* activity, triggered through /bookPackage request

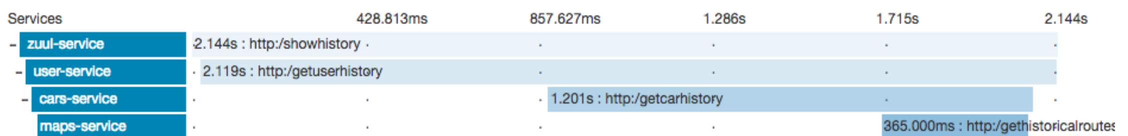


Figure A.8.: *Show driving history* activity, triggered through /showHistory request

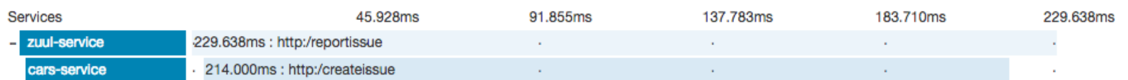


Figure A.9.: *Report issue* activity, triggered through /reportIssue request

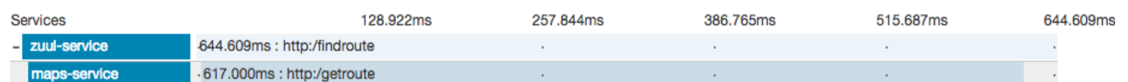


Figure A.10.: *Find route* activity, triggered through /findRoute request

Table A.1.: Sample data of a `/bookCar` request from the *spans* table

trace_id	id	name	parent_id	start_ts	duration
a	a	http:/bookcar	NULL	1498134578920000	621682
a	b	book-car	a	1498134578924000	609215
a	c	http:/bookcar	b	1498134578925000	595000
a	d	book-car	c	1498134578935000	578378
a	e	http:/handlebooking	d	1498134578956000	568670
a	f	handle-booking	e	1498134578978000	521484

Table A.2.: Sample data of a `/bookCar` request from the *annonations* table

trace _id	span _id	a_key	a_value	a_timestamp	endpoint _service_name
a	a	sr		1498134578921000	webui
a	b	lc	unknown	1498134578924000	webui
a	b	mvc.controller.class	WebUIController	1498134578924000	webui
a	b	mvc.controller.method	bookCar	1498134578924000	webui
a	b	SessionID	vjddkk70r03s6lqin86prfc4of	1498134578924000	webui
a	c	http.host	localhost	1498134578925000	webui
a	c	http.method	GET	1498134578925000	webui
a	c	http.path	/bookCar	1498134578925000	webui
a	c	http.url	http://localhost:9090/bookCar	1498134578925000	webui
a	c	sa		1498134578925000	webui
a	c	cs		1498134578926000	webui
a	c	sr		1498134578932000	carmanagement
a	d	lc	unknown	1498134578935000	carmanagement
a	d	mvc.controller.class	CarManagementController	1498134578935000	carmanagement
a	d	mvc.controller.method	bookCar	1498134578935000	carmanagement
a	e	http.host	localhost	1498134578936000	carmanagement
a	e	http.method	GET	1498134578936000	carmanagement
a	e	http.path	/handleBooking	1498134578936000	carmanagement
a	e	http.url	http://localhost:6060/handleBooking	1498134578936000	carmanagement
a	e	sa		1498134578936000	carmanagement
a	e	cs		1498134578937000	carmanagement
a	e	sr		1498134578956000	accounting
a	f	lc	unknown	1498134578978000	accounting
a	f	mvc.controller.class	AccountingController	1498134578978000	accounting
a	f	mvc.controller.method	handleBooking	1498134578978000	accounting
a	e	cr		1498134579511000	carmanagement
a	e	ss		1498134579514000	carmanagement
a	c	cr		1498134579521000	webui
a	c	ss		1498134579525000	accounting
a	a	ss		1498134579542000	webui

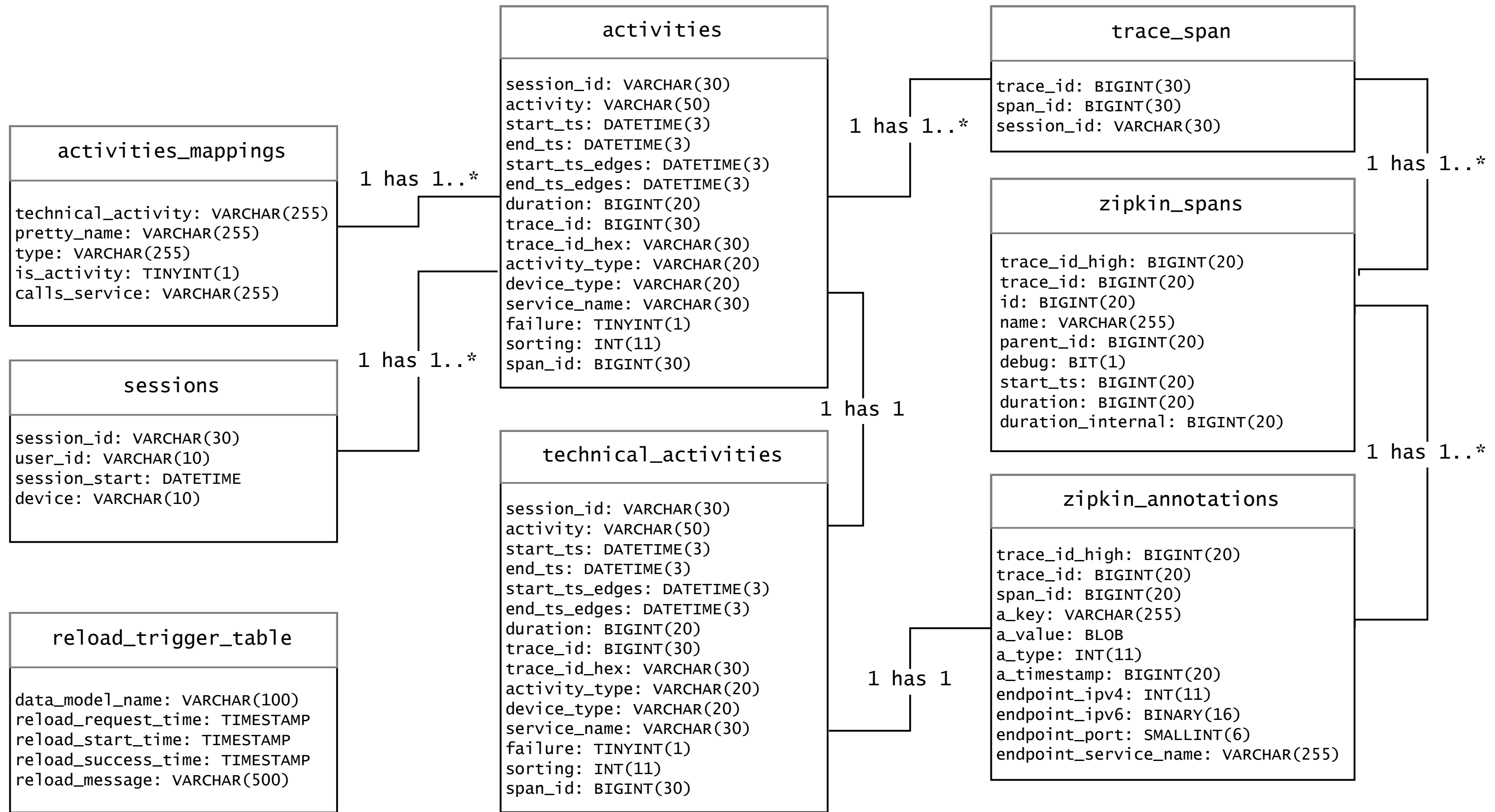


Figure A.11.: Complete data model in UML notation

A.2. Technical documentation of analyses in the PMT

Business Analysis (BA)

Component	**BA_3**
Title	Development of Conversions vs. # Sessions
Filter	-
Dimensions	**Eventtime in Days** `ROUND_DAY("sessions"."session_start")`
KPIs	**Conversion Rate** `SUM(CASE WHEN "activities"."activity" LIKE 'Book car' THEN 1.0 ELSE 0.0 END)/COUNT_TABLE("activities_CASES")` **# Sessions** `COUNT_TABLE("activities_CASES")` **# Bookings** `SUM(CASE WHEN "activities"."activity" LIKE 'Book car' THEN 1.0 ELSE 0.0 END)`

Component	**BA_4**
Title	Activities' conversion
Filter	`FILTER "activities"."activity" NOT LIKE 'Book car';FILTER "activities"."activity" NOT LIKE 'http:/%';`
Dimensions	**Activity Name** `"activities"."activity")`
KPIs	**# Sessions** `COUNT_TABLE("activities_CASES")` **# Car bookings** `SUM(CASE WHEN "activities"."activity" LIKE 'Book car' THEN 1.0 ELSE 0.0 END)` **# Package Bookings** `SUM(CASE WHEN "activities"."activity" LIKE 'Book package' THEN 1.0 ELSE 0.0 END)` **Conversion Rate** `AVG(CASE WHEN PROCESS EQUALS 'Book car' THEN 1.0 ELSE 0.0 END)`

Component	**BA_5**
Title	Bookings and Sessions
Filter	-
Dimensions	**User** <%= business_dim %>
KPIs	**# Sessions** COUNT_TABLE("activities_CASES") **# Car bookings** SUM(CASE WHEN "activities"."activity" LIKE 'Book car' THEN 1.0 ELSE 0.0 END) **# Package Bookings** SUM(CASE WHEN "activities"."activity" LIKE 'Book package' THEN 1.0 ELSE 0.0 END)

Component	**BA_6**
Title	Durations between user activities
Filter	-
Dimensions	**Throughput Time** `CASE WHEN CALC_THROUGHPUT(FIRST_OCCURRENCE['<%= a_from %>'] TO FIRST_OCCURRENCE['<%= a_to %>'], REMAP_TIMESTAMPS("activities"."start_ts", SECONDS)) <= 5 THEN '0 - 5 s' WHEN CALC_THROUGHPUT(FIRST_OCCURRENCE['<%= a_from %>'] TO FIRST_OCCURRENCE['<%= a_to %>'], REMAP_TIMESTAMPS("activities"."start_ts", SECONDS)) <= 10 THEN '5 - 10 s' WHEN CALC_THROUGHPUT(FIRST_OCCURRENCE['<%= a_from %>'] TO FIRST_OCCURRENCE['<%= a_to %>'], REMAP_TIMESTAMPS("activities"."start_ts", SECONDS)) <= 15 THEN '10 - 15 s' WHEN CALC_THROUGHPUT(FIRST_OCCURRENCE['<%= a_from %>'] TO FIRST_OCCURRENCE['<%= a_to %>'], REMAP_TIMESTAMPS("activities"."start_ts", SECONDS)) <= 20 THEN '15 - 20 s' WHEN CALC_THROUGHPUT(FIRST_OCCURRENCE['<%= a_from %>'] TO FIRST_OCCURRENCE['<%= a_to %>'], REMAP_TIMESTAMPS("activities"."start_ts", SECONDS)) <= 25 THEN '20 - 25 s' WHEN CALC_THROUGHPUT(FIRST_OCCURRENCE['<%= a_from %>'] TO FIRST_OCCURRENCE['<%= a_to %>'], REMAP_TIMESTAMPS("activities"."start_ts", SECONDS)) <= 30 THEN '25 - 30 s' ELSE '> 30' END)`
Dimensions	**Case count** `COUNT_TABLE("activities_CASES"))`

Application Analysis (AA)

Component	**AA_3**
Title	Microservice processing success rate
Filter	-
Dimensions	**Eventtime in years** `ROUND_WEEK("sessions"."session_start")`

KPIs	**accounting service** `1-(SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 1 AND "activities"."service_name" = 'accounting-service' THEN 1.0 ELSE 0.0 END)/ SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 0 AND "activities"."service_name" = 'accounting-service' THEN 1.0 ELSE 0.0 END))` **payments service** `1-(SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 1 AND "activities"."service_name" = 'payments-service' THEN 1.0 ELSE 0.0 END)/ SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 0 AND "activities"."service_name" = 'payments-service' THEN 1.0 ELSE 0.0 END))` **maps service** `1-(SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 1 AND "activities"."service_name" = 'maps-service' THEN 1.0 ELSE 0.0 END)/ SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 0 AND "activities"."service_name" = 'maps-service' THEN 1.0 ELSE 0.0 END))` **notifications service** `1-(SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 1 AND "activities"."service_name" = 'notifications-service' THEN 1.0 ELSE 0.0 END)/ SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 0 AND "activities"."service_name" = 'notifications-service' THEN 1.0 ELSE 0.0 END))` **cars service:** `1-(SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 1 AND "activities"."service_name" = 'cars-service' THEN 1.0 ELSE 0.0 END)/ SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 0 AND "activities"."service_name" = 'cars-service' THEN 1.0 ELSE 0.0 END))`

	user service `1-(SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 1 AND "activities"."service_name" = 'user-service' THEN 1.0 ELSE 0.0 END)/ SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 0 AND "activities"."service_name" = 'user-service' THEN 1.0 ELSE 0.0 END))` **webui service** `1-(SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 1 AND "activities"."service_name" = 'webui-service' THEN 1.0 ELSE 0.0 END)/ SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 0 AND "activities"."service_name" = 'webui-service' THEN 1.0 ELSE 0.0 END))`

Component	**AA_4**
Title	Most used & failed microservices
Filter	`FILTER "activities"."activity_type" LIKE 'system';`
Dimensions	**Service Name** `"activities"."service_name"`
KPIs	**# failed requests** `COUNT(SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 1 THEN 1.0 ELSE 0.0 END) )` **# total requests** `COUNT("activities"."activity")`

Component	**AA_5**
Title	Technical activity performance
Filter	`FILTER "activities"."activity" LIKE '/';`
Dimensions	**activity** `"activities"."activity"`
KPIs	**# Requests** `COUNT("activities"."activity")` **AVG duration** `AVG("activities"."duration")`

Component	AA_6
Title	Microservice SLA
Filter	`FILTER "activities"."activity" LIKE '/';`
Dimensions	**service_name** `"activities"."service_name"`
KPIs	**# Requests** `COUNT("activities"."activity")` **# failed requests:** `SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 1 THEN 1.0 ELSE 0.0 END)` **Service Availability** `1-(SUM(CASE WHEN "activities"."activity" LIKE 'http:/' AND "activities"."failure" = 1 THEN 1.0 ELSE 0.0 END)/COUNT("activities"."activity"))`

Component	AA_7
Title	Error types
Filter	`FILTER "zipkin_annotations"."a_key" = 'error';`
Dimensions	**a_value** `"zipkin_annotations"."a_value"`
KPIs	**Count** `COUNT_TABLE("zipkin_spans")`

Cross-Domain Analysis (CDA)

Component	CDA_3
Title	Conversion vs. Activity Failure
Filter	-
Dimensions	**Eventtime in years** `ROUND_DAY("sessions"."session_start")`
KPIs	**Conversion Rate** `AVG(CASE WHEN PROCESS EQUALS 'Book car' THEN 1.0 ELSE 0.0 END)` **User Activity Failure Rate** `AVG(CASE WHEN "activities"."activity" LIKE 'failed' AND "activities"."activity_type" = 'user' THEN 1.0 ELSE 0.0 END)`

Component	CDA_4
Title	Duration: Session Start to Conversion
Filter	`FILTER "activities"."activity_type" = 'user';`
Dimensions	**device_type** `"sessions"."device"`
KPIs	**Duration in s** `AVG(CALC_THROUGHPUT(ALL_OCCURRENCE[] TO LAST_OCCURRENCE['Book car'], REMAP_TIMESTAMPS("activities"."start_ts", SECONDS)))`

Component	CDA_6
Title	User Process Activity Performance
Filter	`FILTER "activities"."activity_type" = 'user';`
Dimensions	**Activity** `"activities"."activity"`
KPIs	**Activities count** `COUNT("activities"."activity")` **Duration** `AVG("activities"."duration")` **Act. Conversion Impact** `AVG(CASE WHEN PROCESS EQUALS 'Book car' THEN 1.0 ELSE 0.0 END)`

Component	AA_7
Title	Most critical services
Filter	`FILTER "activities"."activity_type" = 'user';`
Dimensions	**technical_activities.service_name** `"system_activities"."service_name"`
KPIs	**Activites count** `COUNT("activities"."activity")`

Single User Activtiy Analysis (SUAA)

Component	CDA_3
Title	User activities
Filter	`FILTER "activities"."activity" NOT LIKE 'http:/';`
Dimensio	**Activitiy** `"activities"."activity"` **trace_id_hex** `"activities"."trace_id_hex"`
KPIs	-

Component	CDA_4
Title	System activities
Filter	-
Dimension	**technical_activities.activity** `"system_activities"."activity"`
KPIs	**Activities count** `COUNT("activities"."activity")`

Component	CDA_5
Title	User Process Activity Performance
Filter	`FILTER "activities"."activity_type" = 'user';`
Dimension	**Activity** `"activities"."activity"`
KPIs	**Activities count** `COUNT("activities"."activity")` **Duration** `AVG("activities"."duration")` **Act. Conversion Impact** `AVG(CASE WHEN PROCESS EQUALS 'Book car' THEN 1.0 ELSE 0.0 END)`

Component	CDA_6
Title	activity_name performance
Filter	FILTER "activities"."activity" LIKE '<%=activity_dim%>';
Dimensions	**Activity** "activities"."activity" **trace_id_hex** "activities"."trace_id_hex"
KPIs	**Duration** AVG("activities"."duration") **Duration above AVG** AVG("activities"."duration")

A.3. Installation guide for prototype setup

Install Celonis

1. Apply for an academic *enterprise* licence at `academic.alliance@celonis.de`
2. Download Celonis installer with obtained login credentials at `http://my.celonis.com`
3. Download the required additional MySQL driver installer from `https://dev.mysql.com/downloads/connector/j/5.1.html`
4. Extract the file and copy it in the following path: `C:\Program Files\Celonis 4 Enterprise\appfiles\app\WEB-INF\lib`
5. Install Celonis and configure it to run on port 9000 (default).
6. Activate license.

Deploy a MySQL database

1. Download and install a MySQL database from `https://dev.mysql.com/downloads/mysql/`
2. Configure it to run on port `1024`, with `root` as user and password.
3. Create a database with the name `zipkin`.
4. Start the database.
5. Execute the `initialize_datamodel.sql` script for initialising the tables of the database.

Deploy the prototype

1. Deploy and start the the *configuration-service* first.
2. Continue with the *discovery-service*. After that, deploy and start all other service in an arbitrary order.

Configure the analysis in Celonis

1. Import the `analysis_full` transport, which includes the analysis, the data model and test data. This is accomplished via clicking on the user name (top left) -> Transport. From here the transport can be imported. The password is `EnhancedPM2017`. This analysis can be tested without further configuration.

2. For establishing the live seeting with automatic analysis generation form the SUS, please import the `analysis_operational` transport, which includes the analysis, the data model and test data but is configured for operation usage. The password again is `EnhancedPM2017`
3. Configure a new database connection by opening the data model named `Datamodel`. Click on Data Sources -> Add new data source. Enter the configuration as described below:

Database connection settings

Template:

Name:
MySQL

Connection String:
jdbc:mysql://localhost:1024/zipkin

User name:
root

Password:
****

Schema Name:
zipkin

Driver Name:
com.mysql.jdbc.Driver

4. Now configure the trigger table in the data model. Thereby, in the open data model to 'Loading' and enter the configuration as described below:

Load scheduling

Save schedule

☑ Activate automatic re-loading

Next Scheduled reload:

Last Scheduled Reload: 8/15/17 4:56 PM

When to reload data model

External trigger

Select DB connection

MySQL

Column configuration

The reload tracking table is the table into which your integration job has to insert the trigger rows.

Reload tracking table

reload_trigger_table

The data model identifier is used to identify the data model in the table. This is required.

Data model identifier

Datamodel

The data model column contains the above name of the data model. This column is required.

Data model key column

DATA_MODEL_NAME

The reload request date column contains the timestamp when your data model reload job requested the data reload. This column is required.

Reload request date column

RELOAD_REQUEST_TIME

The reload started date column will be updated with the timestamp when the data model reload has been started. This column is required. It is used to identify, if a reload is required. A reload will be executed when this column is NULL.

Reload started date column

RELOAD_START_TIME

The reload finished date column will be updated with the timestamp when the data model reload has finished. This column is optional.

Reload finished date column

RELOAD_SUCCESS_TIME

The message varchar column will be updated with the resulting message when the load has finished. If an error occurred during the load, then the column will be updated with the error message. This column is optional.

Reload message varchar column

RELOAD_MESSAGE

Usage

1. For creating click paths, open the web UI in a browser at `http://localhost:8080`.
2. Create a new user and perform random activities.
3. Create more sessions and users in order to create click paths.
4. Optional: Shut down a random service and continue performing activities. When the request fails go back to the main page.
5. After performing activities, click on "Generate activites & reload Data model" or wait max. 5 minutes for the *log generation service* to execute the activity generation automatically.
6. The process mining tool should now refresh the data model after new activities are generated. This can take up to 5 minutes. If the automatic refresh does not work, open the data model and navigate to Status. Click 'Reload from source' to refresh the model manually.
7. Go to the main menu and open the `Dashboard` analysis. See in the different tabs the four analyses described as in subsection 4.6.4. Start discovering the process from multiple perspectives!