Master's Degree in Informatics Engineering

Final Dissertation

Observing and Controlling Performance in Microservices

Author:

André Pascoal Bento

Supervisor:
Prof. Filipe João Boavida Mendonça Machado Araújo

Co-Supervisor:
Prof. António Jorge Silva Cardoso

UNIVERSIDADE Ð COIMBRA

July 2019

This page is intentionally left blank.

Abstract

Microservice based software architecture are growing in usage and one type of data generated to keep history of the work performed by this kind of systems is called tracing data. Tracing can be used to help Development and Operations (DevOps) perceive problems such as latency and request work-flow in their systems. Diving into this data is difficult due to its complexity, plethora of information and lack of tools. Hence, it gets hard for DevOps to analyse the system behaviour in order to find faulty services using tracing data. The most common and general tools existing nowadays for this kind of data, are aiming only for a more human-readable data visualisation to relieve the effort of the DevOps when searching for issues in their systems. However, these tools do not provide good ways to filter this kind of data neither perform any kind of tracing data analysis and therefore, they do not automate the task of searching for any issue presented in the system, which stands for a big problem because they rely in the system administrators to do it manually. In this thesis is present a possible solution for this problem, capable of use tracing data to extract metrics of the services dependency graph, namely the number of incoming and outgoing calls in each service and their corresponding average response time, with the purpose of detecting any faulty service presented in the system and identifying them in a specific time-frame. Also, a possible solution for quality tracing analysis is covered checking for quality of tracing structure against OpenTracing specification and checking time coverage of tracing for specific services. Regarding the approach to solve the presented problem, we have relied in the implementation of some prototype tools to process tracing data and performed experiments using the metrics extracted from tracing data provided by Huawei. With this proposed solution, we expect that solutions for tracing data analysis start to appear and be integrated in tools that exist nowadays for distributed tracing systems.

Keywords

Microservices, Cloud Computing, Observability, Monitoring, Tracing.

Resumo

A arquitetura de software baseada em micro-serviços está a crescer em uso e um dos tipos de dados gerados para manter o histórico do trabalho executado por este tipo de sistemas é denominado de tracing. Mergulhar nestes dados é díficil devido à sua complexidade, abundância e falta de ferramentas. Consequentemente, é díficil para os DevOps de analisarem o comportamento dos sistemas e encontrar serviços defeituosos usando tracing. Hoje em dia, as ferramentas mais gerais e comuns que existem para processar este tipo de dados, visam apenas apresentar a informação de uma forma mais clara, aliviando assim o esforço dos DevOps ao pesquisar por problemas existentes nos sistemas. No entanto, estas ferramentas não fornecem bons filtros para este tipo de dados, nem formas de executar análises dos dados e, assim sendo, não automatizam o processo de procura por problemas presentes no sistema, o que gera um grande problema porque recaem nos utilizadores para o fazer manualmente. Nesta tese é apresentada uma possivel solução para este problema, capaz de utilizar dados de tracing para extrair metricas do grafo de dependências dos serviços, nomeadamente o número de chamadas de entrada e saída em cada serviço e os tempos de resposta coorepondentes, com o propósito de detectar qualquer serviço defeituoso presente no sistema e identificar as falhas em espaços temporais especificos. Além disto, é apresentada também uma possivel solução para uma análise da qualidade do tracing com foco em verificar a qualidade da estrutura do tracing face à especificação do OpenTracing e a cobertura do tracing a nível temporal para serviços especificos. A abordagem que seguimos para resolver o problema apresentado foi implementar ferramentas protótipo para processar dados de tracing de modo a executar experiências com as métricas extraidas do tracing fornecido pela Huawei. Com esta proposta de solução, esperamos que soluções para processar e analisar tracing comecem a surgir e a serem integradas em ferramentas de sistemas distribuidos.

Palavras-Chave

Micro-serviços, Computação na nuvem, Observabilidade, Monitorização, Tracing.

This page is intentionally left blank.

Acknowledgements

This work would not be possible to be accomplished without effort, help and support from my family, fellows and colleagues. Thus, in this section I would like to give my sincere thanks to all of them.

Starting by giving thanks to my mother and to my whole family, who have supported me through this entire and long journey, and who always gave and will always give me some of the most important and beautiful things in life, love and friendship.

In second place, I would like to thank all people that were involved directly in this project. To my supervisor, Professor Filipe Araújo, who contributed with his vast wisdom and experience, to my co-supervisor, Professor Jorge Cardoso, who contributed with is vision and guidance about the main road we should take and to Engineer Jaime Correia, who "breathes" these kind of topics through him and helped a lot with is enormous knowledge and enthusiasm.

In third place, I would like to thank Department of Informatics Engineering and the Centre for Informatics and Systems, both from the University of Coimbra, for allowing and provide the resources and facilities to to be carried out this project.

In fourth place, to the Foundation for Science and Technology (FCT), for financing this project facilitating its accomplishment, to Huawei, for providing tracing data, core for this whole research, and to Portugal National Distributed Computing Infrastructure (INCD) for providing hardware to run experiments.

And finally, my sincere thanks to everyone that I have not mentioned and contributed to everything that I am today.

This page is intentionally left blank.

Contents

Acronyms

API Application Programming Interface. 9, 10, 22, 39, 49, 64, 67

CPU Central Processing Unit. 1, 2, 11, 16, 19

CSV Comma-separated values. 47, 54, 55, 56, 57, 60

DEI Department of Informatics Engineering. 1

DevOps Development and Operations. i, iii, 1, 2, 11, 17, 29, 64, 66

GDB Graph Database. 20, 21, 22, 39, 50

HTTP Hypertext Transfer Protocol. 12, 13, 23, 30, 39, 42, 43, 45, 47, 48

JSON JavaScript Object Notation. 41, 46, 55

OTP OpenTracing processor. 8, 37, 41, 43, 45, 47, 48, 50, 51, 54, 56, 57, 60, 61, 65

QA Quality Attribute. xi, 35, 36, 39

RPC Remote Procedure Call. 12, 30, 42, 43

TSDB Time Series Database. 22, 23, 39, 47, 48, 49, 54, 56

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

This document presents the *Master Thesis* in *Informatics Engineering* of the student *André Pascoal Bento's* during the school year of 2018/2019, taking place in the *Department of Informatics Engineering (DEI)*, *Faculty of Sciences and Technology* of the *University of Coimbra.*

1.1 Context

Software systems are becoming larger and more distributed than ever, thus requiring new solutions and new development patterns. One approach that emerged in recent years is to decouple large monolithic components into interconnected "small pieces" that encapsulate and provide specific functions. These components are known as "Microservices" and have become mainstream in the enterprise software development industry [1], [2]. Besides their impact on latency, fine-grained distributed systems, including microservices, increase system complexity, thus turning anomaly detecting into a more challenging task [3].

To tackle this problem, Development and Operations (DevOps) resort to techniques like monitoring [4], logging [5], and end-to-end tracing [6], to observe and maintain records of the work performed in a microservices system. Monitoring consists of measuring aspects like Central Processing Unit (CPU) and hard drive usage, network latency and other infrastructure metrics around the system and components. Logging provides an overview to a discrete, event-triggered log. Tracing is similar to logging, but focuses on registering the flow of execution of the program, as requests travel through several system modules and boundaries. Distributed tracing can also preserve causality relationships when state is partitioned over multiple threads, processes, machines and even geographical locations. Subsection 2.1.3 - Distributed Tracing.

The main problem with this is that there are not many implemented tools for processing tracing data and none for performing analysis of this type of data. For monitoring it tend to be easier, because data is represented in charts and diagrams, however for logging and tracing it gets harder to manually analyse the data due to multiple factors like its complexity, plethora and increasing quantity of information. There are some visualisation tools for the DevOps to use, like the ones presented in Subsection 2.2.1 - Distributed Tracing Tools, however none of them gets to the point of analysing the system using tracing, has they tend to be developed only for visualisation and display of tracing data in a more human readable way. Distributed tracing data can be used by DevOps because it is particularly well-suited to debugging and monitoring modern distributed software architectures, such as microservices. This kind of data contains critical information about request paths, response time and status, services presented in the system and their relationship and, for this reason, can be further analysed to detect anomalous behaviours in requests, response times and services in these systems. Nevertheless, this is critical information about the system behaviour, and thus there is the need for performing automatic tracing analysis.

1.2 Motivation

Exploring and develop ways to perform tracing analysis in microservice based systems lay down the motivation behind this work. The analysis of this kind of systems tend to be very complex and hard to perform due to their properties and characteristics, as it is explained in Subsection 2.1.1 - Microservices, and to the type of data to be analysed, presented in Subsection 2.1.3 - Distributed Tracing.

DevOps teams have lots of problems when they need to identify and understand problems with distributed systems. They usually detect the problems when a client complains about the quality of service, and after that, DevOps dive in monitoring metrics like, e.g, CPU usage, usage, hard drive usage and network latency. Later on, they use distributed tracing data visualisations and logs to find some explanation to what is causing the reported problem. This involves a very hard and tedious work of look-up through lots of data that represents the history of work performed by the system and, in most cases, this tedious work reveals like a big “find a needle in the haystack” problem. For this reason, DevOps have hard time finding problems in services and end up “killing” and rebooting services, which can be bad for the whole system. However, due to lack of time and difficulty in identifying anomalous services precisely this is the best approach to perform.

Problems regarding the system operation are more common in distributed systems and their identification must be simplified. This need of simplification comes from the exponential increase in the amount of data needed to retain information and the increasing difficulty in manually managing distributed infrastructures. The work presented in this thesis, aims to perform a research around these needs and focus on presenting some solutions and methods to perform tracing analysis.

1.3 Goals

The main goals for this thesis consist on the main points exposed bellow:

1. Search for existing technology and methodologies used to help DevOps teams in their current daily work, with the objective of gathering the best practices about handling tracing data. Also, we aim to understand how these systems are used, what are their advantages and disadvantages to better know how we can use them to design and produce a possible solution capable of performing tracing analysis. From this we expect to learn the state of the field for this research, covering the core concepts related work and technologies, presented in Chapter 2 - State of the Art.

2. Perform a research about the main needs of DevOps teams, to better understand what are their biggest concerns that lead to their approaches when performing pinpointing of microservices based systems problems. Relate these approaches with related work in the area, with the objective of understanding what other companies and groups have done in the field of automatic tracing analysis. The processes

used to tackle this type of data, their main difficulties and conclusions provide a better insight about the problem. From this we expected to have our research objectives clearly defined and a compilation of questions to be evaluated and answered, presented in Chapter 3 - Research Objectives and Approach.

3. Reason about all the information gathered to design and produce a possible solution that provides a different approach to perform tracing analysis. From this we expect first to propose a possible solution, presented in Chapter 4. Then we implemented it using state of the art technologies, feed it with tracing data provided by Huawei and collect results, presented in Chapters 5 - Implementation Process and 6 - Results, Analysis and Limitations. Finally, we provide conclusions to this research work in Chapter 7 - Conclusion and Future Work.

1.4 Work Plan

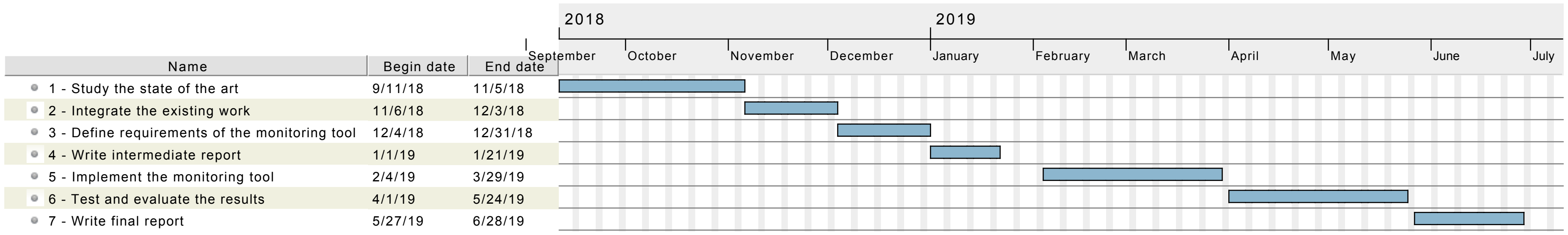
This work represents an investigation and was mainly an exploratory work, therefore, no development methodology was adopted. Meetings were scheduled to happen every two weeks. We gathered with the objective of discussing the work carried out and define new courses of research. The main focus in the first semester were topics like published papers, state of the art, analysis of related work and a proposition of solution. In the second semester, two more colleagues joined the project (DataScience4NP) and started participating in meetings, which contributed with wider discussions of ideas. In these meeting, the main topics covered were: implementation of the proposed solution, research for algorithms and methods for trace processing and analysis of gathered data. In the end, these meetings were more than enough to keep the productivity and good work.

Total time spent in each semester, by week, were sixteen (16) hours for the first semester and forty (40) hours for the second. In the end, it was spent a total of three-hundred and four (304) hours for the first semester, starting in 11.09.2018 and ending in 21.01.2019 (19 weeks $*$ 16 hours per week). For the second semester, eight-hundred and forty (840) hours were spent, starting in 04.02.2019 and ending in 28.06.2019 (21 weeks $*$ 40 hours per week).

As we can see in Figures 1.1 and 1.2, the proposed work for the first semester has suffered changes, when comparing it to the real work plan. Task 1 - Study the state of the art (Fig. 1.1), was branched in two, 1 - Project Contextualisation and Background and 2 - State of the Art, however, these last ones tocked more time to accomplish due to lack of work in the field of trace processing and trace analysis, core topics for this thesis. Task 2 - Integrate the existing work (Fig. 1.1) was replaced by task 3 - Prototyping and Technologies Hand-On (Fig. 1.2) due to redirections in the work course. This redirection was done due to interest increase in testing state of the art technologies, allowing us to get a better visualisation of the data provided by Huawei and enhancing our investigation work. The remaining tasks took almost the predicted time to accomplish.

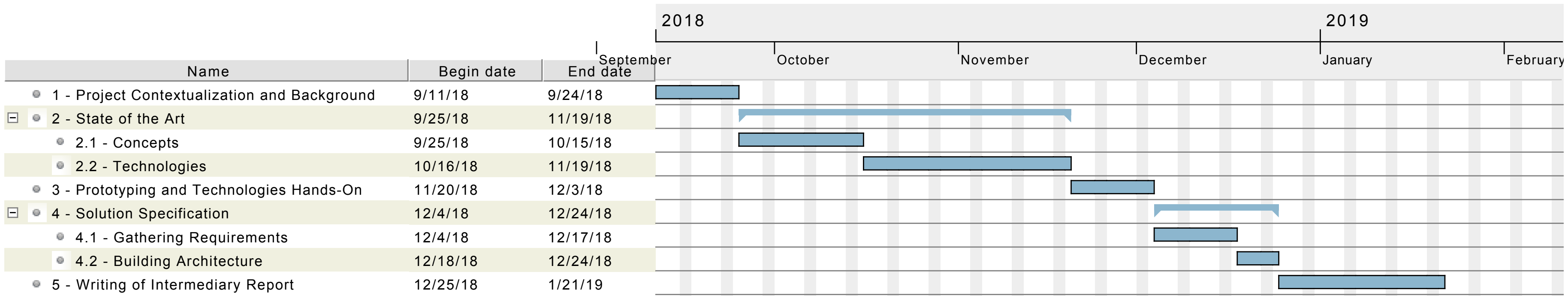
For the second semester, an "expected" work plan was defined with respect to the proposed work, presented in Figure 1.1, and the state of the research at the time. The expected work plan can be visualized in Figure 1.3. This Figure contains the expected (Grey) and real (Blue) work for the second semester.

Three main changes were made over time in the work plan. The first one involved a reduction in task 1 - Metrics collector tool. When the solution was being implemented and the prototype was capable to extract a set of metrics, we decided to stop the implementation process to analyse the research questions. Second, this analysis lead to an emergence of ideas, “2 - Restructuring research questions’,’ and thus a project redirection. Tests were removed from planning and the project followed with the objective of producing the data analyser, “3 - Data Analyser tool”, and with it, answer two main questions regarding anomalous services and quality of tracing. Third, the introduction of a new task, “4 - Write paper to NCA 2019”, covering the work presented in this thesis.



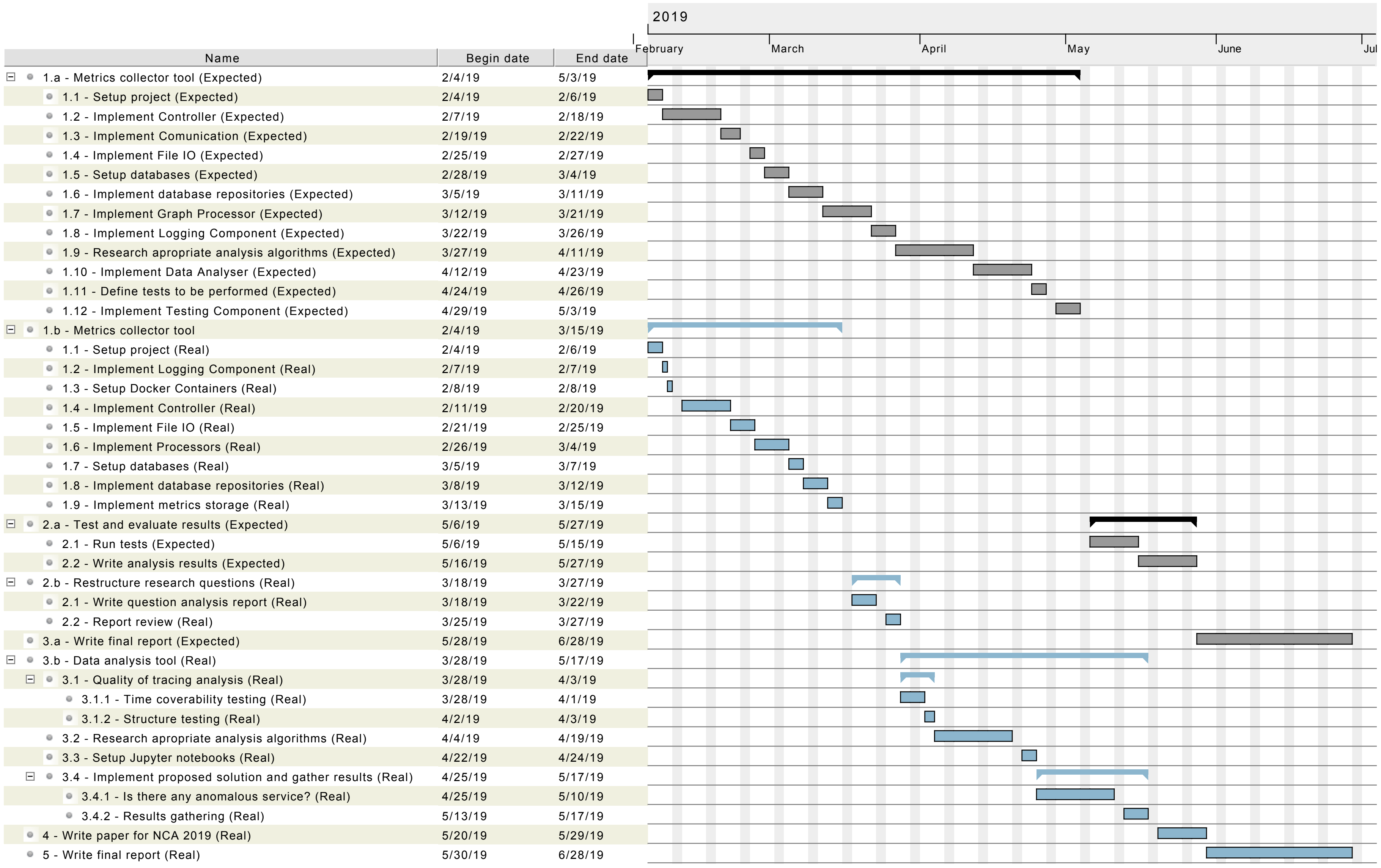
Name	Begin date	End date
1 - Study the state of the art	9/11/18	11/5/18
2 - Integrate the existing work	11/6/18	12/3/18
3 - Define requirements of the monitoring tool	12/4/18	12/31/18
4 - Write intermediate report	1/1/19	1/21/19
5 - Implement the monitoring tool	2/4/19	3/29/19
6 - Test and evaluate the results	4/1/19	5/24/19
7 - Write final report	5/27/19	6/28/19

Figure 1.1: Proposed work plan for first and second semesters.



Name	Begin date	End date
1 - Project Contextualization and Background	9/11/18	9/24/18
2 - State of the Art	9/25/18	11/19/18
2.1 - Concepts	9/25/18	10/15/18
2.2 - Technologies	10/16/18	11/19/18
3 - Prototyping and Technologies Hands-On	11/20/18	12/3/18
4 - Solution Specification	12/4/18	12/24/18
4.1 - Gathering Requirements	12/4/18	12/17/18
4.2 - Building Architecture	12/18/18	12/24/18
5 - Writing of Intermediary Report	12/25/18	1/21/19

Figure 1.2: Real work plan for first semester.



Name	Begin date	End date
1.a - Metrics collector tool (Expected)	2/4/19	5/3/19
1.1 - Setup project (Expected)	2/4/19	2/6/19
1.2 - Implement Controller (Expected)	2/7/19	2/18/19
1.3 - Implement Comunication (Expected)	2/19/19	2/22/19
1.4 - Implement File IO (Expected)	2/25/19	2/27/19
1.5 - Setup databases (Expected)	2/28/19	3/4/19
1.6 - Implement database repositories (Expected)	3/5/19	3/11/19
1.7 - Implement Graph Processor (Expected)	3/12/19	3/21/19
1.8 - Implement Logging Component (Expected)	3/22/19	3/26/19
1.9 - Research apropriate analysis algorithms (Expected)	3/27/19	4/11/19
1.10 - Implement Data Analyser (Expected)	4/12/19	4/23/19
1.11 - Define tests to be performed (Expected)	4/24/19	4/26/19
1.12 - Implement Testing Component (Expected)	4/29/19	5/3/19
1.b - Metrics collector tool	2/4/19	3/15/19
1.1 - Setup project (Real)	2/4/19	2/6/19
1.2 - Implement Logging Component (Real)	2/7/19	2/7/19
1.3 - Setup Docker Containers (Real)	2/8/19	2/8/19
1.4 - Implement Controller (Real)	2/11/19	2/20/19
1.5 - Implement File IO (Real)	2/21/19	2/25/19
1.6 - Implement Processors (Real)	2/26/19	3/4/19
1.7 - Setup databases (Real)	3/5/19	3/7/19
1.8 - Implement database repositories (Real)	3/8/19	3/12/19
1.9 - Implement metrics storage (Real)	3/13/19	3/15/19
2.a - Test and evaluate results (Expected)	5/6/19	5/27/19
2.1 - Run tests (Expected)	5/6/19	5/15/19
2.2 - Write analysis results (Expected)	5/16/19	5/27/19
2.b - Restructure research questions (Real)	3/18/19	3/27/19
2.1 - Write question analysis report (Real)	3/18/19	3/22/19
2.2 - Report review (Real)	3/25/19	3/27/19
3.a - Write final report (Expected)	5/28/19	6/28/19
3.b - Data analysis tool (Real)	3/28/19	5/17/19
3.1 - Quality of tracing analysis (Real)	3/28/19	4/3/19
3.1.1 - Time coverability testing (Real)	3/28/19	4/1/19
3.1.2 - Structure testing (Real)	4/2/19	4/3/19
3.2 - Research apropriate analysis algorithms (Real)	4/4/19	4/19/19
3.3 - Setup Jupyter notebooks (Real)	4/22/19	4/24/19
3.4 - Implement proposed solution and gather results (Real)	4/25/19	5/17/19
3.4.1 - Is there any anomalous service? (Real)	4/25/19	5/10/19
3.4.2 - Results gathering (Real)	5/13/19	5/17/19
4 - Write paper for NCA 2019 (Real)	5/20/19	5/29/19
5 - Write final report (Real)	5/30/19	6/28/19

Figure 1.3: Real and expected work plans for second semester.

1.5 Research Contributions

From the work presented on this thesis, the following research contribution were made:

- Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo and Jorge Cardoso. On the Limits of Automated Analysis of OpenTracing. International Symposium on Network Computing and Applications (IEEE NCA 2019).

1.6 Document Structure

This section presents the document structure in this report, with a brief explanation of the contents in every section. This document contains a total of eight chapters, including this one, Chapter 1 - Introduction. The remaining six are presented as follows:

- In Chapter 2 - State of the Art the current state of the field for this kind of problem is presented. This chapter is divided in three sections. The first one, Section 2.1 - Concepts introduces the reader to the core concepts to know as a requirement for a full understanding of the topics discussed in this thesis. The second, Section 2.2 - Technologies presents the result of a research for current technologies, that are able to help solving this problem and produce a proposed solution to be implemented. Finally, Section 2.3 - Related Work presents the reader to related researches produced in the field of distributed tracing data handling.

- In Chapter 3 - Research Objectives and Approach the problem is approached in detail and the objectives of this research are presented. This chapter is divided in two sections. First, Section 3.1 - Research Objectives, provides a concrete definition of the problem, how we tackled it, the main difficulties that were found and the objectives involved in order to propose a solution. Second, Section 3.2 - Research Questions, a compilation of questions are presented and evaluated with some reasoning about possible ways to answer them.

- In Chapter 4 - Proposed Solution a possible solution for the presented problem is exposed and explained in detail. This chapter is divided in four sections. The first one, Section 4.1 - Functional Requirements, expose the functional requirements with their corresponding priority levels and a brief explanation to every single one of them. The second one, Section 4.2 - Quality Attributes, contains the gathered non-functional requirements that were used to build the solution architecture. The third one, Section 4.3 - Technical Restrictions, presents the defined technical restrictions for this project. The last one, Section 4.4 - Architecture, presents the possible solution architecture using some representational diagrams, and ends with an analysis and validation to check if the presented architecture meets up the restrictions involved in the architectural drivers.

- In Chapter 5 - Implementation Process, the implementation process of the possible solution is presented with detail. This chapter is divided in three main sections covering the whole implementation process, from the input data set through the pair of components presented in the previous chapter. The first one, Section 5.1 - Huawei Tracing Data Set, the tracing data set provided by Huawei to be used as the core data for research is exposed with some detail. Second, in Section 5.2 - OpenTracing Processor Component we present the possible solution for the first component, namely

“Graphy OpenTracing processor (OTP)”, that processes and extracts metrics from tracing data. The final Section 5.3 - Data Analysis Component presents the possible solution for the second component, namely “Data Analyser”, that handles data produced by the first component and produces the analysis reports. Also, in the last two sections presented, the used algorithms and methods in the implementations are properly detailed and explained.

- In Chapter 6 - Research Objectives and Approach, the gathered results, corresponding analysis and limitations of tracing data are presented. This chapter is divided in three main sections. The first one, Section 6.1 - Anomaly Detection, the results regarding the gathered observations on the extracted metrics of anomalous service detection are presented and explained. Second, in Section 6.2 - Trace Quality Analysis the results obtained from the quality analysis methods applied to the tracing data set are presented and explained. The final Section 6.3 - Limitations of OpenTracing Data we present the limitations felted when designing a solution to process tracing data, more precisely OpenTracing data.

- Last, in Chapter 7 - Conclusion and Future Work, the main conclusions for this research work are presented. To present this chapter, a reflection about the implemented tools, methods produced and the open paths from this research are exposed. Also a reflection of the main difficulties felted with this research regarding the handling of tracing data are presented. After this, the future work that can be addressed, considering this work, is properly explained.

Next, Chapter 2 - State of the Art, the state of the field is covered with core concepts, technologies and related work.

Chapter 2

State of the Art

In this Chapter, we discuss the core concepts regarding the project, the most modern technology for the purpose today and related work in the area. All the information presented results from work of research through published articles, knowledge exchange and web searching.

First, the main purpose of Section 2.1 - Concepts is to introduce and provide a brief explanation about the core concepts to the reader. Second, Section 2.2 - Technologies, all the relevant technologies are analysed and discussed. In the final Section 2.3 - Related Work, published articles and posts of related work are presented and possible research directions are discussed.

2.1 Concepts

The following concepts represents the baseline to understand the work related to this research project. First an explanation of higher level of concepts that composes the title of this thesis are presented in Subsections 2.1.1 and 2.1.2. The following Subsections 2.1.3 to 2.1.5, aim to cover topics related to previous concepts: Distributed Tracing, Graphs and Time-Series.

2.1.1 Microservices

The term "micro web services" was first used by Dr. Peter Rogers during a conference on cloud computing in 2005, and evolved later on to "Microservices" at an event for software architects in 2011, where the term was used to describe a style of architecture that many attendees were experimenting with at the time. Netflix and Amazon were among the early pioneers of microservices [7].

Microservices is "an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities" [1], [2].

This style of software development has a very long history and has being introduced and evolving due to software engineering achievements in the later years regarding cloud distributed computing infrastructures, Application Programming Interface (API) improvements, agile development methodologies and the emergence of the recent phenomenon of containerized applications. "A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another, communicating with others through an API" [8].

In Microservices, services are small, specifically calibrated to perform a single function, also each service is designed to be autonomous, resilient, minimal and composable. This framework brings a culture of rapid iteration, automation, testing, and continuous deployment, enabling teams to create products and deploy code exponentially faster than ever before [9].

Until the rising of Microservices based architecture, the Monolithic architectural style was the most used. This style has a the particularity of produce software composed all in one piece. All features are bundled, packaged and deployed in a single tier application using a single code base.

Figure 2.1 aims to give a comparison between both architectural styles, Monolithic and Microservices, and provide an insight about the differences between them.

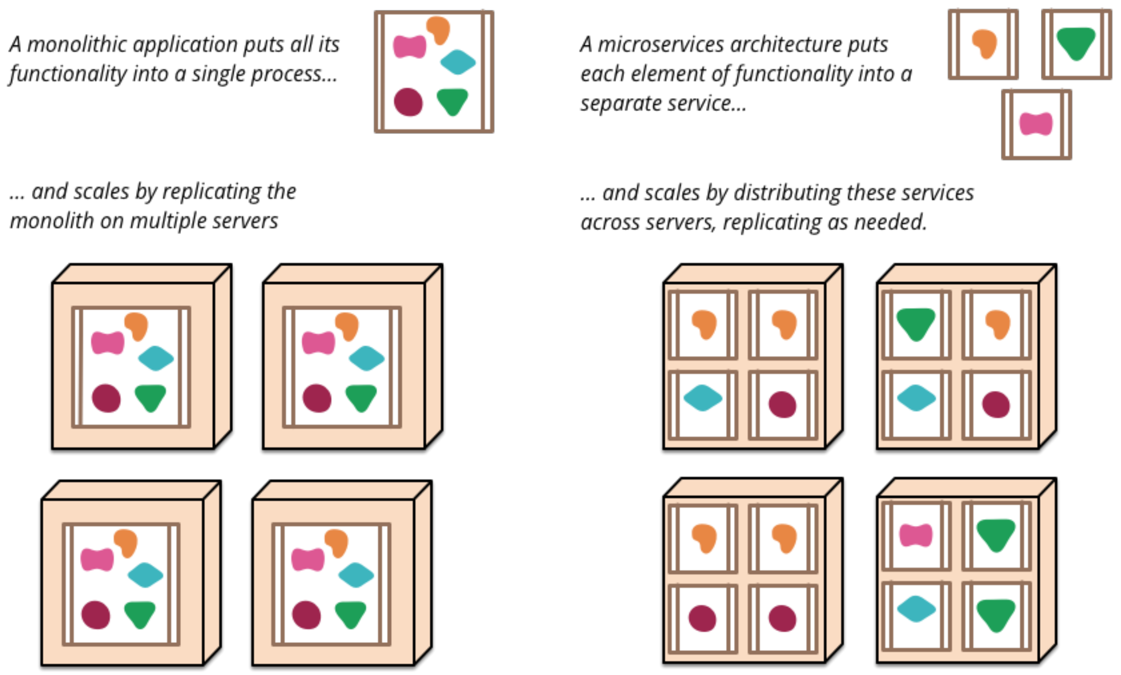

A monolithic application puts all its functionality into a single process...
... and scales by replicating the monolith on multiple servers
A microservices architecture puts each element of functionality into a separate service...
... and scales by distributing these services across servers, replicating as needed.

Figure 2.1: Monolithic and Microservices architectural styles [10].

Both styles presented have their own advantages and disadvantages. To briefly present some of them, two examples are provided, one for each architectural style. First example: if one team needs to develop a single process system, e.g., e-Commerce application, that authorizes customer, takes an order, check products inventory, authorize payment and ships ordered products. The best alternative is to use Monolithic architecture, because they can develop every feature in a single software package due to the application simplicity, however, if the client starts to demand hard changes and additional features in the solution, the code base may tend to increase into "out of control", leading to more challenging and time consuming changes. Second example, if one team needs to develop a complex and huge service that needs to scale, e.g., Video streaming service, the best alternative is to use Microservices architecture, because they can tackle the problem of complexity by decomposing the application into a set of manageable small services which are much faster to develop and test by individual organized teams, and thus, it will be easier to maintain the code base due to decoupling, however, it will be harder to monitor and manage the entire platform due to additional complexity associated with distributed systems.

Taking into consideration this increasing difficulty in monitoring and managing large Microservice based platforms, one must be aware and observe system behaviour to be able to control it. Therefore, in the next Subsection 2.1.2, the core concept of Observability and Controlling Performance is explained.

2.1.2 Observability and Controlling Performance

This Subsection aims to provide an introduction to some theory concepts about Observability and Performance Controlling, regarding distributed software systems.

Observability is a meaningfully extension of the word observing. Observing is “to be or become aware of, especially through careful and directed attention; to notice’ ’[11]. The term Observability comes from the world of engineering and control theory. Observability is not a new term in the industry, however it has gain more focus in the last years due to Development and Operations (DevOps) raising. It means by definition “to measure of how well internal states of a system can be inferred from knowledge of its external outputs” [12]. Therefore, if our good old software systems and applications do not adequately externalize their state, then even the best monitoring can fall short.

Controlling in control systems is “to manage the behaviour of a certain system” [13]. Controlling and Observability are dual aspects of the same problem [12], as we need to have information to infer state and be able take action. E.g., When observing an exponential increase in the Central Processing Unit (CPU) load, the system scales horizontally invoking more machines and spreading the work between them to easy handle the work. This is a clear and simple example that conjugates the terms presented, we have: values that are observed “Observability” and action that leads to system control “Controlling Performance”.

When we want to understand the working and behaviour of a system, we need to watch it very closely and pay special attention to all details and information it provides. Microservice based systems produce multiple types of information if instrumented. These type of information are the ones mentioned in Chapter 1: Monitoring, Tracing and Logging. In this thesis, the goal is to use tracing data thus, this type of produced information is the one to focus.

In the next Subsection 2.1.3 - Distributed Tracing, the type of data mentioned before is presented and explained in detail.

2.1.3 Distributed Tracing

Distributed tracing [14] is a method that comes from traditional tracing, but applied to a distributed system at the work-flow level. It profiles and monitor applications, especially those built using microservice architectures and, in the end, it can be used to help DevOps teams pinpoint where failures occur and why.

A number of tools and standards emerged from this concept. For example, the *Open-Tracing* standard [15] follows the model proposed by Fonseca *et al.* [16], which defines traces as a tree of spans representing scopes or units of work (i.e., thread, function, service). These traces enable following such units of work through the system.

OpenTracing uses dynamic, fixed-width metadata to propagate causality between spans, meaning that each span has a *trace identifier* common to all spans of the same trace, as well as a *span identifier* and *parent identifier* representing parent/child relationships between spans [17]. The standard defines the format for spans and the semantic [18], [19] conventions for their content/annotations.

Figure 2.2 provides a clear insight about how spans are related to time and with each other.

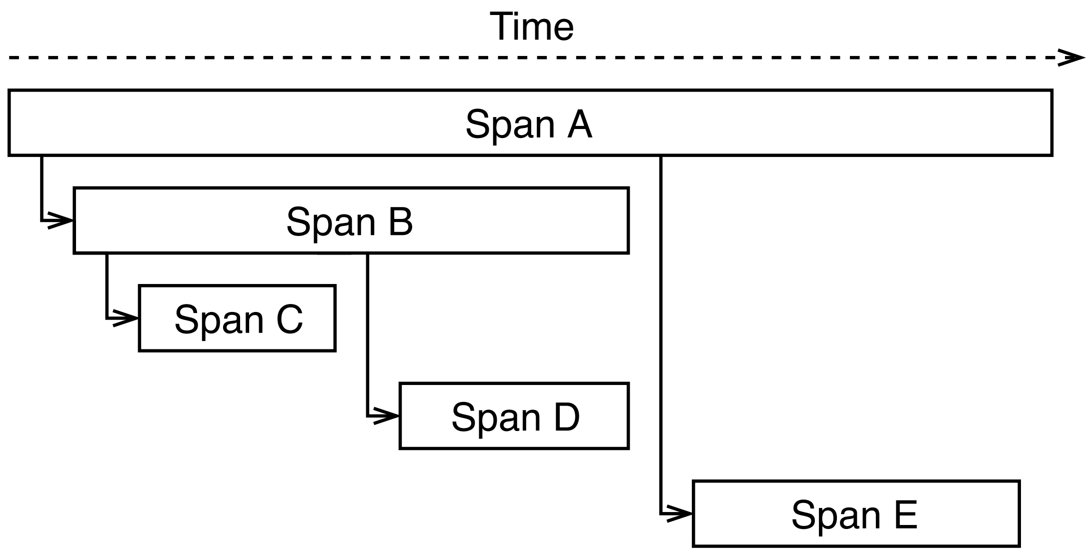


Figure 2.2: Sample trace over time.

In Figure 2.2 there are a group of five spans spread through time that represents a trace. A trace is a group of spans that share the same *TraceID*. A trace is a representation of a data/execution path in the system. A span represents the logical unit of work in the system. A trace can also be a span, if there is only one span presented in the trace. One span can cause another.

Causality relationship between spans can be observed in Figure 2.2, where “Span A” causes “Span B” and “Span E”, moreover, “Span B” causes “Span C” and “Span D”. From this we say that “Span A” is parent of “Span B” and “Span E”. Likewise, “Span B” and “Span E” are children of “Span A”. In this case, “Span A” does not have a parent, it is an “orphan span” and therefore, is the root span and the origin of this whole trace. Spans carry with them metadata like e.g., *SpanID* and *ParentID*, that allows to infer this relationships.

Disposition of spans over time is another clear fact that can be observed from the representation in Figure 2.2. Spans have a begin and an end in time. This causes them to have a duration. Spans are spread through time, however they usually stay inside parent boundaries, this means that the duration of a parent span always covers durations of their children. Considering a parent and a child spans, if they are related, the parent span always start before child span, also, the parent span always end after child span. Note that nothing prevents multiple spans to start in the same exact moment. Span also carry with them metadata like e.g., *Timestamp* and *Duration*, that allows to infer their position in time and when they end.

An example of a span can be an Hypertext Transfer Protocol (HTTP) call or a Remote Procedure Call (RPC) call. We may think of the following cases to define each operation inherent to each box presented in Figure 2.2: A - "Get user info", B - "Fetch user data from database", C - "Connect to MySQL server", D - "Can't connect to MySQL server" and E - "Send error result to client".

In the data model specification, the creators of *OpenTracing* say that: "with a couple of spans, we might be able to generate a span tree and model a directed graph of a portion of the system" [15]. This is due to the causal relationships they represent. Apart from the root span every other span must have a parent. Figure 2.3 provides an example of a span tree.

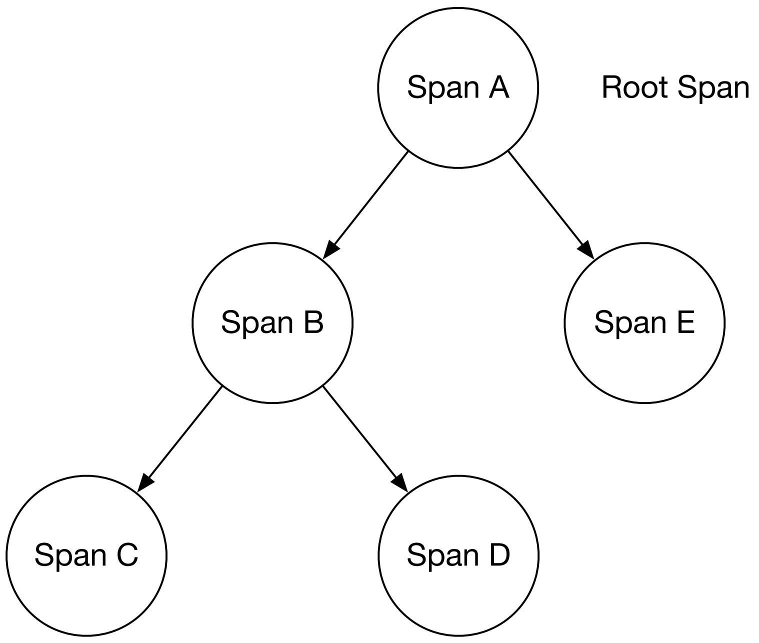


Figure 2.3: Span Tree example.

Figure 2.3 contains a span tree representation with a trace containing five spans. Apart from the root span every other span must have a parent. With this causal relationship, a path through the system can be retrieved. For example, if every span processes in a different endpoint represented by letters presented in the span tree, one may generate the request path: A $\rightarrow$ B $\rightarrow$ D. This means that our hypothetical request passed through machine A, B and D, or if it were services, the request passed from service A, to B and finally to D. From this, we can generate the dependency graph of the system (explained in the Subsection 2.1.4 - Graphs).

This type of data is extracted as trace files or streamed over transfer protocols like e.g., HTTP, from technologies like Kubernetes [20], OpenStack [21], and other cloud or distributed management system technologies that implements some kind of system or code instrumentation using, for example, *OpenTracing* [22] or OpenCensus [23]. Tracing contains some vital system details as they are the result of system instrumentation and therefore, this data can be used as a resource to provide observability over the distributed system.

As said before, from the causality relationship between spans we can generate a dependency graph of the system. The next Subsection 2.1.4 - Graphs aims to provide a clear understand of this concept and how they relate with distributed tracing.

2.1.4 Graphs

From distributed tracing we can be able to extract the system dependency graph from a representative set of traces. To introduce the concept of Graph, “A Graph is a set of vertices and a collection of directed edges that each connects an ordered pair of vertices” [24].

Taking the very common sense of the term and to provide notation, a graph, G, is an ordered pair $G = (V, E)$, where V are the vertices/nodes and E are the edges.

Graphs are defined by:

- Node: Are the entities in the graph. They can hold any number of attributes (key-value pairs) called properties. Nodes can be tagged with labels, representing their different roles in a domain. Node labels may also serve to attach metadata (such as index or constraint information) to certain nodes;
- Edge (or Relationships): provide directed, named, semantically-relevant connections between two node entities;
- Property: can be any kind of metadata attached to a certain Node or a certain Edge.

Also, there are multiple types of graphs, they can be:

1. Undirected-Graph: the set of edges without orientation between a pair of nodes;
2. Directed-Graph: the set of edges have one and only one direction between a pair of nodes;
3. Multi-Directed-Graph: multiple edges have more than one connection between a pair of nodes that represents the same relationship.

Figure 2.4 gives us a simple visual representation of what a graph really is for a more clear understanding.

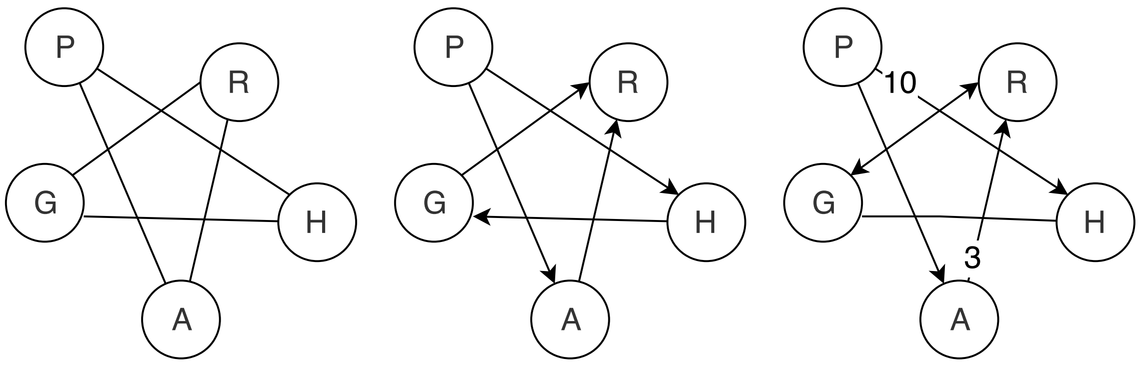


Figure 2.4: Graphs types.

In Figure 2.4 three identical graphs are presented and each one is composed by five nodes, however, they are not equal because each one has it own type. They belong respectively to each type enumerated above. From left to right, the first graph is a Undirected-Graph, the second one is a Directed-Graph and the last one is a Multi-Directed-Graph.

The last graph has some numbers in some edges. Every graph can have this annotations. These can provide some information about the connection between the pair of nodes. For example, in distributed systems context, if this graph represents our system dependency graph, and nodes H and P hypothetical services, the edge between them could represent calls between these two service and the notation number the number of calls with respect to the edge direction. Therefore, in this case, we would have 10 requests from incoming from P to H.

Figure 2.5 provides a clear insight about service dependency graphs.

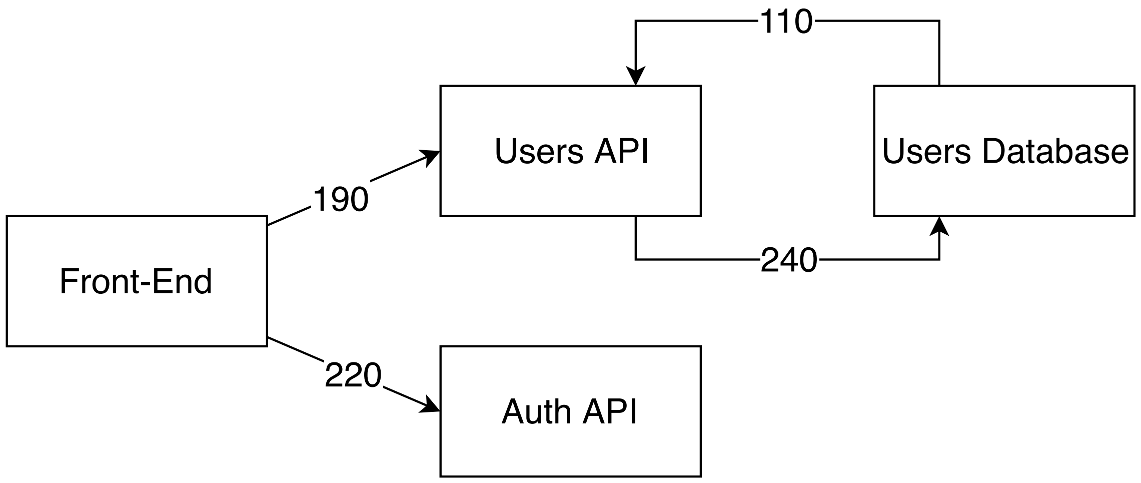


Figure 2.5: Service dependency graph.

In Figure 2.5, a representation of a service dependency graph is provided. Service dependency graphs are graphs of type Multi-Directed-Graph, because they have multiple edges with more than one direction between a pair of services(Nodes). In this representation, there are multiple services involved, each inside a box. The edges between boxes (Nodes), indicate the number of calls that each pair of services invoked, e.g., "Users API" called "Users Database" 240 times. These dependency graphs gives the state of the system in a given time interval. This can be useful to study the changes in the morphology of the system, e.g., a service disappeared and a set of new ones appeared. Other interesting study could be the variation in the amount of call between services.

Graphs are a way to model and extract information from tracing data. Another interesting approach could be to extract metrics in time from tracing because traces and spans are spread in time, and they have information about the state of the system at a given instant. The next Subsection 2.1.5 - Time-Series provides an introduction to a data representation model.

2.1.5 Time-Series

Time-Series are a way of representing data as a time-indexed series of values. This kind of data is often arise when monitoring systems, industrial processes, tracking corporate business metrics or sensor measurements. Figure 2.6 provides a visual example of this way of data representation.

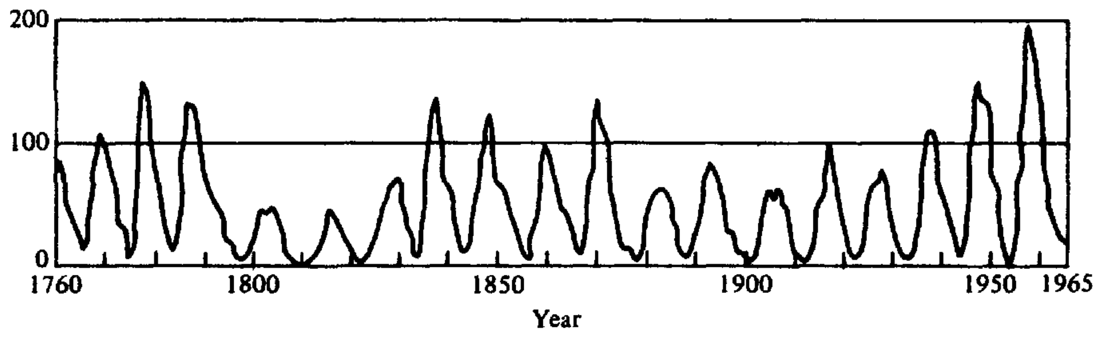


Figure 2.6: time-series: Annual mean sunspot numbers for 1760-1965 [25].

In Figure 2.6, Brillinger *D.* [25] presents a visual representation of a time-series as a collection of values in time. These values are measurements of sunspot means gathered from 1960-1965. In this case, measurements come from natural origin, however, one can perform observations of e.g., CPU load, system uptime / downtime and network latency.

As these processes are not random, autocorrelation can be exploited to extract insight from the data, such as predict patterns or detect anomalies. Therefore, time-series data can be analysed to detect anomalies present in the system. One way to do this is to look for outliers [26] in the multidimensional feature set. Anomaly detection in time-series data is a data mining process used to determine types of anomalies found in a data set and to determine details about their occurrences. Anomaly detection methods are particularly interesting for our data set since it would be impossible to manually tag the set of interesting anomalous points. Figure 2.7 provides a simple visual representation of anomaly detection in time-series data.

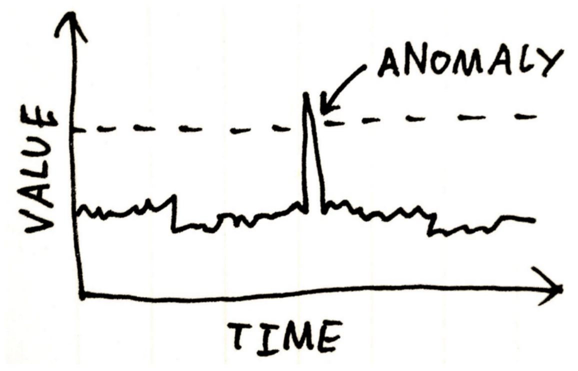


Figure 2.7: Anomaly detection in Time-Series [27].

In Figure 2.7, there is a clear spike in values from this time-series measurements. This can be declared an outlier because it is a strange value considering the range of remaining measurements and therefore, it is considered an anomaly. In this example, anomaly detection is easy to perform by a Human, however, in mostly cases nowadays, due to great variation of values and plethora of information that can be gathered, perform this detection manually is impracticable, thus automatic anomaly detection using Machine Learning techniques are used nowadays.

Anomaly detection in time-series data is a data mining process used to determine types of anomalies found in a data set and to determine details about their occurrences. This auto anomaly detection method has lots of usage due to the impossible work of tag manually the interesting set of anomalous points. Auto anomaly detection has a wide range of applications such as fraud detection, system health monitoring, fault detection, event detection systems in sensor networks, and so on.

After explaining the core concepts, foundations for the work presented in this thesis, to the reader, technologies capable of handling this types of information are presented and discussed in next Section 2.2 - Technologies.

2.2 Technologies

In this section are presented technologies and tools capable of handling the types of information discussed in the previous Section 2.1 - Concepts.

The main tools covered are: 2.2.1 - Distributed Tracing Tools, for distributed tracing data handling, 2.2.2 - Graph Manipulation and Processing Tools and 2.2.3 - Graph Database Tools, for graph processing and storage, and 2.2.4 - Time-Series Database Tools, for time-series value storage.

2.2.1 Distributed Tracing Tools

This Subsection presents the most used and known distributed tracing tools. These tools are mainly oriented for tracing distributed systems like microservices-based applications. What they do is to fetch or receive trace data from this kind of complex systems, treat the information, and then present it to the user using charts and diagrams in order to explore the data in a more human-readable way. One of the best features presented in this tools, is the possibility to perform queries on the tracing (e.g., by trace id and by time-frame). Table 2.1 presents the most well-known open source tracing tools.

In Table 2.1, we can see that these two tools are very similar. Both are open source projects, allow docker containerization and provide a browser ui to simplify user interaction. Jaeger was created by Uber and the design was based on Zipkin, however, it does not provide much more features. The best feature that was released for Jaeger in the past year was the capability of perform trace comparison, where the user can select a pair of traces and compare them in terms of structure. This is a good effort in additional features, but it is short in versatility because we can only compare a pair of traces in a “sea” of thousands, or even millions.

These tools aim to collect trace information and provide a user interface with some query capabilities for DevOps to use. However they are always focused on span and trace lookup and presentation, and do not provide a more interesting analysis of the system, for example to determine if there is any problem related to some microservice presented in the system. This kind of work falls into the user, DevOps, as they need to perform the tedious work of investigation and analyse the tracing with the objective of find anything wrong with them.

This kind of tools can be a good starting point for the problem that we face, because they already do some work for us like grouping the data generated by the system and provide a good representation for them.

In next Subsection 2.2.2, graph manipulation and processing tools are presented and discussed.

Table 2.1: Distributed tracing tools comparison.

	Jaeger [28]	Zipkin [29]
Brief description	Released as open source by Uber Technologies. Used for monitoring and troubleshooting microservices-based distributed systems. Was inspired by Zipkin.	Helps gathering timing data needed to troubleshoot latency problems in microservice applications. It manages both the collection and lookup of this data. Zipkin's design is based on the Google Dapper paper.

Pros	Open source;
	Docker-ready;
	Collector interface is compatible with Zipkin protocol;
	Dynamic sampling rate;
	Browser user interface.
		Open source;
		Docker-ready;
		Allows multiple span transport technologies (HTTP, Kafka, Scribe, AMQP);
		Browser user interface.

Cons	Only supports two span transport ways (Thrift and HTTP).	Fixed sampling rate.
Analysis	Dependency graph view; Trace comparison (End 2018).	Dependency graph view.

Used by	Red Hat;
	Symantec;
	Uber.
		AirBnb;
		IBM;
		Lightstep.

2.2.2 Graph Manipulation and Processing Tools

Distributed tracing is a type of data produced by Microservice based architectures. This type of data is composed by traces and spans. With a set of related spans, a service dependency graph can be produced. This dependency graph is a Multi-Directed-Graph, as presented in Subsection 2.1.4. Therefore, with this data at our disposal, there is the need of a graph manipulation and processing tool.

In this Subsection, the state of the art about graph manipulation and processing tools is presented. Graphs are non-linear data structure representations consisting of nodes and edges. Nodes are sometimes also referred to as vertices and edges are lines or arcs that connect any pair of nodes in the graph. This data structure takes some particular approaches when handling their contents, because there are some special attributes related. For example, perform the calculation of the degree of some node – degree of a node is the number of edges that connect to the node itself; Calculate how many nodes entered and exited the graph by comparing it to another one; Know the difference in edges between two distinct graphs [30].

Taking into consideration this data structure, the particularities involved and the need to use graphs to manipulate service dependencies, frameworks with features capable of handling and retrieving graphs are a need. Therefore, Table 2.2 presents a comparison of the main tools available at the time for graph manipulation and processing.

Table 2.2: Graph manipulation and processing tools comparison.

	Apache Giraph [31]	Ligra [32]	NetworkX [33]
Description	An iterative graph processing system built for high scalability. Currently used at Facebook to analyse the social graph formed by users and their relationships.	A library collection for graph creation, analysis and manipulation of networks.	A Python package for the creation, manipulation, and study of structure, dynamics, and functions of complex networks.
Licence [34]	Free Apache 2.	MIT.	BSD - New License.
Supported languages	Java and Scala.	C and C++.	Python.
Pros	Distributed and very scalable; Excellent performance – Process one trillion edges using 200 modest machines in 4 minutes.	Handles very large graphs; Exploit large memory and multi-core CPU – Vertically scalable.	Good support and very easy to install with Python; Lots of graph algorithms already implemented and tested.
Cons	Uses “Think-Like-a-Vertex” programming model that often forces into using sub-optimal algorithms, thus is quite limited and sacrifices performance for scaling out; Unable to perform many complex graph analysis tasks because it primarily supports Bulk synchronous parallel.	Lack of documentation and therefore, very hard to use; Does not have many usage in the community.	Not scalable (single-machine); High learning curve due to the maturity of the project; Begins to slow down when processing high amount of data – 400.000+ nodes.

Table 2.2 presents some key points to consider when choosing a graph manipulation and processing tool.

First, one aspect to be considered when comparing them is the scalability and performance that each provide. Apache Giraph is the best tool in this field, since it is implemented with distributed and parallel computation, which allows it to scale to multiple-machines, sharing the load between them, and processing data large quantities of data in less time than the remaining presented tools. On the opposite side, NetworkX, only works in a single-machine environment which does not allow it scale to multiple-machines. Ligra, like the previous tool, works in a single-machine environment, however it benefits from vertical scale on a single-machine, which allows to exploit multi-core CPU and large memory. NetworkX and Ligra are tools that can present a bottleneck in a system where the main focus is to handle large amounts of data in short times.

Secondly, another aspect to be considered is the support and quantity of implemented graph algorithms available on the frameworks. NetworkX have advantages in this aspect, because it contains implementation of the majority graph algorithms defined and studied in graph and networking theory. Also, due to project maturity, it has a good documentation support from the community who keeps all the information updated. Ligra framework has lack of documentation, which causes tremendous difficulty for developers to use and know what are the implemented features. Apache Giraph, does not support a large set of graph processing algorithms due to implementation constraints.

Figure 2.8 gives a clear insight when comparing these tools from two features – scalability / performance against implementation of graph algorithms.

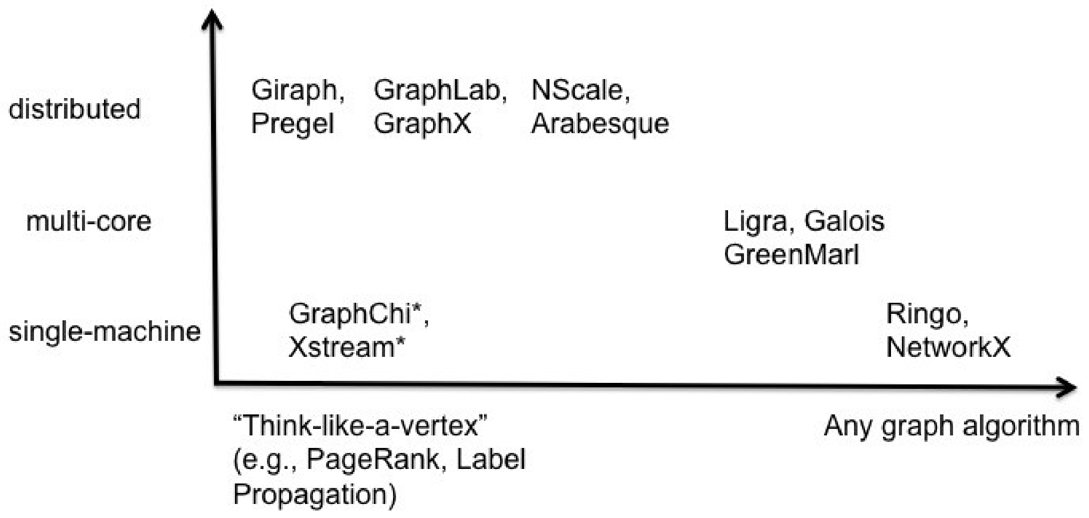


Figure 2.8: Graph tools: Scalability vs. Algorithm implementation [35].

In Figure 2.8 we can observe tools disposition regarding the two aspect key points explained before. This figure contains all tools presented over two featured axis: one for scalability and the other for implementation of graph algorithms. Tools placement in this chart proves and reinforces the comparison presented before. Apache Giraph and NetworkX are placed in the edges of these features, which means that Apache Giraph can be found in the upper left region of the chart – highly distributed but minimally in graph algorithms implementation –, and NetworkX is in the lower right region – minimally distributed but highly in graph algorithms implementation.

After discussing tools capable of manipulate and process graphs, their storage is a need for later usage. Graph Database (GDB) storage technologies are presented in next Subsection 2.2.3 - Graph Database Tools.

2.2.3 Graph Database Tools

Graph databases represent a way of persisting graph information. After having instantiated a Graph, processed it in volatile memory, they can be stored in persistent memory for later use. To do this one can use a GDB. A GDB is "a database that allows graph data storing and uses graph structures for semantic queries with nodes, edges and properties to represent them" [36].

Based upon the concept of a mathematical graph, a graph database contains a collection of nodes and edges. A node represents an object, and an edge represents the connection or relationship between two objects. Each node in a graph database is identified by a unique identifier that expresses key $\rightarrow$ value pairs. Additionally, each edge is defined by a unique identifier that details a starting or ending node, along with a set of properties. Graph databases are becoming popular due to Machine Learning and Artificial Intelligence grows, since a number of Machine Learning algorithms are inherently graph algorithms [37].

Furthermore, in this research service dependency graphs are highly used, thus the need to use a GDB. Table 2.3 contains the most well-known GDB.

Table 2.3: Graph databases comparison.

	ArangoDB [38]	Facebook TAO [39]	Neo4J [40]
Description	A NoSQL database that uses a proper query language to access the database.	TAO, "The Associations and Objects", is a proprietary graph database, developed by Facebook, used to store the social network.	The most popular open source graph database, completely open to the community.
Licence [34]	Free Apache 2.	Proprietary.	GPLv3 CE.
Supported languages	C++; Go; Java; JavaScript; Python and Scala.	Go; Java; JavaScript; Python and Scala.	Java; JavaScript; Python and Scala.
Pros	Multi data-type support (key/value, documents and graphs); Allows combination of different data access patterns in a single query; Supports cluster deployment.	Low latency ($= 100ms$); Accepts millions of calls per second; Distributed database.	Supports ACID(Atomicity, Consistency, Isolation, Durability) [41]; Most popular open source graph database.
Cons	High learning curve due to AQL (Arango Query Language); Has paid version with high price tag.	Not accessible to use.	Not able to scale horizontally.

From Table 2.3 we can notice that the state of the art for GDB is not very pleasant. Interest for this type of databases has began in the later years due to artificial intelligence and machine learning trends, therefore, the offer presented in the field are limited.

Back in time, when social network tendency emerged, the development of this type of databases raised, and the most powerful technologies for graph storage where developed in closed source. One example is *Facebook TAO* database presented in Table 2.3, a database developed by the company to support the entire social network, storing users in nodes and their relationships in edges. This database is described by having very low latency, which stands for high response time, however, very few information regarding this tool can be found – just some scientific papers [42], [43].

The remaining tools presented are available for usage. ArangoDB has multi data-type support, which means that a wider type of data structures are supported for storing in nodes and edges metadata. Also, it supports scalability through cluster deployment, however, this feature is only available in paid versions – Arango SmartGraphs storage improves the writing of graph in distributed systems environment [44]. The biggest disadvantage of this database is the high learning curve associated with the usage of *AQL (Arango Query Language)*, however, this disadvantage can be surpassed by using provided API clients with the trade-off of loosing some control.

Neo4J is the most accepted GDB by the open source community. This GDB has increased in popularity in the past years due to simplicity and easy support [45]. Trade-offs from this database consists in lack of support for scalability, which means that this database can only run on a single-machine environment, however, there are some users reporting that they were able to perform implementations and surpass the lack of support for horizontal scaling, but this is not tested [46].

Choosing a graph database can be hard because these tools are growing and the tendency for changes in features and tooling support is very high, however, the decision falls on the question of easy of usage and horizontal scalability. This means that *ArangoDB* is a database more advised for big projects, where the size of graphs to store may surpass the limit of a single-machine, and Neo4J for simpler projects, where the focus are functionality testing and prototyping, and graph storage represents a side concern.

Next Subsection 2.2.4 - Time-Series Database Tools covers the state of the art for tooling capable of storage values based in time.

2.2.4 Time-Series Database Tools

In this Subsection, tools for storing time-indexed series of values are presented. This type of data is a need for this research due to the tight relation between distributed tracing and time, as explained in Subsections 2.1.3 and 2.1.5. Also, service dependency graphs, as a representation of the system at a given time, can contain valuable information for monitoring Microservice systems. For this purpose, Time Series Database (TSDB) are databases capable of storing time-series based values.

A TSDB is "A database optimised for time-stamped or time-series data like arrays of numbers indexed by time (a date time or a date time range)" [47]. These databases are natively implemented using specialised time-series theory algorithms to enhance their performance and efficiency, due to widely variance of access possible. The way this databases use to work on efficiency is to treat time as a discrete quantity rather than as a continuous mathematical dimension. Usually a TSDB allows operations like create, enumerate, update, organise and destroy various time-series entries in short access times.

This type of database is growing in usage and popularity because of Internet of Things (IoT) trend. Discussions in this area have increased over the past few years, and is expected that it keeps increasing, due to Ubiquitous Computing – Raise of omnipresent and universal technologies. At the same time, TSDB grows with this IoT tendency, because data mining from sensor spread geographically and sensors gather information through measurements in specific points in time. This information are usually stored in TSDB [48].

Table 2.4 presents a comparison between two TSDB: *InfluxDb* and *OpenTSDB*.

Table 2.4: Time-series databases comparison.

	InfluxDB [49]	OpenTSDB [50]
Description	An open-source time-series database written in Go and optimised for fast, high-availability storage and retrieval of time-series data in fields such as operations monitoring, application metrics, Internet of Things sensor data, and real-time analytics's.	A distributed and scalable TSDB written on top of HBase; OpenTSDB was written to address a common need: store, index and serve metrics collected from computer systems (network gear, operating systems and applications) at a large scale, therefore, making this data easily accessible and displayed.
Licence [34]	MIT.	GPL.
Supported languages	Erlang, Go, Java, JavaScript, Lisp, Python, R and Scala.	Erlang, Go, Java, Python, R and Ruby.

Pros	Scalable in the enterprise version;
	Outstanding high performance;
	Accepts data from HTTP, TCP, and UDP protocols;
	SQL like query language;
	Allows real-time analytics's.
		Massively scalable;
		Great for large amounts of time-based events or logs;
		Accepts data from HTTP and TCP protocols;
		Good platform for future analytical research into particular aggregations on event / log data;
		Does not have paid version.

Cons	Enterprise high price tag; Clustering support only available in the enterprise version.	Hard to set up; Not a good choice for general-purpose application data.

From Table 2.4, we can notice some similarities between these two TSDB databases. Both TSDB are capable scalable and accept HTTP and TCP transfer protocols for communication. InfluxDB and OpenTSDB are two open source time-series databases, however, the first one, InfluxDB, is not completely free, as it has an enterprise paid version, which is not very visible in the offer. This enterprise version offers, clustering support, high availability and scalability [51], features that OpenTSDB offer for free. In terms of performance, InfluxDB surpasses and outperforms OpenTSDB in almost every benchmarks [52]. OpenTSDB has the benefits of being completely free and support the most relevant features, however it is very hard to set up and to develop for this database.

In the end, both TSDB are bundled with good features, and the decision falls into how much performance is needed when choosing one. If the need is performance and access to the database in short amounts of time, with low latency responses, InfluxDB is the way to go, by other way, if there no restriction about the performance needed to query the database and money is a concern, the choice should be OpenTSDB.

Tooling for this project is presented. We have covered the most used technologies and core concepts in related to the field of tracing Microservices. Next Section 2.3 - Related Work, will cover the related work performed in this area. Some ideas, approaches and developed solutions will be discussed.

2.3 Related Work

This section aims to present the related work in the field of distributed tracing data handling and analysis. It is divided in three Subsections: first, 2.3.1 - Mastering AIOps, which covers a work carried out by Huawei, that uses machine learning – deep learning – methods to analyse data from distributed traces. Secondly, 2.3.2 - Anomaly Detection using Zipkin Tracing Data, a work of performed by Salesforce with the objective of analyse tracing from a distributed tracing tool. Finally, 2.3.3 - Analysing distributed trace data, a work by Pinterest, where the objective is to study latency in tracing data.

2.3.1 Mastering AIOps

Distributed tracing has only started to gain widespread acceptance in the industry recently, as a result of new architectural and software engineering practices, such as cloud-native, fine-grained systems and agile methodologies. Additionally, the increase in complexity resulting from the rise of web-scale distributed applications is a recent phenomenon. As a consequence of its novelty, there has been little research in the field so far.

A recent example, AIOps, an application of Artificial Intelligence to operations [53] was introduced in 2016 [54]. This trend aims to use Artificial Intelligence for IT Operations in order to develop new methods to automate the enhance IT Operations. Driving this "revolution" are the following points:

- First there is the additional difficulty of manually managing distributed infrastructures and system state;
- Secondly, the amount of data that has to be retained is increasing, creating a plethora of problems to the operators handling it;
- Third, the infrastructure itself is becoming more distributed across geography and organizations, as evidenced by trends like cloud-first development and fog computing;
- Finally, due to the overwhelming amount of new technologies and frameworks, it is an herculean task for operators to keep in pace with the new trends.

The work performed and presented by Huawei, entitled Mastering AIOps with Deep Learning, Time-Series Analysis and Distributed Tracing [55], aims to use distributed tracing data and aforementioned technologies to detect anomalous tracing. The proposed method encodes the traces and trains a deep learning neural network to detect significant differences in tracing. This is a very perceptive approach, taking into account the amounts of data that is needed to analyse, however is limited to classifying a trace as normal or abnormal, losing detail and interpretability i.e., no justification for the classification.

2.3.2 Anomaly Detection using Zipkin Tracing Data

Tooling in this field are not taking the expected relevance. Their usage is starting in industry and production environments involving distributed systems, however, the concerns in are not well aligned with the needs of operators, and this leads to increasing effort when monitoring large scale and complex architectures, such as Microservices.

In a post from Salesforce, a work of research about using tracing data gathered by Zipkin, to detect some anomalies in a Microservice based system [56]. At Salesforce, Zipkin is used to perform distributed tracing for Microservices, collecting traces from their systems and providing performance insights in both production monitoring and pre-production testing. However, the current Zipkin open source instrumentation and UI offers only primitive data tracing functionality and does not have in-depth performance analysis of the span data. The focus on their work was to detect and identify potential network bottlenecks and microservices performance issues.

The approach carried out was to implement scripts using that used Python AI packages, with the objective of extracting values from their network of services, namely service dependency graph, in order to identify high traffic areas in the network. The values that were extracted were the number of connections from each service, which means, the degree of the service at specific times. This allows to notice which services are establishing more connections with other services.

From this approach, it was possible to visualize the high traffic areas within the production network topology. Therefore, they have identified services with the most connections. This finding was an helpful feedback for service networking architects that designed those microservices. Those services, identified with too many connections, may potentially become choking points in the system design. If one of the services fail, a huge impact on a large number of depending services occur. Additionally, there could be also potential performance impacts in the system since a large number of services depending on them. Those are valuable information for system designers and architect to optimize their designs.

The conclusions from Salesforce research identified that, with Zipkin tracing data, it is possible to identify network congestion, bottlenecks, efficiencies and the heat map in the production network. However, this tool does not provide analysis of tracing data at this level. This was the main conclusion and possible working direction from this research: “features like the ones presented, can be added to Zipkin or other distributed tracing tool product line, including UI and dashboards. Capabilities like daily metrics or correlation between microservices load and latency, able to generate alerts if bottleneck or heat map is identified, should be added” [56].

2.3.3 Analysing distributed trace data

At Pinterest, the focus was to research for latency problems in their Microservices solution. Pinterest claims to have tens of services and hundreds of network calls per-trace. One big problem identified at start is the huge difficulty of looking to trace data due to the overwhelming quantity of information – “thousands of traces logged each minute (let alone the millions of requests per minute these traces are sampled from)”.

Pinterest felt the problem of monitoring Microservices early due to their service popularity in the past years. With this popularity, systems usage increased significantly. This lead them to take action and create their closed source distributed tracing analysis tool called “Pintrace Trace Analyser” [57].

This tool gathers tracing data from Distributed Tracing Tools, more precisely from Zipkin, and processes a sample of these tracing to detect mainly latency problems in the service dependency network. Looking at stats from thousands of traces over a longer period of time not only weeds out the outliers/buggy traces, but provides a holistic view of performance.

The conclusions from Pinterest, where that there is a great need to develop tooling for distributed tracing analysis, with the main objective of ease the life of operators. The following points were considered:

1. Automatically generate reports so engineers can easily check the status of each deployment;
2. Setting up alerts for when latency or number of calls hits a certain threshold.

2.3.4 Research possible directions

One thing to notice from the related work presented is that there is few research accomplished in the area and trace tooling development, however, these works are from the past year and the tendency is to increase in the following years. Enlargement and usage of distributed systems are fuel to feed the need of research in this field and develop new methodologies and tools to monitor and control operations.

From the first work presented, “Mastering AIOps”, some final results and conclusions were provided. They point out that the benefits of this approach were: first, very high accuracy in detection $99,7\%$, and secondly, extremely fast detection in $O(n)$ time. However, some limitations involving requiring very long training times for long traces (with decent machines) were noted. Also, improvements were pointed: truncate traces, to lower the quantity of tracing and therefore, summarizing traces.

The second work presented, “Anomaly Detection using Zipkin Tracing Data”, point down the lack of features in the existing tools. These features include automatic anomaly detection using distributed tracing data. The main idea consists in extending functionality presented in this tools, to provide autonomous anomaly detection and alerting based on information presented in tracing from services.

The third work presented, “Analysing distributed trace data”, crucial points considered were to represent autonomous generation of reports, allowing operators to check the status of deployments, and therefore, providing more control over the system regarding detection of anomalous values in service latency.

Finally, from the multiple related work presented the final assumptions for possible research directions in this field are:

- Focus on the most important traces, reducing the quantity of tracing;
- Develop new methods that leverage features of existing distributed tracing tools;
- Automate the detection of anomalies presented in distributed systems;

After providing the state of the art for this research to the reader, next Chapter 3 - Research Objectives and Approach will cover the objectives of this research, the approach used to tackle the problem and the compiled research questions.

Chapter 3

Research Objectives and Approach

In this Chapter, the problem is approached in detail and the objectives for this research are presented. The problem definition, how we tackled it and our main difficulties and the objectives involved to provide a possible solution are presented in Section 3.1 - Research Objectives. Also, a compilation of research questions are presented and evaluated with some reasoning about possible ways to answer them, later in Section 3.2 - Research Questions.

3.1 Research Objectives

Modern distributed services are large, complex, and increasingly built upon other similarly complex distributed services. Debugging systems is not an easy task to perform. It involves the collection, interpretation, and display of information concerning the interactions among concurrently executing processes operating in distributed machines. Distributed Tracing data helps keeping an history of work performed by these systems.

End-to-end tracing captures the causally-related activity (e.g., work done to process a request) within and among the components of a distributed system. As distributed systems grow in scale and complexity, such tracing is becoming a critical tool for management tasks like diagnosis and resource accounting. However, as systems grow, resulting tracing data from system execution is growing as well [17].

Tracing data growth raises some problems. This data is used by system operators to gain insight and observability of the distributed system, however, with this tendency to increase in quantity and complexity, it is becoming an overwhelming task for operators.

There are some tools that help handling tracing data, such as the ones presented in Subsection 2.2.1 - Distributed Tracing Tools, however, they only perform the job of collecting tracing data, present this information to the user in more human-readable formats and provide forms of querying this type of data. For this reason, manually managing these growing microservice architectures is becoming an outdated approach due to their incomportability.

To address this issue, there is a great need of improving tracing data processing and automate the task of tracing analysis. However, at this point, we did not have any tracing data at our disposal to start working, thus acquire tracing data from a distributed system was an urgency. Obtain tracing data for study is hard because it represents working from these systems and contain confidential information about them. However, through a

NDA (Non-Disclosure Agreement) and the help of professor Jorge Cardoso, representing Huawei, we were able to gain access to confidential tracing data generated by the company. To ensure confidentiality, in this thesis, direct data fields are presented using fictitious information.

This tracing data set was the starting point for this research. It is in `OpenTracing` format and was provided by Huawei. This data had been gathered from an experimental `OpenStack` cluster used by the company for testing purposes, and covered two days of operation. This data is addressed in detail in Section 5.1 - Huawei Tracing Data Set.

After having access to tracing data, we have developed some prototype tools for data ingestion and setted up a distributed tracing tool. *Zipkin* was used as a distributed tracing tool to ingest tracing data provided by Huawei. The decision to use *Zipkin* instead of *Jaeger*, fell in the fact that it were much simpler due to lesser feature configuration. This was done with the purpose of gain a clear visualization about the data that were given. From this we decided to perform several meetings with the objective of defining a research direction and be able to propose a solution.

In these meetings, the elements of this research project gathered to debate ideas and define a set of questions to answer, taking into consideration the defined problem. Professor Jorge Cardoso, representing Huawei, was the client of the designed solution. The approach taken was to create a shared Kanban Board [58], containing multiple lanes, to perform the of generation and refinement of prototype questions. This process involved having prototype question in the first lane, and move them through every lane reaching the last one, transforming a prototype question into a final research question. These research questions were built taking into consideration:

1. Main needs felt by operators in normal day-to-day tasks, troubleshooting distributed systems;
2. Most common issues presented in these systems;
3. Variables involved when these issues appear;
4. Relationship between these variables and the most common issues.

In the next Section 3.2 - Research Questions, the process to generate the research questions is explained and the research questions, in their final state, are presented.

3.2 Research Questions

In this Section, we start by explaining the process to generate the research questions. In the end, these questions are defined and a possible approach for each one of them is discussed.

A Kanban Board was created with five lanes: “Initial Prototype”, “To Refine (1)”, “Interesting”, “To Refine (2)” and “Final Research Questions”. Throughout these lanes, questions were improved and filtered before reaching their final state.

Initial prototype questions were generated based on the four points enumerated at the end of the previous Section 3.1. Therefore, prototype questions where:

1. What is the neighbourhood of one service?
2. Is there any problem (Which are the associated heuristics)?
3. Is there any faults related to the system design/architecture?
4. What is the root problem, when A, B, C services are slow?
5. How are requests coming from the client?
6. How endpoints orders distributions are done?
7. What is the behaviour of the instances?
8. What is the length of each queue in a service?

All these questions represent needs felt by operators when monitoring and troubleshooting distributed systems. To generate them, we gathered in meetings and discussed what are the main needs of Development and Operations (DevOps) based on research, state of the art tooling, related work developed in the past years and opinions from colleagues working in the area. However, these initial questions were too general, therefore they were passed through every lane defined before. This refinement leaded to the generation of final state questions. Final questions, with their corresponding description **(D)** and a starting point for the expected work **(W)** involved, are defined bellow:

1. Does any service present a significant change in the number of incoming requests?
2. Does any service present a significant change in the number of outgoing requests?
   - D. The number of requests are the number of calls performed to a service. These metrics represent a very important measurement for service monitoring, because it measures the service usage in time.
   - W. To obtain these metrics, one must generate the service dependency graph throughout defined time-frames and retrieve the number of connections between every node presented in the graph.
3. Does any service present a significant change in response time?
   - D. Response time represents the amount of time needed to respond to a call. It is considered one of the most important measurements in systems because represents their performance.
   - W. Get the response time for every span (difference between end and start time present in the structure).
4. Is there a problem related to the work-flow of one (or more) requests?
   - D. Work-flow of one request represents the interaction path triggered throughout the system.
   - W. Generate service dependency graph, retrieve work-flow paths presented in the graph and gather information about the number of unique paths and type variation.
5. How do requests are being handled by a specific service? (Identify services that are experiencing unreliability periods)

   - D. In the end, requests have success or not. This is represented by a status code in Hypertext Transfer Protocol (HTTP) or an exception in Remote Procedure Call (RPC). Measure the ratio of these values can help identify unreliability periods in services.
   - W. Gather status codes or exceptions from spans and generate a ratio of success and error.

6. Which services are the most popular in the system? (Number of established connections)

   - D. Popularity of a service stands for the number of established connections. This measurement is important because a failure in a very popular service can compromise the entire system.
   - W. Generate service dependency graph, and calculate the degree of each node. Services with higher degree are the most popular in the system.

7. Does any service present a significant change in the services it uses to fulfil requests?

   - D. Services tend to communicate with a set of other services. These services do not change often, therefore, patterns in service communication can be observed. If these patterns are violated without service redeployment and networking changes, one might be facing a possible traffic redirection.
   - W. Generate service dependency graph, and retrieve the set of services that each service communicates. Gathering these values in time, lead to a history of communication between services and, therefore, pattern recognition can be applied to detect strange variations.

8. Is there a problem related to the constitution of the system?

   - D. Constitution in microservices architecture represent which services are presented in the system. The study of entries and exits of services in the overall system network can help identifying problems in system constitution.
   - W. Generate service dependency graph in consecutive time-frames and retrieve the entry / exit of services. Variation analysis of this data can lead to detect constitution problems presented in distributed systems.

9. Do traces follow *OpenTracing* specification? (Structural quality testing)

   - D. Structure quality is always an important factor when using some dataset to analyse a system. This question aims to perform a structural test of spans presented in tracing against the defined specification.
   - W. Produce a structural schema based on the proposed open source tracing specification – *OpenTracing* –, and check every span.

10. How is time coverage of tracing? (Coverability quality testing)

   - D. Time coverage is an important aspect in tracing, because this measurement can pinpoint possible failures in system instrumentation.
   - W. In tracing, child spans should cover almost the total duration of their parent span. To perform this test, a span tree for each trace must be assembled and times ratios of the durations must be extracted.

After having generated these final state questions, an analysis report was performed in order to group them in similar fields of end-to-end tracing use cases [17]. Table 3.1 present the defined groups and the associated questions.

Table 3.1: Final state questions groups.

Group	**Question numbers**
1. Anomaly detection	1. Does any service present a significant change in the number of incoming requests? 2. Does any service present a significant change in the number of outgoing requests? 3. Does any service present a significant change in response time?
2. Steady state problems	4. Is there a problem related to the work-flow of one (or more) requests? 5. How do requests are being handled by a specific service? (Identify services that are experiencing unreliability periods)
3. Distributed resource profiling	6. Which services are the most popular in the system? (Number of established connections) 7. Does any service present a significant change in the services it uses to fulfil requests? 8. Is there a problem related to the constitution of the system?
4. Quality of tracing	9. Do traces follow *OpenTracing* specification? (Structural quality testing); 10. How is time coverage of tracing? (Coverability quality testing).

Table 3.1 presents us with questions grouped in four classes: *anomaly detection*, *steady state problems*, *distributed resource profiling* and *quality of tracing*. Questions were grouped in these four mentioned classes due to their affinity. The first one, Anomaly detection, is “diagnosis-related case that involves identifying and debugging problems related to correctness (e.g., component time-outs or connection failures)”, therefore grouped questions are related with response time and number of calls performed to services. Secondly, Steady state problems, is “another diagnosis-related, which involves identifying and debugging problems that manifest in work-flows (and so are not anomalies)”, thus questions are related with work-flow and status of requests. Thirdly, Distributed resource profiling, is “identify slow components or functions.”, so questions associated with service usage and system constitution. Finally, Quality of tracing, involve questions related to tracing quality testing.

The following general questions were composed for each group:

- Group 1 - Is there any anomalous service?
- Group 2 - What is the overall reliability of the service?
- Group 3 - Which service consumes more time when considering the entire set of requests?
- Group 4 - How can we measure the quality of tracing?

From these general questions, we decided to tackle two groups: 1. Anomaly detection and 4. Quality of tracing and therefore, the selected general questions were: 1. Is there any anomalous service? and 4. How can we measure the quality of tracing?. Questions presented in the remaining groups were not studied further in this research project.

The first question can be reduced to finding anomalies in observations of service or system behaviour, namely metrics and morphology. In particular, we considered three metrics: number of incoming service calls, outgoing service calls and average response time. Our proposed solution in Chapter 4 - Proposed Solution must have this into consideration – extract and analyse these metrics from tracing data.

For the second question, there are multiple ways to analyse quality in tracing. We explore two directions, first performing a trace structure testing against the defined *Open-Tracing* specification –structural testing –, to determine if the tracing data complies with all the predefined requirements. Secondly, coverage testing for tracing data to determine how much of the duration of Span is covered by its children – time coverability testing. The first kind would be more valuable if the specification was stricter, however, changing the standard was not an option at the time as the data had external providence – discussed in Chapter 7 - Conclusion and Future Work.

Next Chapter 4 - Proposed Solution covers our proposed solution, taking into considerations the main problem, the data to process and the research questions to be answered in this project.

Chapter 4

Proposed Solution

In this Chapter, we present and discuss a possible solution to be implemented regarding the main problem to solve in this research, the data to process and the research questions to be answered. To present the solution and explain it, we will cover some aspects considered when defining a software based solution: functional requirements 4.1, quality attributes (non-functional requirements) 4.2, technical restrictions 4.3 and finally, the architecture 4.4 produced based on all previous topics.

The starting point for our proposed solution is the tracing data provided by Huawei. Tracing must be ingested by an entry component, capable of extracting metrics from tracing data. The outcome of this module are metrics and metadata in files to be further processed by a second component. This second component has the duty of analysing the output data from the first module, and point out service anomalies.

For a clear insight about our solution, the proposed approach in high level of abstraction is presented in the Figure 4.1.

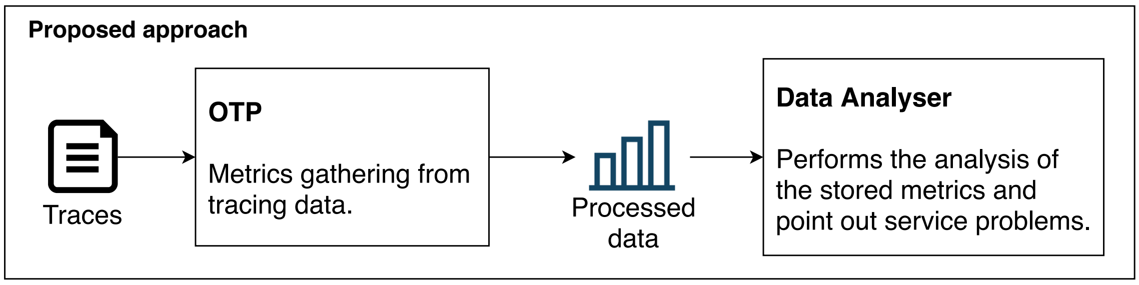

Proposed approach
Traces
OTP
Metrics gathering from tracing data.
Processed data
Data Analyser
Performs the analysis of the stored metrics and point out service problems.

Figure 4.1: Proposed approach.

Figure 4.1 shows the proposed process order for tracing data. We expect to have two main components, one for data extraction and another for data analysis. The input for each are tracing data and processed data from the first component respectively. The outcome is to answer the research questions defined in Section 3.2 - Research Questions.

Next Section 4.1 - Functional Requirements covers the functional requirements for this solution.

4.1 Functional Requirements

In software engineering, functional requirements define the intended function of a system and its components. To present the functional requirements for our solution proposition, an id, the corresponding name and its priority are provided. The notation used in priority was based on the urgency that we expected from feature implementation. Three priority levels were used: High, Medium and Low. Therefore, the functional requirements for the proposed solution, sorted by priority levels, are presented in Table 4.1.

Table 4.1: Functional requirements specification.

ID	**Name**	**Priority**
FR-1	The system must be able to ingest tracing data from a files or external distributed tracing tools.	High
FR-2	The system must be able to retrieve service dependency graphs from distributed tracing tools.	High
FR-3	The system must be able to store service dependency graphs in a graph database.	High
FR-4	The system must be able to store time-series metrics extracted from tracing data in a time-series database.	High
FR-5	The system must be able to extract the number of calls per service (total, incoming and outgoing) from tracing data.	Medium
FR-6	The system must be able to extract the response time per service from tracing data.	Medium
FR-7	The system must be able to generate request work-flow paths from tracing data.	Medium
FR-8	The system must be able to calculate request ratio of success and error, for specific services, from tracing data.	Medium
FR-9	The system must be able to calculate the degree (total, in and out) of services from service dependency graphs.	Medium
FR-10	The system must be able to retrieve the difference between two service dependency graphs.	Medium
FR-11	The system must be able to produce a report about spans structure using a defined OpenTracing structural schema.	Low
FR-12	The system must be able to calculate the time coverage of traces in a given time-frame.	Low
FR-13	The system must be able to identify regions of outliers presented in multiple time-series.	Low

Functional requirements defined in Table 4.1 were written based on defined research questions presented in Section 3.2. These functional requirements can be grouped in three groups due to their priority levels. The first four (FR-1 to FR-4) are presented with high level of priority, because they represent the base functionality needed to implement the remaining requirements. The next eight functional requirements (FR-5 to FR-11), are time based metric extraction from tracing. The remaining three (FR-12 to FR-14) are related with trace testing and anomaly detection based in time-series thus the low priority.

The relationship between these functional requirements and questions presented in previous chapter, Section 3.2, as well has the verification that these requirements are fulfilled by the solution, is covered in next chapter, Sections 5.2 and 5.3.

Next Section 4.2 - Quality Attributes covers the proposed approach non-functional requirements.

4.2 Quality Attributes

Another important consideration, when designing a software system, is to specify all the quality attributes (also called non-functional requirements). These type of requirements are usually Architecturally Significant Requirements and are the ones that require more from software architect's attention, as they reflect directly all architecture decisions. To specify them, a representation called utility tree is often used. In this tree, the Quality Attribute (QA) are placed by an order of priority considering their impact for the architecture and for the business. The priority codification for the QA is:

- H. High
- M. Medium
- L. Low

To describe them properly, six important aspects must be included in QA definition: *stimulus source*, *stimulus*, *environment*, *artefact*, *response* and *measure of the response.*

Figure 4.2 contains all raised QA for this proposed solution exposed in an utility tree structure, sorted alphabetically by their general QA name, and after by the architectural impact pair (Architecture and Business).

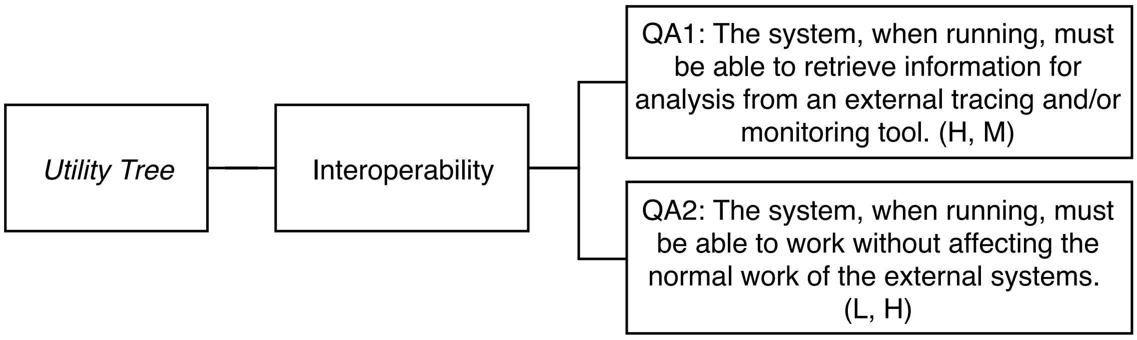

Utility Tree
Interoperability
QA1: The system, when running, must be able to retrieve information for analysis from an external tracing and/or monitoring tool. (H, M)
QA2: The system, when running, must be able to work without affecting the normal work of the external systems. (L, H)

Figure 4.2: QA utility tree.

Figure 4.2 shows us that only two Interoperability QA were defined. An explanation for both is provided bellow:

QA1 (Interoperability): Since the proposed solution must ingest tracing data. This information is usually found in distributed tracing tools already used by operators. To gather this information, access to an external distributed tracing tool is an important feature.As this is considered the starting point to obtain our data, we considered a Medium level for the architecture and a Low for the business.

QA2 (Interoperability): Since the proposed solution will be accessing an external distributed tracing system or outputs generated by it, all interactions with these systems must not cause conflicts. This is very important in the business perspective, because if our solution is not co-habitable with already used systems, it may be completely rejected. For the architectural perspective it does not represent a big impact, and therefore a Low level was assigned.

4.3 Technical Restrictions

In this Section, technical restrictions considered in proposed solution are presented.

In software engineering, after specifying functional and non-functional requirements for a solution, comes the specification of business restrictions, however, in this project none were raised due to the fact that this work is focused on exploration and research.

To define the technical restrictions, we used an id and its corresponding description. Table 4.2 presents the technical restrictions considered for the proposed solution.

Table 4.2: Technical restrictions specification.

ID	**Description**
TR-1	Use *OpenTSDB* as a Time-Series database.

Table 4.2 shows that we have raised only one technical restriction. This technical restriction was considered because Professor Jorge Cardoso, acting as a client for this solution demanded it. *OpenTSDB* is the database that they are currently using in their projects at Huawei Research Center. This restriction will ease their work to introduce changes if needed.

4.4 Architecture

In this Section, the architecture is presented based on all previous topics with resource to the defined Simon Brown's C4 Model [59]. This approach of defining an architecture uses four diagrams: *1 - Context Diagram*, *2 - Container Diagram*, *3 - Component Diagram* and *4 - Code Diagram*. To define the architecture for our solution, only the first three representations were considered. Every representation will be exposed with a explanation of the decisions taken to draw each diagram. After presenting the representations and the corresponding explanations, we will cycle thought all architectural drivers: QA, business and technical restrictions, in order to explain where they are reflected and the considerations taken to produce this architecture.

4.4.1 Context Diagram

In this Subsection the context diagram is presented. This diagram allows us to see "the big picture" of the overall system as it represents the system as a "big box" and the corresponding interactions with users and external software systems. Figure 4.3 presents the context diagram for this solution.

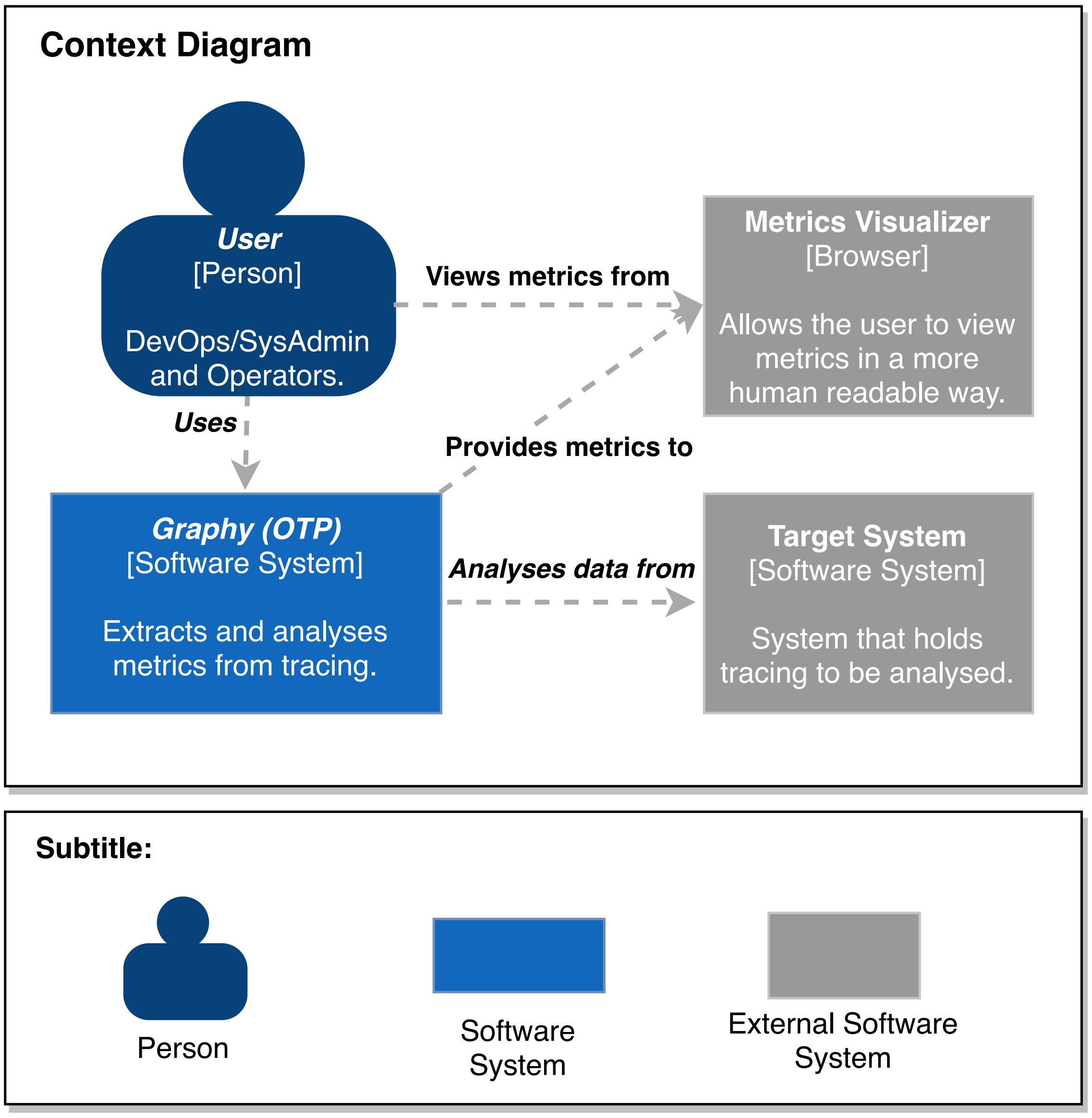


Figure 4.3: Context diagram.

From Figure 4.3, we can see that our solution, named *Graphy OpenTracing processor (OTP)*, receives interactions from users, as it need someone to start the whole process. This piece of software analyses data from an external target system that holds the tracing information and consequently, provides extracted metrics to an external metrics visualizer component. Users can view extracted metrics from this last component. Also, reports are produced and stored within our solution, when it performs tracing analysis.

4.4.2 Container Diagram

The container diagram is presented in this Subsection. This type of diagram allows us to "zoom-in" in the context diagram, and get a new overview of our solution. Therefore, in this diagram we are able to see a high-level shape of the software architecture and how responsibilities are distributed across containers. Figure 4.4 presents the container diagram for our proposed solution.

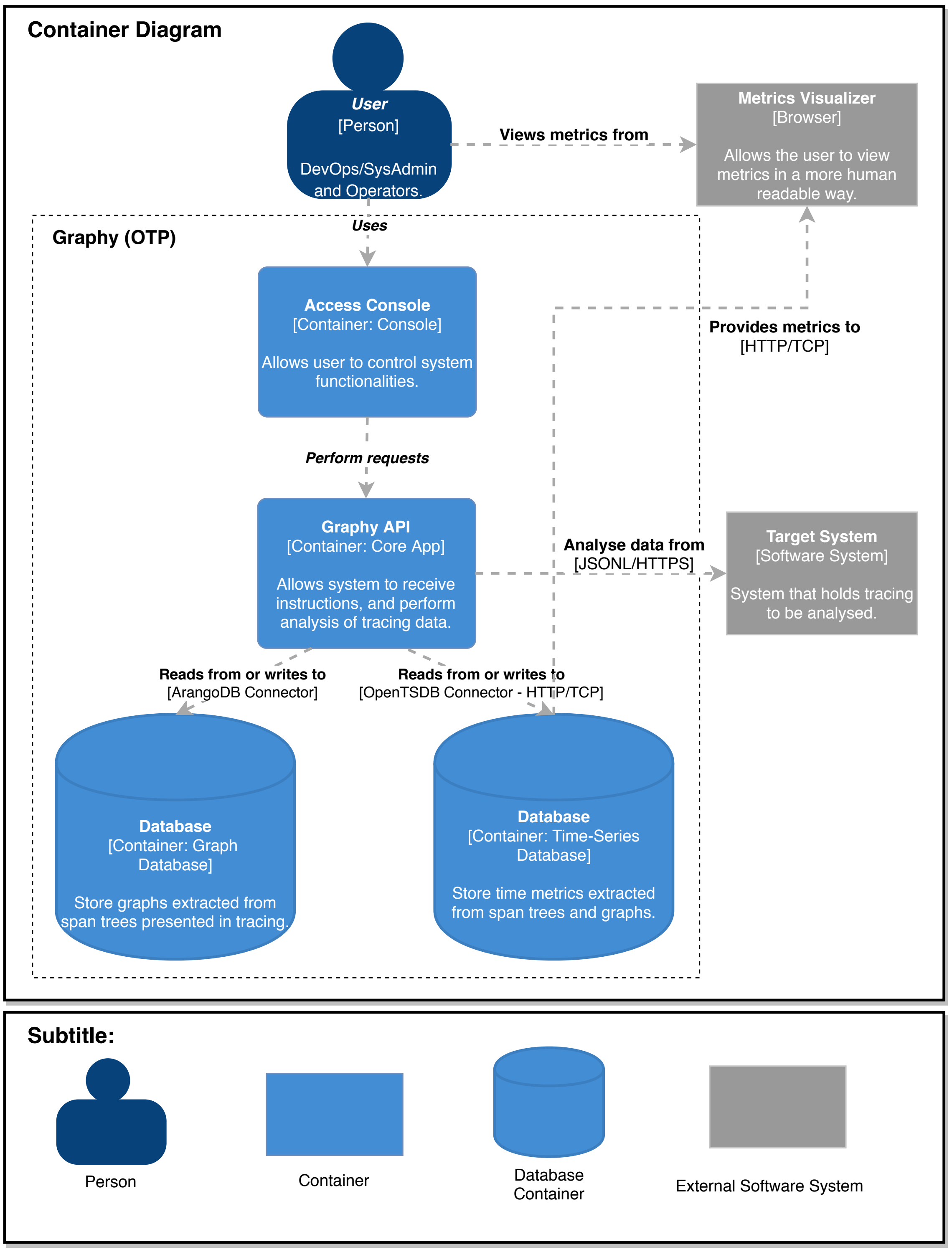

Container Diagram
User
[Person]
DevOps/SysAdmin and Operators.
Views metrics from
Metrics Visualizer
[Browser]
Allows the user to view metrics in a more human readable way.
Graphy (OTP)
Uses
Access Console
[Container: Console]
Allows user to control system functionalities.
Provides metrics to
[HTTP/TCP]
Perform requests
Graphy API
[Container: Core App]
Allows system to receive instructions, and perform analysis of tracing data.
Analyse data from
[JSONL/HTTPS]
Target System
[Software System]
System that holds tracing to be analysed.
Reads from or writes to
[ArangoDB Connector]
Reads from or writes to
[OpenTSDB Connector - HTTP/TCP]
Database
[Container: Graph Database]
Store graphs extracted from span trees presented in tracing.
Database
[Container: Time-Series Database]
Store time metrics extracted from span trees and graphs.
Subtitle:
Person
Container
Database Container
External Software System

Figure 4.4: Container diagram.

Figure 4.4 contains the main containers involved in our solution. The first one, from top to bottom, is the *Access Console* and this container was considered as it is needed for the user to be able interact with the *Graphy Application Programming Interface (API)*. This last one controls the entire OpenTracing system, uses a communication protocol to retrieve tracing information from external target system, and two databases to store the information resulted from processing tracing data – a Graph Database (GDB) and a Time Series Database (TSDB). The second database provides metrics to be visualized in an external metrics visualizer system.

4.4.3 Component Diagram

This Subsection contains the last diagram, the component diagram. This type of diagram gives a more deeper vision about the system, and therefore, it reveals the main components. Figure 4.5 presents the component diagram for this solution.

Figure 4.5 provides us with a lower level visualization of *Graphy API* container composed by eight components. At its core we have *Graphy Controller*, a component with the responsibility of receiving requests from the user through *Access Console* and control *OpenTracing Processor*, *Tracing Collector* and *Data Analyser* components. The first one has the objective of mapping tracing data, span trees and service dependency graphs instantiation into memory. The second one collects tracing, the information that feeds this entire application, from local files or from external systems, e.g., *Zipkin*. The last one, *Data Analyser*, identifies outliers presented in time-series metrics extracted from tracing, allowing our solution to detect anomalous services presented in distributed systems. *Graph Processor* is the component for graphs handling, thus it has the capability of performing operations over graphs, e.g., subtract one graph from another, extract node degrees and count connections between nodes. The remaining components, *Graphs Repository* and *Metrics Repository*, are used to map graphs and time-series metrics, respectively, into and from their corresponding databases.

To check the architecture produced, we will now cycle between both QA and check were they are reflected in the architecture presented for this solution, explaining the trade-off involved and what were our considerations about each one.

QA1 and QA2 are satisfied by the fact that the system is able to collect data from an external system. Using a communication protocol where data is exchanged thought Hypertext Transfer Protocol (HTTP) and exposed API, allows to externally request little chucks of data from target systems without interfering with their normal function.

Finally, for the only technical restriction raised, we can see that it is satisfied by the usage of *OpenTSDB* as the main TSDB for our solution.

This solution does not have many architectural drivers: quality attributes, business constraints and technical restrictions, due to being a prototype. The main objective is to produce a solution capable of explore tracing data allowing us to conduct a research about what we can do with tracing, therefore it does not have many architectural constraints. Nevertheless, with the presentation of these four sections, we conclude that our solution satisfies all the architectural drivers, and therefore, we may claim that the proposed architecture fits our needs as a solution.

Next Chapter, 5 - Implementation Process, covers the implementation of the solution presented in the current chapter. All implemented algorithms and technical decisions are discussed and explained in detail.

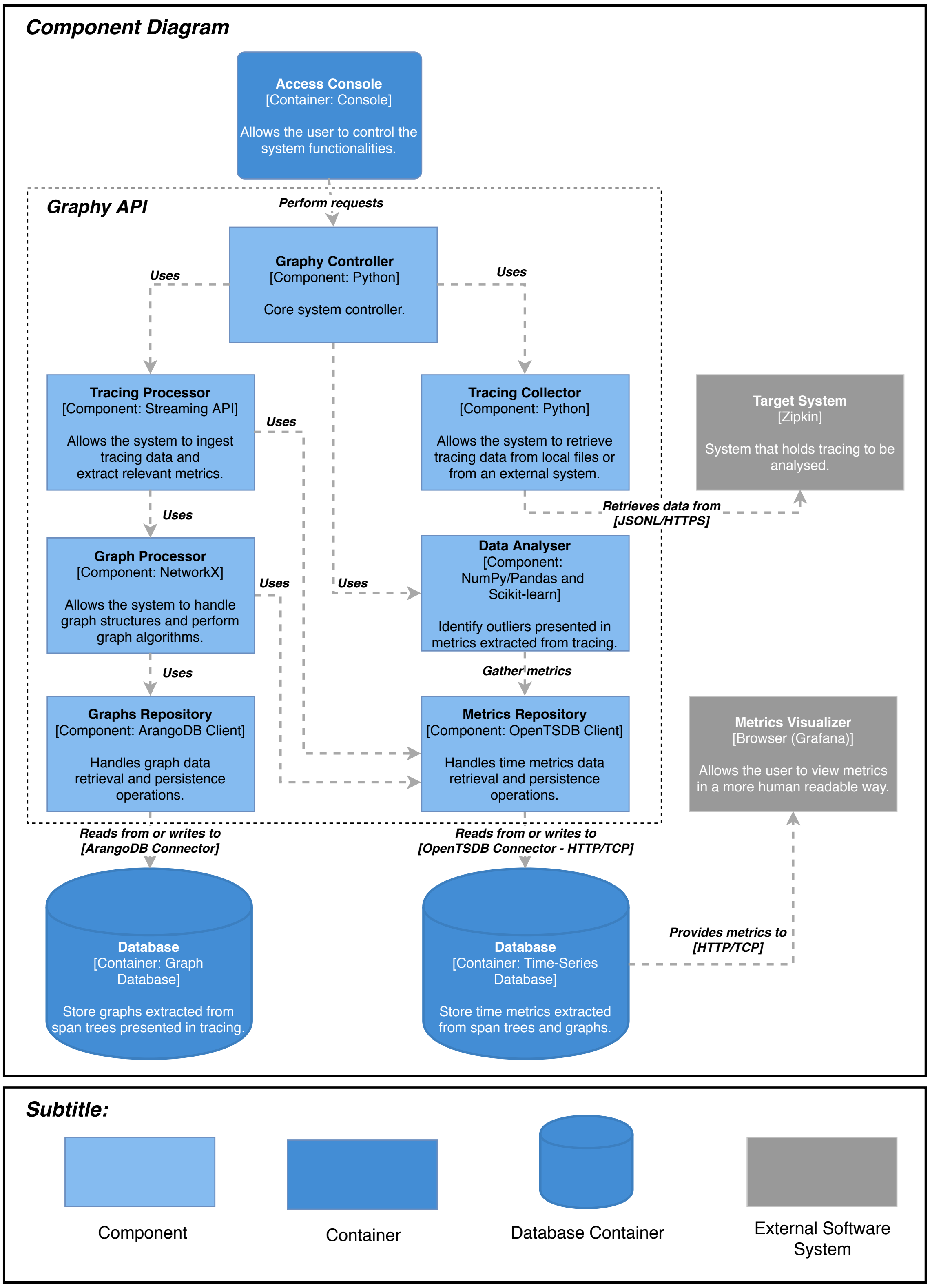

Component Diagram
Access Console
[Container: Console]
Allows the user to control the system functionalities.
Graphy API
Perform requests
Graphy Controller
[Component: Python]
Core system controller.
Uses
Uses
Tracing Processor
[Component: Streaming API]
Allows the system to ingest tracing data and extract relevant metrics.
Tracing Collector
[Component: Python]
Allows the system to retrieve tracing data from local files or from an external system.
Target System
[Zipkin]
System that holds tracing to be analysed.
Retrieves data from [JSONL/HTTPS]
Uses
Uses
Graph Processor
[Component: NetworkX]
Allows the system to handle graph structures and perform graph algorithms.
Uses
Uses
Data Analyser
[Component: NumPy/Pandas and Scikit-learn]
Identify outliers presented in metrics extracted from tracing.
Uses
Gather metrics
Graphs Repository
[Component: ArangoDB Client]
Handles graph data retrieval and persistence operations.
Metrics Repository
[Component: OpenTSDB Client]
Handles time metrics data retrieval and persistence operations.
Metrics Visualizer
[Browser (Grafana)]
Allows the user to view metrics in a more human readable way.
Reads from or writes to [ArangoDB Connector]
Reads from or writes to [OpenTSDB Connector - HTTP/TCP]
Database
[Container: Graph Database]
Store graphs extracted from span trees presented in tracing.
Database
[Container: Time-Series Database]
Store time metrics extracted from span trees and graphs.
Provides metrics to [HTTP/TCP]
Subtitle:
Component
Container
Database Container
External Software System

Figure 4.5: Component diagram.

Chapter 5

Implementation Process

This Chapter presents the implementation process of the proposed solution explained in previous Chapter. Three main sections are covered in this Chapter: Firstly, in Section 5.1 - Huawei Tracing Data Set, the data provided to perform this research is presented and analysed. Secondly, in Section 5.2 - OpenTracing Processor Component, the implementation of (OpenTracing processor (OTP)), our proposed solution to collect and store metrics from tracing data is explained in detail with intermediate results. Finally, in Section 5.3 - Data Analysis Component, the approach and methods for analysis of the stored observations are presented.

5.1 Huawei Tracing Data Set

The starting point for this solution and every method developed within it was a data set provided by Huawei, represented by professor Jorge Cardoso. To gain access to this information, a NDA: Non-disclosure agreement was signed by both parts. This data set contains the results of tracing data gathered from an experimental *OpenStack* cluster used by the company for testing purposes, and covers two days of operation. Consequently, two files were provided, one for each day. These files were generated in 10 of July, 2018 and, for protection, some fields of the data set were obfuscated during the generation process. Table 5.1 contains some details about the provided data set.

Table 5.1: Huawei tracing data set provided for this research.

File Date	2018-06-28	2018-06-29
Spans count	190 202	239 693
Traces count	64 394	74 331

From Table 5.1, we can see some detail regarding spans and trace counting for each day. Both files were written in JSONL format [60]. This file format is an extension to the lightweight data-interchange standard JavaScript Object Notation (JSON): JavaScript Object Notation, however, in JSONL format multiple JSON are separated by a new line character. Each span is presented by a single JSON, therefore, each line contains a span encoded in JSON format. To count spans a line count in each file was enough. To count traces, spans must be mapped to span trees, and then the total of trees represent the trace count. Algorithms to perform this conversion are presented further, in Section 5.2 - OpenTracing Processor Component.

Span data format is defined in an open source specification called OpenTracing [61], however, companies and software developers are not obliged to follow it, thus they can produce their own span data format, leading to difficulties developing a general purpose tool for tracing analysis. Therefore, to ease the interpretation of spans presented in the data set, a file with instructions about the specification was provided. In this file, a definition was given about possible fields and their corresponding data types. A sample of the fields and their descriptions are exposed in Table 5.2.

Table 5.2: Span structure definition.

Field	**Description**
traceId	Unique id of a trace (128-bit string).
name	Human-readable title of the instrumented function.
timestamp	UNIX epoch in milliseconds.
id	Unique id of the span (64-bit string).
parentId	Reference to id of parent span.
duration	Span duration in microseconds.

binaryAnnotations	protocol - “HTTP” or “function” for RPC calls;
	http.url - HTTP endpoint;
	http.status_code - Result of the HTTP operation.

annotations	value - Describes the position in trace (based on *Zipkin* format). Could be one of the following values or other: “cs” (client send), “cr” (client receive), “ss” (server send) or “sr” (server receive); timestamp - UNIX epoch in microseconds; endpoint - Which endpoint generated a trace event.

Also, the file contained two notes. To point each one, has they are very important, we present them bellow.

1. Time units are not consistent, some fields are in milliseconds and some are in microseconds.
2. Trace spans may contain more or less fields, except those mentioned here.

From Table 5.2, we get a notion about the fields that can be found in spans. These fields are defined by OpenTracing specification, therefore, is important that companies follows the specification, even if open source.

In this data set, spans are composed by: “traceId”, “name”, “timestamp”, “id”, “parentId” and “duration”. These are the main required fields, because they represent the foundations for tracing data, containing the identification, relation and temporal track of the span. Also, these fields are fixed, meaning that they are always represented by the defined field name. The same can not be said from the remaining fields: “binaryAnnotations” and “annotations”. These tow fields are always identified by these field names, however, their values are maps and therefore, have values stored in key - value pairs. This brings some consistency problems and we might not know clearly what is available in a span, when working with it. As said in the second point presented in the list above: “Trace spans may contain more or less fields, except those mentioned here”, and for this reason, there is a tremendous explosion in possibilities, because there might be keys with corresponding values for some particular spans and it gets hard to generalise this in a uniform span structure.

The notion of span data only depends on the quality of communication and documentation of the ones that produce tracing. To be certain that one crafts good tracing data, there must be an implemented standard for everyone to follow. The formalization and unification of one tracing specification should be a thing to consider, for the reason that it is an endeavour to analyse inconstant fields. For example, in the data provided spans can be of two types: HTTP span or RPC span, and the only field that distinguishes them is a field named “exec”, which stands for the execution process id, and is not presented in the HTTP span type. Another example, fields having the same key should have one and only one measurement unit, because distributed tracing tools (like the ones presented in Subsection 2.2.1) are not expecting different measurement units for the same field and therefore, assume wrong values when spans have timestamps declared in milliseconds and others in microseconds, like in this case. To fix this, we decided to convert all time measurements to milliseconds.

To provide notion of how traces and spans are spread throughout time, we have used our tool, Graphy OTP, to generate two charts that represents the counting of traces and spans for each hour in each day. We decided to generate two split charts due to the simple fact that we have one file for each day. To count the number of spans in time, in this case by hour, the tool only needed to group every span by hour and count them, however for traces, the tool has more work because it needs to merge all spans in their corresponding span tree (explained in Section 5.2). After having all span trees it just needs to count them, and the result is the number of traces. Note that if a span or trace starts at a given time $t1$ contained in a time-frame, and with its duration $d1$ surpassing the next time-frame $tf1$, $(t1 + d1 > tf1)$, it is considered to be in the first time-frame, or by other words, only the starting time of the trace or span is considered for the counting. Figures 5.1 and 5.2 presents the data set traces and spans counting throughout time.

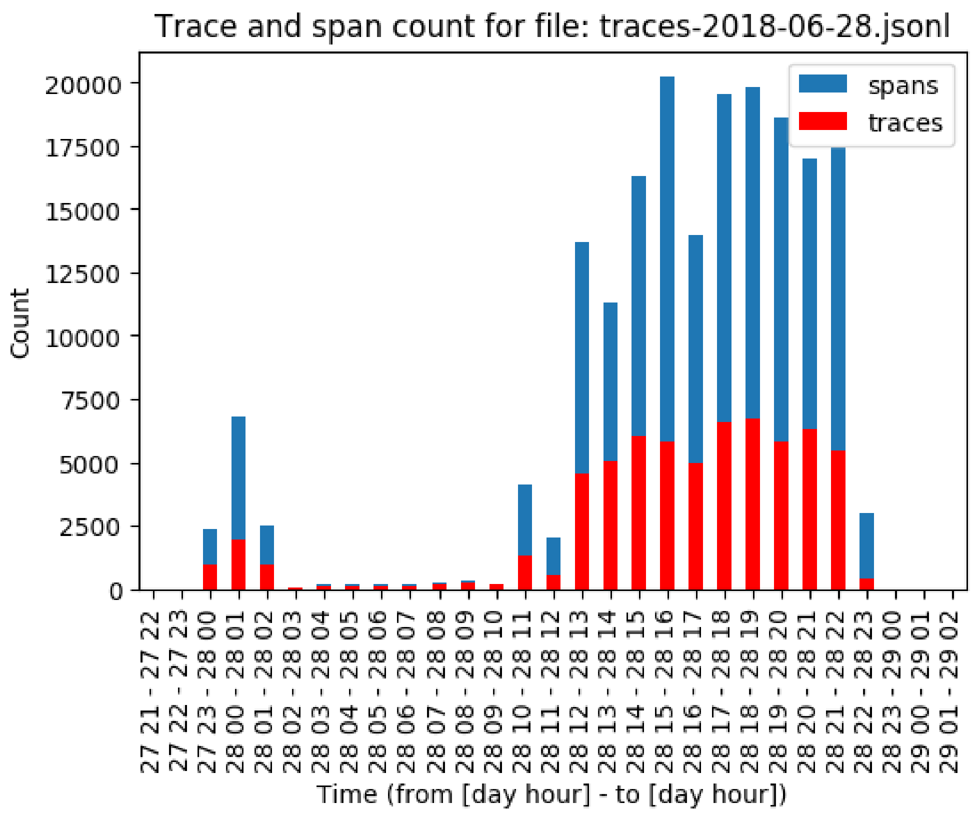


Figure 5.1: Trace file count for 2018-06-28.

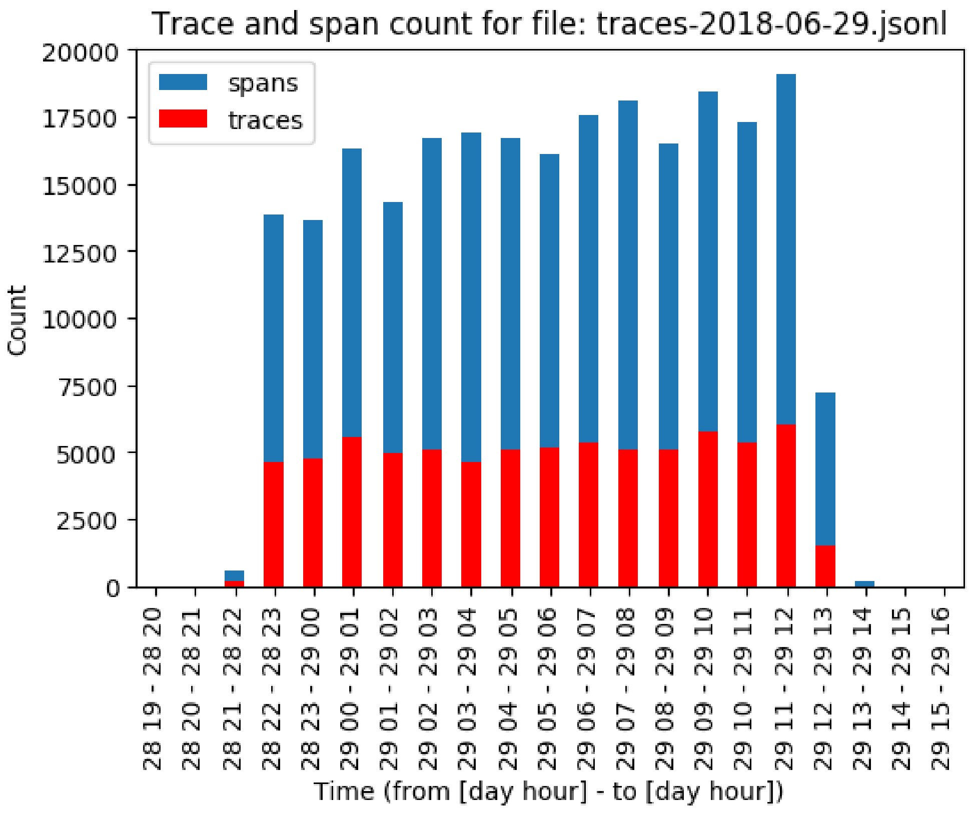


Figure 5.2: Trace file count for 2018-06-29.

Figure 5.1 presents the counting of traces and spans for the 28th of June, 2018. In this Figure we can spot a "pit" in quantity from 2AM to 10AM. No explanation for this was given, however, at this point we assumed that extracting metrics from data in this time interval would produce less points, thus less resolution. This is visible in metrics presented in Figure 5.3, reproduced using *Grafana*. The quantity of data for the rest of the day is somehow inconstant, however, there is no lack of data like in the previous day.

Figure 5.2 presents the counting of traces and spans for the 28th of June, 2018. In this Figure there are no "pits", and consequently, the quantity of information is more constant throughout time.

To summarise, this system produces an average of 5000 traces an hour and 15000 spans an hour. Also, the quantity of information provided in the second day (29th of June) is more constant, and therefore, better for analysis, than in the first day (28th of June). Nevertheless, this data set has sufficient information to study tracing data and develop methods for tracing data, and then, it is a suitable data set for this research project.

Next Chapter, 5.2 - OpenTracing Processor Component, covers the explanation and algorithms used over this data set for tracing metrics extraction and quality analysis. Also, some visualizations of metrics extracted from tracing data are provided.

5.2 OpenTracing Processor Component

In this Chapter, the implementation for the first component of the proposed solution, OTP, is presented and explained, hence, functional requirements defined in Table 4.1, service dependency graph handling, span trees generation and methods for metrics extraction and storing from tracing data will be covered.

Starting by the first two functional requirements (FR-1 to FR-2). These require communication with distributed tracing tools, to obtain tracing data and to retrieve service dependency graphs. We have decided to use *Zipkin*, as a distributed tracing tool for holding our data set, instead of Jaeger only due to simplicity in setup configuration. To setup this tool a *Docker* container was instantiated in an external server. Communication methods are implemented in *Tracing Collector* component. To feed information to our solution, one can use two ways: collect tracing data from local files, or export them to *Zipkin* and ingest it through HTTP requests. This configurations can be changed by editing a configuration file provided with the solution. The configurations to edit are file locations in local machine and *Zipkin* IP (Internet Protocol) address.

After collecting information from one of the two defined sources by *Tracing Collector*, data is passed to *Tracing Processor* which ingests and maps all the information into in memory data structures. Data can be either trace data or service dependency graphs. If it is a graph, it is transferred to *Graph Processor* for process, graph metrics extraction and later storage, otherwise, it is processed in *Tracing Processor* to extract defined metrics from tracing. The algorithm for metrics extraction from tracing and service dependency graphs is presented at a high abstraction level in Algorithm 1.

Algorithm 1: Algorithm for metrics extraction from tracing.

```
   Data: Trace files/Trace data.
   Result: Trace metrics written in the time-series database.
1  Connect to Time-Series database;
2  Read time_resolution, start_time and end_time from configuration;
3  Read traces from trace files/trace data;
4  Post traces to Zipkin;
5  Get services from Zipkin;
6  Calculate time_intervals using start_time, end_time and time_resolution;
7  while time_interval in time_intervals do
8      Get service_dependencies from Zipkin;
9      Build service_dependency_graph using service_dependencies;
10     Extract graph_metrics from service_dependency_graph;
11     while service in services do
12         Get traces from Zipkin;
13         Map traces in SpanTrees;
14         Extract service_metrics from SpanTrees;
15     Post graph_metrics to Time-Series database;
16     Post service_metrics to Time-Series database;
```

Algorithm 1 contains some core functionalities implemented in components presented in OTP solution. This algorithm aims for metrics extraction from tracing data and perform this procedure using two main data structures: service dependency graphs and tracing data mapped into SpanTrees.

Service dependency graphs are obtained from *Zipkin* and parsed directly into a *NetworkX* graph structure, presented in component *Graph Processor*. We decided to chose *NetworkX*, a framework for graph processing written in *Python*, due to tooling versatility has it contains a large implementation set of the majority graph algorithms. At this point we preferred this trade-off over processing power and scalability. *Zipkin* provides service dependency graphs through an explicit endpoint – */dependencies*, and a start and end timestamps in epoch milliseconds must be passed as parameters. The information comes in JSON format as presented in Listing 5.1.

```
[
    {
        "parent": "string",
        "child": "string",
        "callCount": 0,
        "errorCount": 0
    },
    { /* ... */ }
]
```

Listing 5.1: Zipkin dependencies result schema.

Listing 5.1 shows that dependencies come in an array of JSON objects. Each object contains the information about one relationship between services: parent “from”, child “to” and the number of calls. Therefore, having this information grant the creation of service dependency graph using *NetworkX*. Note that this information assembles a graph containing the information of system services at a specific time interval defined by provided parameters to *Zipkin /dependencies* endpoint. After having this information mapped into *NetworkX* graphs in memory, their visual representation are identical to the one demonstrated in Figure 2.5, presented in Subsection 2.1.4.

SpanTrees are a representation of a trace in a tree format. Method for their creation from a span list is presented in Algorithm 2.

Algorithm 2: Algorithm for SpanTree mapping from spans.

```
Data: Span list.
Result: Spans mapped into SpanTrees.
1 Index spans by ids from span list into SpanIndex;
2 while span in span list do
3     Read parentId from span;
4     Index span using parentId into SpanIndex;
```

Algorithm 2 shows that to transform a list of spans (unordered traces) into *SpanTrees*, one must index them by span id an then read every span, indexing the span using their *parentId*. After applying this method, spans will be properly indexed and a list of *SpanTrees* are produced. Also, a SpanTree is a representation of a trace and these structures ease tracing handling due to distinct causal relationships between spans. For example, one can use span trees to map TraceInfos. This data structure was created to hold relevant information from span trees: for example request work-flows. The process involves pinpointing requests between services, presented in spans throughout their causal relationship, and then store request paths through services, generating the corresponding request work-flow. For each span tree, one work-flow is generated, however, from root to leafs, multiple paths are possible. Note that not always do spans contain information to produce the path, and therefore, some request paths are dubious, depending only on the completeness quality of tracing. The method to produce request work-flows is described in Algorithm 3.

Algorithm 3: Work-flow type algorithm.

```
   Data: Trace files/Trace data.
   Result: Comma-separated values (CSV) with unique work-flow types, their
           corresponding count and times.
1  Read start_time and end_time from configuration;
2  Read SpanList from trace files/trace data within defined time_frame;
3  while have Spans in SpanList do
4      Read Span;
5      Map Span to SpanTrees;
6  while have SpanTree in SpanTrees do
7      Read SpanTree;
8      Map SpanTree to TraceInfos;
9      Read TraceInfo;
10     Read work-flows, work-flow count, times, (others) from TraceInfo;
11     Write fields to CSV files;
```

The method described in Algorithm 3 aims to use tracing to produce span trees, and then generate TraceInfos to retrieve request work-flow paths.

These two data structures, service dependency graphs and span trees, are the foundations to extract metrics from tracing data, satisfy the functional requirements presented in Section 4.1 and answer the final research questions defined in Section 3.2.

The metrics that OTP is able to extract from tracing data, for a defined time interval, are the following:

1. Number of incoming/outgoing service calls;
2. Average response time by service;
3. Service connection, i.e., other services invoking and being invoked by the system, i.e., the service dependency graph variation.
4. Service degree (in/out/total);
5. Service HTTP status code ratio. (sum of success or failure count over total status code count)

These metrics are all related with time and represent observations of values extracted from tracing data, therefore, as time-series metrics they are stored in a Time Series Database (TSDB). Explanation of used technology and procedure is provided later on this Section.

Table 5.3 relates each metric with a functional requirement, and correspondent final research question. Functional requirements are identified by an *id* from Table 4.1.

Table 5.3: Relations between final research questions, functional requirements and metrics.

Research Question	**Functional Requirements**	**Metrics**
1. Is there any anomalous service?	FR-5; FR-5; FR-6.	Number of incoming service calls; Number of outgoing service calls; Average response time by service.
2. What is the overall reliability of the service?	FR-7; FR-8.	No metric extracted; Service HTTP status code ratio.
3. Which service consumes more time when considering the entire set of requests?	FR-9; FR-10.	Service degree; Service dependency graph variation.

For Table 5.3, only functional requirements from numbers 5 to 10 were considered, due to being the ones related with metrics extraction. As said before, only question number 1 was considered for metrics extraction. The remaining, defined at grey colour, were implemented and OTP extracts them, however, they were not further analysed in this research. Almost all functional requirements have one metric associated except one, *FR-7*. This functional requirement was implemented, and our solution allows to generate work-flow paths from tracing data, however, no metric was defined. Nevertheless, the implementation of this functionality helped us understanding results for the first final research question – Method for work-flow generation from tracing data is explained Algorithm 3.

Span trees are a representation of causal relationship between spans. Two types of time based metrics are extracted from span trees: 1. Average response time by service in time; and 2. Service HTTP status code ratio in time. To extract the first metric type, *duration* and *annotations/endpoint/serviceName* values presented in spans , when defined, are used to calculate the average response time by service. For each span tree a list of services and their corresponding average times are obtained. After gathering all values from every span tree presented in the defined time-frame, the values are merged and posted to the TSDB. The second metric, is extracted through a calculation of status codes ratio by each service. For this, *binaryAnnotations/http.status_code* and *annotations/endpoint/serviceName* values are used. Also, equally to the previous metric, values are merged and posted to the TSDB.

Service dependency graphs are a representation of dependencies of services at a specific time-frame. Three types of time based metrics are extracted from service dependency graphs: 1. Number of incoming/outgoing service calls in time; 2. Entry/exit of services in time (service dependency graph node variation); and 3. Service degree (in/out/total) in time. To extract the first metric type, the values in between (Edges) services (Nodes) are retrieved. These values are dispatched for storage with service name, flow indication (incoming/outgoing), timestamp and number of calls. The second metric type is extracted having two successive graphs and performing their difference. For example, if $GraphA$ has nodes A, B, C and $GraphB$, nodes A, C, D, E, the difference between them will result in two service entries D, E and one exit. Last metric type, service degree, is extracted by retrieving the number of connections from each service. For example, consider that $GraphC$ has a service A connected from itself to services B, C, D. In this graph, service A has an out degree of three and an in degree of zero. The remaining services have an out degree of zero and an in degree of one. Methods to extract these metrics are implemented in *Graph Processor* and resource to *NetworkX* to handle graph structures. All these metrics are then posted to the TSDB.

At this point, our solution is able to retrieve and store time-series metrics from tracing data. For the TSDB, we have decided to use *OpenTSDB*, due to technical restrictions imposed in Section 4.3. There was a client implementation for usage in *Python*, however, the support was not good due to lack of updates and clear documentation. For this reason, we decided to implement our own *OpenTSDB* client in *Python* using their Application Programming Interface (API) specification, – *Metrics Repository* component. Later, when all implementation from tracing collection through trace metrics storage in the TSDB, we used a browser metrics visualizer. To do this, a *Docker* container with *Grafana*, a data visualization tool capable of rendering time-series metrics in charts and present them in dashboards. The decision to use this tool, was due to easy to setup and integrated compatibility with our TSDB. We just needed to create a container and, through a url configuration in *Grafana*, we established a link to the TSDB.

Figures 5.3, 5.4, 5.5, and 5.6, contain sample representations of extracted time-series metrics stored in our TSDB.

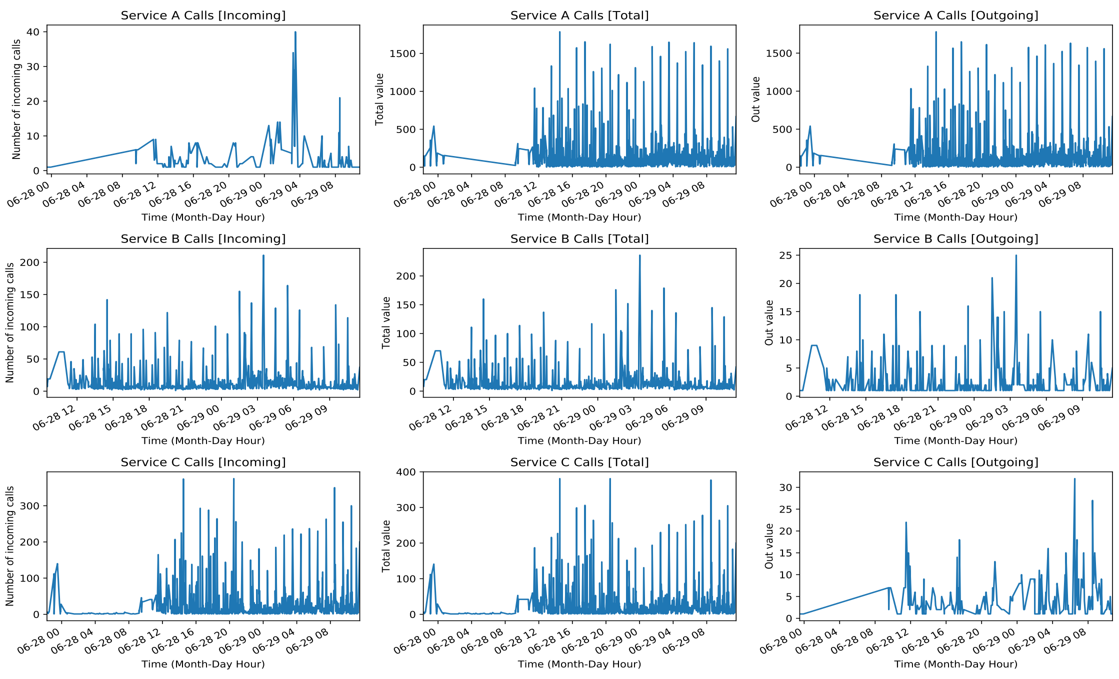


Figure 5.3: Service calls samples.

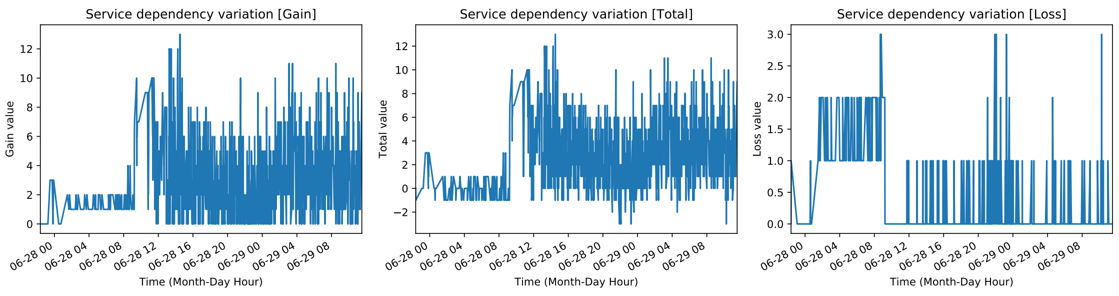


Figure 5.4: Service dependency variation samples.

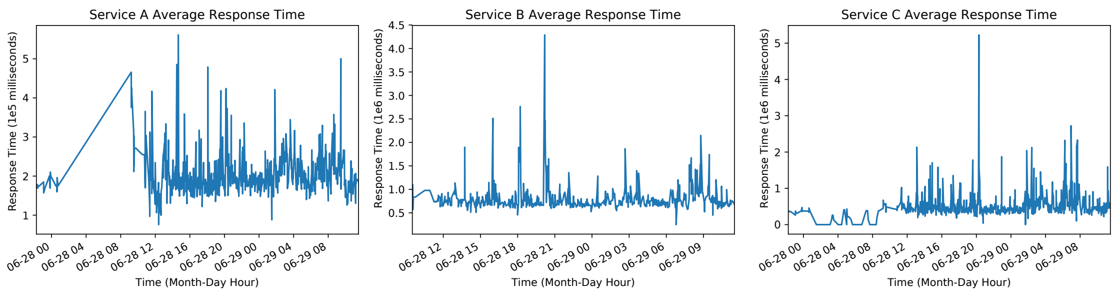


Figure 5.5: Service average response time samples.

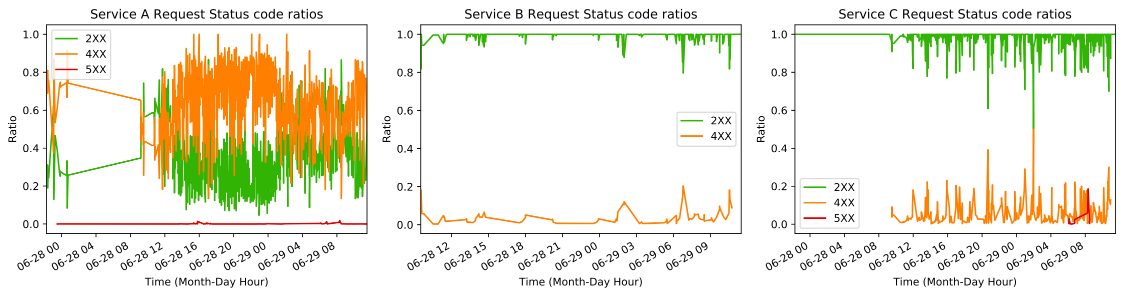


Figure 5.6: Service status code ratio samples.

Figure 5.3 represent samples about the number of service request calls metric. In this Figure, we have 9 plots, three in each row, representing three variations (incoming, outgoing and total) of this metric for one service. In this metric we can clearly see the lack of information presented in tracing for the beginning of the first day.

Figure 5.4 contain samples about service dependency variation, one for each metric (gain, loss and total). Total are the result of $gain - loss$. Gain stands for the number of new service entries in system, and loss, represent the number of service exits in system.

Last Figure, 5.6, shows the gathering of status code ratio samples for three distinct services. The ratio varies from 0.0 to 1.0, and represent the proportion of status code groups (2xx – Success, 4xx – Client error and 5xx – Other errors).

Also, service dependency graphs are stored for further access after being processed by *Graph Processor* component. We have decided to use *ArangoDB* as our Graph Database (GDB). This decision was based in the "Multi data-type support" provided by this database, allowing us to extend our graph structures to whatever we wanted, enhancing our graph storage possibilities and relieving the implementation from parsing data-types. This database has a *Python* client, *pyArango* [62], which revealed lack of features, leading to propositions for functionality creation and issue declarations in GitHub. However, the answers were not pleasant due to lack of support and people to maintain the project [63]. This have lead to some difficulties when implementing *Graphs Repository* component in OTP. Difficulties from storing graphs with custom names to custom graph retrieval were felted. The solution was to fork the project, perform changes and use our custom *pyArango* client. This changes were committed for review to the original project. We could not mitigate these problems in advance because they were only perceived when using the client.

After presenting the first component, OTP, from our proposed solution, next Section 5.3 - Data Analysis Component covers the implementation of the second component presented in our solution.

5.3 Data Analysis Component

In this Section, the implementation of the second component presented in our proposed solution, "Data Analysis" component, is presented and expected outcomes from each analysis are discussed.

"Data Analysis" component has the main objective of detecting anomalies, presented in services, using time-series metrics extracted from tracing data using the component presented in previous Section and perform tracing quality analysis.

In our implementation, this component is detached from the remaining components, however, in architectural terms we have decided to place it has being part of the overall solution. This is because there is nothing preventing total integration with other components presented in the solution. The reason to implement these methods detached from the remaining, was to ease our research path and increase flexibility. This means that, to ease our data exploration, implement these methods in Notebooks detached from the overall components, allowed us to change code effortlessly and conceded focus on methods development. Jupyter Notebook [64] was the notebook chosen for method implementation, hence, one server was created to hold our implementations in notebooks.

In this case, extracted time-series data belong to unlabelled data group. Data can belong to unlabelled or labelled groups. Unlabelled data are information sampled from of natural, or human-created artefacts, that one can obtain from observing and record values. In this group, there is no "explanation" for each piece of data, as it just contains the data, and nothing else. Labelled data typically takes a set of unlabelled data and augments each piece of data with some sort of meaningful "tag", "label", or "class" that is somehow informative or desirable to know. For example, for this solution, having labelled data would help in identifying anomalies presented in our data set. However, only unlabelled data was provided, and therefore, we needed to work with unlabelled data and perform anomaly detection with it [65].

So, the approach was to use processed data produced from OTP, and perform the analysis using "Data Analysis" component to point out service problems and perform tracing quality analysis, as defined in Figure 4.1, to answer questions defined in Chapter 3:

1. Is there any anomalous behaviour in the system? (If yes, where?);
2. How can we measure the quality of tracing?

To answer the first question, using our proposed solution, one must use metrics extracted from tracing data, namely the number of incoming / outgoing requests and the average response time for each service. These metrics are time-series metrics, and therefore, anomaly detection using unsupervised learning algorithms are the way to do it [25]. Metrics were obtained using methods defined in our *OpenTSDB* client, implemented in *Metrics Repository* component.

After extracting these metrics, they are allocated in a data structure called *dataframe* from *Pandas*, an open source library that provides high-performance, easy-to-use data structures and data analysis tools. This library was chosen due to being one of the most used and popular for this purpose [66] – data science and data analysis. A *Dataframe* is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labelled axes (rows and columns). In these data structures, values from time-series metrics were stored in columns: *timestamp* (index), *datetime*, *number_of_incoming_re-*

quests, *number_of_outgoing_requests* and *average_response_time*. In the end, a list of *dataframes* are created, one *dataframe* for each service.

Before performing an analysis to detect if there are outliers presented in our data, the information must be checked and tested to verify if data have missing values. This is done because metrics are extracted from multiple sources and thus generates missing values. For example, we may have missing information for one of the three features in a row of one *dataframe*. Missing values are a pain in data analysis and are represented by *NaN* in *dataframes*, and for this reason,one can not apply anomaly detection algorithms over data with missing values. To fulfil missing information there are two approaches:

1. Remove rows with missing values, which degrades the overall data and may result in insufficient data;

2. Impute missing values, however, it may be dangerous because it introduces ”wrong” observations.

We decided to impute missing values because there were too keep information quantity. However, there are multiple ways for imputation of missing values into time-series data, depending on factors of trend and seasonality. Trending is the increasing or decreasing value in the series, and seasonality is the repeating short-term cycle in the series [25]. Figure 5.7 shows the path to chose the correct method to fulfil information in time-series data.

So, before applying the method, our component tests the data to chose the correct method to fulfil data. Figure 5.8 contains trend and seasonality sample tests performed over our data.

Figure 5.8 shows that there are clearly trends in our data, however, no seasonality was detected. For this reason, the selected method to fulfil data presented in *dataframes* is Linear Interpolation – Figure 5.7.

These *dataframes* are then processed by an unsupervised learning algorithm to detect if there are outliers. For the unsupervised learning algorithm there were: Isolation Forests and OneClassSVM [68]. The first one uses binary decision trees to isolate data points and identify outliers presented in the data set, the second one, generates density areas using max-margin methods, i.e. they do not model a probability distribution, hence the idea is to find a function that is positive for regions with high density of points, and negative for small densities, identifying outliers presented in data. Figure 5.9 displays the error comparison of these two methods.

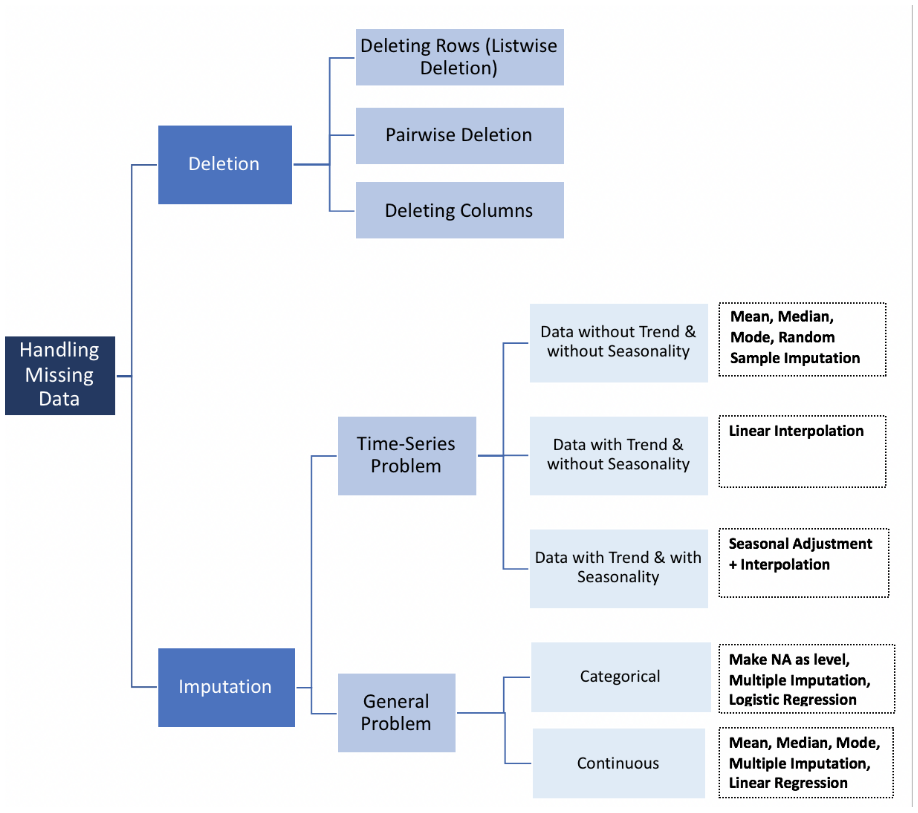

Handling Missing Data
Deletion
Deleting Rows (Listwise Deletion)
Pairwise Deletion
Deleting Columns
Imputation
Time-Series Problem
Data without Trend & without Seasonality
Mean, Median, Mode, Random Sample Imputation
Data with Trend & without Seasonality
Linear Interpolation
Data with Trend & with Seasonality
Seasonal Adjustment + Interpolation
General Problem
Categorical
Make NA as level, Multiple Imputation, Logistic Regression
Continuous
Mean, Median, Mode, Multiple Imputation, Linear Regression

Figure 5.7: Methods to handle missing data [67].

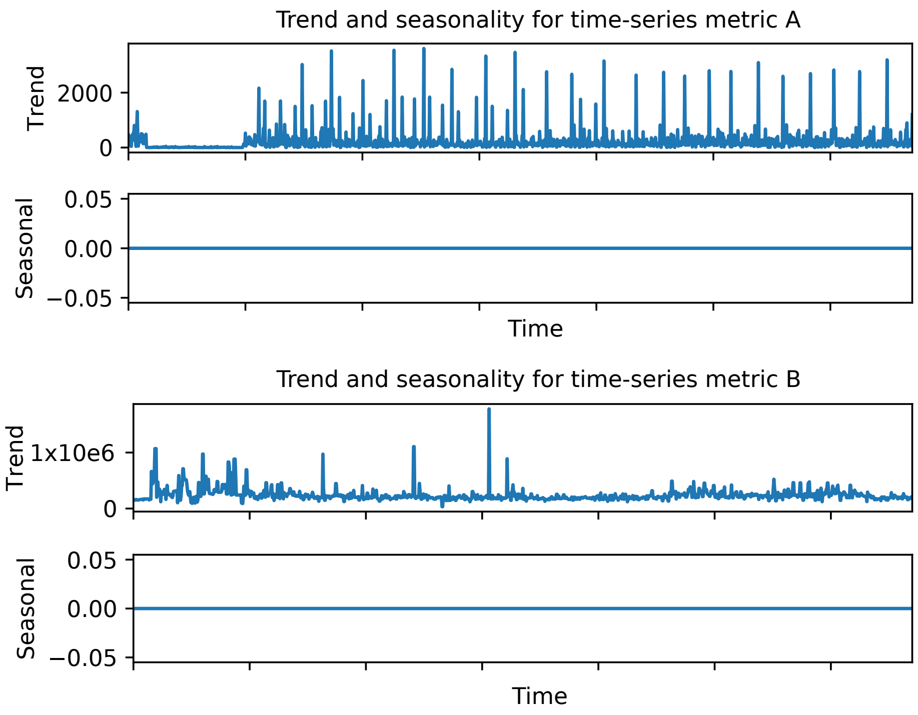


Figure 5.8: Trend and seasonality results.

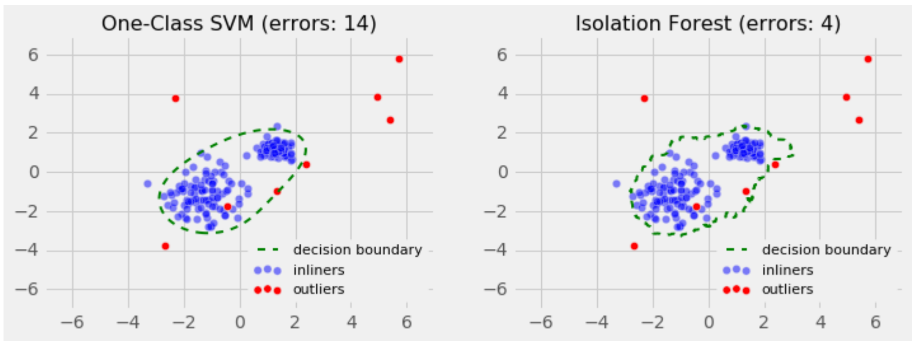


Figure 5.9: Isolation Forests and OneClassSVM methods comparison [69].

From Figure 5.9, isolation forests prove to be a better method for outlier detection because, from this test, it resulted in fewer errors as it did not construct a parametric representation of the search space. For this reason, we decided to use Isolation Forests, to detect and identify outliers presented in time-series metrics extracted from tracing data. To implement Isolation Forests method we used Scikit-Learn, a library full of simple and efficient tools for data mining, data analysis and machine learning. All configurations used from this library to implement Isolation Forests were setted to default. Therefore, Algorithm 4 presents the whole process to identify anomalous services presented in the system.

Algorithm 4: Anomalous service detection algorithm.

```
Data: Processed data from tracing using OTP.
Result: Report, in CSV file, containing identified anomalous services and
        correspondent times.
1 Read start_timestamp, end_timestamp, db_settings from configuration;
2 Connect to TSDB;
3 Retrieve metrics from TSDB using database connection, start_timestamp and
   end_timestamp;
4 Create dataframes with metrics data;
5 Perform data imputation over dataframes;
6 Feed Isolation Forests with metric columns from dataframes;
7 Fire Isolation Forests method (Adds new column “anomaly” with -1 “Anomalous”
   or 1 “Non-anomalous”);
8 Filter “anomaly” column with -1 values from dataframes into
   anomalous_dataframes;
9 Write report with anomalous service names and times from
   anomalous_dataframes data;
```

Algorithm 4 contains all the process explained above. The final outcome from this algorithm is a report containing all anomalous services and correspondent times identified. Also, later we decided to study further the pattern observed in anomalous regions. For this, the approach was to use the algorithm defined in 3 to analyse what happens to work-flows in “anomalous” and “non-anomalous” regions.

To answer the second question, it requires to perform a structural and time coverage analysis. For the first analysis, the approach is to define a specification schema based on *OpenTracing* open source specification. This schema aims to test span structures in order to detect structural problems present in spans, e.g., missed fields, wrong data types, typos presented in structure. The method implemented is presented in Algorithm 5.

Algorithm 5: Span structure analysis algorithm.

```
  Data: Trace files/Trace data.
  Result: CSV file reporting span structure analysis.
1 Read specification from open_tracing_specification_schema.json;
2 while not end of tracing do
3     Read Span;
4     Check Span against specification;
5 Write results from “Check” to CSV file;
```

As we can see in Algorithm 5, our method aims to produce a report containing the results of span structural analysis. To do this, first it needs to read the *OpenTracing* specification schema. This schema is written in a JSON file, where the fields are annotated with tags: *required*, *data-type: <string, int, other>* and others. JSON Schema [70] was the library used to verify if each span complies with the specification. For the second analysis, the approach is to use spans presented in trace data to analyse the coverage of each trace. Figure 5.10 presents an example for time coverage in a trace.

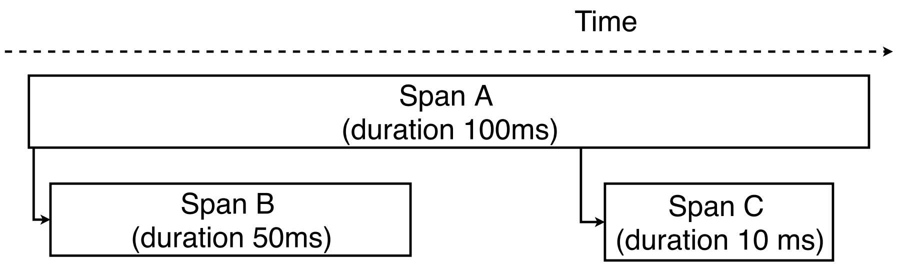


Figure 5.10: Trace time coverage example.

Figure 5.10 gives us an example in which we have a trace with a root span of 100 milliseconds of duration, and this root span has two children spans, one with $50ms$, the other one with $10ms$, the entire trace has a coverage of $(50+10)/100 = 60\%$. This method is applied to every trace, and the results are the stored in a CSV file to be plotted for visualisation. In this case we apply it and split the results by service, with the objective of perceive the time coverability of tracing in each service. The method is presented in Algorithm 6.

Algorithm 6: Trace coverability analysis algorithm.

```
  Data: Trace files/Trace data.
  Result: CSV file for each service reporting the coverability analysis.
1 Read start_time and end_time from configuration;
2 Get services from Zipkin;
3 while service in services do
4     Get traces from Zipkin using service, start_time and end_time;
5     Map traces in SpanTrees;
6     Calculate trace_coverability using SpanTrees;
7     Write trace_coverability to CSV file;
```

Algorithm 6 uses *SpanTrees* to calculate *trace_coverability*, this is due to causal relationships presented in these trees. As explained above, through Figure 5.10, one must have a trace mounted in span relationships (span trees), to know when a span is child of another, and be able to calculate the coverability presented in a trace. This method performs this calculation for every service and, in the end, stores information about trace coverability into a CSV file. This file is later used to produce plots about the service trace coverability. What is expected from this method is that we achieve a plotting, where every service has a counting of traces that cover a certain amount of time.

To summarise, this tools gathers processed data and time-series data from our TSDB, extracted using OTP from original trace information. Then it perform data imputation to solve missing values problems, analyses resulting data using Isolation Forests, an unsupervised multiple feature machine learning algorithm, to identify outliers presented in our extracted metrics, and therefore, detect anomalies presented in services, identifying their occurrences in time. Also, this tools uses tracing to perform an analysis about the structure of spans presented in tracing, and uses processed data from OTP, to perform an analysis of time coverage provided by tracing data.

Next Chapter 6, we will cover results obtained by this component, discuss these results and present *OpenTracing* data limitations.

Chapter 6

Results, Analysis and Limitations

In this Chapter we present the final results gathered from the "Data Analysis" component presented in Chapter 4 - Proposed Solution, to answer the questions defined in Section 3.2. Results for both questions, "1. Is there any anomalous service?" and "2. How can we measure the quality of tracing?", are presented as well as a brief discussion regarding both results in Sections 6.1 and 6.2 respectively. Later, in the end of this Chapter, in Section 6.3, we explore some limitations regarding the *OpenTracing* data.

6.1 Anomaly Detection

For the first question, the approach was to use the OpenTracing processor (OTP) tool to extract metrics from tracing data to further analyse it using the unsupervised learning algorithm. The implemented algorithm used for metrics extraction is presented in Algorithm 1.

After extract metrics, a tool for metrics visualisation (e.g., Grafana) was used to visualise metrics from *OpenTSDB* database. Samples from these visualizations were presented in Figures 5.3, 5.4, 5.5 and 5.6. Therefore, the method explained in Algorithm 4 was applied to metrics extracted from tracing data. From this algorithm, a Comma-separated values (CSV) file is generated containing candidates to "possible anomalous regions" for each service presented in the system. Figure 6.1 shows a sample of the result of outliers identified in time-series data for a given hypothetical service.

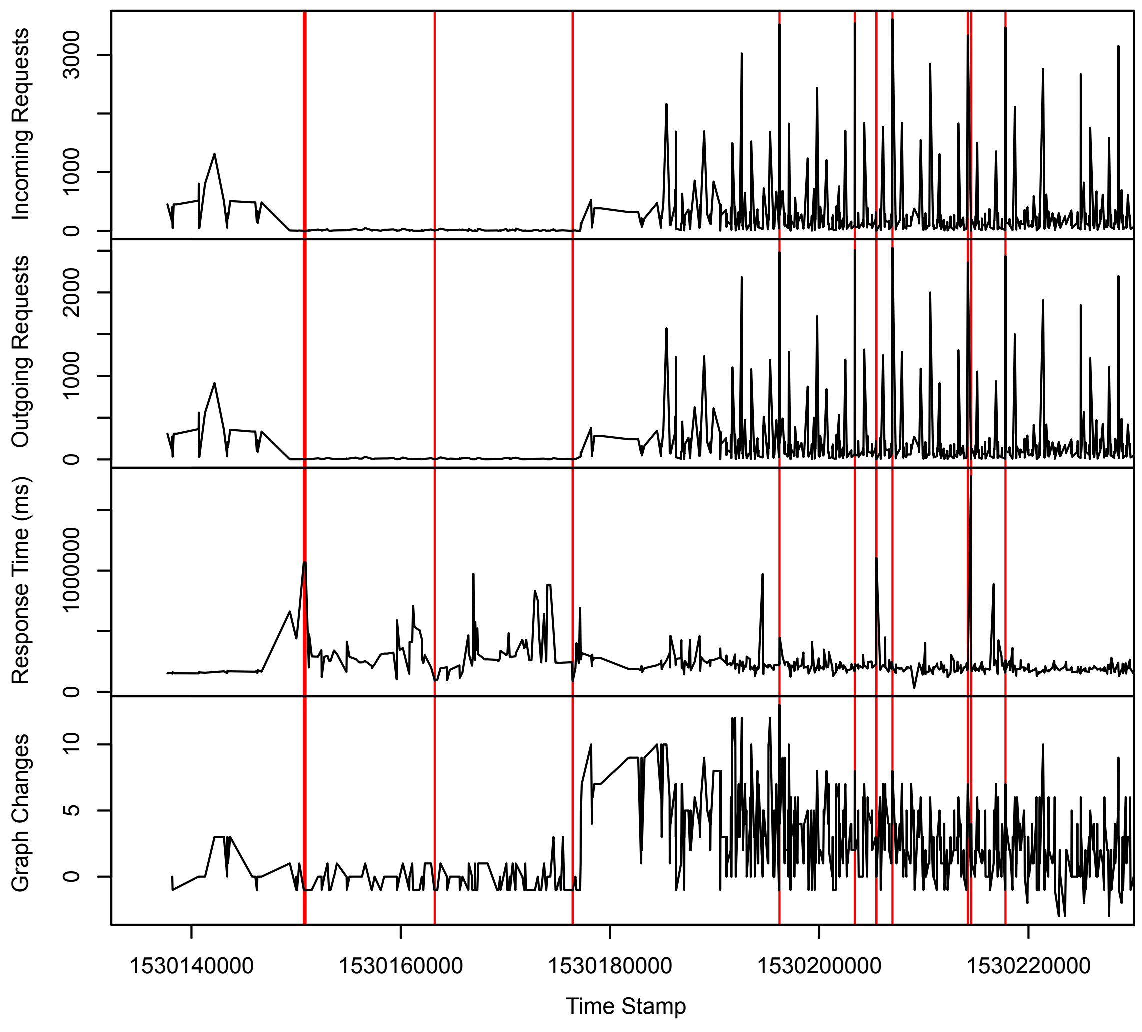


Figure 6.1: Sample of detection, using multiple feature, of "Anomalous" and "Non-Anomalous" time-frame regions for a service.

Figure 6.1 contains a set of vertical red lines representing the points of identified anomalies in time, involving the three distinct time-series metrics. From this outlier detection, using *Isolation Forests*, plots containing candidates to "possible anomalous regions" were generated. The outcome expected from these plots, were a clustering of values in normal ("non-anomalous") time-frame regions against clustering of values with outliers scattered in distant regions ("anomalous").

Figure 6.2 provides a representation of two time-frame samples, one for the "anomalous" region, and the other for the "non-anomalous" region considering the same service. In these samples we retrieved data to analyse and give answers to the first question. For this, we considered three features (as shown in the samples bellow): the number of incoming requests, the number of outgoing requests and the average response time. The sample resolution for the time-frame is 10 minutes centred in a given *timestamp*.

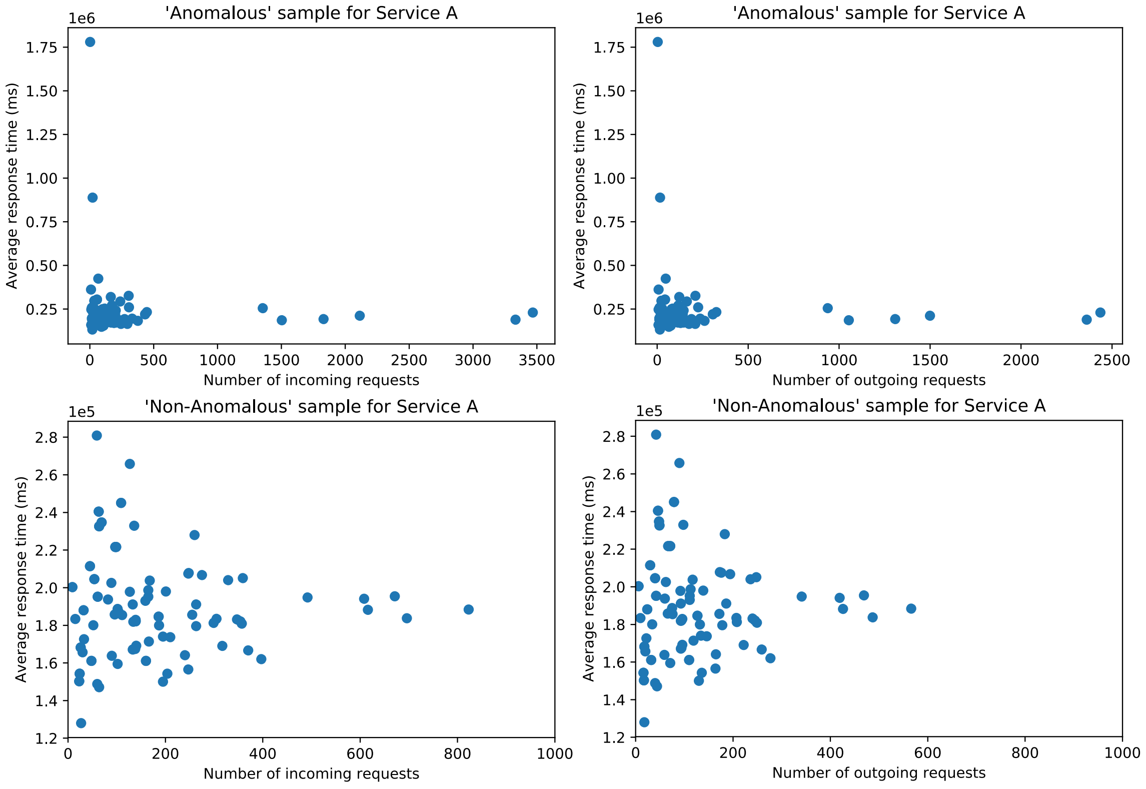


Figure 6.2: Comparison between "Anomalous" and "Non-Anomalous" service time-frame regions.

As we can see in Figure 6.2, there is a clear difference between anomalous and non-anomalous regions. There is a drastic change in the range of values between the anomalous and non-anomalous regions, where the maximum for each feature changes greatly and therefore, outliers are visible and evident in the observations. In the anomalous samples, it is possible to notice a clear crowding of points near the origin point of the chart and some outliers in the upper-left and down-right regions of the chart. On the other side, in the non-anomalous samples, all that is possible to notice is the crowding of points near the origin point of the chart. The crowding of points is what is expected to be the normal behaviour for services, which means that is expected that the service can handle the load with good response times. Furthermore, after this observations, what is expected is to investigate what these points represent and what is causing this unexpected increment in the number of incoming/outgoing requests and the average response time.

There are two anomalous situations observed:

1. Services are increasing the response time when there are few incoming/outgoing requests.
2. Services are receiving more incoming/outgoing requests, however it is having a good response time.

The first situation is much worse than the second one. The expectation is that services can handle more requests and keep the average response time, however, this system is being used for testing purposes, and it has been target of several load and fault injection tests. Furthermore, we do not have access to information regarding this tests, thus we can not be certain if the detected outliers can be considered real "anomalies" presented in services, however, they are interesting points to care about due to their unusual values. The worst case scenario would be to find points in the upper-right section of the charts, however this was not observed in this tracing data which leads to the assumption that this system is able to scale their workload well and therefore, it is capable of keeping response time low with large amounts of requests.

To study both situations, and further our anomaly detection presented in services, an analysis of trace request work-flow types was performed. The objective of this analysis is to perceive if there is some strange occurrences in request work-flow paths. To be able to perform this process, the OTP must be able to get the tracing data and map each unique trace work-flow for the given time-frame. We have used the method presented in Algorithm 3 to retrieve this information.

As presented in algorithm 3, parameters from TraceInfo are written to CSV files. These files are then processed in the "Data Analyser" component and, afterwards, a grouping of work-flow types from "Anomalous" and "Non-Anomalous" regions are retrieved for plotting. Results from this method are presented in Figure 6.3.

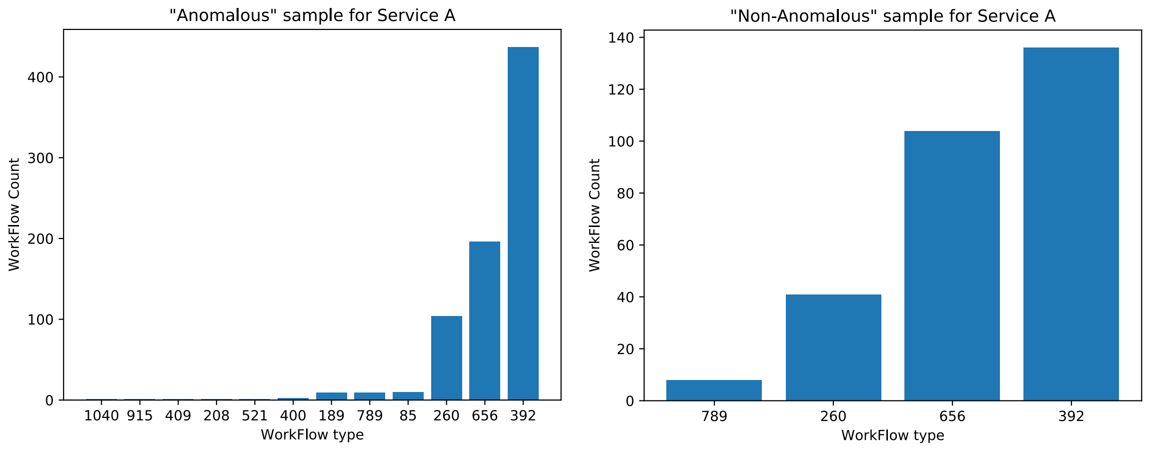


Figure 6.3: Comparison between "Anomalous" and "Non-Anomalous" service work-flow types.

Figure 3 shows a clear difference between work-flow types presented in "anomalous" and "non-anomalous" regions. One interesting thing to notice and that gives more evidence to prove anomalies presented in these regions is that, in the anomalous regions, more quantity and more types of request work-flow types were observed. The next step was to check what was causing this by retrieving the most "called" work-flows, however, the results were not good because of the completeness of the tracing data. The flows were not relevant to further our analysis because they were just calls from point A to point B, or represented a not so interesting request path due to involved services. Also, in some of these requests work-flow path, high values of response time were observed and therefore, they tocked longer to execute, however, like for the previous explanation, their path was not relevant to study. For these reasons, there were no possibility to extend our analysis and identify the root cause for these abnormal observed behaviours. Therefore, at this point and for this question, it is possible to say that this data set was exhaustively analysed, and an improvement of the tracing data, or the gathering of other types of data, e.g., monitoring and logging, should be a path to take. One point to note for future work is to test this method with other tracing data, to evaluate them and understand if this approach can lead to identification of the root cause of anomalous behaviour presented in services. For this reason, the data provided and thus, the OpenTracing in general has some limitations. These limitations are covered and explained further in Section 6.3.

6.2 Trace Quality Analysis

For the second question, the main approach was the same as in for the previous question, we need to use the OTP to process the tracing data and gather the results to be further analysed in the "Data Analysis" component. However, in this case, the results obtained by the first component were directly used by the second one.

In this question the analysis is divided in two procedures as explained in Chapter 4. The first procedure aims to check if the spans comply with the OpenTracing specification. This method is rather simple and is presented in Algorithm 5.

The results obtained by the application of this method were that every span structure complies with the specification. This is not a very good test because the specification of the *OpenTracing* is not very strict and therefore, the created method for testing does not provide a very accurate kind of results. To better explain this topic, we give two examples with some solutions for each one. First example, the units for timestamps are not uniform, one can use milliseconds and in other field of a span presented in the same trace, other can use microseconds. This leads to problems in time measurements and is not covered by this test. The solution for this problem can be the standardization of values and use only one measurement unit. Second example, there are multiple declarations for fields with key $\rightarrow$ value pairs, and thus, this brings inconsistency and uncertainty with the possible values that can appear. One solution for this is to redefine the semantic specification and terminology for programmers to adopt in their implementations. The limitations of this specifications and the redefinition of the *OpenTracing* specification is discussed later in Section 6.

The second procedure aims to check if tracing covers the entire time of the root spans. For a simple example, if we have a trace with a root span of 100 milliseconds of duration, and this root span has two children spans, one with $50ms$, the other one with $10ms$, the entire trace has a coverage of $(50 + 10)/100 = 60\%$. This method is applied to every trace, and the results are plotted for visualisation. In this case we apply it and split the results by service, with the objective of perceive the time coverability of tracing in each service. The method is presented in Algorithm 6 and the corresponding results, regarding two different services, are presented in Figure 6.4.

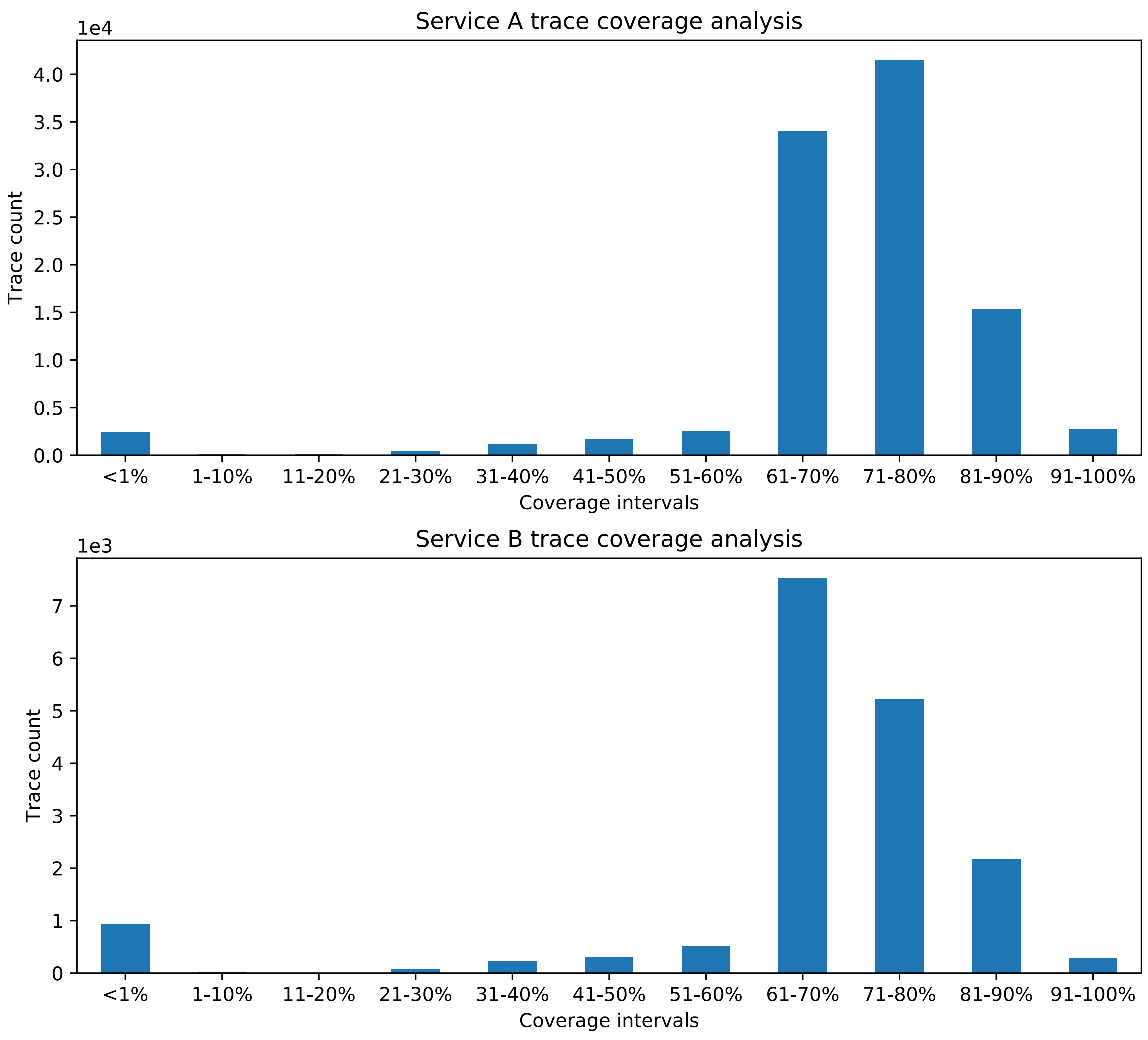


Figure 6.4: Services coverability analysis.

Figure 6.4 allows the user to visualise the tracing coverability, in terms of how many the overall tracing covers their entire execution duration. The most important thing to notice for this tracing data is the presence of higher bar values in $60\% - 100\%$ regions. This means that coverability for this tracing could be better, but in overall is good. What is expected by the result of this kind of analysis is that the coverability of tracing remains closer to the last interval ($90\% - 100\%$), which means that our service is fully/almost fully covered by this kind of data and therefore, the analysis of this data is worthy and the results provided by the usage of this it are trusty. From this data set, the results for the remaining services where close to the ones presented and shown by Figure 6.4.

After checking these results, one can use them to see which services developers can analyse in order to improve the coverage of tracing. To improve this coverage, changes in code instrumentation must be performed. Later on, after performing changes in code instrumentation and gather new trace information, the method for coverability test must be executed over the new tracing data to see if results have changed. What is expected is that trace coverability raises, which means that, for example, for service B presented in this figure, after developers changed the implementation, the trace counting for coverage must shift into higher intervals, and for this reason, one must observe lower values for $1\% - 70\%$ and higher values for $71\% - 100\%$. From this analysis, one thing to improve is to develop a method to analyse the gathered results in order to detect traces that do not cover their duration with respect to a predefined threshold. For example, the method could be applied to newer services after some time (to gather sufficient trace information), and then report or notify the developer if the service does not comply with the predefined coverage threshold. This would allow developers to improve their tracing coverage.

6.3 Limitations of OpenTracing Data

In this Section, we explore limitations felted when using and only using *OpenTracing* data in this research and therefore, give some solutions to improve this work and present a brand new project that emerged in the end of this research.

Limitations of *OpenTracing* were exposed in previous topics, Sections 6.1 and 6.2. These limitations are presented bellow:

1. There is no definition in the specification for which measurement units can be used when defining numeric values in spans, neither an exclusive field or in-field to indicate them;

2. Spans do not contain any field to indicate causally-related spans from different traces;

3. Specification does not provide a set of possible values for keys in key $\rightarrow$ value fields presented in spans;

4. Spans do not contain any field to identify correlated logs;

5. There are no defined way to record raw measurements or metrics with predefined aggregation and set of labels from tracing.

The first limitation brings problems regarding the definition of time units in spans. Without the clear indication of units used in these metrics, one may confuse the measurement and make the mistake of inferring misleading values, resulting in wrong spread of spans throughout time. This scenario occurred when posting our tracing data to distributed tracing tools, and to solve this, we needed to check the measurement unit defined by the tool. For this reason, this is a big problem, and therefore, this should be defined in the specification.

Second limitation causes the inability of knowing which spans are related with other ones when they are presented in different traces. Not having this information leads to lack of understanding of causally relationships between operations performed by distributed components. Therefore, to solve this issue, an additional field of causally-related spans or traces should be added to the span structure.

Third limitation consists in having fields of key $\rightarrow$ value pairs, when there is no definition of which keys can appear. This can be solved by creating a predefined schema where all possible key values must be indicated. It looks easy to fix this issue, however, to change the specification, there must be consensus in a unified structure and create new tools to process this new tracing data.

Fourth limitation, tracing contains relevant information about system work and can be used to map the flow of execution throughout the system, however, it could be much more complete if it contained a correlation between spans and logs. This could be solved if the span structure contained a field to declare related logs. Furthermore, the explicit declarations of logs would ease Development and Operations (DevOps) work because they retrieve logs manually by time intervals after searching in tracing for e.g., longer spans.

Last limitation says that tracing specification does not have a defined way to record raw measurements or metrics. This can be solved if specification and *OpenTracing* provided an Application Programming Interface (API) with defined metrics that could be exploited from tracing to be further analysed. This limitation was surpassed by creating a metrics extractor from tracing data in our proposed solution.

These limitations are generated by some issues presented in the specification of *OpenTracing*. Provide changes was not the focus of this research, however, these limitations carried out barriers for our own research because they bring difficulty when processing this type of data. For this reason, revise and perform adjustments to the whole specification is a job that must be done to ease tracing handling and analysis.

Near the end of this research, a project started with the support of big companies such as Google, Lightstep and Uber. This project, named as *OpenTelemetry* [71], is backed by CNCF: Cloud Native Computing Foundation and for this reason is open source. Started in April 2019 and has a defined roadmap to November 2019 with the main objective of merging *OpenCensus* and *OpenTracing*. The last one was the main focus of the research presented in this thesis because we only had access to tracing data. *OpenCensus* is a set of libraries for various languages that allows to collect application metrics, furthermore, this data can be analysed by developers and administrators to understand the health of the applications and debug problems [23].

The creation of *OpenTelemetry* and the interest from all these companies proves and emphasizes the whole work carried out during this research. All starting points for the creation of *OpenTelemetry* solution, stands in the problems and limitations of *OpenTracing* felted during this work and presented above. Furthermore, in June 2019, a revision of tracing specification was planned and worked out with the objective of introducing new standard tags, log fields, and change span context reference types [72]. Also, in this project, creators are planning to develop a metrics API, however, at time of writing, the only decision that was made is to use time-series to handle this kind of data but there is no specification created so far.

The usage of metrics, logs and other information come from the main objective of this project, merge *OpenCensus* and *OpenTracing*. Though the various components will be loosely coupled and consumed separately, the scope of the merged project includes data sources beyond distributed transaction traces. After all, instrumentation and observability involve other data sources, too. So the surface area of merged project API will incorporate a variety of signals, like metrics, traces and logs providing higher observability.

Observability stems from the discipline of control theory and refers to how well a system can be understood on the basis of the telemetry that it produces. From distributed systems, three major vertical data types are generated: Tracing, Metrics and Logging, and therefore, because they are tightly interconnected, one should use all of them to fully achieve observability of these systems. For this reason, Metrics can be used to pinpoint, for example, a subset of misbehaving traces. Logs associated with those traces could help to find the root cause of this behaviour. And then new metrics can be configured, based on this discovery, to catch this issue earlier next time. Furthermore, *OpenTelemetry* is an effort to combine all three verticals into a single set of system components and language-specific telemetry libraries and, in the end, replace both the *OpenTracing* project, which focused exclusively on tracing, and the *OpenCensus* project, which focused on tracing and metrics.

Chapter 7

Conclusion and Future Work

This Chapter covers three main topics: a summary of what we did and the main conclusions we reached from this research; followed by brief reflections regarding this whole research topic; and ending with the future work and research paths that seem to be promising for the future.

After this whole research, we are able to state that tracing data is useful and required to find anomalies related to service morphology. However, this type of data is hard to handle and one must use it if some issue was detected in metrics easier to analyse, e.g. monitoring. For this type of data to be easier to analyse, a discussion is provided about this difficulty bellow. So, in the end our perception is that, there are issues that we can only perceive using tracing data, but it is very expensive to analyse this data directly.

From tracing quality analysis, both tests are very interesting but, due to lack of required and strict specification, the tests and results of the "structural quality analysis" using spans are not very useful however, one can state that this is all we can do taking into consideration the *OpenTracing* specification.

In the end, our analysis of the provided tracing data generated by *OpenStack* – Huawei Cluster, took us to the following conclusions about *OpenTracing*:

1. *OpenTracing* suffers from a lack of tools for data processing and visualisation.
2. The *OpenTracing* specification is ambiguous.
3. The lack of tools to control instrumentation quality jeopardizes the tracing effort.

Firstly, we found it difficult to find appropriate tools for tracing data processing and visualisation. Only *Zipkin* and *Jaegger*, presented in the Subsection 2.2.1, are useful, as they allow distributed tracing visualisation in a human readable way. Unfortunately, they do not present any kind of tracing analysis. The need for additional open-source tools that can perform tracing analysis and visualisation is therefore quite real.

Secondly, one of the main difficulties in implementing the OpenTracing processor (OTP) and Data Analysis tools we mentioned in this thesis is the ambiguity in tracing data. The specification includes many fields that are not strictly defined. As mentioned in Section 6.2, one of the problems is the lack of standardization of measurement units, which led to different ones being used in the data provided. Other problem resides in some fields that contain very important information about the path of the request. These fields are defined as key-value pairs, where the keys vary freely according to the programmer's needs. This raises a major challenge for tools, which must infer the units, or assume that some data is unsuitable for analysis. A simple solution could be to redefine the specification and reduce this kind of fields, transforming the specification into a more strict schema. This would allow the implementation of more general trace processing tools.

Therefore, from this work, the following research paths are considered for future work:

1. Improve and develop new tools for *OpenTracing* processing.
2. Perform a research to redefine the *OpenTracing* specification.
3. Explore and analyse the remaining extracted tracing metrics.
4. Use tracing data from other systems.
5. Develop a simulated system with the capability of fault-injection to prove the analysis observations.
6. Conciliate the results from tracing data with other kinds of data like monitoring and logging.
7. Follow closely the development and the community of *OpenTelemetry* project, and contribute with ideas generated by this research.

First, today there are not many tools for processing and handling *OpenTracing* data. This increased difficulty is felt when we needed to process this kind of data in a different way, because we always ended up developing everything from scratch.

Second, there must be a way to eliminate or reduce the ambiguity and uncertainty of data presented in tracing generated by non-strict fields. If the specification can not be changed, a new way to transform tracing data to ease the analysis is very welcome. However, this is a topic that should be covered by the development of *OpenTelemetry* project, as mentioned in Section 6.3.

Third, these developed tools extract many more metrics. The majority of them were not explored due to lack of time, and therefore, here resides the opportunity to do it. The path starts by defining new research questions or analyse the remaining ones, presented in Section 3.2, that use these metrics and develop ways to analyse them.

Fourth, just one data set of tracing data was used in this research. Test the tools and methods with other tracing data could be an interesting path.

Fifth, the system were the data was gathered was a company testing system. One good future approach was to have a microservice based simulated system, were the developers could inject faults like request flow redirection, latency issues, and others, point them out and test the developed tools and methods.

Sixth, only tracing data was used in this research, one interesting path to follow is to have more kinds of data like monitoring and logging from the target system. This could help the analysis of the system, due to more knowing about it.

Seventh, after developed, *OpenTelemetry* solution could cover points 2 and 5 mentioned here. Also, expectation of success is high in the community, and commitment is visible in the project pulse. For these reasons, this project is a must watch in the following months.

We started with only tracing data provided by Huawei, and walked a path were we defined research questions based in Development and Operations (DevOps) needs and in *OpenTracing* characteristics. Later on, we designed a proposed solution capable of processing tracing data and extract metrics from this type of data. Then we implemented this solution and used it to retrieve results. These results proved some issues presented in *OpenTracing* specification and the difficulty that is to analyse a distributed system only using tracing information.

OpenTelemetry was created and started near the end of the research work presented in this thesis, with the core objective of merging *OpenCensus* and *OpenTracing* into a single Application Programming Interface (API), and consequently, review both specifications in order to modify and improve them. This projects emphasizes the work performed in this research, because the raised problems in this thesis are covered by it.

In the end, given the imposed limitations, one may conclude that this work was a success because the research directions are in the vanguard of the state of the art, related work and general community of tracing usage and analysis. Also, tools for tracing processing and analysis were developed and the created methods and conclusions were used to produce a scientific paper submitted to the International Symposium on Network Computing and Applications (IEEE NCA 2019).

This page is intentionally left blank.

References

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: Yesterday, today, and tomorrow”, in *Present and Ulterior Software Engineering*, Cham: Springer International Publishing, 2017, pp. 195–216, ISBN: 9783319674254. DOI: 10.1007/978-3-319-67425-4_12. [Online]. Available: https://hal.inria.fr/hal-01631455.

[2] C. Richardson, *Microservices Definition.* [Online]. Available: https://microservices.io/ (visited on 10/17/2018).

[3] P. D. Francesco, I. Malavolta, and P. Lago, “Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption”, in *2017 IEEE International Conference on Software Architecture (ICSA)*, IEEE, 2017, pp. 21–30, ISBN: 978-1-5090-5729-0. DOI: 10.1109/ICSA.2017.24.

[4] I. O’Reilly Media, *Monitoring Distributed Systems*, 2017. [Online]. Available: https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems/.

[5] S. P. R. Janapati, *Distributed Logging Architecture for Microservices*, 2017. [Online]. Available: https://dzone.com/articles/distributed-logging-architecture-for-microservices.

[6] OpenTracing.io, *What is Distributed Tracing?* [Online]. Available: %7Bhttps://opentracing.io/docs/overview/what-is-tracing%7D.

[7] Laura Mauersberger, *Microservices: What They Are and Why Use Them.* [Online]. Available: https://blog.leanix.net/en/a-brief-history-of-microservices (visited on 06/05/2019).

[8] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, *Cloud Container Technologies: a State-of-the-Art Review*, 2017. DOI: 10.1109/TCC.2017.2702586. [Online]. Available: http://ieeexplore.ieee.org/document/7922500/.

[9] S. Newman, *Building Microservices: Designing Fine-Grained Systems.* 280, ISBN: 978-1-491-95035-7. [Online]. Available: http://ce.sharif.edu/courses/96-97/1/ce924-1/resources/root/Books/building-microservices-designing-fine-grained-systems.pdf.

[10] M. Fowler and J. Lewis, *Microservices, a definition of this architectural term*, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html (visited on 01/07/2018).

[11] *Observing definition.* [Online]. Available: https://www.thefreedictionary.com/observing (visited on 10/13/2018).

[12] Peter Waterhouse, *Monitoring and Observability — What’s the Difference and Why Does It Matter? - The New Stack.* [Online]. Available: https://thenewstack.io/monitoring-and-observability-whats-the-difference-and-why-does-it-matter/ (visited on 06/06/2019).

[13] G. M. Brooker, *Feedback and Control Systems.* 2013, pp. 159–205. DOI: `10.1049/sbcs003e_ch4`. [Online]. Available: `http://people.disim.univaq.it/~costanzo.manes/EDU_stuff/Feedback%20and%20Control%20System_DiStefano_Schaum_Ch01-09.pdf`.

[14] R. R. Sambasivan, I. Shafer, J. Mace, B. H. Sigelman, R. Fonseca, and G. R. Ganger, "Principled workflow-centric tracing of distributed systems", 2016, pp. 401–414. DOI: `10.1145/2987550.2987568`. [Online]. Available: `https://www.rajasambasivan.com/wp-content/uploads/2017/07/sambasivan-socc16.pdf`.

[15] OpenTracing, *OpenTracing Data Model Specification.* [Online]. Available: `https://github.com/opentracing/specification/blob/master/specification.md` (visited on 12/10/2018).

[16] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework", in *Proceedings of the 4th USENIX conference on Networked systems design & implementation (NSDI'07)*, USENIX Association, 2007, p. 20. DOI: `10.1.1.108.2220`.

[17] R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger, "So, you want to trace your distributed system? Key design insights from years of practical experience", p. 25, 2014. [Online]. Available: `http://www.pdl.cmu.edu/PDL-FTP/SelfStar/CMU-PDL-14-102.pdf`.

[18] *The OpenTracing Semantic Specification*, https://github.com/opentracing/specification/blob/master/specification.md.

[19] *The OpenTracing Semantic Conventions*, https://github.com/opentracing/specification/blob/master/semantic_conventions.md.

[20] Cloud Native Computing Foundation, *What is Kubernetes?* [Online]. Available: `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/` (visited on 11/29/2018).

[21] OpenStack, *What is OpenStack?* [Online]. Available: `https://www.openstack.org/software/` (visited on 11/29/2018).

[22] OpenTracing.io, *What is OpenTracing?* [Online]. Available: `https://opentracing.io/docs/overview/what-is-tracing/` (visited on 11/29/2018).

[23] Google LLC, *What is OpenCensus?* [Online]. Available: `https://opencensus.io/` (visited on 11/29/2018).

[24] R. Sedgewick and K. Wayne, *Algorithms, 4th Edition - Graphs.* Addison-Wesley Professional, 2011. [Online]. Available: `https://algs4.cs.princeton.edu/42digraph/`.

[25] D. R. Brillinger, *Time Series: Data Analysis and Theory.* 4. Society for Industrial and Applied Mathematics, 2006, vol. 37, p. 869, ISBN: 0898715016. DOI: `10.2307/2530198`. [Online]. Available: `https://books.google.pt/books/about/Time_Series.html?id=PX5HExMKER0C&redir_esc=y`.

[26] H. Liu, S. Shah, and W. Jiang, "On-line outlier detection and data cleaning", *Computers and Chemical Engineering*, vol. 28, no. 9, pp. 1635–1647, 2004, ISSN: 00981354. DOI: `10.1016/j.compchemeng.2004.01.009`.

[27] Nikolaj Bomann Mertz, *Anomaly Detection in Google Analytics — A New Kind of Alerting.* [Online]. Available: `https://medium.com/the-data-dynasty/anomaly-detection-in-google-analytics-a-new-kind-of-alerting-9c31c13e5237` (visited on 06/06/2019).

[28] *Jaeger: open source, end-to-end distributed tracing.* [Online]. Available: https://www.jaegertracing.io/ (visited on 06/09/2019).

[29] *Apache Zipkin · A distributed tracing system.* [Online]. Available: https://zipkin.apache.org/ (visited on 06/09/2019).

[30] R. J. Trudeau and R. J. Trudeau, *Introduction to graph theory.* Dover Pub, 1993, p. 209, ISBN: 0486318664. [Online]. Available: https://books.google.pt/books/about/Introduction_to_Graph_Theory.html?id=eRLEAgAAQBAJ&redir_esc=y.

[31] Apache Software Foundation, *Apache Giraph.* [Online]. Available: http://giraph.apache.org/ (visited on 12/03/2018).

[32] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory", Pittsburgh, [Online]. Available: https://www.cs.cmu.edu/%7B~%7Djshun/ligra.pdf.

[33] *NetworkX.* [Online]. Available: https://networkx.github.io/ (visited on 12/03/2018).

[34] A. Morin, J. Urban, and P. Sliz, "A Quick Guide to Software Licensing for the Scientist-Programmer", *PLoS Computational Biology*, vol. 8, no. 7, F. Lewitter, Ed., e1002598, 2012, ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1002598. [Online]. Available: https://dx.plos.org/10.1371/journal.pcbi.1002598.

[35] A. Deshpande, *Surveying the Landscape of Graph Data Management Systems.* [Online]. Available: https://medium.com/@amolumd/graph-data-management-systems-f679b60dd9e0 (visited on 11/24/2018).

[36] J. Celko, "Graph Databases", in *Joe Celko's Complete Guide to NoSQL*, 2013, pp. 27–46, ISBN: 1449356265. DOI: 10.1016/b978-0-12-407192-6.00003-0.

[37] Favio Vázquez, *Graph Databases. What's the Big Deal? – Towards Data Science*, 2019. [Online]. Available: https://towardsdatascience.com/graph-databases-whats-the-big-deal-ec310b1bc0ed (visited on 06/07/2019).

[38] ArangoDB Inc., *ArangoDB Documentation.* [Online]. Available: https://www.arangodb.com/documentation/ (visited on 10/16/2018).

[39] Amenya, *TAO — Facebook's Distributed database for Social Graph*, 2018. [Online]. Available: https://medium.com/coinmonks/tao-facebooks-distributed-database-for-social-graph-c2b45f5346ea (visited on 06/07/2019).

[40] Neo4J Inc., *No Title.* [Online]. Available: https://neo4j.com/docs/ (visited on 10/16/2018).

[41] B. M. Sasaki, J. Chao, and R. Howard, *Graph Databases for Beginners.* 2018, p. 45. [Online]. Available: https://go.neo4j.com/rs/710-RRC-335/images/Graph_Databases_for_Beginners.pdf?_ga=2.124112970.1994598198.1521285291-1141717847.1521285291&_gac=1.180373973.1521290471.CjwKCAjw-bLVBRBMEiwAmKSB.

[42] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's Distributed Data Store for the Social Graph", [Online]. Available: https://cs.uwaterloo.ca/~brecht/courses/854-Emerging-2014/readings/data-store/tao-facebook-distributed-datastore-atc-2013.pdf.

[43] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One Trillion Edges: Graph Processing at Facebook-Scale Avery", *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015, ISSN: 15782190. DOI: 10.1016/S0001-7310(16)30012-6. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1804-ching.pdf.

[44] ArangoDB Inc., *ArangoDB Enterprise: SmartGraphs.* [Online]. Available: https://www.arangodb.com/why-arangodb/arangodb-enterprise/arangodb-enterprise-smart-graphs/ (visited on 12/15/2018).

[45] A. Turu, P. Ozge, K. Supervisor, and E. Zimányi, "Université libre de Bruxelles Graph Databases and Neo4J", Tech. Rep., 2017. [Online]. Available: https://cs.ulb.ac.be/public/_media/teaching/neo4jj_2017.pdf.

[46] K. V. Gundy, *Infographic: Understanding Scalability with Neo4j.* [Online]. Available: https://neo4j.com/blog/neo4j-scalability-infographic/ (visited on 12/15/2018).

[47] T. Dunning and E. Friedman, *Time Series Databases New Ways to Store and Access Data.* 2015, p. 71, ISBN: 9781491917022. [Online]. Available: https://www.academia.edu/29891282/Time_Series_Databases_New_Ways_to_Store_and_Access_Data.

[48] Tanay Pant, *Ingesting IoT and Sensor Data at Scale – Hacker Noon*, 2019. [Online]. Available: https://hackernoon.com/ingesting-iot-and-sensor-data-at-scale-ee548e0f8b78 (visited on 06/07/2019).

[49] InfluxData, *InfluxDB GitHub.* [Online]. Available: https://github.com/influxdata/influxdb (visited on 12/12/2018).

[50] OpenTSDB, *OpenTSDB.* [Online]. Available: https://github.com/OpenTSDB/opentsdb (visited on 12/12/2018).

[51] C. Churilo, *InfluxDB Markedly Outperforms OpenTSDB in Time Series Data & Metrics Benchmark.* [Online]. Available: https://www.influxdata.com/blog/influxdb-markedly-outperforms-opentsdb-in-time-series-data-metrics-benchmark/ (visited on 12/12/2018).

[52] S. Noor, Z. Naqvi, S. Yfantidou, E. Zimányi, and Z. Zimányi, "Time Series Databases and InfluxDB", Tech. Rep., 2017. [Online]. Available: %7Bhttps://cs.ulb.ac.be/public/_media/teaching/influxdb_2017.pdf%7D.

[53] S. Jacob, *The Rise of AIOps: How Data, Machine Learning, and AI Will Transform Performance Monitoring*, https://www.appdynamics.com/blog/aiops/aiops-platforms-transform-performance-monitoring/.

[54] A. Lerner, *AIOps Platforms*, https://blogs.gartner.com/andrew-lerner/2017/08/09/aiops-platforms/.

[55] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly Detection and Classification using Distributed Tracing and Deep Learning", 2018, [Online]. Available: https://pt.slideshare.net/JorgeCardoso4/mastering-aiops-with-deep-learning.

[56] W. Li, *Anomaly Detection in Zipkin Trace Data*, 2018. [Online]. Available: https://engineering.salesforce.com/anomaly-detection-in-zipkin-trace-data-87c8a2ded8a1.

[57] B. Herr and N. Abbas, *Analyzing distributed trace data*, 2017. [Online]. Available: https://medium.com/@Pinterest_Engineering/analyzing-distributed-trace-data-6aae58919949.

[58] M. Ikonen, E. Pirinen, F. Fagerholm, P. Kettunen, and P. Abrahamsson, "On the impact of Kanban on software project work: An empirical case study investigation", in *Proceedings - 2011 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011*, IEEE, 2011, pp. 305–314, ISBN: 9780769543819. DOI: 10.1109/ICECCS.2011.37. [Online]. Available: http://ieeexplore.ieee.org/document/5773404/.

[59] S. Brown, *The C4 model for software architecture.* [Online]. Available: https://c4model.com/ (visited on 12/12/2018).

[60] I. Ward, *JSON Lines.* [Online]. Available: http://jsonlines.org/ (visited on 04/18/2018).

[61] OpenTracing.io, *The OpenTracing Specification repository.* [Online]. Available: https://github.com/opentracing/specification.

[62] *pyArango: Python Client Driver for ArangoDB.* [Online]. Available: https://github.com/ArangoDB-Community/pyArango (visited on 06/14/2019).

[63] A. Bento and T. Daouda, *pyArango Issues*, 2019. [Online]. Available: https://github.com/ArangoDB-Community/pyArango/issues/137.

[64] D. Avila and M. Bussonnier, *Jupyter Notebooks.* [Online]. Available: https://jupyter.org/.

[65] R. Kothari and V. Jain, "Learning from labeled and unlabeled data", [Online]. Available: http://ieeexplore.ieee.org/document/1007592/.

[66] Pandas-dev, *Pandas - Flexible and powerfull time-series data analysis.* [Online]. Available: https://github.com/pandas-dev/pandas.

[67] A. Swalin, "How to Handle Missing Data", pp. 1–10, 2019. [Online]. Available: https://towardsdatascience.com/how-to-handle-missing-data-8646b18db0d4.

[68] C. Zhou and R. C. Paffenroth, "Anomaly Detection with Robust Deep Autoencoders", in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, New York, New York, USA: ACM Press, 2017, pp. 665–674, ISBN: 9781450348874. DOI: 10.1145/3097983.3098052. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3097983.3098052.

[69] A. C. Bahnsen, *Isolation forests for anomaly detection improve fraud detection.* 2016. [Online]. Available: https://blog.easysol.net/using-isolation-forests-anamoly-detection/ (visited on 06/18/2019).

[70] *JSON Schema.* [Online]. Available: https://json-schema.org/ (visited on 06/16/2019).

[71] *OpenTelemetry.* [Online]. Available: https://opentelemetry.io/ (visited on 06/21/2019).

[72] *OpenTelemetry Semantic Conventions.* [Online]. Available: https://github.com/open-telemetry/opentelemetry-specification/blob/master/semantic-conventions.md (visited on 06/22/2019).