Pedro Henrique Jones Deodato

Bachelor in Computer Science and Informatics Engineering

Runtime Tracing of Low-Code Applications: A Case Study for the OutSystems Platform

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João M. Lourenço, Associate Professor, NOVA University Lisbon

Co-adviser: Hélder Gregório, Senior Software Engineer, OutSystems

Examination Committee:

Chair:	Prof. Hervé Paulino, Associate Professor, NOVA University Lisbon
Rapporteur:	Prof. João Pascoal Faria, Associate Professor, University of Porto
Member:	Prof. João M. Lourenço, Associate Professor, NOVA University Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2021

Runtime Tracing of Low-Code Applications: A Case Study for the OutSystems Platform

À Família, que me ensina a Ser.
Aos Amigos, que me ensinam a Viver.

Acknowledgements

This dissertation was only made possible by the people and entities supporting its author. Albeit being the one writing, the strength to do so was not entirely mine.

The first line of appreciation is directed towards the *Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa*. Here did I endured all the challenges that built up the academic experience, leading to great personal and intelectual enrichment, essential for thriving in the professional world.

Secondly, I thank *OutSystems*, not only for providing a scholarship for this dissertation, but mostly for providing the purpose and meaning for this dissertation work. Working towards the improvement of a product so vast as *OutSystems* gave a nurturing feeling throughout the project. The provided conditions and tools are also to be appreciated, as they enabled the work process.

This thesis was developed in a thriad. As one of its members, I would like to immensely thank the other two: *Professor João Lourenço, PhD* and *Lead Software Engineer Hélder Gregório*. Without them this work would have never been possible. A special thank you to *Professor João* for teaching me when to keep thinks simple and clear (I apologize for when such teachings have not been remembered...).

As mentioned at the start, this work was not only made possible by the people actually working on it, by rather by the support behind it. Mostly emotional and affective.

The greatest thank you in the page goes to Pedro Rodrigues, the person who was always present. Pedro, I commend you: for not only have you been the most brilliant partner I could ever wish, but also have put up with all the days that I overslept and my overall tardiness. Not only did we had so much fun throughout the years, but also yelled at each other in such an efficient way that I cannot remember why were we yelling: I can just remember that we did and that we always laughed at the end and moved on. The sleepless nights, the nervous laughter, the funny conclusions: all memories that I will treasure for life. We are the most iconic duo that I can be proud to be a part of, enduring all that the academic and personal life as challenged us with. We are a team, we are *P&PCI*. And I won't ever let that go. Pedro, you are *impecável*. Thank you so much.

To all my friends at *Sociedade Filarmónica Humanitária*, a thank you for the fun that allowed me to relax, and the work that allow me renew the content in my mind. *Humanitária* has been more of a first home to me than the actual house where I live in. Being a part of such a huge cultural and social endeavour is an indescribable joyful feeling.

To my two partners at *ImpecAudio*, a thank you for enabling me to grow in other ventures besides my professional career, such as the show business. That is so dear to me. Thank you for your comradery, and for the fun we all had creating pieces of art to present to our costumers.

To my girls at *#CdF*, thank you for always been there since the beginning of times. I love you all.

To all my travelling companions to *Lavre*, thank you for the fun and the magical moments we all had. I hope that next year we can all get together in our natural habitat.

Finally, and most importantly, a huge thank you to my family. To my dear parents and grandparents, I am not able to fully express my feelings of gratitude for the teachings, the nurturing, and the care. You always made me feel loved and supported. You are my cornerstones.

"Happiness can be found, even in the darkest of times,
if one only remembers to turn on the light."
- J. K. Rowling

Abstract

Low-Code Development platforms enable users to rapidly develop applications relying on a form of abstraction of the actual code run by the system. The developer interacts mostly with a visual programming language, needing to write few or no lines of code at all. The abstraction and the degree of automation working underlyingly diminish greatly the time required to implement a fully functional application.

However, the same abstraction that accelerates development also contributes to the mapping gap between running and written code. The problem arises as the information exposed by the current running systems complies with the generated running code (e.g.: through verbose logs), rather than the low-code abstraction employed by the developer.

This work's case study is of the *OutSystems* platform, on which it addresses the improvement of the relation between a runtime problem and the OutSystems code that led to it. Particularly, the problem will be faced in the context of most OutSystems running applications, i.e., of a multi-machine setting. The proposed solution modifies the *OutSystems* compiler, so that the application publishing process automatically generates *OpenTelemetry* instrumentation code, that exports tracing information on the same abstraction level as the one developed. Said information is currently presented using external tools, such as *Jaeger*.

As a whole, the approach shall provide relevant information about the system's runtime state, facilitating the task of finding a possible root-cause of an encountered problem.

The proposed solution was tested against its impact regarding performance overhead, namely on the server and client machines, as well as on network activity. The results pointed out some overhead, particularly on the client-side's CPU consumption and number of KBytes sent. Real users were also tested, to analyse the overall usability of the solution (and its collected and presented information), leading to a success rate of, approximately, 90% on user interpretation of the information and usability scoring.

Keywords: Low-Code Development, OutSystems, Distributed Systems, Observability, Monitoring, Tracing, Performance Analytics, Application Performance Management, OpenTelemetry, Jaeger

Resumo

As plataformas de desenvolvimento *low-code* permitem que os utilizadores desenvolvam aplicações rapidamente, baseando-se numa forma de abstração do código que efetivamente é executado pelo sistema. O programador interage maioritariamente com uma linguagem de programação visual, necessitando apenas de escrever poucas ou nenhumas linhas de código. A abstração e o grau de automação subjacentes diminuem sobejamente o tempo necessário para a implementação de uma aplicação completamente funcional.

Contudo, a mesma abstração que acelera o desenvolvimento também contribui para a separação entre o código executado e o código escrito. O problema surge uma vez que a informação exposta pelo sistema em execução diz respeito ao código gerado que estiver em execução (e.g.: através de *logs* verbosos), ao invés da abstração *low-code* empregue pelo programador.

O caso de estudo deste projeto é o da plataforma *OutSystems*, no qual é abordada a melhoria da relação entre um problema encontrado em tempo de execução e o código OutSystems que o originou. Particularmente, o problema será enfrentado no contexto no qual a maior parte das aplicações OutSystems se situam, i.e., de uma configuração de múltiplas máquinas. A solução proposta altera o compilador *OutSystems*, para que, no processe de publicação de uma nova aplicação, este gere código de instrumentação *OpenTelemetry*, que exportará informação de *tracing* ao mesmo nível de abstração que o usado no processo de desenvolvimento. Esta informação é atualmente apresentada em ferramentas externas, como o *Jaeger*.

Como um todo, esta abordagem deverá providenciar informação relevante sobre o estado do sistema em tempo de execução, facilitando a identificação de possíveis causas de um problema encontrado.

A solução proposta foi testada contra o seu impacto ao nível de *performance*, nomeadamente nas máquinas de servidor e cliente, bem como quanto à atividade de rede. Os resultados apontam para a presença de impacto, particularmente no consumod e CPU e número de KBytes enviados pelo cliente. Foram também inquiridos utilizadores reais, para analisar a usabilidade geral da solução, apontando para uma taxa de sucesso de, sensivelmente, 90%.

Palavras-chave: Desenvolvimento *Low-Code*, OutSystems, Sistemas Distribuídos, Observabilidade, Monitorização, *Tracing*, Análise de *Performance*, Gestão de *Performance* de Aplicações, *OpenTelemetry*, *Jaeger*

Contents

List of Figures

Listings

Acronyms

aPaaS Application Platform as a Service i, 13, 15
API Application Programming Interface i, 30, 31, 32
APM Application Performance Management i, xvii, 12, 13
AST Abstract Syntax Tree i

DAG Directed Acyclic Graph i, xvii, 10

GUI Graphical User Interface i

IDE Integrated Development Environment i, 14
IT Information Technology i, 12

JSON JavaScript Object Notation i, 9

RCA Root Cause Analysis i, 30
REST Representational State Transfer i, 15, 31
RPC Remote Procedure Call i

SOAP Simple Object Access Protocol i, 15, 31

UI User Interface i, 15, 16
UID Unique Identifier i, 9

XML Extensible Markup Language i, 14

Chapter 1

Introduction

This chapter serves to introduce the overall content of this document. It begins with the presentation of the Context of said document and the definition of the problem that this thesis approaches. After that, an overview of the global document's structure is presented.

1.1 Context

The work reported in this dissertation was developed in the context of a collaboration between *Departamento de Informática* (DI) of the *Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa* (FCT NOVA) and *OutSystems*. Therefore, besides its academic relevance, this work also aims at having an industrial impact, serving as valuable know-how to be incorporated into the *OutSystems* product line.

Furthermore, the title itself could be enlightening. Breaking down *"Runtime Tracing of Low-Code Applications"* one could easily mark "Runtime Tracing", as it is the analysis of the behavior of an application while it is running. This "running" state should be comprehended from an industrial standpoint, where the most sensible form of a running application is of one which is on a production environment (i.e., running and available to end-users). Therefore, arises the problem of analyzing an application that can not be submitted to a process of thorough examination, through methods like *debugging* which are not possible in production environments. After that, the way of applying such techniques on a *"Low-Code Application"* must also be considered as it focuses the scope of the present work, in the sense that low-code offers additional abstraction over traditional languages, which makes mapping of the application's behavior to its original code more difficult. Not all applications shall be taken into consideration, as it is the specific case of low-code development that is being studied.

On the initial thesis enunciation, it is mentioned an arising problem with the great

ease of development that the OutSystems platform brings: an inexperienced developer may not possess the knowledge needed to maintain the applications that he creates. The document will refer to these "inexperienced developers" as "citizen developers", as one could be any citizen and not necessarily someone with informatics, computer science or application development background and might be, therefore, unqualified to understand fully the runtime state of the application (and the myriad of conditioning factors that might require such developer attention). The sole requirement to properly publish an *OutSystems* application is to be familiar with the *OutSystems* language and not the technologies that support it. There is, certainly, a gap between the provided ease of development and the provided ease of monitoring, with the OutSystems technology. As such, the relation of a problem in runtime to the low-code abstraction that the developer wrote might not be immediate and, when it is not, it might be extremely difficult, to a developer unknowing of the underlying behavior of his system, to resolve. This dissertation will focus the problems that could be, indeed, resolved by the developer: ergo, problems that stem from the application logic and not any other aspects of the infrastructure or configurations that could (and sometimes, really are) be done automatically or by a different person.

1.2 Motivation

As the present work is being developed in an industrial environment, it is easily understood that, besides all of the aforementioned academic value, it is needed to take into careful consideration all of the project's stakeholder's needs and suggestions, being them in the possession of knowledge of real-world problems related to the subject. Those may establish baselines on goals but may also define and limit the scope.

Stakeholders share a common goal of providing OutSystems' customers tools to be able to maintain and evaluate the performance of their applications in an independent way, not needing an OutSystems intervention in case of a problem. That way, the main issue is - as the title of this project states - the traceability of an encountered problem. This can be extended to observability over runtime, as this concept encompasses the former, as it will be presented in the next chapter, Background. However, the interest could be twofold. It could be directed towards an approach in which an OutSystems client could be able to monitor his applications at any given time, enabling performance and correctness evaluation and diagnostics. It could also be directed - probably in a more ambitious endeavor - towards a proactive approach, in which an OutSystems application should be able to monitor itself, determining whether or not a bottleneck was encountered and/or if a crash is imminent (and why), alerting the developer to the need of human intervention.

The first easily relates to problems found in the literature, approaching concepts related to systems supervision and diagnostics, such as *Observability* and *Monitoring*, both further discussed on the following chapter 2. As for the second, it encompasses the insights obtained by observing a system's state and behavior at a given moment but

comprehends a self-diagnostic feature that is, indeed, useful to the end-user but might be impractical to accomplish in the time and effort scope of this project.

1.3 Problem Definition

The dimension of the problem of lack of Observability, discussed in the previous section, deepens in severity when an error occurs in a setup of shared execution, i.e. when the full execution trace of an application is shared between multiple machines: a *Client* and a (set of) *Server*(s), including scenarios such as the use of microservices or multiple intervenients in the communication. This kind of environment partially relates to the one of a distributed system, in the sense that the execution is not centralized by a single process running on a single machine. As this is the reality of most current applications, especially the ones employed by OutSystems customers, it is now beginning to appear a problem waiting to be solved that fits the scope of this project. In this case, in the context of OutSystems applications and its ecosystem, the lack of linkage between a problem and the code from which it arose is greater, as the information retained from a Client's machine is minimal (compared to that of a Server) and the instrumentation currently available to obtain useful monitoring data (i.e., data that enables the comprehension of the execution steps, allowing for a detailed diagnostic) on the client-side is also scarce.

That being said, a problem to solve can be identified: the need to trace the origin of a runtime problem to the OutSystems code that led to it. This work addresses a generalization of the above problem, namely at answering the question

> *How to trace the origin of a runtime problem in a low-code application to its original source code, when the application is executing in a multi-machine (clients/servers) setting?*

Those problems could be organized into two categories:

- *Loss of Availability* occurs when a system stops serving its clients. This could be perceived as more of a serious problem than the one of a system which is taking longer than expected to respond to its clients. It could be related to a myriad of causes that go far beyond the scope of the OutSystems platform, namely problems intrinsic to an Amazon Web Services machines where OutSystems Cloud is hosted or to additional network appliances.
- *Performance Problems* occurs when the system's performance is below expected. This could translate in an increase in response time, for example, due to a specific code path that only runs with a specific input or a simple loop that is quite fast but runs millions of times. These deficiencies are not easy to track in the source code supporting the OutSystems generated code. Mapping deficiencies to the OutSystems language is not always trivial and, currently, what can be tracked is dependent on the manual instrumentation of the code by the developer.

However, due to sparse origin set of problem of the *Loss of Availability* kind, tracing the problem to the OutSystems code that led to it (i.e., the ultimate goal of the work) would be infeasible in the timespan of the project, as it can originate in a component that is not affected by the user's code.

There are also aspects of the resolution of said problems that shall not be approached, namely automation and proactivity on solving detected issues and data visualization. Regarding the former, the work shall focus on the detection of issues and the relation between its effect and the written OutSystems code that led to it, rather than solving them, as that would also be impractical in the timespan of this project. As for the latter, the presentation of the collected data would be time-consuming and would not worth its while, as there are many external tools - refer to chapter Related Work to see proper examples - that can perform this when fed with the correct data: the project's work will focus on the collection of said insightful data (which is currently unreachable on the platform) and then use such external tools to present them to the users. This will not only relieve the project's planning, enabling the sum of all efforts into the means of collecting the information, but also ensure that the presentation will be made by a tool that is already prepared to do so in an aesthetically pondered, user-friendlier and overall better manner than any implemented solution on this project could do (not forgetting that it is also professionally refined and approved by the community).

The absence of additional effort to developer is of utmost important. This means that instrumentation should not be required from the developer, as if it were, it may easily be discarded or even forgotten. A situation of such kind might endanger the overall observability approach to the system, as it would introduce non-instrumented components, without reachable runtime information.

1.4 Key Contributions

The work present in this dissertation provided the following key contributions:

- Extensive research on the *Observability* field, leading to a detailed written description of the state-of-the-art, in all of its tools, methods and techniques. This research focused on achieving a solution to the aforemention problem, but detailed the findings on a set of diverse approaches that the field has to offer, nonetheless.

- A proof of concept solution, materialized on an improvement to the *OutSystems* compiler, so that it generates instrumentation code on the applications it publishes. This information was designed to be collected and presented on the same level of abstraction of the remaining *OutSystems* ecosystem (i.e., low-code) rather than the generated high-code.

- Evaluation of said proof of concept, regarding user satisfaction. This lead to the

conclusions that this problem is felt by real users and a solution like the one proposed in this dissertation stands as a major improvement on the area. Satisfaction surveys and user feedback confirmed that the solution is a valuable asset towards the enablement of low-code runtime tracing.

- Evaluation of said proof of concept, regarding performance overhead impact. This appointed the known implementation problems, that should be addressed in further approaches.

1.5 Document Structure

Following the present chapter, this document comprehends another six: Background, Related Work, Approach, Implementation Details, Validation and Conclusion.

The next chapter, 2 - Background, shall focus the Key Concepts and theoretical bases in which this document relies to a full comprehension. As the scope of this study is not a complete review on the literature regarding the aforementioned subject, this chapter will focus only on key aspects, trying to present, as promptly as possible, its relation to the discussed matter, in order to maintain the reader aware of its relevance on any further mention. Ending the chapter, a brief overview of the OutSystems platform shall be conducted (as mentioned above, agnosticism is a goal of this project so the need for an extensive review is discarded).

Having a knowledge baseline defined on that chapter, the document continues to chapter 3 - Related Work. There, cases from an industrial and academic standpoint will be taken into account, described and analyzed in order to establish key points that can be useful in the following phase of this project. It is not guaranteed that the entirety of the problems covered in this chapter will be useful, as they might be partially out of the scope of the present report, and so, besides presenting the relevant problem, possible solutions, and approaches, it will be also presented and discussed the degree of relation with the present project and how it might be important in the understanding of the investigation as well as possible contributions to a possible solution. In a summarised way, this chapter serves to relate the aforementioned problem with similar problems found in other contexts, revealing important discoveries that those made and using their work as a reference for future investigation.

Chapter 4 - Approach described the solution put to practice, after reviewing all research done and designing what should be a proposal to solve the identified problem. The chapter contains not only remarks regarding design choices and characteristics of the technologies and tools put to practice, but also a wide set of figures to illustrate what is being discussed. These images reflect real scenarios where the proposed solution was employed, and translate to what an end-user would see.

On chapter 5 - Implementation Details, the specific details of the presented implementation are presented. This chapter complements the previous one, by pointing out

detailed characteristics of the solution and how to replicate them in the future.

Having shown the proposed solution, its validation process is depicted in chapter 6 - Validation. The solution was evaluated regarding two factors: user satisfaction and performance impact. Results for both are presented and discussed in this chapter.

Finally, chapter 7 - Conclusion holds the concluding remarks, summarizing the entire document and pointing out the most important aspect to keep in mind. It also highlights the contributions of this work, and ends with a list of suggested topics to address in future work.

Chapter 2

Background

In this chapter, specifically in its first section, several key concepts are presented in order to provide an overview of the assumed knowledge to understand the following chapters. Each subsection of that section should focus on a specific concept, defining it and explaining some of its most important features. The chapter ends with a summary of the OutSystems technology, that shall serve as a use case of this project, being an example of a low-code platform.

2.1 Key Concepts

Throughout this document, several concepts will be addressed enough times for them to be considered key in understanding the overall discussed subject and all logic and arguments made. These concepts will be presented and explained in this section.

2.1.1 Observability

The concept of **Observability** will be explored extensively throughout the rest of the present document as it represents a goal defined to achieve on the project, specifically, applying it to the context of a low-code application. Its precise definition is not explicitly documented as the scientific community does not yet agree on a full and clear (group of) sentence(s) that encompasses the entirety of its scope. One could start by analyzing the concrete meaning of the word in the dictionary, only to find that the exact term, as a noun, is not properly defined. The best approximations found, either on Lexico as well as on Merriam-Webster's, were to the adjective "Observable" that was defined as "Able to be noticed or perceived; discernible"[36] and "noteworthy; capable of being observed"[47], respectively.

The lack of clarification then extends to the literature that usually does not present a clear definition, but rather relates the term to an older, contextually bounded, one: *Monitoring*. However, it does not comply as a synonym, as will be presented later in this section. As shall be discussed, Monitoring is by itself a goal to achieve, having Observability at its foundation, as a requisite [17]. For this motive, albeit presenting the term - just to raise the attention to its existence and difference from the one being described here -, it will only be purely defined and explained in the next section.

Indeed, the concept of Observability becomes clearer when applying its formal definition (the one in the dictionaries) to the context of a distributed system. It is, at least, much more relatable to that than to the context of debugging a single process on a single machine, which falls short when applied to a shared execution [17] This relation is important because the available literature encompasses numerous works on the subject applied to the specific context of distributed systems, whereas it is scarce when it comes to situations exactly similar to the one approached: not of a distributed service, but a service that is at least running outside of the client's machine.

Focusing on the term itself, *Observability* is a property that any given system could achieve if its design and development acknowledged the facts that a real system is bound to fail and that the universe of possible failures is infinite, which makes the overall system unpredictable at every given moment [17]. Ergo, instead of predicting all error cases, and without the existence of a distributed debugger (especially in a production environment, where an application's execution can not be halted at the developer's demand), extracting runtime information is key to evaluate system performance. To extract runtime information could be perceived as "to observe" the current system state, not from a static viewpoint but rather throughout a given period. This is why the definition of Observability is not usually found as a clear and concrete sentence, as it is formed upon the knowledge of a set of features about a system. It could be summed up in a sentence such as "Observability is a property of that a given system has if it allows its state to be perceived by an external observer".

As a result of the complexity of this concept, it is only natural that it fundamentally relies on other key concepts. In the specific case of "Observability", three concepts can be appointed and some authors refer to those as "The Three Pillars of Observability" [17], to distinguish them for their relevance. Those are Event Logs, Metrics and Tracing and will be subsequently addressed.

2.1.1.1 Event Logs

Event Logs are immutable records, that are exposed when a certain instrumentation point is reached in the code.

The type of information they held is defined by the type of log emmitted [17]. *Plaintext* and *Structured* logs are both text-based, although the first might hold any string with any kind of information that can be reproduced textually, while the second is typically a

JavaScript Object Notation (JSON) object. *Binary* logs are the most diverse, as they can hold any type of data, after being converted to a binary representation. Logs are normally timestamped.

Although they are easy to generate and expose, the performance overhead related to logging might be significant, becoming problematic to the system.

2.1.1.2 Metrics

Metrics are a system-centric abstraction built on top of logs.

Over time, data collected about the systems, normally on the form of logs, can be represented in numerous ways. Metrics are a form of said representation, in a numeric form, enabling easier analytic and querying features. They are also timestamped and carry a value representing the data they expose. This value could be any instrumented insightful information, such as the time spent on a request, the number of times a certain request has been served by a given service, or the amount of memory and/or CPU consumed over a certain amount of time. A metric is identified by its name and a set of labels that also ease querying.

Due to their numeric and simpler nature, the overhead related with metrics is much less than the one related with event logs.

2.1.1.3 Tracing

Traces can be the most insightful observability feature that a system can expose.

They are also an abstraction built on top of logs, acting as a representation of them. Unlike metrics, they are request-centric. This means that the timing related to a trace is equal to the time related to the request it exposes.

Traces are directed acyclic graphs, being each node a *span*, connected to other nodes by edges called *references*. A span is a representation of computation time spent on a certain component, needed to serve the request. The direction of the graph's edges represents the flow of execution calls, maintaining the order and causality of the spans. As such, it is used a *happens-before semantics*. This can be visualized in figure 2.1.

The trace is constructed via the exposing of information at instrumentation points related to a certain Unique Identifier (UID). This UID is created at the first instrumented span of the trace and passed to the next component or layer of the system, following the request call. At each stage, before passing the UID to the next, information and metadata might be added, to enrich the trace.

By exposing not only the components involved in the processing of a request but also the time spent on said processing and other enriched data, the benefits of tracing are clear: the ease of troubleshooting the system, and also the pinpointing of an encountered runtime situation to its root cause.

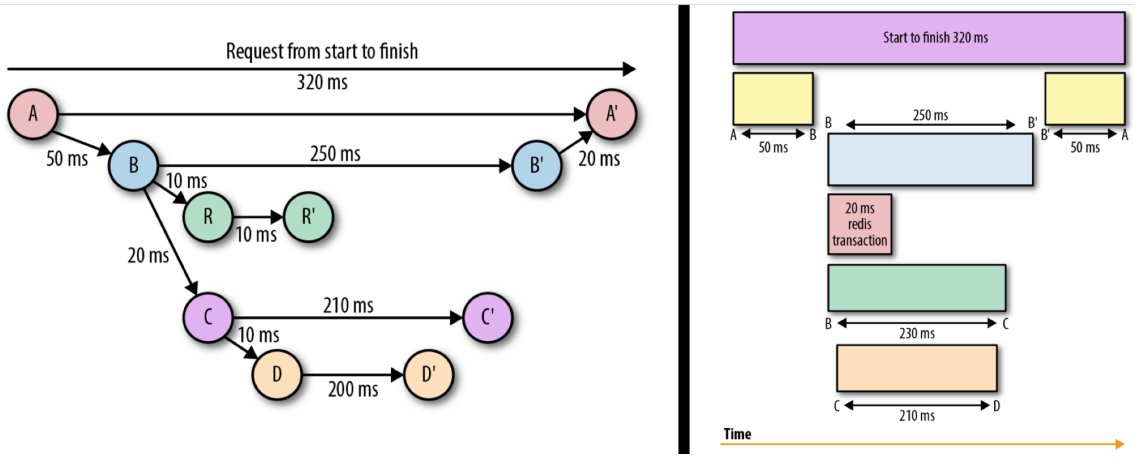


Figure 2.1 – A trace example represented as a DAG (left) and as a set of spans (right) [17].

However, the difficulty of tracing lies in the hard retrofitting process needed for it to work properly on already implemented systems. To achieve a complete and proper trace, every component on the request path must be instrumented.

Being the context of this work the one of a multi-machine setting, relatable to a distributed system, the approached tracing concept will be of *Distributed Tracing*. This comprehends the trace of a request for its entire execution, including all the components and/or machines in which it is processed, throughout the distributed system.

Having the concept of Observability defined, in all of its features, it is also important to refer its Goal as well as its Value, as those two characteristics help to determine the scope and relevance of the concept. Regarding its Goal, observability could be approached as "not about logs, metrics, or traces, but about being data-driven during debugging and using the feedback to iterate on and improve the product"[17]. This means that the concept, although relying on the features presented above, it is not a mere sum of them - in the sense that applying them all does not guarantee that a system is fully observable. Rather, it consists of a paradigm that should be approached, towards the iterative development process, acquiring always important data by observing the system. On the same book, it is also mentioned that "The value of the observability of a system primarily stems from the business and organizational value derived from it"[17], meaning that the source of its value is the set of accomplishments that it might achieve, and so its value *per se* is not significant.

Continuing that reasoning, it is easily understandable that it is of utmost importance to perceive as much information as possible on the most valuable environment that a system can be subject to, i.e., the production environment: it is only there that the system meets its most rigorous demands while performing directly to its most demanding and harsh evaluators, the end-users. However, typical debugging techniques do not comprehend this endeavor, as their practices of stalling the system on-demand to reach for its

data can not be put to practice when a system is facing the production environment (as they would be perceived in lacks of availability to end-users, even for brief moments). Furthermore, even performing a debugging session on a system while it is running on the production environment could not prove itself useful, as some problems are time-dependent and exacerbate their effect as time goes by, leading to degradation of service. Facing these challenges, an intuitive and widely practiced solution was to test the system in a previous stage to the production environment, attempting to mimic the known conditions and challenges that the system would face on production, reasoning that conducting a test in a scenario that resembles the real world should give a certain amount of insight about the performance of the system in real occurrences.

The aforementioned book [17] raises a problem, with a caricature: "Verifying in environments kept as identical to production as possible is akin to a dress rehearsal; while there are some benefits to this, it's not quite the same as performing in front of a full house". To overcome this, there must be a paradigm shift in the system's design and code stages, where the previous default was to code (and test) for success, whereas it now must be to code (and test) for failure. This new approach should instill on the development team the principle defined by Edsger Dijkstra "Program testing can be used very effectively to show the presence of bugs, but never to show their absence", meaning that it shall be assumed that it is impossible to guarantee via testing that a system is fail-safe. Combining the inexistence of a guarantee of correctness (and/or performance) and the impossibility to predict each of the possible error causes, constant observable feedback of the system's current state is needed, to dynamically assess its overall condition.

2.1.2 Monitoring

The concept of **Monitoring** is not dissociated from the previously presented *Observability*. In fact, "Observability is a superset of monitoring" [17]. Monitor focuses simply on the reporting of system health and on alerting when a problem occurs. Observability encompasses both Monitoring's failure prediction through system health analysis and the "best-effort verification of correctness" achievable through testing, in all their possible permutations. This is why Observability exposes fine details about the inner workings and possible failure modes of the system.

There are two main variants of Monitoring [11, 3]:

- **Blackbox Monitoring** is the most realistic approach, as it comprehends the system's observations from an outside perspective. It makes sole use of the information the system exposes by nature, without the modification of its code nor accessing directly its internal resources. In the vast majority of cases, this is the only approach available, since most monitoring is done to machines not available to directly observe (e.g.: external services or servers on the cloud) nor is the code available to reinstrument or even reimplement.

- On the other hand, **Whitebox Monitoring** has a greater chance of positive results and achievements, but it is also far more difficult to achieve itself. It translates to the exposure of monitoring information from inside the system, made available by instrumentation. It may be feasible when the team implementing the monitoring solution is allowed to reinstrument or reimplement the application code when said code is not already prepared to expose monitoring information. As some systems are already designed to encompass this type of feature, the results extrapolated are far more meaningful than the ones achieved only by *Blackbox Monitoring*. However, as mentioned, this is not always possible.

There are multiple ways to approach Monitoring on a system. As an already well-known topic to the Information Technology (IT) community, there are several discussed patterns [11]. One of them is *Composable Monitoring* that states that a rich monitoring infrastructure is composed of multiple, loosely coupled, tools. These tools range from activities such as *Data Collection* to *Visualization* and *Analytics and Reporting*. It is based on this pattern that the next chapter Related Work is organized and documented, presented tools and techniques that address such activities. Another important pattern is *Monitor from the User Perspective*, which translates to the need of placing instrumentation and monitoring tools to, as much as possible, the entirety of the system. Monitoring that addresses only the deep layers of an architecture is impractical to pinpoint a detected problem to the user activity that lead to it, difficulting the task of identifying the cause of said problem. An also clear pattern is *Continual Improvement* that restates the need for continuous development of monitoring techniques applied to the systems at hand, in order to maintain a healthy monitoring infrastructure, suitable for the task and context in which it is applied.

2.1.3 Application Performance Management

Application Performance Management (**APM**) is an IT discipline that encloses tools and activities related with achieving "an adequate level of [application] performance during operations" [9]. Its focus is twofold: not only to monitor the system's overall state but also to use the gathered data to detect and aid the resolution of occurrences of performance-related problems.

This concept is greatly inspired by the aforementioned *Composable Monitoring* pattern, presented on the previous subsection 2.1.2. To achieve its goal, the concept relies on four activities:

- **Data Collection** - The activity were the information about the system state and performance metrics are gathered. The goal is to collect as much data as possible, from as many layers and components as possible, without causing a significant overhead. The reason to seek to gather sparse information is that it contributes to the observability of a wider portion of the system.

- **Data Storage and Processing** - After collecting the data, one must process it in order to enable reasoning on its findings. To do so, data needs to be stored, so that it is processed *a posteriori*. Processing might approach a data representation technique, such as the already mentioned execution traces or a time-series. Traces are more detailed, in the sense that they represent the internal application flow that forms a response to a request. On the other hand, time-series results, although also being presented in relation to time, are simply summarized numeric statistic data.
- **Data Presentation** - It may be approached in different ways, depending on the tool and/or on the context, as it information may be presented with different levels of abstraction and refer to different scopes (e.g.: business processes information might be useful on some applications, while on other cases the intel might only need to come from technological data).
- **Data Interpretation and Use** - This activity might either be performed manually or automatically. Goals range from detecting the presence of a problem to its diagnosis and root cause analysis. Eventually, it may lead to changes in the system's code or architecture, if the concluded solution to the encountered problem or state depends on so.

These four activities are related in a way that forms a cycle, in the sense that one activity's results serve to feed the next activity, as input. Such cyclic relation is depicted on figure 2.2.

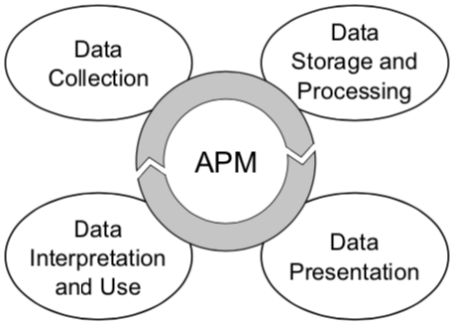


Figure 2.2 – APM Activity Cycle [9].

2.2 OutSystems

The *OutSystems* platform allows for rapid application development, through a low-code approach. It functions as an Application Platform as a Service (aPaaS), a specific type of cloud computing, that provides users the means to develop and run an application without the concern of building or maintaining the underlying infrastructure [21]. That is already directed towards the OutSystems goal: to relieve developers of the burden of hand-coding every single part of their application, removing the need to waste time on

implementing well-known coding patterns by automating those tasks [72]. Applications are built visually, dragging and dropping available widgets and using already built tools.

There multiple applications and tools available to the development and management of an OutSystems application. Their overall architecture can be summarized and depicted in figure 2.3. The three main components are:

- The **Service Studio**, the Integrated Development Environment (IDE) on which the developer builds the application, employing low-code resources provided by this application. As previously mentioned, constructing an interaction flow is achieved by dragging and dropping pre-built and easily available visual widgets. A flow is constructed based on the widgets that compose it, and the interactions between them. This visual representation of the application developed is stored in a Extensible Markup Language (XML) file, that is shipped to the following component, as the application is published.

- The **Platform Server** is the core component of the platform, receiving the low-code abstraction description, translating it and generating optimized native .NET code. Is it also there that deployment and application services take action. This component is aided by the analytic capabilities of the **Service Center** and **LifeTime** tools, that allow for insights on the application system details, such as performance metrics, system availability, errors and user management. The **Integration Studio** tool is responsible for managing *Extensions*: OutSystems components made to enable integration with external services, or code explicitly implemented by the developers. These extensions, after compilation, are shipped to Platform Server, for it to use on the application bein run.

- The **Application Server** is the actual server, running the generated code, interacting with client-side applications and responding to its requests. Its execution is based on the logic defined by the code, comprehending interactions with external services and databases.

The first mentioned component, the Service Studio [61], is the one closer to the developer and, therefore, its layer of abstraction is the one to be achieved by any means of information delivery to the developer. It is composed of several areas, as depicted on figure 2.4. On the **Toolbox**, the available widgets are displayed, ready to be dragged to the **Main Editor** area, where the developer drops the widget, relating it to the others already places, constructing the application flow. On the **Application Layer Tabs**, the components that mark the structure of the application are presented, displayed on four tabs, according to their respective application layer:

- In the **Processes** tab, *Business Process Technology* is addressed, i.e., it is defined the handlers for specific business-related tasks, such as "handling invoices, processing orders, or handling complaints" [65].

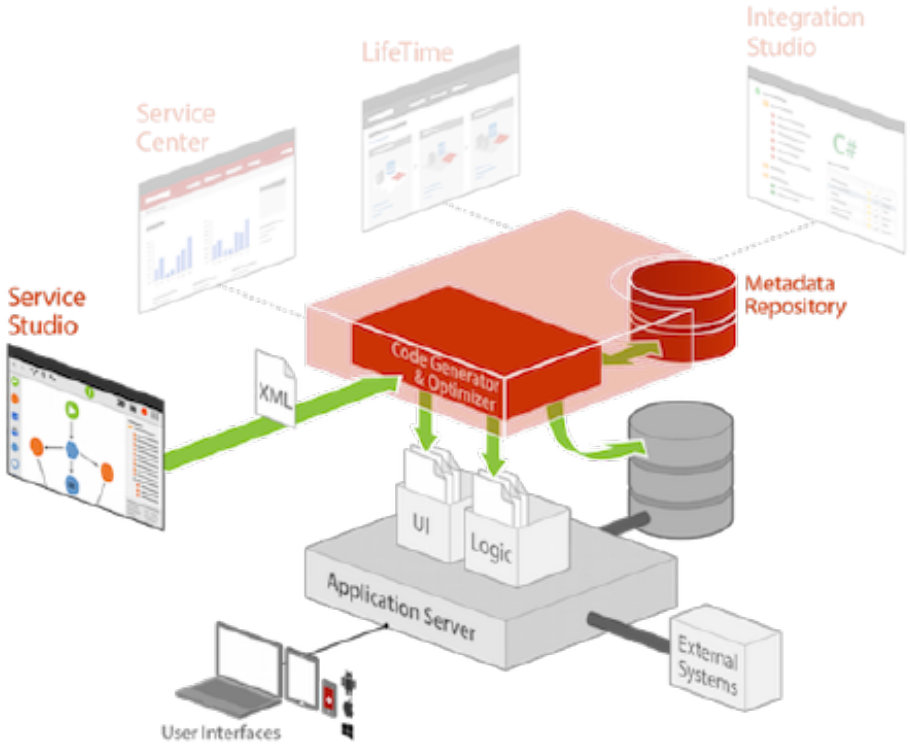


Figure 2.3 – OutSystems Architecture [59].

- The **Interface** tab, the User Interface (UI) of the application is implemented. Besides designing, interaction flows between *Screens* are also defined.
- The **Logic** tab is where the application and business logic is defined, mainly by implementing *Actions*: logic flows that can be executed when called by other components or triggered by user interaction. On figure 2.4, an example of a *Server Action* is being implemented, on the *Main Editor* area. It is also in this tab that *Representational State Transfer (REST)* and *Simple Object Access Protocol (SOAP)* integrations are defined, as well as *user roles*.
- Tab **Data** is where data model definition occurs, specifying *Entities* and *Data Structures* that will be present on the database, and other data-helding features, such as *Session Variables* and *Site Properties*.

The **Properties Editor** provides property costumization for the selected widget or component. The **Development Tabs** display details related with the development of the application, such as errors and warnings, a debugger, publish results and a search field. The **Toolbar** provides basic and common interaction operations, such as navigation between screens, *Undo*, among others. As an aPaaS, it is on the **Status Bar** that cloud

development environment and basic user information is displayed. Finally, the **1-Click Publish Button** serves to start an application deployment.

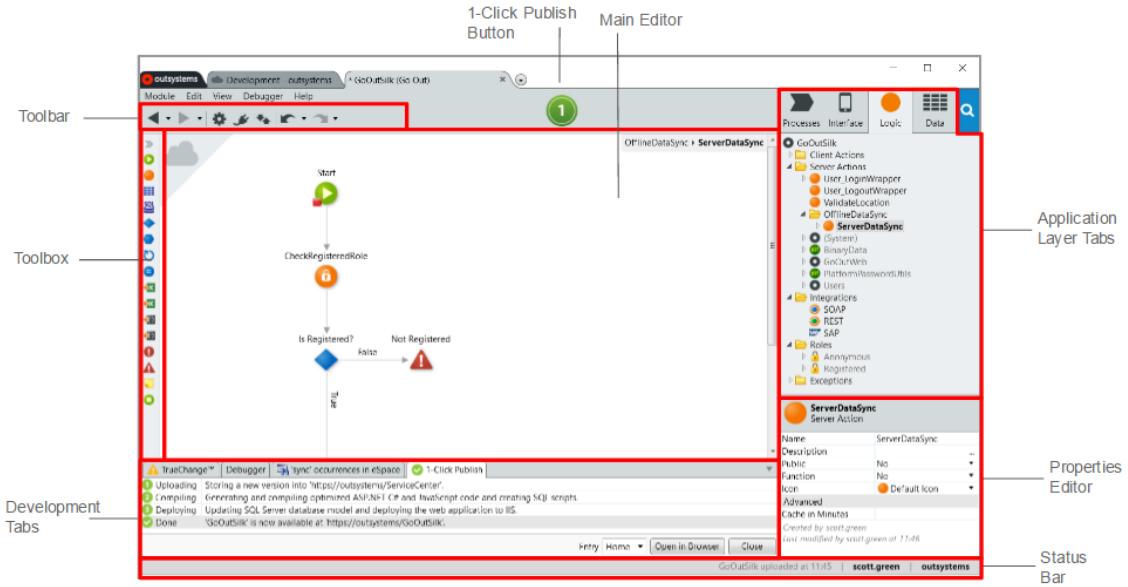


Figure 2.4 – Service Studio Overview [61].

There are several types of applications that can be built on OutSystems [55]:

- **Traditional Web Apps** are based on server-side development, where logic and page rendering are done on the server and then distributed to the client that requested it. They are still employed, but they are the earliest of the referred types: the newer kinds propose better approaches to application development, as they introduce reactive paradigms.

- **Reactive Web Apps** approach the reactive paradigm, introducing client-side logic, asynchronous data fetching and reactive client-side rendering of the application. Besides commuting to the newest development paradigm, it also improves responsiveness and scalability. This way, it "consolidates the development experience across mobile and web" [77]. It also run on the client's browser.

- **Mobile Apps** are actually presented as **Tablet Apps** and **Phone Apps**: both work on either device, as the difference relies only on UI optimization. These types of applications behave in a similar way of the previous Reactive Web Apps, as they share the same runtime. They were actually released on the platform first. The main difference is that these are not supported on the browser, and rather they run natively on the user's device.

- **Services** provide a way to "centralize and isolate reusable logic and data" [55]. Employing a service-oriented architecture, the low-code developed is to be consumed by some other type of application.

The OutSystems tools and techniques already available related with this work, and the problem it faces, will be described on the next chapter, namely on the Low-Code Monitoring Approaches.

Chapter 3

Related Work

To propose a solution to the problem defined in Chapter 1, it is useful to address contexts similar to the one to be approached, as solutions may be already documented in a way that shows themselves useful to the case to be studied on this project. This context could be either in industrial or academic settings. For obvious reasons, some solutions may appear as too specific to the underlying problem in each case and, as such, this chapter shall perform an overview on multiple solutions and tools that are already implemented, used and that have proved themselves worthy of recognition and consideration, as references to the solution to be planned, designed and implemented on the following stages of this project.

An important caveat is that Chapter 1 referred that the goal is to trace a problem occurring in runtime back to the low-code abstraction that led to it and, to achieve that, one must have observability over the system. However, as explained on chapter 2, that is a complex concept. Some tools and solutions might not approach the entirety of this complexity, and might even focus exclusively on one particular topic, however that could contribute to a partial solution that might help to overcome the problem as a whole.

3.1 Instrumentation and Data Collection

The main focus of the work to be developed and implemented shall be about data collection. To observe a system is to become aware of its state and, as such, to become knowing of certain important data, described in detail in the previous chapter 2 (namely on the Observability and Monitoring subsections). To become aware of such data, one must first collect it. In this section, tools that enable such data collection will be approached, as they already enable or at least facilitate system observability in many settings. Some presented tools might not approach said data collection exclusively, but that is their main focus and

the most well-achieved feature.

The first intuitive approach to provide observability over a system is to implement it in a way such that it reports its state on key occasions. The difficulty of this approach is to achieve a method capable of predicting, in a reliable manner, which occasions (i.e., source-code locations) to instrument, i.e., to implement monitoring report code. A proposed initial solution to this problem was **Pythia** [2], by Ates *et al*. It was designed as an automated instrumentation framework, targetting distributed applications in their multiple stack layers. The focus was to achieve its goal by comparing traces of executions, matching executions that are known to be expected to perform similarly. In case of mismatch, instrumentation was considered required to analyze thoroughly this possible performance deficiency. To do so, Pythia requires as input initial expectations of similarity in request execution's performance and *workflow skeletons*, i.e., a set of instrumentation points that are always enabled throughout the execution and serve as an initial proposed solution. Its complete operation cycle is depicted on figure 3.1.

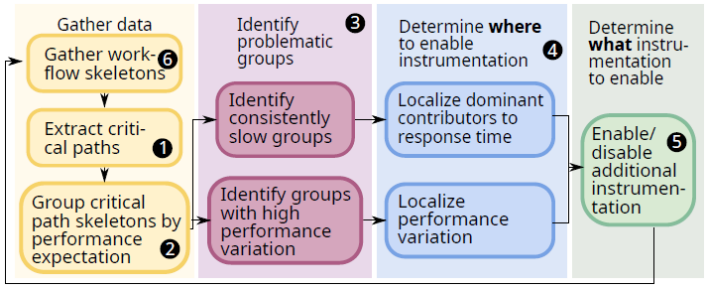


Figure 3.1 – Pythia's continous cycle of operation [2]

Pythia's design could be separated into two planes: Control and Instrumentation. The instrumentation plane is the one closer to the running, implemented by the developer, code of the system. From said code, as requests pass through components instrumented to expose monitoring information, traces are collected and redirected to Pythia's components on the Control Pane. There, critical paths from traces are extracted and grouped, so that problem localization can take place. Localized problems are then analyzed, also regarding information from the Search Space, a collection of known traces and hierarchically-described ("caller/callee" [2]) examples of interactions between components. The result of this analysis is a set (that can be, naturally, empty) containing instrumentation points needed on the system's code. As this set can be large enough to lead to a significant overhead after its implementation, a prioritization component is responsible for filtering the most important instrumentation points, to a certain predetermined limit of information bandwidth. This decision is based on metrics recovered from the system, and can be adjusted to prioritize groups of problems such as "groups with extremely slow performance" [2] first. This design is represented schematically in figure 3.2.

This approach presents two limitations: first, it does not collect the tracing data

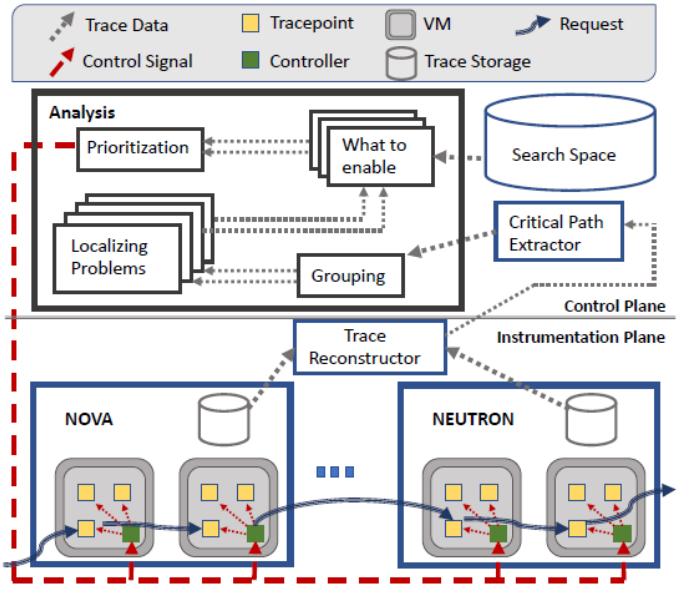


Figure 3.2 – Pythia's Design [2].

needed to refine its results; second, it requires a predetermined instrumentation format, so that its automatic instrumentation works properly and complies with other already defined instrumentation. To approach the first, there are already numerous tools that could serve the purpose, that shall be described followingly. To approach the second, there is a well-known and well-adopted API specification called *OpenTelemetry* [51], that resulted from the merge of two previous projects *OpenTracing* [54] and *OpenCensus* [48, 89, 76].

Historically, in 2007, Fonseca *et al* proposed **X-Trace** [8], a "distributed causal logging framework" [82]. Relying on logs collected on the system's multiple components and shipped to a centralized processor, it builds a graph denoting the path of an execution call throughout the various system components, appending each request with metadata tags, that are sent along with the original request data. Instrumentation requires numerous changes, not only to support the metadata placeholders on the data already exchanged, but also to implement operations related to the propagation of said data. X-Trace already used a primitive, that was only coined years later, called *Baggage*, also referred to as *Baggage Context* [42]. Its formal presentation was made on the paper that introduced the **Tracing Plane** instrumentation library [13]. This primitive focused on solving the problem of context-propagation, propagating captured metadata related to each request or operation, by passing it along its tracing information, as a tuple. The **Pivot Tracing** framework makes use of this primitive and approaches it with dynamic instrumentation, thus enabling certain instrumentation to be made at runtime. This is done by requiring a set of "tracepoints" introduced by the systems developer: those are not instrumentation points, in the sense that they do not represent points in the code

where instrumentation code is placed, but rather signaled code points where Pivot Tracing can place instrumentation code when needed to answer a specific query (hence the "dynamic" behavior). When a query is running, when running code reaches a tracepoint, it triggers the instrumentation placed by the framework generating the data and exporting it to a collector. Pivot Tracing was the first proposed system to combine said approach with causal tracing, as it introduces the "happened-before join query operator", enabling the querying of causally related data. With the insight given by carrying baggage on a system's request, this operator proves itself remarkably useful.

Dapper [16], a distributed systems tracing platform introduced by *Google* in 2010, was also one of the first approaches to distributed tracing. It had then documented results on its effectiveness, with minimum overhead and application-level code modifications, throughout Googles's plethora of running systems (most of which of great dimensions, complexity and demand). After that, other approaches picked up on that work, focusing on mitigating some of its limitations, some of which shared with Pivot Tracing. One of the aspects focused on other projects was the massiveness of the already available approaches, that led to the development of solutions such as **Panorama** [10]. An overview of this solution is depicted on figure 3.3.

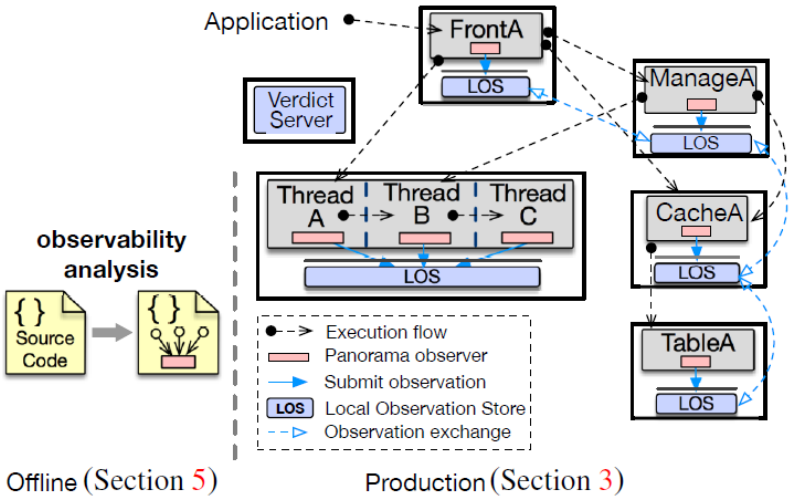


Figure 3.3 – Panorama's Overview [10].

Panorama's activity is twofold: "Offline" (the left side of the figure) and "Production" (the right side of the figure). In the "Offline" pane, static analysis is ran to inspect the code, injecting hooks on "critical points" (for example, near exception throws, RPC or external thread calls). These hooks consist in code, based on an instrumentation library, that exposes trace data, such as information about its caller and the context in which it arises, in order to provide observability over the execution situation in which it was reached. The set of these hooks in a process constitutes a logical "Panorama observer". Ergo, all processes' code deploys its own observer, responsible for exposing its specific observability information, rather than executing another thread to serve this purpose. In

each system component, it is deployed (this time, as a separate thread) a "Local Observation Store" ("LOS"), responsible for gathering information shipped from the observers (in the figure, this interaction is depicted with solid blue arrows). These stores also perform analytic processing to determine the existence of a possible encountered problem and are capable of interacting with each other directly (as it is depicted with blue dashed arrows on the figure). A "Verdict Server" serves as an interface component to the Panorama system by a result-fetching user, providing querying capability over the set of Stores.

Commercially, both **Zipkin** [90], initially presented by *Twitter* and **Jaeger** [41], initially presented by *Uber* and supported by the *Cloud Native Computing Foundation* [39], act as request-tracing collectors and analyzers. Companies whose product's goal is to provide and/or enhance observability over distributed systems, like *Logz.io* [45] or *Epsagon* [37] have already described and compared the architecture and functionality of such tools, respectively on [46] and [38]. Both solutions rely on instrumentation of the application and server code (of the system to be observed, i.e., outside of the Jaeger/Zipkin architecture) to release tracing data: that data is to be collected by a component that then sends it to another component responsible for storage. This approach was presented by Dapper, which these tools based themselves on. By gathering sufficient data, tracing information can then be retrieved by a component associated with Jaeger/Zipkin or by a third party tool. Moreover, picking up on the Pythia's example, these tools can then be responsible for collection the data that the system exposes, feeding Pythia with information that can be used to enrich its model and determining new tracing points, as a cycle. Conclusions of those comparisons both point out that, although their purpose is the same and its architecture and functionality, Jaeger might be more transversely useful, due to its built-in compatibility with all *OpenTracing* libraries, as well as more flexible as it is also with Zipkin's API, from which its own instrumentation is inspired.

Zipkin might be more practical to employ on infrastructures with no containerization, due to a significantly lower number of "moving pieces", i.e., it consists of a single process running all aforementioned components on a *Docker* image, rather than Jaeger's approach of a dedicated process for each component. Among others, it is used at *Salesforce* [44] to allow for distributed tracing in an infrastructure rich in microservices. As traces are useful to plot call graphs, identifying the components involved in processing a request, Zipkin enables the diagnosis of performance situations, namely by identifying bottlenecks. However, Zipkin does not possess any analyzing features, capable of summarizing the collected data and/or present results of it (e.g.: dashboards) to the user: these problems will be mitigated by the use of a different set of tools, like *Grafana*, *Datadog* or *Honeycomb*, described below.

Jaeger shares Zipkin's tracing benefits (from which it is inspired), as well as the lack of analyzing features, being also prepared to address context propagation [74], as it is fully compliant with the OpenTracing API (and, more recently, with the OpenTelemetry API) that provides the aforementioned *Baggage* mechanism. This compliance is due to the fact that the Jaeger's components running on the client-side (i.e., along with the application

code) are actually "language-specific implementations of the OpenTracing API" [81].

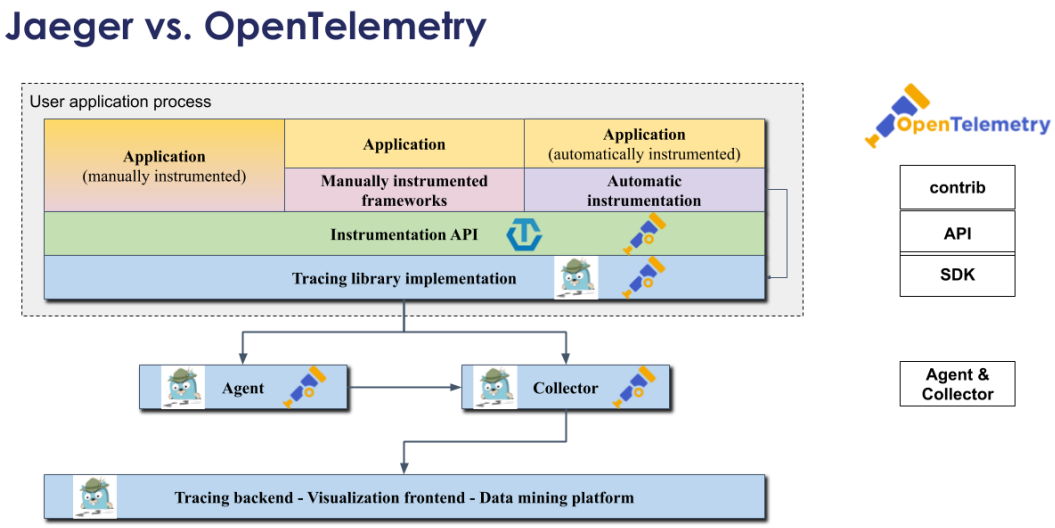


Figure 3.4 – Jaeger interaction and problem-space overlapping with OpenTelemetry [75].

In fact, there is currently a major overlap in problem-space between Jaeger and **OpenTelemetry**, as depicted on figure 3.4. To clarify interpretation, the layers are tagged with the icons of the tools that can be applied, being the blue animal the icon for Jaeger, the telescope the icon for OpenTelemetry and the blue hexagon the icon for OpenTelemetry's ancestor project, OpenTracing. The OpenTelemetry open-source framework focuses on easing the task of instrumenting the system's code, supplying multiple components, starting at the API and library that act directly on the "User application process", exposing internal information. Moreover, the "Agent" component is responsible for gathering exposed data and forwarding it to the "Collector" component, which also gathers information but whose prime function is to translate and ship said information to applications responsible for processing and providing visualization for it. As such, one can conclude that OpenTelemetry covers most of the layers already covered by Jaeger, with the addition of doing so with a vendor-agnostic API, capable of being employed on any system and enabling the integration of a variety of backend trace analysis components. An extensible approach would be to employ OpenTelemetry layers of instrumentation on the application process, using also OpenTelemetry's Agent and Collector components, and finally ship the gathered data to an external service, like Jaeger or any other preferred similar product.

3.2 Elastic Observability

Some solutions approach multiple APM activities, particularly *Data Storage and Processing* and *Data Presentation*. As referred on the Problem Definition section, those are the

problematic features that will not be addressed by this project. As such, the project's output could be fed into a solution of this kind, so that the user can use its output to put the last APM activity - *Data Interpretation and Use* - to practice.

To approach this kind of situation the company Elastic B.V. provides a set of solutions for various kinds of observability targets, which are easily put together to form a customizable and complete solution, to suit the developer's needs [30, 12]. These are already well known to the community, as well as widely accepted and put to practice [32], with use cases such as the ones of Microsoft, eBay or Netflix.

One example is the **Elastic Stack**, also referred to as the *ELK stack* [26], which stands for the stack's three main components - Elasticsearch, Logstash and Kibana:

- **Elasticsearch** [27] is the core component of the stack, responsible for storing, analyzing features and providing a search engine for the fed data. It is prepared to ingest all types of data, such as logs, metrics and trace data collected from the system.

- **Logstash** [29] is the stack's component responsible for data collection. As such, it is also the stack's most valuable tool in terms of insight, as it related directly to the solution to be developed. It has multiple plugins, enabling redirecting the collected data to several commercially available monitoring tools. It is capable of capturing either application or web logs, collecting network metrics, interacting with HTTP requests and endpoints and enriching the data set.

- **Kibana** [28] serves the purpose of presenting the results of the underlying layers of the stack, using dashboards and other visual tools. The end user can visualize data stored on Elasticsearch, its indexations and so perform data analysis.

This stack relies on another platform to complement its purpose: *Beats* [23]. It ships data from clients machines to the environment in which the Elastic functionalities are being processed, normally to Logstash (as the stack's collector), but can also feed Elasticsearch directly. It is formed by a set of different single-purpose shippers (for logs, metrics, network, etc). The purpose of this shippers is to ease the transmission of data from remote machines to the ones where monitoring processing is held (automating processes like connection and transfer). It also has the benefit of shipping data according to an Elastic specification: the *Elastic Common Schema* (ECS) [31], using fields that tag the collected data, further improving analytic capabilities and correlation when processing.

The relation between the stack's components can be easily described as a diagram, as seen on figure 3.5.

Elastic also provides an APM solution, named **Elastic APM** [25, 24]. As the name implies, it allows to monitor the application's performance in realtime, along with the analysis already made available by the Stack. It relies on libraries running on the machines where the application is running, the APM Agents [22], which are available on multiple languages, support frameworks that the application might employ and do so by

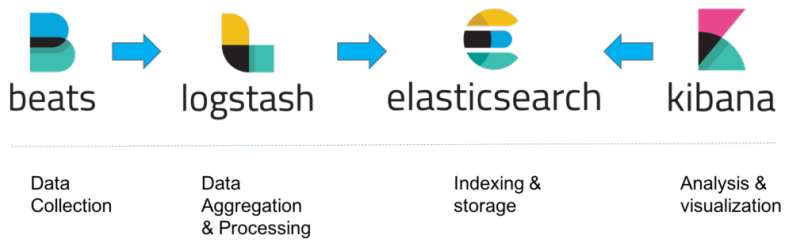


Figure 3.5 – Elastic Stack component relation [87].

auto-instrumenting the code (being automatic, it relieves the developer of the burden of instrumenting the code following ECS in order to be truly useful on the Stack). Those Agents feed the APM Server, which is a component built with the Beats framework, responsible for receiving the data, preprocessing and making it inteligible by Elasticsearch. Being stored on Elasticsearch, this data becomes available to the developer by the rest of the data (namely, it is presented on Kibana). The Elastic APM integrating with the Stack can be described as seen in figure 3.6. As this figure states, only the Agents need to run on machines related to the system to be monitored, the other components can run elsewhere - as commercially suggested, the Elastic Cloud.

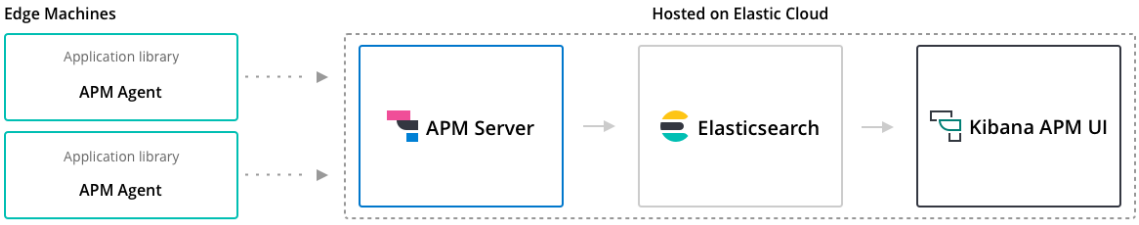


Figure 3.6 – Elastic APM relation with the Elastic Stack [24].

Employing a stack mitigates one of the problems found by Brebner on [4], namely the difficulty in combining tools, due to the "lack of standards for APM data". As the aforementioned tools on this stack were already designed to be put to use in a combined setting, each offering a solution for a specific layer or APM activity, the consumed and generated data by each tool is already proper for *intra*-stack communication.

3.3 Visualization Tools

It has been already said that the focus of this work is the collection of the data and further correlation with the code that might led to it. However, data visualization might also be an important feature to achieve the problem's solution, in its whole. As it shall not be implemented anew, it is required to employ an already available tool.

As mentioned in the previous section, Kibana specializes in displaying log data. It is fed from Elasticsearch's data, enabling querying and providing dashboards for a practical summary of the system's runtime performance. On the other hand, **Grafana** [43]

directs its focus towards metrics, such as CPU and memory consumption over time, for example. It might be also fed by Elasticsearch, if metrics were previously translated into logs and shipped to Logstash, but it is normally fed by usual metrics collectors, such as **Graphite** [40] or **Prometheus** [69]. Being a not-so-verbose type of observability data, it might be practical to achieve an overview monitoring over the system, but might appear as scarce when troubleshooting and/or deeper investigating over a specific system runtime state. Comparisons have been made between the two tools [88], concluding great relevance of the two in each of their target areas, and pointing out Kibana's more powerful querying features and the greater power available by applying logging: as a more extense, and verbose, approach, logs can be fitted with nearly all kinds of information, even the already described tracing data, key to a proper insight on the system.

3.4 Context-Based Analytics

Now that the means to gather and visualize data about the system have been presented, it is important to relate said data to the code that led to it. Cito et al. proposed in [6] an approach to do so with **Context-Based Analytics**. Research on the field, seeking to achieve a proper way of relating runtime data to improve the development process, concluded numerous challenges in retrofitting the data to a specific problem's cause, namely due to the bulk collection of undirected data (meaning, without a specific purpose set at collection-time), only resorting to *post-mortem* analysis [1]. Similar research, in DevOps practice, pointed out the lack of tools that approach this issue, mentioning the relevance of automated traceability to the development of reliable distributed software [14].

Traditionally, the aforementioned relation could only be established by a human intervenient who is aware of a certain level of functionality and characteristics of the system at hand and uses that knowledge as base for analysing dashboards, then searching the recent history of changes on the code (normally, on a Version Control System) and finally roaming the project's code in the hope of finding some code chunck that could be responsible for the detected behaviour. This practice is significantly delayed by the fact that normally a problem does not arise from a single source or, at least, not directly. As such, one finding of possibly malfunctioning code could lead to many others.

In the new approach, the authors provide a framework [34] capable of retrieving application state data from Elasticsearch, compares it with insights that result from code changes on Git and uses ANTLR [15] to establish an AST. It is prepared to do so automatically and, as such, the aforementioned human intervenient needs only to evaluate the code blocks signaled by the frameworks as potential causes to a detected performance deficiency. The key functionality is the establishment of an AST, i.e. a graph, that explicitly exposes previously implicit links between runtime information and the source code that led to it.

A comparison of the two methodologies is ilustrated on figure 3.7.

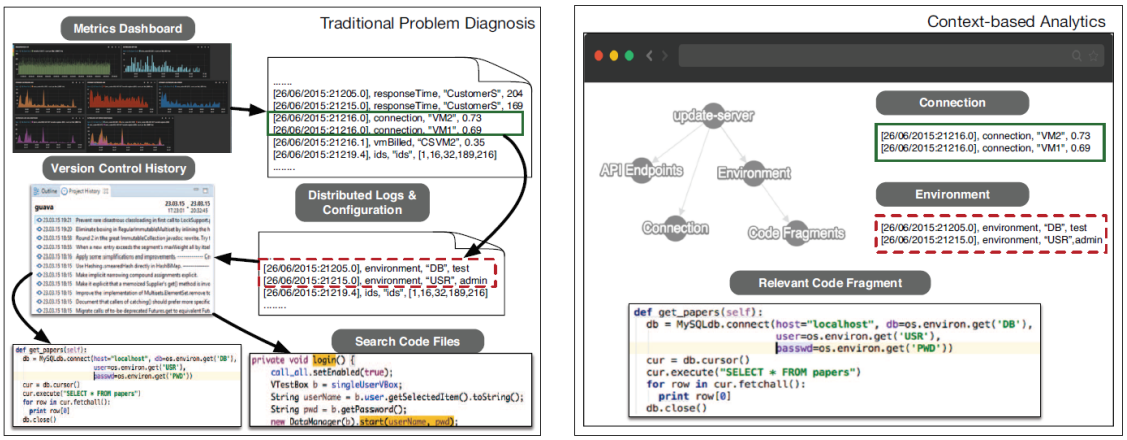


Figure 3.7 – Comparison between traditional problem diagnosis (left) and the proposed context-based analytics solution (right) [6].

On the left, the aforementioned traditional and human-intervened method of problem diagnosis is depicted. The steps may vary, or can even be repeated throughout the cycle, but the main procedure is clearly described: it starts with analyzing metrics dashboards and the distributed logs associated with an encountered situation, check the code alterations on Version Control History and finally diagnosing one or more code chunks as possible causes for the encountered problem. On the right, the newly proposed computationally ran algorithm is presented as a pseudo-GUI, depicting the graph that resumes the gathered information and its interaction, as well as context information (labeled "Connection" and "Environment") and, again, the possible cause of the problem, i.e., the "Relevant Code Fragment" that led to it.

The graph that is used in practice, the *context graph* is the sum result of the *application model* that, itself, instantiates the *meta-model*. Briefly, the meta-model defines entities that can be Runtime Entities (observable execution facts) and Program Files (source code), as an abstraction of the system's execution. The application model relates the previous abstract meta-model with concrete instantiation: uniquely defining entities (and their attributes), the data provider, the project's repository and other concrete information about the system's operation. The stages where these two models are defined are referred to as *Model Definition*, depicted on the left side of figure 3.8: on the upper left side, the meta-model is depicted, introducing its brief and abstract concepts; on the lower left side, the application model instantiates the previous one, with the system's concrete characteristics. After the definition of the model, online processing is achievable, as the context graph is constructed, assigning real gathered runtime data to its relative positions according to the model. This stage, depicted on the left side of figure 3.8, is referred to as *Online Processing* and it traverses three layers: selecting a node in the graph obtained on "Graph Constructing", enables its "Context Expansion", relating a specific system point to a certain characteristic (e.g.: memory consumption), which can then be observed on "Feature Visualization"; besides that, for each expanded node, it is also obtained a set of

related nodes, which can also be expanded.

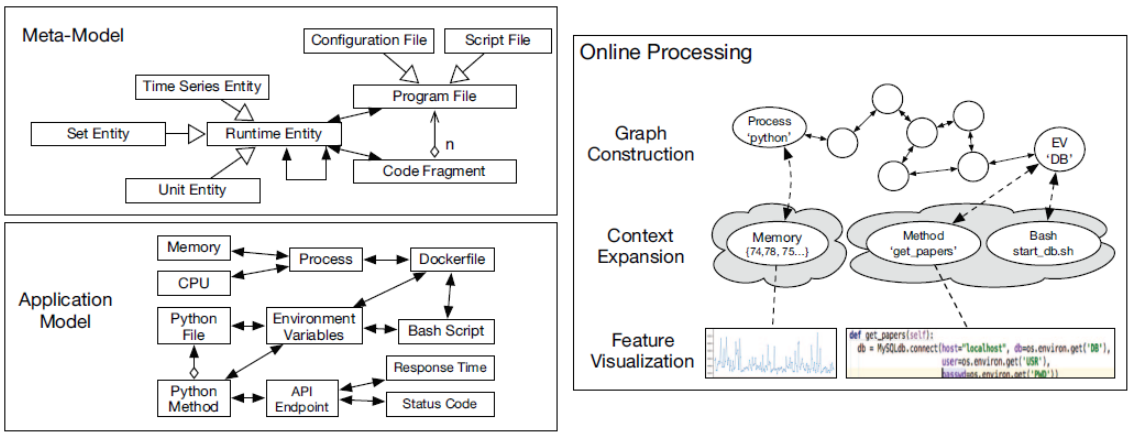


Figure 3.8 – Stages of Model Definition (left) and Online Processing (right) [6].

The aforementioned proof-of-concept framework served as a prototype to test their new approach. Tests were conducted on two separate cloud applications, the results were positive, when comparing the new *Context-Based Analytics* to the traditional methods: the number of required analysis steps was noted to decrease by an average of 48%, as well as the number of needed inspected traces that also decreased, in average, by 40% [6].

There were already available solutions that attempted to provide a relation between a change in the code and a specific occasion in performance result deviation, like *TemPerf* [5]. Some commercial solutions, like Datadog [35] already provide this kind of approach. However, this solution does not analyze the source code in order to pinpoint the source of a performance effect, but rather establishes a temporal correlation between it and a recent change in the code. For obvious reasons, this can be helpful to reduce the search space but still has the limitation of presenting results too sparse to be practical, as those depend only on timing and not on a cause-effect relation, determined from code and system's interaction analysis.

Following the work on *Context-Based Analytics*, another team also led by Cito proposed *PerformanceHat* [7], a technology presented as an Eclipse IDE plugin but whose underlying functionality is composed of a profiling system that constructs a global performance model from the system's production environment's gathered performance data and results. Briefly overviewing, the model could be described as a tuple $\langle A, D, S, \Gamma \rangle$, based on:

- An **Abstract Syntax Tree (AST), A,** that represents the source code, establishing the relations between methods, concerning method calls, branching and loops;
- A **dataset of runtime traces, D,** provided by an external monitoring tool;
- A **specification query, S,** that consists of a mapping, returning true for each match of source code's excerpt to a data point on the runtime traces dataset;

- A **inference function,** Γ**,** used to predict the performance effect of changes on the source code (from previous contexts)

After testing their prototype on 20 individuals tasked with software maintenance, the results of employing *PerformanceHat* show a significant decrease in time required to find the "root-cause" of a runtime problem, after its detection. As such, this solution showed itself practical and useful, presenting and documenting the construction of a model that could be employed on the project at hand. Moreover, the introduced overhead was only significant on edge cases (full system builds, instead of incremental builds or when the IDE was executing a specific build for the first time) and so did not represent any relevant limitations.

Referring to the cloud-native setting, solutions such as **CloudRanger** [19] or **Grano** [18] also proved themselves relevant. Both apply the construction of a graph relating each system component to the problems it may cause, denoting all inter-component relations throughout the system. Correlations between nodes in the graph are then computed, based on similar registered "anomaly patterns". After that, a Root Cause Analysis (RCA) is performed based on graph traversal algorithms, linking an anomaly or problem detected in runtime to its cause, related to a specific component. The two solutions presented satisfying results, but CloudRanger did so with the advantages of using black-box monitoring (disregarding the need for explicit instrumentation), not needing *a priori* knowledge about the system and being easily mutable and prone to adaptions (a simples change in correlation function can introduce numerous possibilities for customization).

3.5 Low-Code Monitoring Approaches

As the reader can easily observe, the above sections focused on tools and their relation with "common" applications, i.e., those whose source code is written in a typical high-level language. This does not discredit their value to study the problem to be approached on this study, but it does not give any fully applicable solution, either. As such, and bearing in mind the usefulness of the aforementioned tools and processes, research was conducted on the approach of low-code platforms to observability and monitoring.

OutSystems already approaches monitoring and code instrumentation. The generated code, result of translation of the low-code abstraction, is automatically instrumented to expose logs of various key information about the system's state, namely performance metrics [57]. These logs expose several types of captured events, such as errors, invocations of external services, page and database accesses, integrations, among others. Each of these events can be mapped to an element on the OutSystems language.

Through an internal Application Programming Interface (API), an application writes its log data to a specific database, different from the one where application data is stored. This minimizes the perceived performance overhead seen by the user, as intensive log interaction could delay an application data access, if the database was the same. A

PlatformLogs extension is available, to allow for developers to obtain log data from its specific database and use it directly on the application. Such architecture is depicted on figure 3.9.

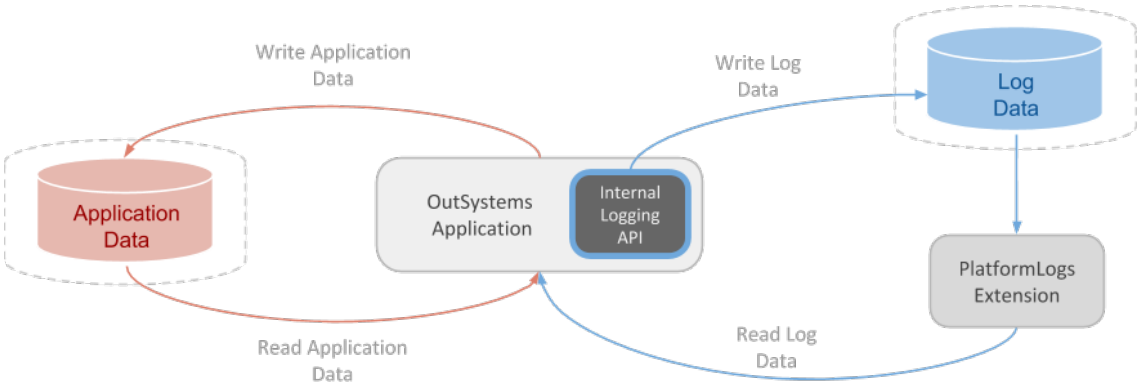


Figure 3.9 – OutSystems' logging database and architecture [58].

Information can then be leveraged by tools inside the OutSystems environment, such as **LifeTime Analytics**. This provides dashboards that depict multiple useful information, regarding client application, server and network analytics [68, 56, 64]. Developers can have an overview on the perceived experience of the end-user by evaluating the time spent in each of the components described previously [62].

The problem on the current *state-of-the-practice* is that not all tracing and monitoring mechanisms are available on all OutSystems' applications paradigms. Most of them are only applied to *Traditional Web Apps*. As described on the OutSystems section of the Background chapter, those represent only a portion of the universe of OutSystems applications. In all paradigms, server-side is always automatically instrumented and ready to expose monitoring information, specifying time spent on Service Actions, REST or SOAP integrations, extensions called, timers and database queries. Such information is available on the OutSystems' **Service Center Console** [67], specifically, on its **Monitoring area**. In Traditional Web's client-side, instrumentation also allows for traceability of processed requests, as the actions that compose them are instrumented with tracepoints, enabling observability of such apps, throughout the entire execution in runtime of a request. However, client-side on *Reactive* and *Mobile* applications is not: it is, actually, representing a blind spot in runtime monitoring. The only information gathered from those paradigms' client-side is the time that took the application server to serve a request to its client. Without the information of what are the execution traces performing on client-side, the global execution trace is not complete, ergo not delivering sufficient information to pinpoint a root cause of a problem in a reliable manner.

Moreover, OutSystems also provides REST APIs [60] to allow for monitoring data to be shipped to external tools and applications, that can also use information exposed by the platform, and leverage it specifically according to their purpose. **MonitorProbe** [80] is an OutSystems component that eases this shipping, exposing its own simplified API:

it is useful in cases where there is the need to export data to external tools, without the need to employ the extensive original API.

Chapter 4

Approach

This chapter describes the approach devised to address the problem identified in section 1.3. The description does not cover the specific implementation details, which are addressed in chapter 5.

4.1 Understanding the Problem

In order to fully understand the design of the solution to be presented, it is important to remember the goal of this dissertation:

> To collect and present tracing information of a running application developed using the *OutSystems* low-code language and ecosystem.

As described in previous chapters, in a low-code application the developer does not write the application code in a conventional (high-level) programming language. Instead, all the development is done by the programmer interacting with a tool that works at a much higher abstraction level, where the logic of the application is described using graphical entities. In the particular case study at hand, the low-code programming and execution ecosystem is the *OutSystems* language and its visual *IDE*: the *Service Studio*. The developer interacts with the *Service Studio* creating a low-code model of the application to-be and publishes it. This triggers an automatic process of translating the low-code program model into a traditional high-level source code (mostly *C#* and *JavaScript*) based on rules defined by the *OutSystems* language. This generated code is then compiled and deployed to a machine where the application will run.

The *OutSystems* experience bases itself on avoiding the necessity of the user to interact with high-level code. Moreover, the *OutSystems* ecosystem is designed to hide the bothersome and complexity of the system configurations and development processes, by

allow the developer to focus on the design of the logic of the desired application. In such very high-level software development environment setting, it does not make sense to have the developer attempting to customize the automatically generated code to add the instrumentation and tracing configuration. Thus, the addition of instrumentation code must be automatic and take place at the source-code generation phase. With such automation, the developer does not need to know when, where or how to instrument an *OutSystems* application. In fact, the tracing information produced by the automatic instrumentation process may also be hidden from the software developer. *Observability* should come by design, being available everywhere and anywhere. In this way, it should be possible to access the monitoring information, collect and process the important data, and present it to the developer using the same concepts and abstraction level that was used to develop the software, i.e, present the information directly in the *Service Studio* and using the concepts of the Outsystems language.

Bearing in mind that, in a common case, an *OutSystems* developer does not have to the generated code, it would not make sense for the tracing information provided by any solution to be written in the context of the generated code. The developer does not know the names of the classes, methods and other entities that are generated. Some are fairly easy to correlate to the ones designed in *Service Studio* using the low-code abstraction, but others are not. And even if all were easily comprehensible, it is not ideal to impose on the user the need to translate each piece of information in order for it to be intelligible. It makes sense that the solution should provide information that transmits the same level of abstraction that the user is used to, the abstraction in which all development was done: the low-code, *OutSystems* language, abstraction. This eases the interpretation of data and minimizes flaws related with that activity.

It is important to note that *OutSystems* as already approached *Observability* in the past, by providing a few tools with monitoring features over *OutSystems* applications. However, these tools fell short as a solution to solve the problem addressed by this dissertation. The majority of software development in *Outsystems* is migrating to a more recent reactive (and mobile) paradigm, having logic executing both on client and server sides of the application. *OutSystems* supports this kind of development, however its current monitoring tool is only available on *Traditional Web* applications, where all the application logic executes on the server side. This tool is called *LifeTime Analytics* [63] and a sample screen is depicted in figure 4.1.

The *LifeTime Analytics* tool provides some insight on the application's runtime performance: *Average Response Times*, *Response Time Distribution*, *Screens Getting Slower*, etc. It is, in fact, a tool that provides a lot of insightful information on the behaviour of the application being monitored, and all of thiubset information is collected automatically and without intervention from the software developer. However, as stated, it is only available for the *Traditional Web* subset of *OutSystems* applications. As developers are now adhering to more recent technologies and using new *reactive Oustystems* applicaitons' model, the existing *LifeTime Analytics* is also loosing ground in the *Outsystems* software

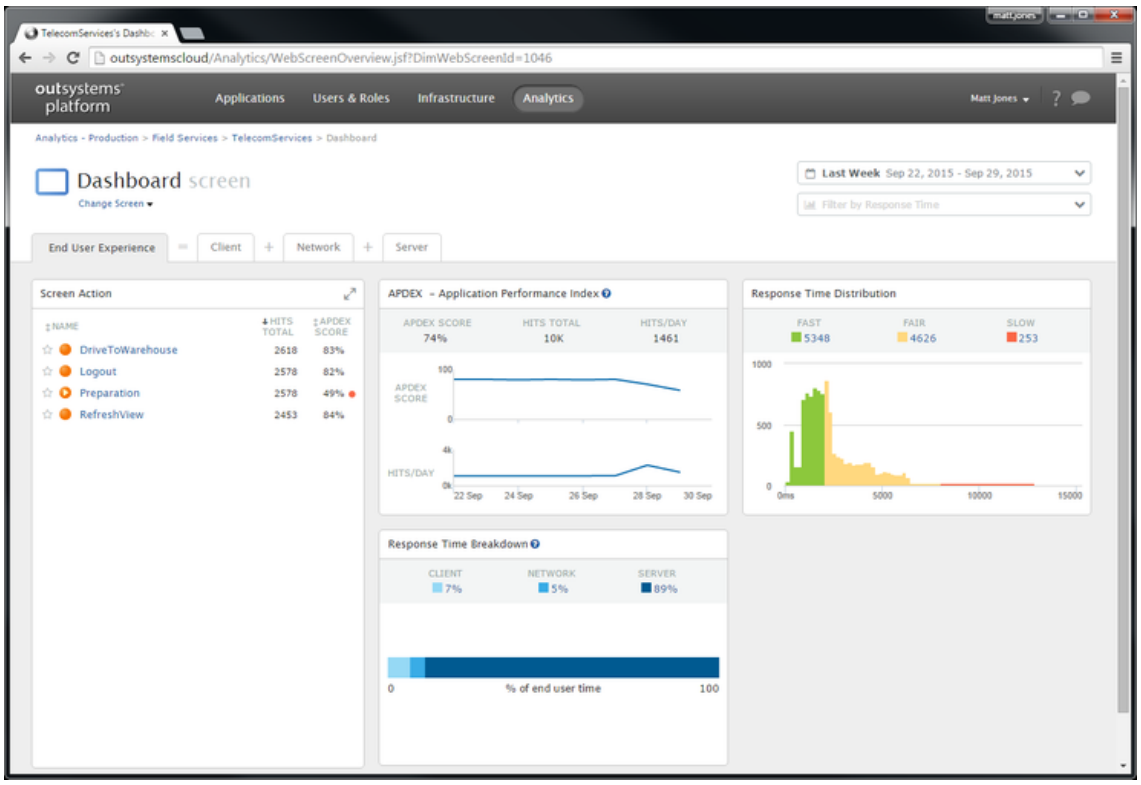
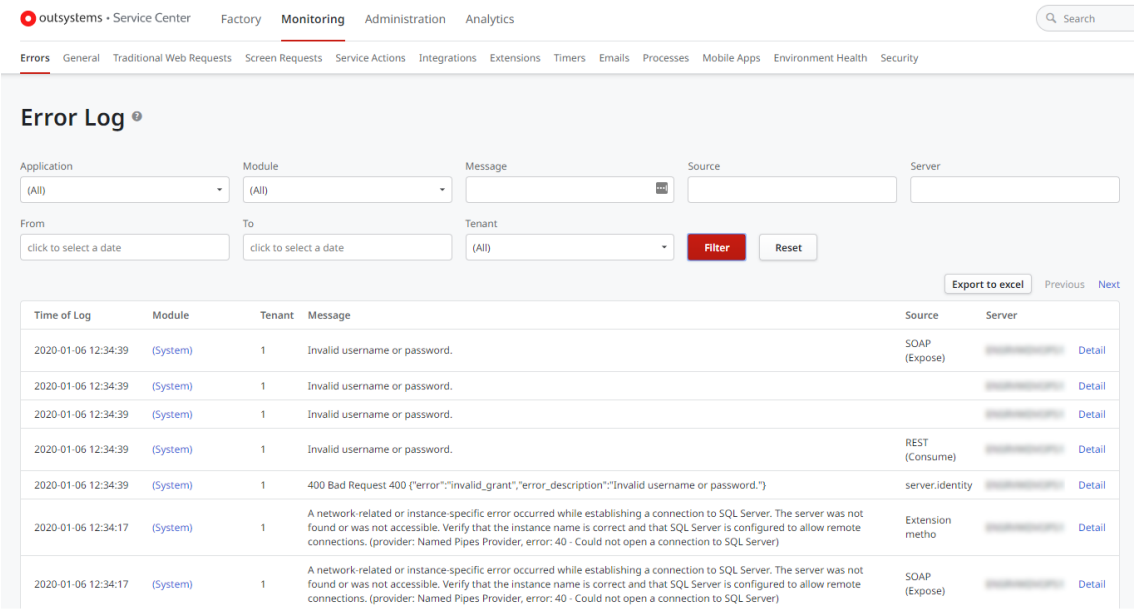


Figure 4.1 – Example screen of *OutSystems' LifeTime Analytics* monitoring tool.

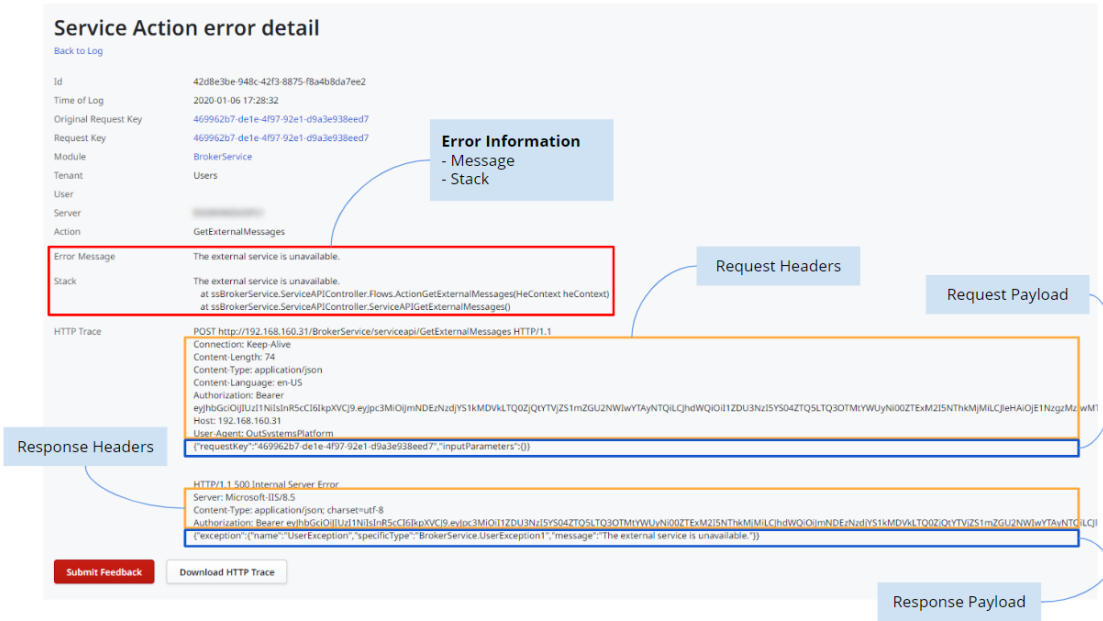
development workflow.

The second aspect of *LifeTime Analytics* that makes it not ideal as a solution to the problem addressed by this dissertation is also present in other monitoring tools provided by *OutSystems*: the difference in the abstraction levels of the provided information in comparison to the one provided by *Outsystems* service studio, which is known by the user. It was already stated the importance of maintaining the same level of abstraction for all the software development process, including the program specification (in the low-code language) and the tracing information provided by the monitoring tool. In any case, as no *OutSystems* current tools provide specific tracing information at the abstraction level of the low-code language, showing detailed information on the *OutSystems* generated code makes it hard to identify the root of the problem. Most of the currently provided information is usually excessively technical and related with the underlying services and components that were not developed by the user, nor did he interacted with them in any way in the development process. Examples of that are present on *Service Center*'s Monitoring tab [66], depicted in figure 4.2. In the *Error List* depicted in this figure, one can see a set of collected logs, none of which report on the *Service Studio*'s low-code abstraction level. In the *Error Detail* screen (figure 4.2b), one can easily notice the overly technical information, when compared to the abstraction known and applied by the user. This does not mean that this data is not relevant: it is, and it serves multiple purposes, namely technical troubleshooting (either by the user or by an *OutSystems* support expert).

However, the different nature and abstraction level of the information provided by the monitoring tool and the one the developer if familiar with, make it hard to map the information from the former with the concepts of the latter. This becomes particularly relevant for developers with low technical background.



(a) List of *Errors* logged



(b) Example of an *Error Detail* screen

Figure 4.2 – Example of *Service Center*'s Monitoring tab, namely *Error Log*.

A third problem was also identified on both tools *LifeTime Analytics* and *Service Center*: the shortage of tracing information. *LifeTime Analytics* provides insights on the overall

performance of the application, presenting aggregated results of execution time of each application screen, among other information. *Service Center* focuses only on logging information. None of the tools from the *OutSystems* ecosystem instrument applications towards distributed traceability and, therefore, are not able to present tracing information from settings where multiple *OutSystems* modules are executing in different nodes/machines. We addressed this problem in this dissertation.

4.2 Design Overview

To better understand the design of the devised solution, one must start by understanding how a typical *OutSystems* application is structured. In particular, one must understand where the application's logic is placed. This application logic dictates the application behaviour, so it is at the root of the runtime behaviour and outcome. As so, the subset of the application's code which holds and manages this application logic must be the focus of the instrumentation process: in order to observe how a running application is behaving, it must report on what is it doing at the (most) relevant actions; so, whenever the application does something relevant, that *something* must be expressed or reported in the tracing information. It is now fairly obvious to understand that the instrumentation code (i.e., the code that will report information about the application's runtime behaviour) should provide information enough to map (as close as possible) the reported actions and events to the original application's low-code and its logic. The details on how we addressed this problem in our prototype will be described in detail in the following chapter 5, while in this remainder of this chapter we will cover the main design choices of our approach.

We can identify several classes of entities that shall be traced in the context of *OutSystems* applications. It is important that the trace reflects the entry and exit points of each of these entities, and also the most relevant and useful attributes of each entity, including a well-defined description of the entities' hierarchy so that a drill-down analysis can be put to practice. The classes of entities we considered are:

Application — An *OutSystems* application is comprised of multiple *Modules*. Modules are independent in terms of implementation, but their workflow dependency and relations must be respected to ensure the application functionality as a whole. One practical example is an application composed of two modules: the *Core* and the *Client* modules that, although having independent codes, must interact and collaborate as the Server and Client sides of the application.

Module — A Module is an entity programmed in the *OutSystems* language, which serves a specific set of tasks in the context of an application. It can implement actions that execute specific tasks, display visual interface contexts, or keep and manage interactions with external entities (such as REST or SOAP calls, and database queries).

As such, a module is delimited by entry and exit points, reached through the aforementioned calls (and exited on their returns).

Screen — Screens are defined by a context on the visual interface shown to the user. In a Screen, information can by displayed through numerous ways, and actions can be triggered by interacting with the visual components.

Action — An action is a programmed set of instructions in the *OutSystems* language (similar to a method or a function in a traditional programming language). It too has entry and exit points, delimited by its calling (and the parameter inputs, if needed) and its return, respectively.

Action component — Can also be referred to as *Nodes*. Any kind of instructions that shall be performed in order to complete the Action's execution: other Actions, Assigns, Database Queries, etc.

The approach we devised focused on the *Action* and *Action component* (or *Node*) classes, as they represent, respectively, the logic and logical elements of an *OutSystems* applications. As a proof of concept, the *Application*, *Module* and *Screen* classes were also covered but with just some brief tracing information, ensuring the continuity in the report of the application execution workflow and easing the interpretation of the presented tracing information.

In *OutSystems*, the tool that presents the low-code abstraction to the developer is the aforementioned *Service Studio*, and the application logic is defined mainly with *Actions*. Depending on the type of the application being designed (*Traditional Web* or *Reactive*) to be executed, those can be either*Server Actions* or *Client Actions* respectively. Both have their own set of *Nodes* and can be composed by different kinds of operations. They also differ in the programming language used by the corresponding code generator. The generated code that represents *Server Actions* is written in *C#*, while the code that represents *Client Actions* is written in *JavaScript*. Graphically, they can be easily identified, as *Server Actions' Nodes* are drawn with fully colored background, while *Client Actions Nodes* are drawn with a hollow background and a thin outline. These makes it easy to distinguish between those two types of *Actions* when dealing with *OutSystems* code, such as the examples presented in the following sections.

Figure 4.3 – *Service Studio* representation of a simple *Server Action*. Notice the *Start* and *End* buttons with a full green background.

As described roughly in the previous chapters, the are several kinds of information that can be useful to achieve an insightful trace of a running low-code application. That

Figure 4.4 – *Service Studio* representation of a simple *Client Action*. Notice the *Start* and *End* buttons with a green border and hollow white background

information could be related to any of the intervening partners a the communication process: the *Client*, the *Server* and the *Network*. As such, instrumentation shall cover all the three communication partners, so that the trace does not exhibit information voids. This means that if a *Client Action* is triggered (say, by the press of a button on the application's UI), a trace collection procedure start, as this marks the beginning of a user's interactive operation. The trace is then be documented with all information of what is executing on the client-side: namely the *Nodes* executing as part of the *Client Action*'s logic graph. Then, one of the *Nodes* of the *Client Action* may invoke a *Server Action* over the network. The same strategy applies to the application logic executing on server-side, retrieving a result back (also over the network) and leading to some final executions on the client-side. All of this before ending the *Client Action*. Our tracing solution will also collect both the information of what is happening (executing) on the server-side and the time spent in such requests over the network. Only this way a full trace is collected, thus enabling a practical observability tool that enables the user to know what is executing in the application when the aforementioned button was pressed, the order by which *Nodes* were executed, how much time did each *Node* take to execute, etc.

To achieve such a solution, a certain set of components has to be put to practice. As mentioned in the previous chapter 3, there are already multiple tools and technologies that enable instrumentation, trace collection and other necessary features for this work. After weighting the pros and cons of those tools, and considering that a portion of the *OutSystems*' target users have a low technological background, this project focused on the technologies that not only enable all the necessary aforementioned features, but also did so in the most user-friendly manner. Also, the selection of tools considered that some tools are more prominent than others to become industry standards, due to their nature and/or to their active supporting community.

The first needed component is *an instrumentation framework*. Such framework provides the methods and entities necessary to collect and export data from a runtime application. As most frameworks, its code will be placed alongside the application code. However, as it was previously mentioned (and will be described in detail in the next chapter), the focus of this project is not to instrument a single application, but rather to create a solution that automatically instruments *OutSystems* applications. This is a major feature of our work: instrumentation code was not placed on *OutSystems* applications itself, but rather the compiler of the *OutSystems* platform was modified to add such

instrumentation automatically when it generates the high-level code for an application developed in *OutSystems*.

A caveat of the current approach (at least, on the timespan of this project) is that it is not compliant with *typical* automatic instrumentation, in the sense that a tool would be used to place instrumentation code automatically on an application. Such tools rely on the analysis of the high-level code and export information at that abstraction level, since they have access to no other. Therefore, if applied to the *Outsystems* case, the tracing information available to the user would present nomenclature and the structure for a code that the user had never access nor seen before. Thus, such approach does not solve the problem addressed by this project, since it is necessary that to present information to the user in the same low-code abstractions that the user is familiar to. To achieve this with automatic instrumentation, some sort of translating process had to take place, so that the low-level information gathered by the tracing system is translated back to the original low-code, in order to be comprehensible to the user. This effort would be massively time-consuming, taking nearly as much effort as placing the instrumentation manually in the first place. As such, aiming to produce a proof of concept, in this project we opted for simulating the instrumented code generations process by the *Outsystems compiler* and used a code manipulation framework to selectively add the required instrumentation lines of code to the plain code generated by the compiler. The compiler has access to the low-code representation of the application, since its job is to generate high-level code based on it, thus it is easy for the instrumentation code to access this data (*Node* names, labels, the exact moment in which a certain *Node* begins to execute, etc) and customize the instrumentation so that it is as complete as possible, but still maintaining the abstraction level. Now, the compiler, when generating high-level code based on the low-code abstraction, will also generate instrumentation code with data gathered from that same low-code abstraction and not the high-level generated code.

The instrumentation framework chosen was *OpenTelemetry* [51]. Although still in early stages of development, it is a follow-up of a major tracing precursor: *OpenTracing* [54]. As such, it provides implementation to important distributed tracing and context propagation techniques, such as *Baggages* (discussed on the previous chapter 3). *OpenTracing* is also widely used in the industry, and aims to become a standard, as most *out-of-the-box* tracing tools either support it or even build on it. Another reason for this framework to be chosen was its simplicity: as it will be discussed in the next chapter, after the configuration stage, placing *OpenTelemetry* code is fairly straightforward and is easily extended and documented, by adding tags, labels, logs, etc. Finally, *OpenTelemetry* also fulfills *Outsystems'* instrumentation needs, as it supports both *C#* and *JavaScript*, allowing for context-propagation between client and server and back to the client.

Having instrumentation in place, there must be a way to properly present it to the user. To achieve this, a component is needed to collect the data exported by the instrumentation code, and provided to a presentation UI. The tool selected for presentation purposes was *Jaeger* [41]. This tool provides presentation support for all of *OpenTelemetry*'s tracing

features in a user-friendly way, guaranteeing ease of access and rapid understanding of what is happening regarding the runtime of the application. Such features will be presented in detail in the following two sections.

The two ends of tracing communication are now defined: instrumentation, running alongside the application's code, and *Jaeger*'s presentation UI, where the user can access tracing data. However, it is still left to explain the communication process between both, as it is a crucial part for the success of this solution. In a strict sense, the instrumentation code does not send any data to a presenter: the instrumentation code merely collects data about the application's runtime. It then hands said data over to a specific component: an *Exporter*. This is the one responsible for sending the data over the network to whoever is listening on the configured address and port. Similarly, *Jaeger*'s presentation service is oblivious to the collection of the tracing information and only focus presenting data. Thus, alongside *Jaeger* is running another component, the *Collector*, responsible for collecting the data sent by the *Exporter*. This *Collector* hands over the data to the *Jaeger* presentation service, in order for it to be shown in the UI.

In a typical use case, it is expected that the tools responsible for the presentation service offer a *Collector*, and the instrumentation framework was configured to use an *Exporter* related to that *Collector*. So, in our case, the instrumentation framework is the *OpenTelemetry* and it uses *Jaeger Exporters* on both the client and server sides of the application, so that data reaches a *Jaeger Collector* over the network, that hands it over to the presentation UI.

An overview of the proposed architecture is depicted in figure 4.5.

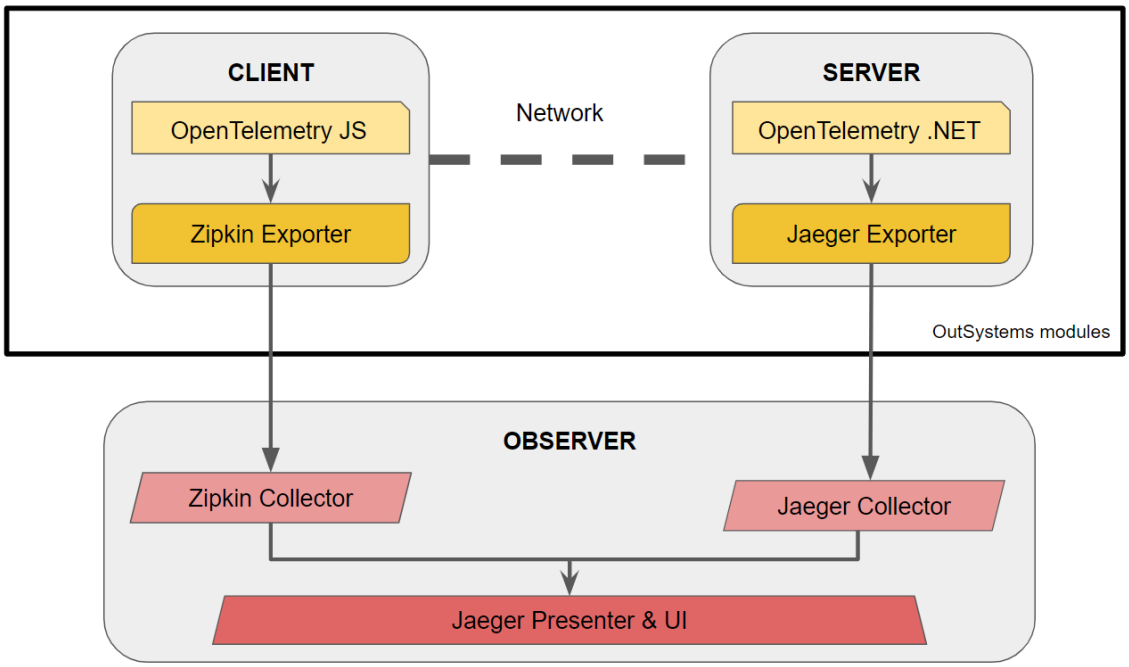


Figure 4.5 – Proposed architecture of the solution to be implemented.

However, in our case, we opted for a different *Exporter* and, therefore, also a different

Collector, for the client-side of the application. The reason for this is that the client-side of *OutSystems* applications does not use *Node.JS*, meaning that the generated *JavaScript* is only compatible with the browser, and the *Jaeger Exporter* is only supported with *Node.JS*.

The server-side instrumentation is configured to hand-over the data to *Jaeger Exporter*, that sends it over the network to the listening *Jaeger Collector*. However, due to the above mentioned limitation limitation, the client-side instrumentation had to be configured to use *Zipkin* [90]. This tool is similar to *Jaeger*, but that presents a less user-friendly interaction. So, for the client-side, the tracer hands over the data to a *Zipkin Exporter*, that sends it over the network to a *Zipkin Collector*.

Fortunately, due to their similarities, *Jaeger*'s presentation service is compatible with *Zipkin Collector* and is ready to correctly interpret data handed-over by this component. *Zipkin Exporter* is also prepared to export data collected by *OpenTelemetry*, just like *Jaeger Exporter* would, so this change does not affect the instrumentation process. So, *Jaeger*'s presentation shows the summary of data from both application sides, meaning that a trace encompasses a complete picture of runtime behaviour of the application.

As some components of the solution run side-by-side, a simplification can be made. A *Client* component can be identified, holding the client-side application code and the *OpenTelemetry JavaScript* (more commonly referred to as *OpenTelemetry JS*), as well as the *Zipkin Exporter*. A *Server* component can also be identified, holding the server-side application code and the *OpenTelemetry C#* (more commonly referred to as *OpenTelemetry .NET*), as well as the *Jaeger Exporter*. Both the sides communicate over the network, as the application sides interact with each other. Finally, a third component, the *Observer*, can be identified, holding both *Zipkin Collector* and *Jaeger Collector*, as well as *Jaeger Presenter & UI*.

Applying this solution, each interaction with an *OutSystems* applications triggers an automatic process that end with the presentation of a trace on *Jaeger*'s UI. A trace is the result of the gathered instrumentation data, that reports what was executed, its timings and other important monitoring information. A trace is characterized by environmental data, such as its start and end date and time, its duration, but most importantly by the services being reported. *Service* is a term used by *Jaeger* that represents an entity that executes application code and reports it. Following figure 4.5, both the *Client* and the *Server* are services, as both have their own application and instrumentation code, as well as their own *Exporter*. The designed solution names these services automatically, in the tracing configuration code, as the name of the *OutSystems* application, concatenated with "*_Client*" or "*_Server*", if it is a client-side or server-side component, respectively. Traces are composed of *Spans*, which represent the time span that any instrumented operation took to execute and can be also documented with tags, and other customizable information. *Spans* can be structured in a hierarchy, as a *Span* can have parent and child *Spans* as needed. This feature is useful to report events such as an *OutSystems Node* that is executed inside an *Action*: the *Span* reporting the *Action* would be the parent of the *Span* reporting the *Node*.

Spans are not executed, but rather convey information about an execution. In an abuse of terminology in this document, the expression *"span is executed"* (or similar) will be used to translate the situation in which a certain instrumented operation was executed, documented and reported, resulting in a *Span*.

A detailed descriptions of a trace will be approached on the following two sections. First, the list of contents that *Jaeger* can present at any given time is depicted on figure 4.6. On figure 4.6a, one can can see the list of available traces related to the service *eSpace_Client* (as selected on the left-side pannel). This list can be filtered by the operations executed (represented as the names of the spans present on a trace), tags present on the trace's spans, time in which the trace reached *Jaeger*, trace duration and the maximum number of results. One can easily observe, on the upper right side, a graphical representation of the lists of collected traces (as blue circles): the x-axis represents the time in which each trace took place, and the trace duration is on the y-axis and on the circle size. This list can be ordered by most recent traces, traces with the longer execution first, etc.

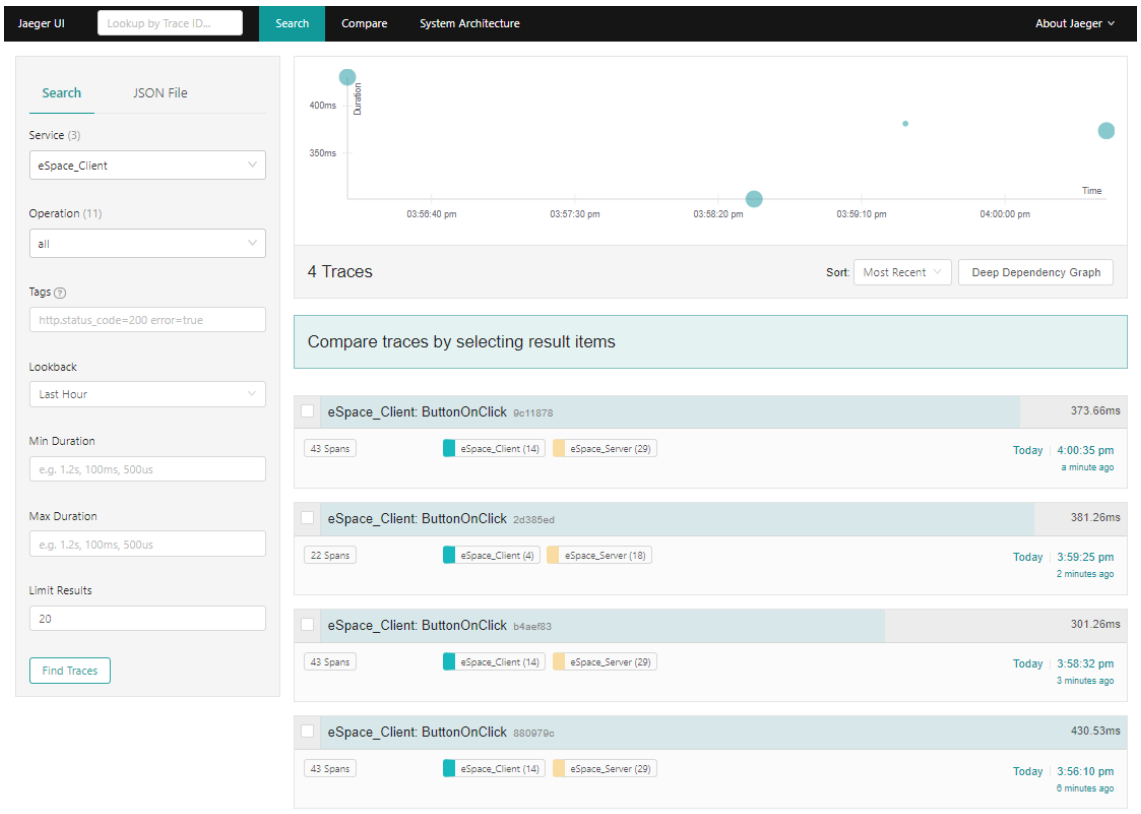
An item of the previous list is zoomed-in in figure 4.6b. Here, besides the name of the service that started the trace (*eSpace_Client*) it is also shown the name of the operation executed *ButtonOnClick*. The number of executed spans (43) and services involved on the trace (*eSpace_Client* and *eSpace_Server*) are also shown, as well as the number of spans each service emitted (14 and 29, respectively). Regarding time, the date and start time are presented on the bottom right of the detail and the total duration of the trace is shown on the upper right. Moreover, the top blue bar represents the relative duration of this trace, in comparison to all others presented on *Jaeger*'s trace list:This comparison can be observed on the trace list 4.6a.

Remembering the identified classes of entities to trace in an *OutSystems* application, some of the summarized classes can easily be identified on the example figures. *Application* and *Module* classes are depicted on the name of the services that hold each trace. The *Screen* class is documented on the spans that are generated in its context, as each of those spans *Screen* hold the *URL* to that screen as a tag: this will be shown in the following section.

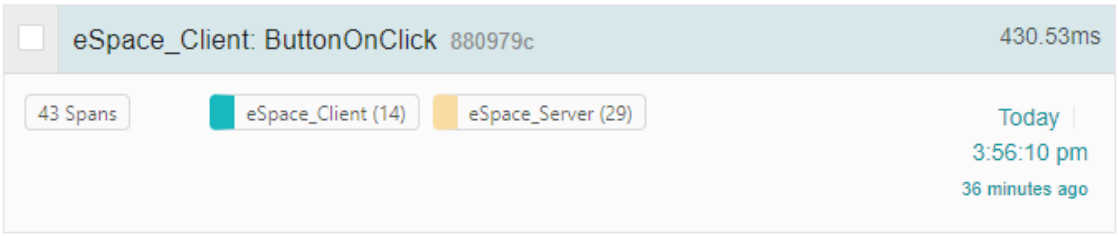
4.3 Simple Running Example: *SimpleClientAction*

In this section, the full range of features provided by the implemented solution will be approached. Bearing in mind all of the project's description already mentioned, from problem identification to the proposed solution, including its most important features and characteristics, the specific details will here be presented and discussed. Again, this description shall maintain its macroscopic view, only transmitting the design choices made and the accomplished working features: to a full overview of the implementation details, refer to the next chapter 5.

On the previous chapter, the list of traces available for presentation on *Jaeger* was



(a) Complete list



(b) Detail of a list's item

Figure 4.6 – List of collected traces on *Jaeger*.

shown. Each of those traces represents an operation executed by an *OutSystems* application. Operation can be triggered automatically, by a recurring process, but mostly they are triggered by an user interaction. To exemplify such interaction, a demo application was developed and will be described below. Considering the time span of this project, the developed solution only focused on instrumenting the application logic, which means no UI elements have been instrumented, nor was the application's accesses (openings and/or closes) or screen loadings.

A typical use case for a simplified example application consisting of a single screen, where a single button can be seen, would be:

1. An *OutSystems* developer designs an application using the *Service Studio* and publishes it.

2. The user then access the application and its main screen. NOTE: This and the previous interactions are not instrumented and will not be reflected in *Jaeger*.
3. The user would follow to press the button. The button press is recognized by the application and triggers the execution of a certain operation. This operation already classifies as "application logic" and is instrumented.
4. A new trace begins, gathering information as operations are executed.
5. At the end of the execution the trace ends.
6. The trace is now visible in *Jaeger*, enabling access to its detailed information.

Picking up on the application described above, remember that the button press triggers the execution of a certain operation. In *OutSystems*, this operation could be the loading of a different screen or the execution of an *Action*. As screen loads were not instrumented, we shall focus on the execution of *Actions*. An action may have logic as complex as the user desires it to be, but first let's focus on a simple example (a complex example will be addressed in section 4.4). This simple *Action* is called *SimpleClientAction* and its *Service Studio* representation (as it could have been designed by the *OutSystems* developer) is depicted on figure 4.7.

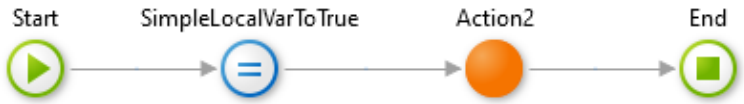


Figure 4.7 – *Service Studio*'s logical representation of *SimpleClientAction*.

As the name suggests, this is a *Client Action*: as it was triggered by an interaction with the UI, it feels natural that it is executed on the client-side, justifying the choice of a *Client Action*, instead of a button press, which directly call a *Server Action* that executes on the server-side. Between its *Start* and *End*, this *Client Action* is comprised of two *Nodes*: *SimpleLocalVarToTrue* and *Action2*. When the user triggers the execution of this *Action* by clicking the button on the application, a new trace appears on *Jaeger*'s trace list. Clicking on this new trace leads to its detailed view, as depicted on figure 4.8.

By comparing the tracing details on *Jaeger* and the *OutSystems* code that designed the *Action* on *Service Studio*, relations are evident. The trace takes the name of the *Action* being executed, and begins with a span with that same name. This span is parent to two spans: each representing one of the nodes programmed in *Service Studio* and taking the same name as the node they represent.

One caveat is that *OpenTelemetry* automatically generates spans that represent network interaction:

HTTP POST — represents the full request as issued by the client-side, including time spent over the network;

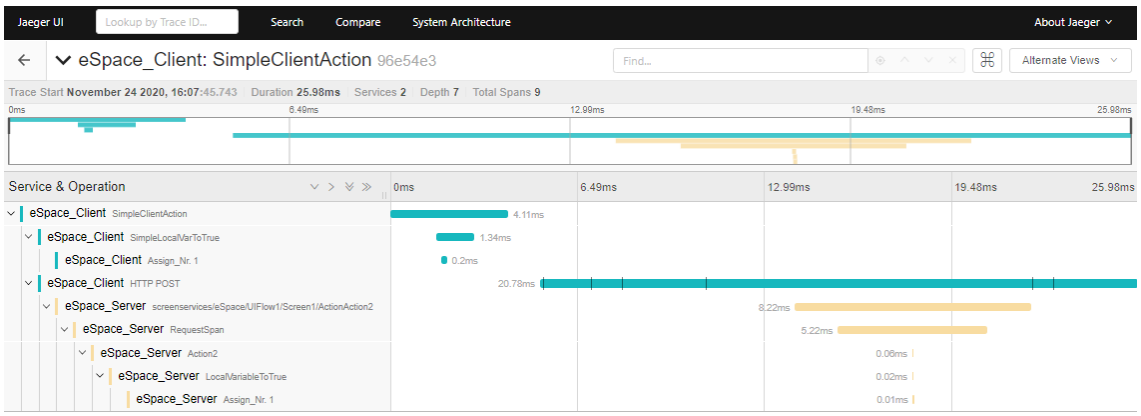


Figure 4.8 – *Jaeger* detailed presentation of a trace named *SimpleClientAction*, on service *eSpace_Client*.

screenservices/eSpace/... — represents the server's reception and handling time of the requests, and takes the automatic name of the URL requested to the server;

RequestSpan — was not named automatically, but rather defined by the author, to solve the need of having one manually created span that encompassed the entirety of the request handling, crucial to store in context and serve as parent to every other span happening on the server. This will be described in detail on the following chapter.

On *Jaeger*'s detailed view, at the top, one can observe generic information about the trace: start time, duration, number of services involved, span depth (number of levels in the span hierarchy, as a span can be parent of other spans) and total spans. After that, a graphical representation of the trace is shown on a thin window. This presents an overview, complementing the more detailed graph below, under *Service & Operation*. These representations follow some basic principles:

- Each line contains only one span. That span is characterized by:
  - The name of the Service which called it (e.g.: *eSpace_Client*);
  - Its name (e.g.: *SimpleClientAction*);
  - Its bar representation, which is sized according to the relative duration of the span. This size follows axis and scale representing time, place on the top of each graphical representation;
  - A label, following or preceding the bar representation, indicating the span's duration.
- Span hierarchy is represented by the tree organization on the left side of the screen.
- Spans are colored according to the service where they have been executed. The color coding is chosen automatically by *Jaeger*'s presentation service: in the example on figures, service *eSpace_Client* is colored blue and service *eSpace_Server* is colored yellow.

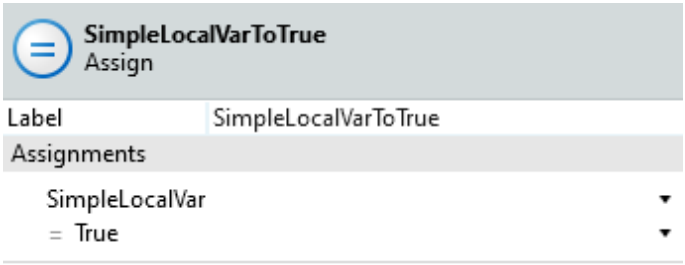
Span representations are also clickable. Clicking on them displays some additional information, most importantly their *Tags*. Those were programmed to be placed with the instrumentation code, in order to further complement and document tracing results. One practical example of *Tags* is on the *Assign* nodes, depicted on figure 4.9. *SimpleLocalVarToTrue* is an *Assign* node comprised of one *Assignment*: after executing, *SimpleLocalVar* will be set to *True*, as seen on 4.9a. On *Jaeger*, an *Assign* node, as any node, is represented by a span with its name. However, in the case of an *Assign* node, there will be a child span for each *Assignment*. In the *OutSystems* language, there is not a name for each *Assignment*, so, in *Jaeger*, they will just appear as *Assign_Nr.#*, being # the order number of the *Assignment*, as seen on *Service Studio*. The usage of *Tags* is useful in the case of *Assign* nodes, as they can document what exactly is happening in runtime: as the user can read, the value of the *Assignment_Value* tag is "*True*" and the value of the *Assignment_Variable* tag is "*SimpleLocalVar*". With this information, when analyzing tracing data, the user can easily identify what was happening at the time of this span's execution. The other tags displayed were automatically generated by *OpenTelemetry*.

Jaeger also presents a zooming feature, to allow for easier reading of the tracing results. This is depicted in figure 4.10. In figure 4.10a the results are zoomed to the time region that comprehends the request sent by the client-side, its processing on server-side and return to the client-side with the response. Figure 4.10b further zooms in, to the region that comprehends the server's reception and handling time of the requests. Finally, figure 4.10c zooms even further in, to the region that comprehends a span representing the execution of a *Server Action*, called *Action2*. From the graphical representation (and based on the approaches explained above), we can infer that this *Server Action* is comprised of a single node, called *LocalVariableToTrue*: an *Assign* comprised of a single assignment operation.

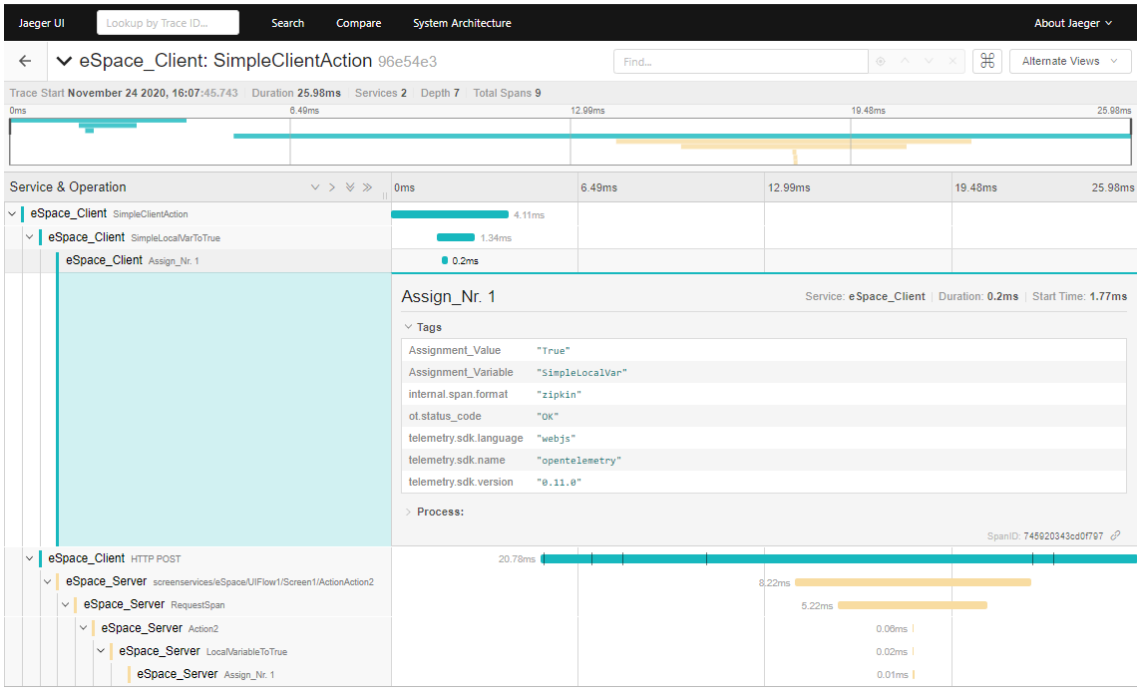
An automatic feature provided by the tools applied is the set of logs provided on the automatic span that represents the *HTTP POST* request sent by the client-side. They mark the time in which certain events related with the request take place, as depicted in figure 4.11. Those could be helpful to users with some technical background or to help when reporting errors to *OutSystems* support.

The *RequestSpan* is the first that was not created automatically by the tools, but rather based on instrumentation code, can benefit from documentation provided by tags, as depicted on figure 4.12. Those can convey any kinds of insightful information, such as the *Request_URI* or *User_Host_Address*.

Besides the representation of spans in a time-oriented axis, *Jaeger* also provides a *Trace Graph* representation, as depicted on figure 4.13. This presents the same number of spans, but as a directed graph. Spans are represented as graph's vertices (or nodes), being that edges connect each vertex representing a parent span to the vertices that represent its child spans. The graph's root is the first span to be executed: usually, the one representing the client-action, since it was the one triggered by the UI interaction, which was responsible for calling the execution of every other subsequent operation .



(a) *Service Studio*'s menu to configure an *Assign* node with one assign operation: when this node is executed, *SimpleLocalVariable* will be set to *True*.



(b) *Jaeger* presentation of tags associated with span *Assign_Nr. 1*. Notice the *Assignment_Value* and *Assignment_Variable* tags, revealing information about what exactly is happening in this assign: *SimpleLocalVar* is now set to *True*.

Figure 4.9 – Comparison between *Service Studio*'s design of *SimpleLocalVarToTrue* and *Jaeger* presentation of its data.

Again, in this representation, spans are easily related to their service by their color. Each vertex represents a span and presents the spans' name and duration. However, it does not provide clickable access to tags or logs. A view of this representation is depicted in figure 4.13a. Figure 4.13b presents a close-up to the graph's region that contains spans from the *eSpace_Client* service (in blue), to allow for better reading. In this close-up, one can easily identify the span that represents the *Client Action* (figure 4.7) as parent of two sub-graphs of spans: the first representing the chain of spans that relate to the execution of the *Assign* node; and the second relating to the calling of the *Action2 Server Action*.

An interesting follow-up feature to this graph representation is the additional color-code that *Jaeger* provides, presented in figure 4.14. Previously, spans were color-coded

according to the service where they executed in. Now, spans are color-coded according to their relative duration: graph vertices that took longer to execute tend to a stronger red colour. Both color-codes are available at any moment, and the user can select whichever better fits the ongoing analysis. Figure 4.14a shows the complete graph and the span representing the *Assign LocalVariableToTrue* is open: notice the light with hue, meaning a small duration (as it is expected for such a simple operation). In figure 4.14b, the complete graph is shown again, but this time the open span is the *RequestSpan*, representing the handling of the incoming request by the server: as it is expected for a more complex operation, the hue is red, indicating a large duration. Finally, in figure 4.14c, the same graph region of the previous figure 4.13b is zoomed in, allowing for a better reading. Particularly the graph representation with a color-code oriented towards span duration is immensely useful, as it allows for nearly instantaneous identification of the spans that take the most time to execute: this could used to pinpoint a zone in the programmed code that is behaving with a lower performance than expected, therefore needing problem-solving or optimization. As such, this feature provides a clearly easy path towards helping to solve the problem identified in this dissertation, particularly allowing to pinpoint applications' sections that underperform the expectations.

4.4 A Complex Running Example: *ButtonOnClick*

Throughout this chapter, every important feature of the implemented solution was already presented. However, the previous example was very simple and did not allow to illustrate some of the implemented features. For example, it did not show the full coverage of instrumentation achieved in the *OutSystems* language.

Thus, this section presents a more complex: the *ButtonOnClick Client Action*, whose logic is depicted on figure 4.15. To maximize the showing of that coverage even further, this *Client Action* also calls a complex *Server Action*: *Action1*, whose logic is depicted on figure 4.16. This interaction shows show nearly every instrumented node in the tracing results, allowing for a summary of the achieved coverage.

These complex *Actions* lead to a rich (and very complex) trace detail, as depicted on figure 4.17. Notice the several zones where interactions between services occur, where a big blue *Client* span is above several yellow *Server* spans, as the *Client Action* nodes call *Server Actions* or other server-side resources.

To allow for better reading, figure 4.18 presents the same trace detail as the prior image, but zoomed in to the region where *Action1 Server Action* (depicted on figure 4.16) is being executed. Notice all that nearly all the nodes present on the *Action* as programmed in *Service Studio* are represented with their own *Span*. It can also be noted that an example of an *Assign* node with two assignments (adding to the previous section's description of a single assignment *Assign* node), as can be confirmed by its *Service Studio* representation, on figure 4.19 and seen on the trace's detail, where the *FirstAssign* span has two child spans, one for each assignment.

Again, a *Trace Graph* representation is made available by *Jaeger*. This much more complex action present an accordingly complex graph, depicted in figure 4.20. The graph is color-coded according to services in figure 4.20a and according to time in figure 4.20b.

In a complex graph, the time color-coding becomes particularly helpful, as it directs the user's attention to the most time-consuming spans, thus enabling a faster comprehension of the executed operations that are affecting response time the most and, so, might need problem-solving and/or performance optimization. By analyzing figure 4.20b, one can notice that the lower half exhibits a discernible higher concentration of redder spans (i.e., more time-consuming ones). Trying to understand the cause of the problem, one would follow the graph's directed edges until reaching leaves, or terminal nodes. Figure 4.21a depicts a terminal region with a high concentration of red spans: this is the region that depicts the execution of the largely complex *Action1 Server Action*, which is expected to be time-consuming due to its large number of operations to execute. However, focusing on this region of the graph, two leaves are redder in comparison to all others.

Figure 4.21b depicts a zoom to this region, revealing that the two most red spans (and, so, more time-consuming) are the ones related with moving a *RecordList* (*OutSystems*' representation of a database table) to or from an *Excel* file (which is expected to be time-consuming, thus confirming that the data collection is behaving according to expectations).

As can be seen on figures 4.15 and 4.16, a lot of *Nodes* of the *OutSystems* language are being executed. And, as one can confirm on figures 4.17 and 4.18, all of the executed nodes are instrumented and reporting their tracing data. The instrumentation coverage can be seen on figure 4.22.

Of all instrumented nodes, only *Client Action*'s *JSON Deserialize* was not executed on the examples, but its instrumentation is placed and fully functional. *Start* and *End* serve no logical purpose on itself, they exist only to delimit the execution of an *Action*: so, said nodes will not be instrumented, as the beginning and end of an *Action* is already defined by the start and end of the span that represents it. *Comment* nodes will also not be instrumented, since they do not represent any executed code.

Exception Handlers were not successfully instrumented in the time span of this project, as well as *If*, *Switch* and *For Each* nodes. These last three nodes (all related with a conditional evaluation to model the execution flow) were left without instrumentation but bearing in mind that, even though the node itself is not instrumented, nearly every *Node* on the execution branch that follows these nodes are indeed instrumented and reporting tracing information: so, the node itself is not instrumented, but the execution path following its evaluation is. A successful instrumentation for *Client Action*'s *Destination* node was also not achieved.

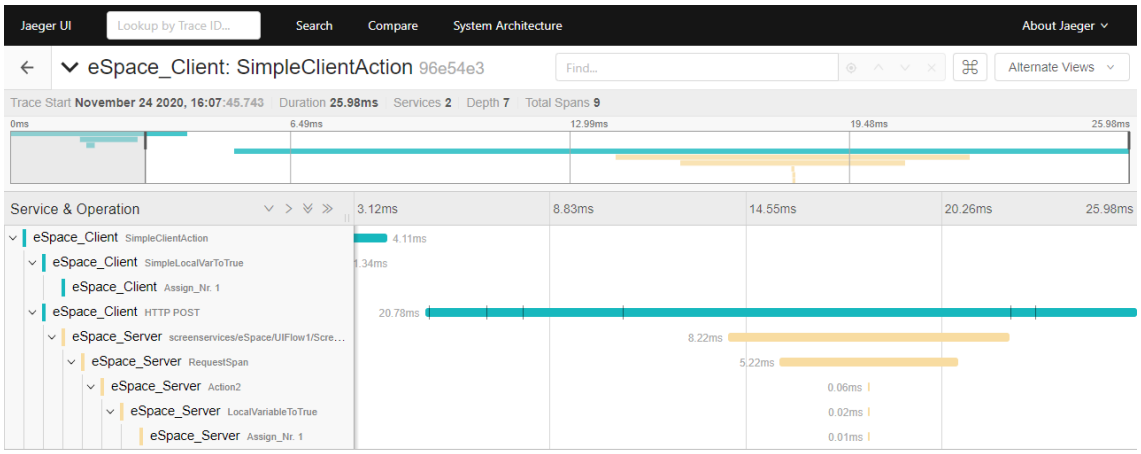
Other *Nodes* that can also be executed in *OutSystems* but are not present in the figures include those that represent *Integrations*, such as *REST*, *SOAP* or *SAP*. Those where also not successfully instrumented in the time span of this project.

These cases of *Nodes* that would be nice to have instrumented are referred as future

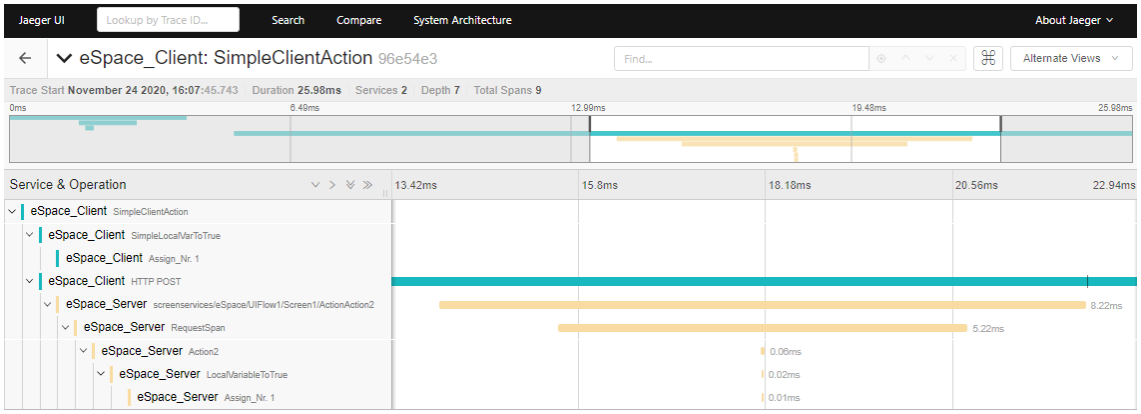
work in section 7.2 of the final chapter.

In summary, out of 18 total *Client Action* nodes, 15 were at least partially instrumented (leaving out *Start*, *End* and *Comment*), and 10 of them were in fact successfully instrumented (as depicted on figure 4.22a).

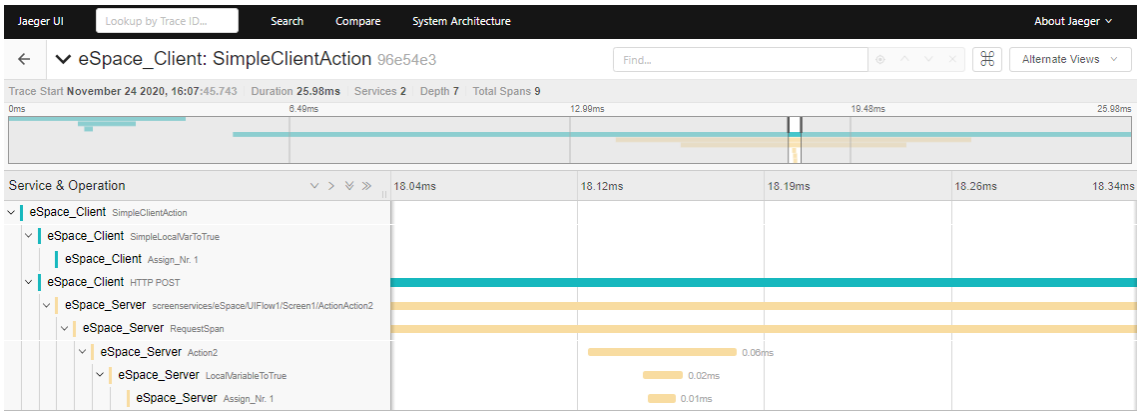
On *Server Actions*, there is a total of 16 total nodes, 13 of which could be instrumented (again, leaving out *Start*, *End* and *Comment*) and, of those, 9 were in fact successfully instrumented (as depicted on figure 4.22b). This accounts for a total of 19 instrumented nodes, out of a set of possibly-instrumented 28 nodes.



(a) Zoomed from the beginning to the end on the Client's request.



(b) Zoomed from the beginning to the end of the Server's handling time.



(c) Further zoomed, enhancing details on the spans representing the executed *OutSystems*.

Figure 4.10 – *SimpleClientAction*'s trace detail, namely the time regions where a request to the server was made.

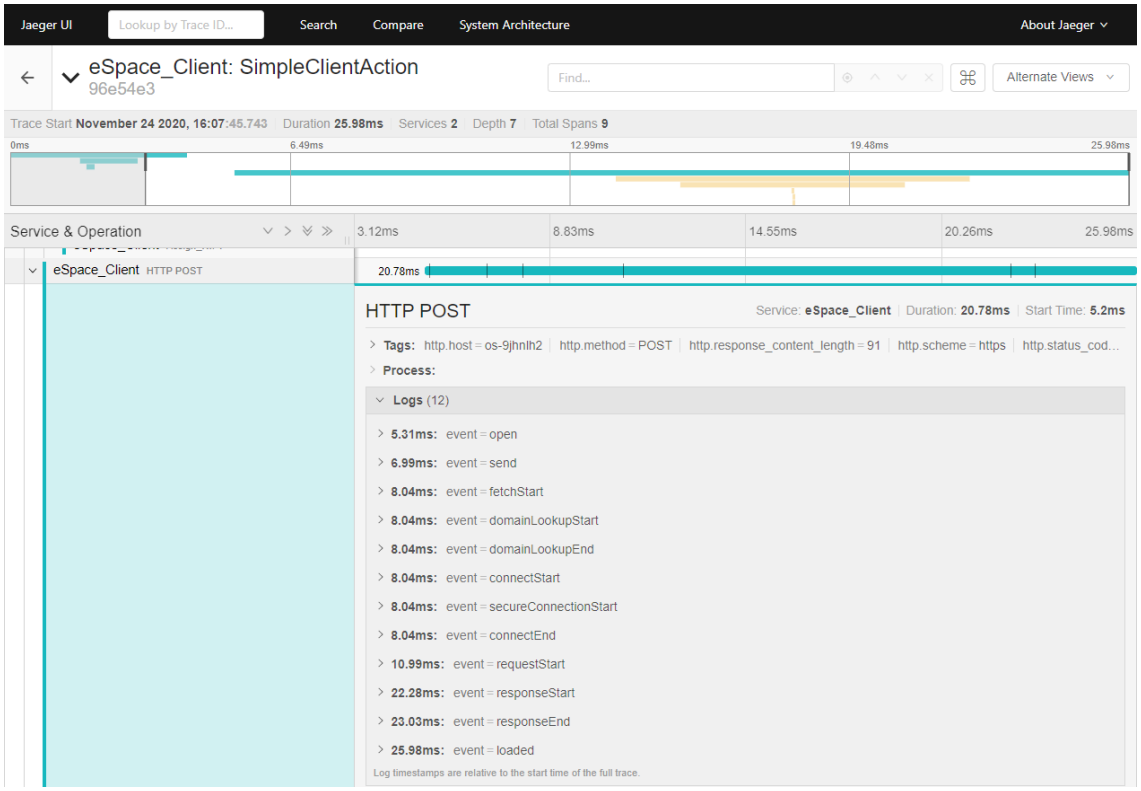


Figure 4.11 – *Zipkin*'s exporter automatically generated logs on *HTTP* requests' events, under the *HTTP POST* span.

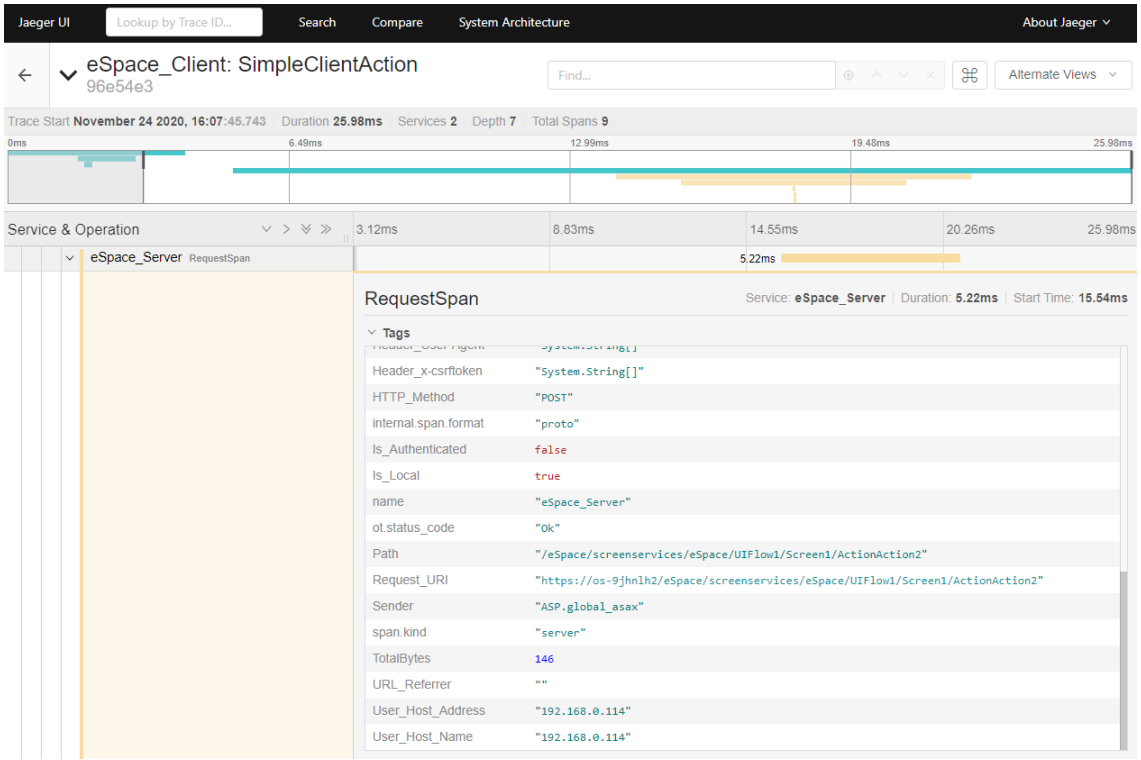
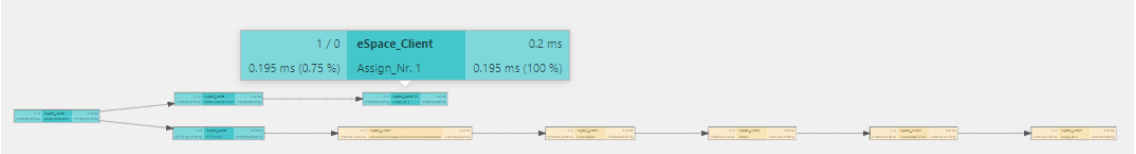
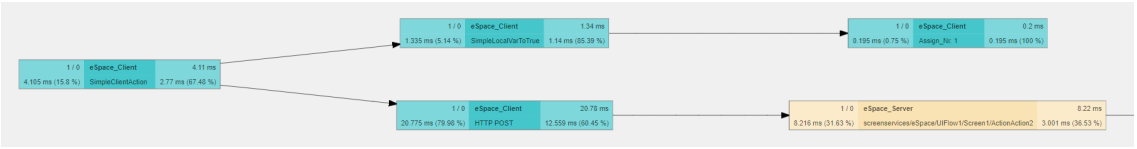


Figure 4.12 – Example of *Tags* used to document *RequestSpan*.

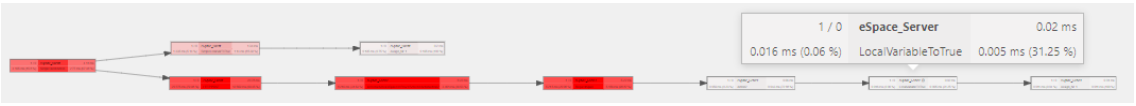


(a) Complete graph representation. The detail for span *Assign_Nr.1* is open.

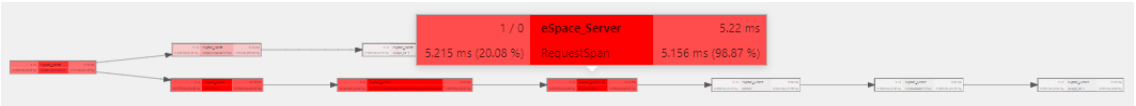


(b) Graph representation zoomed to a section mostly showing *Client* service's spans.

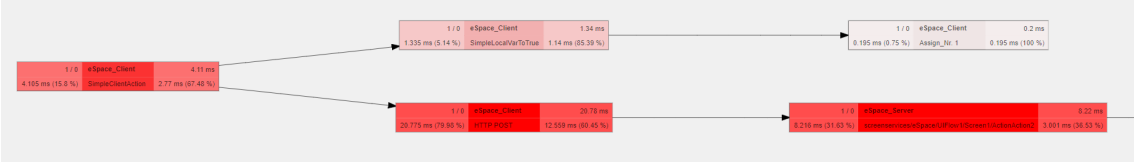
Figure 4.13 – *Jaeger*'s graph representation of *SimpleClientAction*'s trace, color-coded according to *Services*: blue being service *eSpace_Client* and yellow being *eSpace_Server*.



(a) Complete graph representation. The detail for span *LocalVariableToTrue* is open. As it is represented in a light hue (nearly white), one can infer that its time consumption is low.



(b) Complete graph representation. The detail for span *RequestSpan* is open. As it is represented in a bright red hue, one can infer that its time consumption is high.



(c) Graph representation zoomed to a section mostly showing *Client* service's spans, where one can observe spans of different durations (as they are represented in different shades of red).

Figure 4.14 – *Jaeger*'s graph representation of *SimpleClientAction*'s trace, color-coded according to the duration of each span: the redder the hue gets, the longer the duration of the specified span.

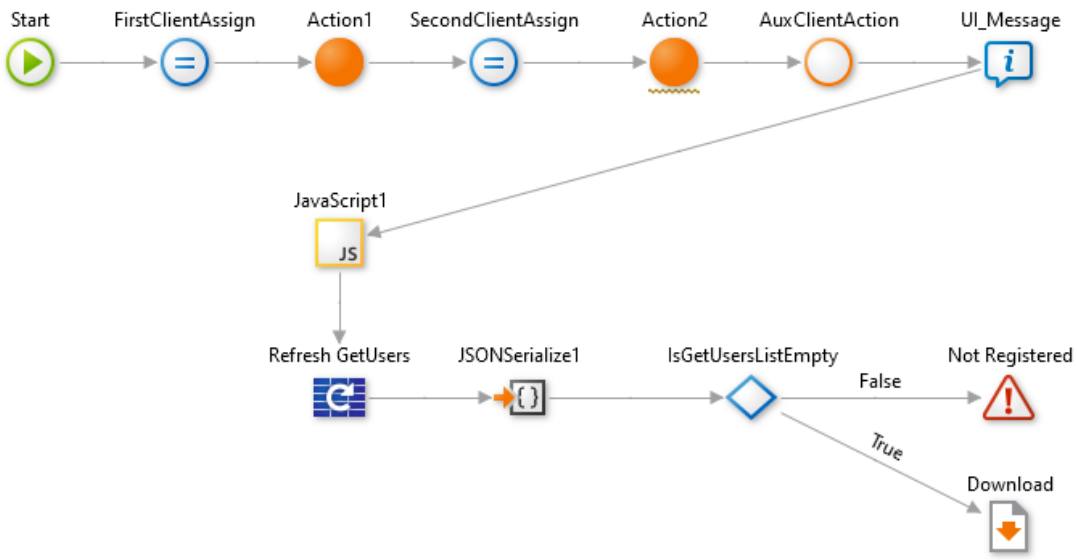


Figure 4.15 – *Service Studio* representation of *ButtonOnClick*, a complex *Client Action*.

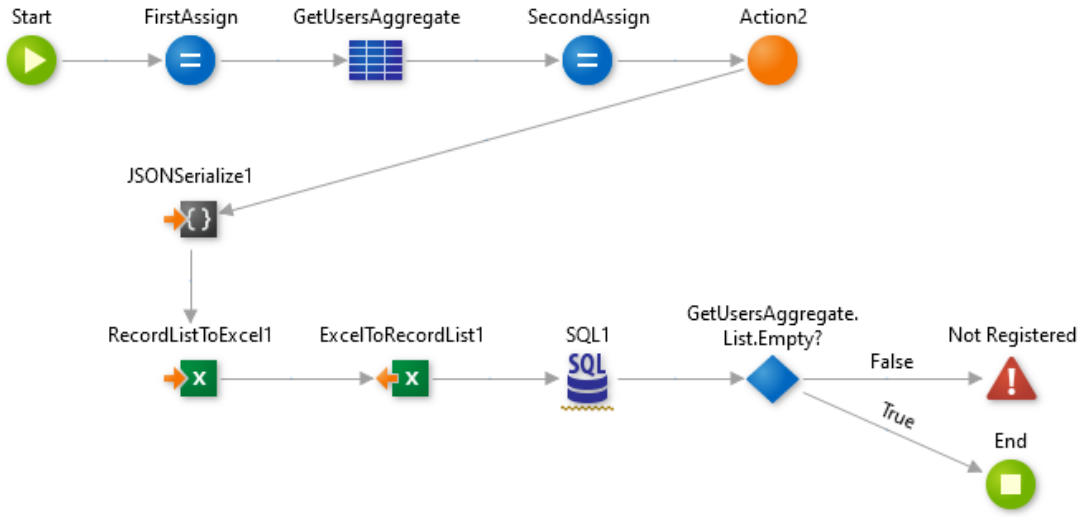


Figure 4.16 – *Service Studio*'s logical representation of *Action1*.

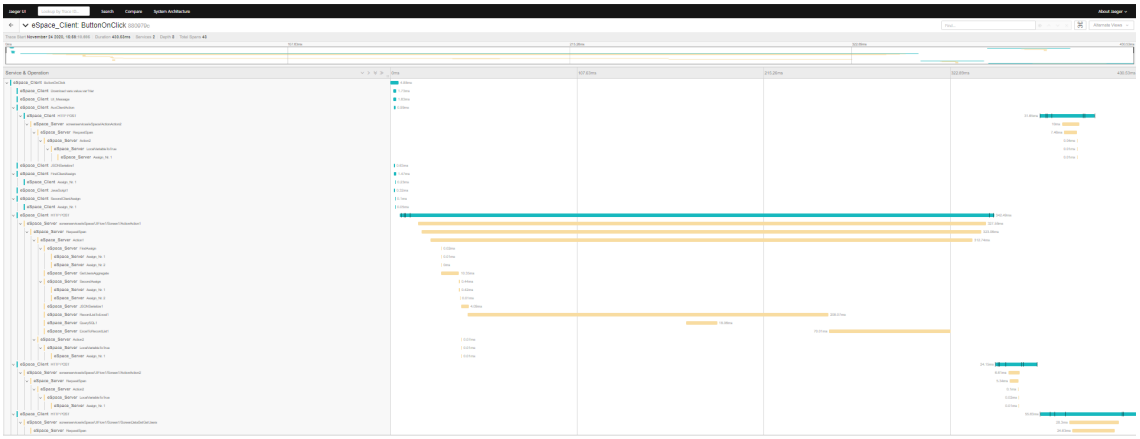


Figure 4.17 – *Jaeger* detailed presentation of a trace named *ButtonOnClick*, on service *eSpace_Client*.

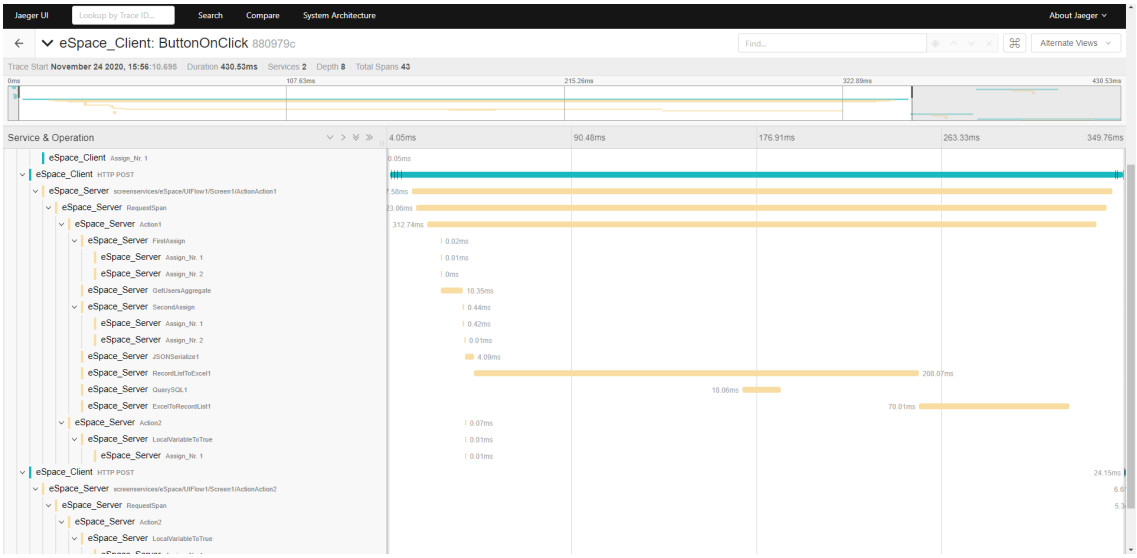


Figure 4.18 – *Jaeger* detailed presentation of a trace named *ButtonOnClick*, on service *eSpace_Client*, zoomed to show only spand related with the calling of *Action1* server action.

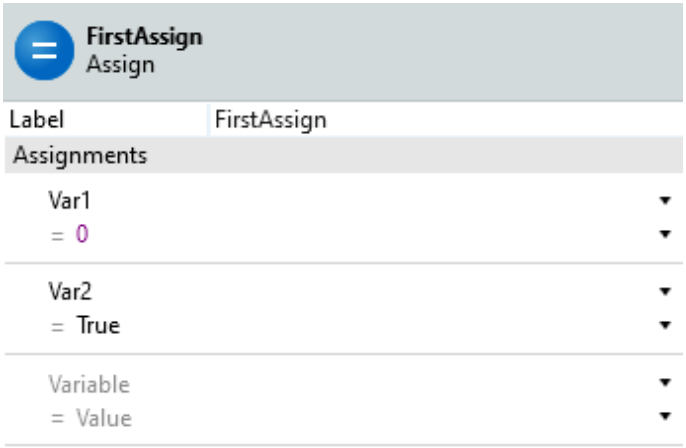
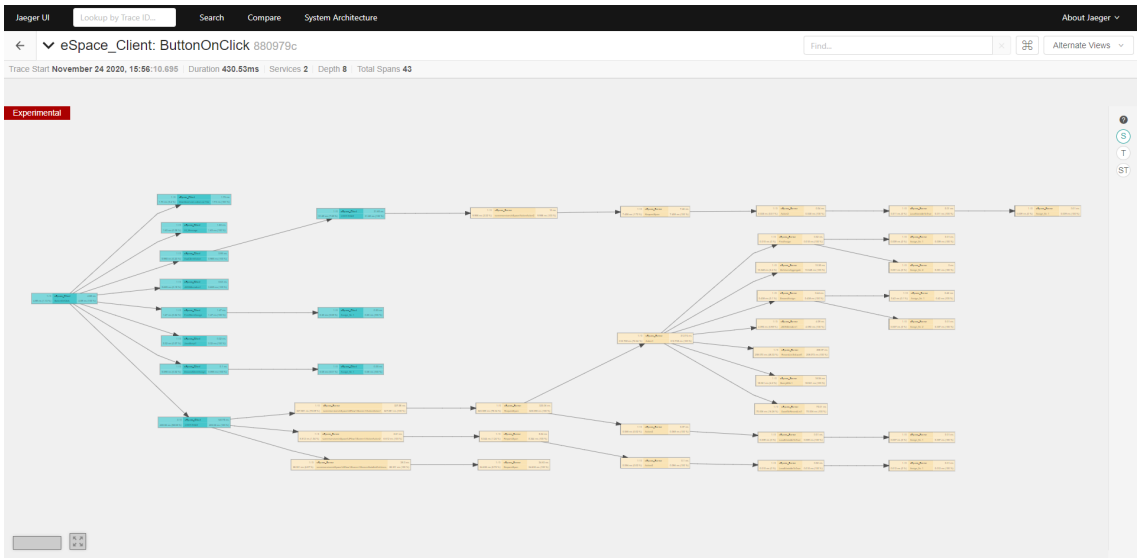


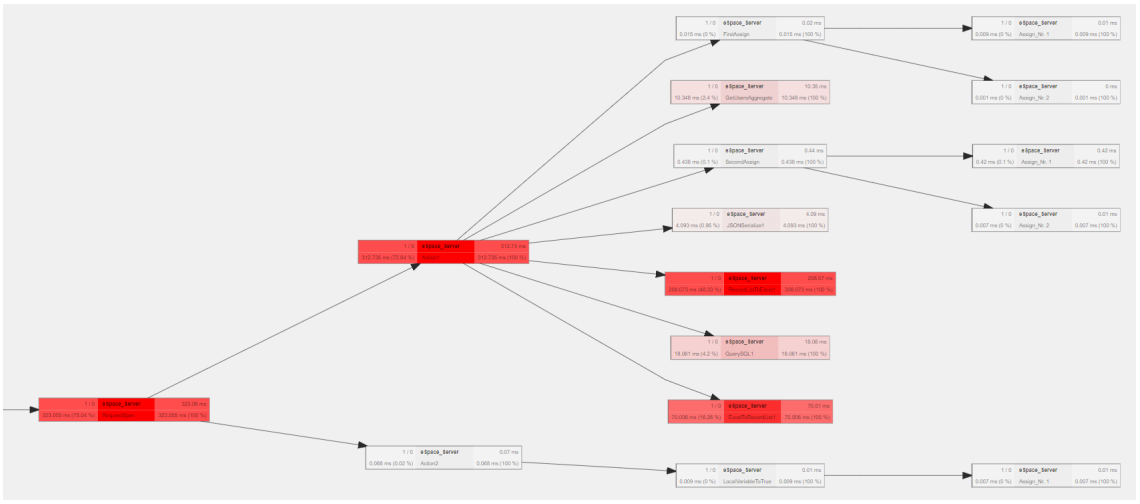
Figure 4.19 – *Service Studio*'s menu to configure an *Assign* node with two assign operations: when this node is executed, *Var1* will be set to *0* and *Var2* will be set to *True*.



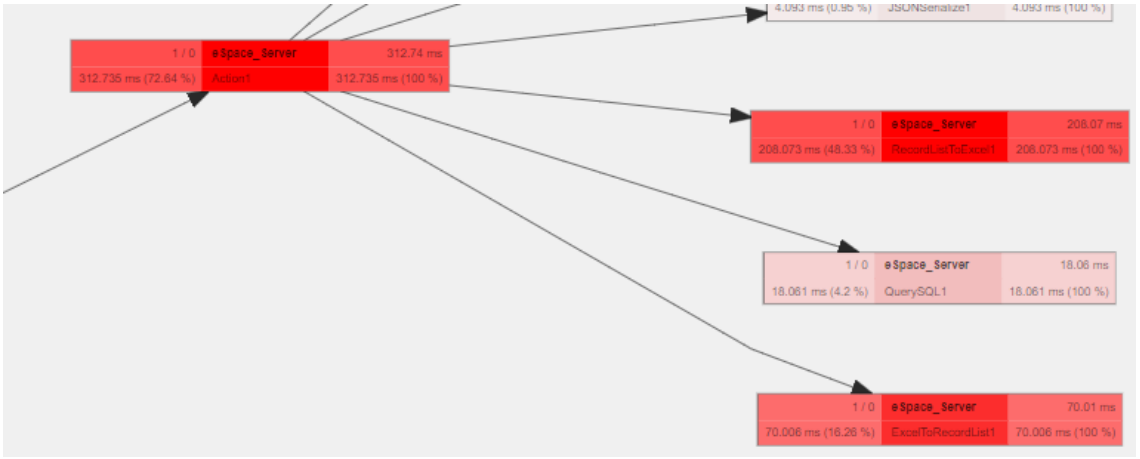
(a) Color-coded according to *Services*: blue being service *eSpace_Client* and yellow being *eSpace_Server*

(b) Color-coded according to the duration of each span: the redder the hue gets, the longer the duration of the specified span

Figure 4.20 – *Jaeger* graph presentation of a trace named *ButtonOnClick*, on service *eSpace_Client*, with two different color-codes.

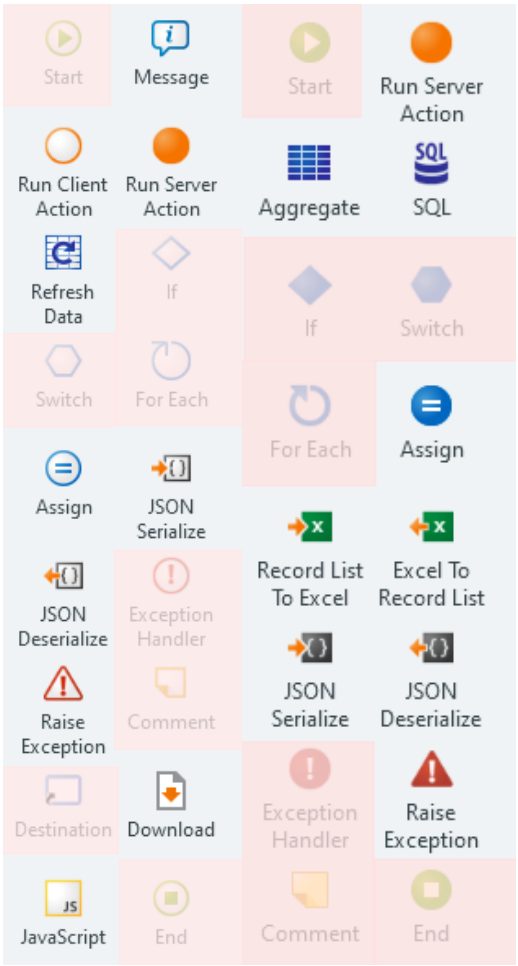


(a) Zoomed in to the view over *Action1*



(b) Zoomed in to the view over *Action1*'s most time-consuming nodes

Figure 4.21 – *Jaeger* graph presentation of a trace named *ButtonOnClick*, on service *eSpace_Client*, color-coded according to duration of each span, zoomed in to the most time-consuming region of the graph.



(a) *Client Actions* instrumentation coverage

(b) *Server Actions* instrumentation coverage

Figure 4.22 – Instrumentation coverage over the *OutSystems* language. *Nodes* not instrumented are shaded out.

Chapter 5

Implementation Details

This chapter focuses on the implementation details of the presented solution, as well as all of its features. The description of said details attempts to be as explicit and complete as possible, so that it covers all the important tasks to be done in order to replicate this work. However, it is important to note that it would be infeasible to place copies of the exact implemented code on this document. As such, most of the presented listings are summarized or presented in an abstract manner (as pseudo-code), just to convey the principal characteristics of the approach. In any case, every approach is described and explained in detail.

An important aspect to highlight is that the *OpenTelemetry* code (namely, the *.NET* part) has not been updated since the beginning of the development of this work. The library was still on its *alpha* stage, so it was prone to changes that would affect severely the structure of the code. So, in order to avoid dealing with such problems regarding this dependency, this code was left without updates. All configurations were based on the *OpenTelemetry*'s pages [49, 50, 52]. Further important pages will be mentioned throughout the chapter, when relevant.

It is important to remember the overall design of the solution and the interaction between all implemented and/or used parts:

- The application as its own running code, just as before the implementation of this solution.
- Instrumentation code is placed on the application's code, to gather tracing information. In this case, this instrumentation code is the one provided by the *OpenTelemetry*'s libraries.
- An important piece of the instrumentation technology is the Exporter: it sends the gathered information to a Collector, listening over the network. As discussed on

the next sections, the Server-side uses a *Jaeger* exporter, while the Client-side uses a *Zipkin* exporter.

- The collector acts as a part of a multifunctional service, receiving data sent by the Exporter that runs along the instrumented code, and handing it over to a service in charge of presentation and UI. As discussed on the next sections, the Server-side data is collected by *Jaeger*'s own collector, while the Client-side data is collected by a *Zipkin* collector running on the *Jaeger* image, that hands over its results to the *Jaeger* presenter.

- After (or while) running the application, the user can see the presentation of tracing data on the UI, in an organized manner, where a trace of an event performed on the application (such as the click of a button) encompasses both sides involved (maintaining the runtime context, supported by automatic context propagation), therefore presenting the results from the beggining of the event to its end, documenting all sub-events execution in the between.

5.1 Handling Dependencies

First and foremost, it is important to enable the technologies and tools used in this solution, so that they may be accessed on the *OutSystems*' platform code. The applications to be dealt with comprise two sides: Client and Server. In the *OutSystems* context, Client-side is written in JavaScript (not using *Node.JS*) and the Server-side in *C#*, and *.NET Framework*.

To enable Server-side, *C#*, dependency code, instructions on [50] were followed. Summarizing, *DLL*s were downloaded and installed on the *OutSystems* compiler's project. Then, scripts responsible for updating the *OutSystems* platform when needed were updated, so that they ensure that dependencies were made available to each new published application, by moving them to the application's files.

Regarding the Client-side, *JavaScript*, dependency code, it was not so easy to understand how such codes were already being handled by the *OutSystems* platform. As this issue was blocking the rest of the implementation process, and was not considerer crucial in the scope of the project, a workaround was put to place: all dependency code was bundled to the tracing configuration code (approached on the following section) and defined, in *Service Studio* as a *Required Script* on each instrumented screen of the *OutSystems* applications used. This way, this code is always loaded when the screen is opened, enabling access to all of the needed dependency code. The code was bundled using *Browserify* [33] and followed instructions on Daniel Wild's example [86] and *XMLHttpRequest* pluging instructions of use [53]. The dependencies contained on the code bundle were made available to all of the application's client-side code by storing them on the *window* Browser Object Model [84]:

Listing 5.1 – *JavaScript* dependencies

```
window._opentelemetry = require('@opentelemetry/api');
window._tracing = require("@opentelemetry/tracing");
window._core = require("@opentelemetry/core");
window._zipkin = require('@opentelemetry/exporter-zipkin');
window._xhr = require('@opentelemetry/plugin-xml-http-request');
window._web = require('@opentelemetry/web');
window._zone = require('@opentelemetry/context-zone');
```

5.2 Tracing Configuration Code

As mentioned above, in the context of this dissertation, applications will have both Client and Server sides, written in different programming languages. So, as it would be expected, configurations differ from one another.

On the Server-side, two important configurations need to take place: the tracing configuration itself and the placement of the needed variables in a context accessible throughout the application. First, *OpenTelemetry* code needs to be initialized and called, and the *Exporter* needs to be configured so that it sends the collected data to the *Jaeger* Collector, as well as maintaining the tracing context received in incoming requests. The designed solution placed tracing configuration code on the *Application_BeginRequest(...)* method of the *ASPNETRequestLifecycle* class. This method occurs at the beginning of the pipeline of events handling an incoming request to the application. For the purpose of this dissertation, its arguments are not relevant and, therefore, will not be discussed.

Listing 5.2 – Tracing configuration code on *Server* side

```
protected void Application_BeginRequest(Object sender, EventArgs e) {

    var serviceName = appInfo.eSpaceName + "_Server";

    this.tracerFactory = TracerFactory.Create(builder => {
        builder
            .UseJaeger(c => {
                c.ServiceName = serviceName;
                c.AgentHost = "localhost";
                c.AgentPort = 6831;
            })
            .AddRequestCollector()
            .AddDependencyCollector(config => {
                config.SetHttpFlavor = true;
            })
            .AddCollector(t => new HttpClientCollector(t));
    });

    var tracer = tracerFactory.GetTracer(serviceName);

    // ...
```

```
22
23      }
```

This block of code starts by defining the service name (that will appear on *Jaeger*) as the name of the *OutSystems* module, concatenated with *"_Server"*. Then, it builds a *TracerFactory*, an object used to retrieve *Tracer* objects, that all follow the configurations placed on the listing above. This means that all tracers created via this TracerFactory will use *Jaeger*'s exporter to send the collected data to the specified address and port, under the specified Service Name. It also initalizes the needed collectors in order for context-propagation automation to work.

A Tracer is an object used to create *Spans*. In tracing, a Span represents an activity or a portion of it. It is characterized by the Service on which it executes, by the time it takes to execute and can be complemented with tags, conveying additional information. If a certain portion of code is to be instrumented, a Span should start just before the beggining of said code and end right after the end of it. In order to do so, a *StartSpan* is called on the tracer, retrieving an object that represent a Span. This new object has its own *EndSpan* method that will end the span's tracing information gathering and dumping all collected data to the exporter. A first example of this is the *RequestSpan*, a span created by this solution at the beggining of every request (right after the tracing configuration code) and ended at the end of the request (at the end of the *Application_EndRequest* method of this same class).

Listing 5.3 – Creation and ending of the *RequestSpan*

```
1   protected void Application_BeginRequest(Object sender, EventArgs e) {
2
3       // ...
4
5       var span = tracer.StartSpan("RequestSpan", tracer.CurrentSpan, SpanKind.Server);
6
7       // ...
8   }
9
10  protected void Application_EndRequest(Object sender, EventArgs e) {
11
12      App.OsContext.OTRequestSpan.End();
13
14      // ...
15  }
```

This starts a span with a "RequestSpan" label (the name that will appear on *Jaeger* to mention it), uses the current span in execution as its parent and that executes on the Server-side of the application. The reason to have this span representing the entirety of the request's execution is not only informational but it is also to allow all subsequent spans that happen in the context of this request to use it as their parent (so that those new spans appear on *Jaeger* as sub-spans of this bigger *RequestSpan*). In order to provide

access to this span, as well as the tracer throughout the generated application's code, both variable will be placed on *OsContext*, a mechanism created by *OutSystems* that already stores numerous others variables related with the context of the requests being handled by the running application and is available throughout its code. To do so, the variables defined on the two previous examples are stored on said *OsContext*:

Listing 5.4 – Placement of the variables on the *OsContext*

```
protected void Application_BeginRequest(Object sender, EventArgs e) {

    // ...

    appInfo.OsContext.OTRequestSpan = span;
    appInfo.OsContext.OTTracer = tracer;

    // ...
}
```

Now, on other regions of the application code (as it will be presented on the following section), both variables will be available from a variable holding *OsContext*, normally called *HeContext* throughout the *OutSystems* platform code.

Client-side configuration is different, as it would be expected, but follows the same generic principles. As stated on the previous section, this code is entirely placed (with the present solution's implementation) as a *Required Script* to each Screen to instrument in an *OutSystems* application, along with the dependency code.

Listing 5.5 – *JavaScript* tracing configuration code

```
window._opentelemetry = require('@opentelemetry/api');
window._tracing = require("@opentelemetry/tracing");
window._core = require("@opentelemetry/core");
window._zipkin = require('@opentelemetry/exporter-zipkin');
window._xhr = require('@opentelemetry/plugin-xml-http-request');
window._web = require('@opentelemetry/web');
window._zone = require('@opentelemetry/context-zone');

const providerWithZone = new window._web.WebTracerProvider({
  plugins: [
    new window._xhr.XMLHttpRequestPlugin({
      propagateTraceHeaderCorsUrls: ['http://localhost:8090']
    })
  ]
});

const options = {
  url: 'http://localhost:8888/http://host.docker.internal:9411/api/v2/spans',
  serviceName: 'eSpace_Client'
}
window.zipkinExporter = new window._zipkin.ZipkinExporter(options);

```

```
23    providerWithZone.addSpanProcessor(new window._tracing.BatchSpanProcessor(window.
          ↪ zipkinExporter));
24
25    // Changing default contextManager to use ZoneContextManager - supports asynchronous
          ↪ operations
26    providerWithZone.register({
27      contextManager: new window._zone.ZoneContextManager(),
28    });
29
30    window.zipkinTracer = providerWithZone.getTracer();
```

As described on the previous section of this chapter and seen on the first lines of this example, dependency code is stored on the *window* object. After that, a *WebTracerProvider* is created and configured with the *XMLHttpRequestPlugin* [53] (using the default settings, described on the reference site), that takes care of propagating trace headers over the network, a vital feature for the automation of context-propagation. Then, following the tutorial by Daniel Wild [85] and its examples [86], *options* were set to configure *ZipkinExporter*. These options make sure that the collected data is sent through the *CORS* proxy, as the URL states so. Then, a *BatchSpanProcessor* is added to the *TracerProvider*. This component is responsible for processing the collected tracing data and hand it over to the *Exporter*, that sends it to the desired *Collector*, over the network. There are several kinds of processors, but this one, being a *BatchSpanProcessor* not only processes the data but also waits some time, before sending it to the *Exporter*. This minimizes the amount of communications over the network, as during the waiting period additional collected data will be added to the "batch" that will be sent, resulting in sending one large batch from time to time, instead of several small batches instantaneously. This minimizes performance impact. Finally, having all the configurations in place, a *Tracer* can be obtained, and that will be used to create *Spans*, as it will be discussed on the following section.

OpenTelemetry only supports *Jaeger*'s exporter on *Node.JS* applications. Luckily, *Jaeger* presenter and UI also work with *Zipkin*'s collector, as they are similar services. That is why the *ZipkinExporter* is used and not a *Jaeger* one. However, all information will appear on *Jaeger*'s UI.

5.3 Placing Instrumentation Code

Now that the configurations are in place, instrumentation code will ensure that information is gathered and exported, so that it can be externally collected and presented. The caveat is that, in this particular case, instrumentation will not be placed on the code that will be executed by an application. Rather, code will be placed on the *OutSystems* compiler so that, when it generates a code for some Node (or Action, for example), it also generates the respective instrumentation code next to it.

In the *OutSystems* platform, the high-level code to be run by final applications can be generated in a myriad of ways. But, summarizing, the general behaviour is of some object dumping lines of code to a file, or files. These generated files are the code of the application to be run. The compiler codes can be summed up as a collection of zones that determine how and where will each *OutSystems* language entity (Actions, Nodes, etc.) be translated to high-level code, i.e., how and where will *C#* and *JavaScript* code be written following the logic designed by the user on *Service Studio*.

On most of the compiler code files, a *Language* class is accessible. This class contains methods to be called by the compiler, when it needs to generate some portion of code. Say the compiler needs to generate *C#* code to represent an *OutSystems* assign: a possible method on the *Language* class to be called could be:

Listing 5.6 – Demo *Assign* code generating method

```
1  public virtual Statement DumpAssign (String variableToBeAssigned, String
       ↪ valueToAssign) {
2      return Statement.FromString("var␣" variableToBeAssigned + "␣=␣" + valueToAssign +
           ↪ ";");
3  }
```

This method generates a Statement object. The compiler dumps the content of this object to the generated code file, at the correct location. Following this approach, methods were created to dump instrumentation code to the generated files, whenever called. Some examples are:

Listing 5.7 – New Language methods to dump instrumentation code

```
1  public virtual Statement DumpStartSpanStatement(String spanName, String label, String
       ↪  parentSpan) {
2      return Statement.FromString("var␣" + spanName + "␣=␣tracer.StartSpan(\"" + label +
           ↪  "\",␣" + parentSpan + ");");
3  }
4
5  public virtual Statement DumpStartSpanStatement(String spanName, String label) {
6      return Statement.FromString("var␣" + spanName + "␣=␣tracer.StartSpan(\"" + label +
           ↪  "\");");
7  }
8
9  public virtual Statement DumpEndSpanStatement(String spanName) {
10     return Statement.FromString(spanName + ".End();");
11 }
12
13 public virtual Statement DumpSpanAttributeStatement(String spanName, String attribute,
       ↪  String value) {
14     return Statement.FromString(spanName + ".SetAttribute(\"" + attribute + "\",␣\"" +
           ↪  value + "\");");
15 }
```

All these methods follow the same approach of the previous example. As explained on the previous section, Spans can have parents, so that their execution is presented as a sub-event of another span. This represents an *Assign* that is happening during the execution of an *Action*, for example. As such, language methods were created to allow for span creation with or without parent spans. As one can easily observe, *OpenTelemetry* code is very straightforward. Finally, having these methods set, it is easy to incorporate the generation of instrumentation code along with the application code. One should simply call the methods needed to start and end a span at the compiler code region where some *OutSystems* language element (also called a *Node*) is being translated. One example would be the generation of an *Assign*:

Listing 5.8 – Calling Language methods to place instrumentation code

```
1   protected readonly Language lang;
2
3   internal AbstractActionNodeCodeGenerator(Language language) {
4       lang = language;
5   }
6
7   public virtual IEnumerable<Statement> Visit(Nodes.Assign node) {
8
9       var spanName = node.Label + "Span";
10      yield return lang.DumpStartSpanStatement(spanName, node.Label, "actionSpan");
11
12      // Code needed to generate the Action's code would appear here
13
14      yield return lang.DumpEndSpanStatement(spanName);
15  }
```

This code would make sure that, to every *Assign* node translated to high-level code, there would be instrumentation creating a span that would represent it. To explain in detail the language method's arguments, according to the example above:

- The first is the name to the local variable that will store the span on the generated high-level code method. This variable would have the name of the *OutSystems* node being translated concatenated with "Span"

- The second is the label to the span, that *Jaeger* will present on its UI. It takes the name of the *OutSystems* node being translated.

- The third is the name of the variable storing the span that should act as parent to the new generated span. The example shows the instrumentation of an *Assign*, all assigns happen within the context of an *Action* so this solution always defines the name of the variable holding the span that represents the *Action* as "actionSpan" so that all child nodes can access it easily and use it as their parent.

On compiler code sections where such methods were not available the instructions to generate instrumentation code were placed directly. A *Text Writer* object was called, to generate the instrumentation code:

Listing 5.9 – Calling a *TextWriter* object to generate instrumentation code

```
1  public override void DumpAction (TextWriter writer, ...) {
2      var actionName = Action.GetName();
3      writer.WriteLine("var␣" + actionName + "Span␣=␣heContext.OTTracer.StartSpan(\"" +
           ↪ actionName + "\");");
4
5      // Code needed to generate the Action's code would appear here
6
7      writer.WriteLine(actionName + "Span.End();");
8  }
```

This way, if the Action's name, as designed by the user on *Service Studio* was "SimpleAction", the generated code of this action would have a "SimpleActionSpan" local variable, created by starting a span through the tracer stored in context. This span would appear on *Jaeger* as "SimpleAction", as the argument to the *StartSpan* method is the label to the associated with the span. On the end of the execution of the action's code, the Span would end. Notice that the tracer is brought from the context variable discussed on the previous section.

Both examples were in *C#* as they refer to the generation of a Server-side Action, but on the Client-side the behaviour is the same, only adjusting for the strings passed as arguments to the Writer. On the Client-side, the language would be *JavaScript* and context would be stored and accessible on the *window* object.

5.4 Building the *OutSystems* Platform

Modifications were made to the *OutSystems* platform's code. As such, in order for those modifications to take effect, it is obviously needed to compile said code, by Building the solution that holds the code to the *OutSystems* platform compiler. This section would otherwise appear as too simple to point out, yet it is important to remember that for the services responsible for generating applications from what the developer designed in low-code, it is crucial to restart all of them, so that they now execute with their new code, the one that includes the modifications placed. To ease this task, the *OutSystems* platform has a script that, among other actions, restarts all platform services in order for them to include new changes: that script is called *UpdateDBG*.

5.5 Running *Docker* Images

In order for the context-propagation automation to work, a *Cross-origin resource sharing (CORS)* proxy is needed. This might be an issue, as it leads to an extra service running

and consuming resources, but the solving of this issue was not in the scope of this project. The proxy used was in the form of a *Docker* image:

Listing 5.10 – Command to run *CORS* proxy

```
1   docker run -p 8888:3000 psimonov/cors-anywhere
```

As long as it is running, this image does not need any further configuration whatsoever.

Another massively important image to run is the one containing *Jaeger* in its two final components in action: the *Collector* and the UI. The Collector is the one responsible for listening at the configured address for information sent by the instrumentation code (and its Exporter). It then delivers this information to the UI service, responsible for presenting it in the manner shown on the previous chapter 4, so that the user can consult it. As the Client-side is configured to use the *Zipkin* exporter, this *Jaeger* image is configured to include the *Zipkin* collector as well. These interact well with each other, and perform the handover of information automatically, so that the entirety of tracing data can be present on *Jaeger*'s UI. The Docker image that comprises all of this configuration and services can be run by executing:

Listing 5.11 – Command to run *Jaeger*'s docker image

```
1   docker run --rm -d --name jaegerZipkin -e COLLECTOR_ZIPKIN_HTTP_PORT=9411 -p
        ↪ 5775:5775/udp -p 6831:6831/udp -p 6832:6832/udp -p 5778:5778 -p 16686:16686 -p
        ↪  14268:14268 -p 9411:9411 jaegertracing/all-in-one:latest
```

As it can be observed, the Docker image shall be named *jaegerZipkin*. Both Collector Services start running and do not need any further configuration: they are ready to receive tracing data and hand it over the presenters on the UI. Instructions on how to access the UI page are described on the following section.

5.6 Publishing and Using the Application

Finally, after all is configured and running, it is reached the point in which the end-user - i.e., the *OutSystems* developer - interacts with the system as he normally would. With *Service Studio* the user develops a low-code application. When it is complete, the user clicks the *1-Click Publish* button, starting the automatic process of Uploading the new application to the *Platform* which will run it, Compiling the code (generating high-level code, following the instruction that the developer programmed using low-code) and finally Deploying the application, allowing it to be executed. To the user, this process stayed precisely the same but, as described above, the *OutSystems* Compiler was altered to place instrumentation code on the high-level generated code. As such, the application ran will be instrumented and exporting tracing information to the configured collector's address. Moreover, said data could be accessed on the collector's presenter UI: in this case,

and if the above configurations were followed, *Jaeger* will present its UI on *localhost:16686*. This page will give access to all the features described on the previous chapter 4.

The running application will continue to export tracing data to its collector. As such, the amount information on the UI page will continue to increase, and data about the most recent runtime activity can be consulted at any time.

Chapter 6

Validation

The present chapter refers to the validation approach that assessed the developed solution's performance overhead and users' satisfaction. Both will be approached on the following sections.

6.1 Users' Satisfaction

Bringing back the context of the problem of this dissertation, the goal was to bridge the gap between the levels of abstraction on the development and monitoring process. This being said, it is important to assess that the information presented on this monitoring solution relates with the information a real user sees on *Service Studio*, the *OutSystems*' IDE, while developing. I.e., if the information provided matches the low-code abstraction, rather than the generated high-code, not visible on the development process. This match between the levels of abstraction on the monitoring and development process will be henceforth described as correctness, as a correct solution to the problem approached would be one that presents information exactly on the same level of abstraction as the low-code one.

At first, this correctness was verified through manual testing of the implemented system, by the author. In the development process, while adding instrumentation to each *OutSystems* node, applications that used that node were deployed to ensure that the instrumentation was collecting information and presenting it as expected. For each node, several tests were conducted to ensure that information regarding an instrumented node was always collected, as well as it was accompanied by the correct information (e.g.: on an *Assign*, tags must show the correct variable and value involved in the assign). Following this process, every time the set of instrumented nodes increased in size, an application would be run to test the set entirely: this application called every instrumented node,

so that the interaction of the nodes' instrumentation could be tested, in order to ensure that none interfered with any other. This method presents no quantitative results, besides proving that each instrumented node has data being collected by the solution and presented on *Jaeger*. This can be confirmed on the demo examples in chapter 4.

At the end of the development process, a set of ten real users were inquired to assess their overall satisfaction about the solution. This assessment was done via individual interviews with each user: in each, a set of three scenarios were shown, i.e., three traces of execution, as presented on the solution proposed in this thesis.

6.1.1 User Testing Scenarios

The conducted test began with a brief explanation over the approached problem and goal of the test itself, and the dissertation as a whole. To put the users in context, they were presented with a sample action on *Service Studio*, particularly, the action described on section 4.4 and presented on figures 6.1 and 6.2. The users were asked not to follow the logical flow in detail, as that was just an example.

Followingly, *Jaeger*'s trace representation of a call to said action was presented, as depicted on figure 6.3. This was used to give the users the main concepts of the UI they would experience in the three testing scenarios. The explanation on *Jaeger*'s UI was brief and users were alerted that the author did not have any intervention on its design and/or implementation: it was used merely as an external presentation tool, to show the collected low-code tracing data (that were, indeed, the product of the author's work). Users were alerted aswell that the focus of the test was to evaluate the accordance of said low-code tracing information to the degree of information (and level of abstraction) present on the *OutSystems* environment and language. In fact, users were only appointed to the fact that on the left of the *Jaeger*'s trace exhibition window, they could see a list of the executed calls, in a tree (with hierarchization) and on the right they could find a visual representation of the execution durations (multiple bars, longer bars mean a longer duration in executing the corresponding call).

It is also important to point out an improvement made to the solution, when designing the user tests: the author recurred to a test subject (who would not enter on the final tests) to assess the overall of the tests themselves. This subject made a valuable suggestion: the need to have the node type, next to the node's label on the tracing visualization. Moreover, the name of the span that started activity on the server side (*RequestSpan*) was also appointed as confusing: it was changed to a more descriptive "*New Request to Server*". This changes are reflected on figure 6.4c and other figures that depict tracing data visualization, on this chapter.

It is important to make clear that users were never presented with the *OutSystems* code that generated the actions they analised on traces. All users were familiar with the *OutSystems* language and had practice developing with it, so the exercise was to infer, looking at a trace, what could be the *OutSystems* code that lead to it. This approach was

used to test the validity of the information presented, and the results were calculated based on the percentage of correct deductions by users, against the known (to the author, who was conducting the tests) code.

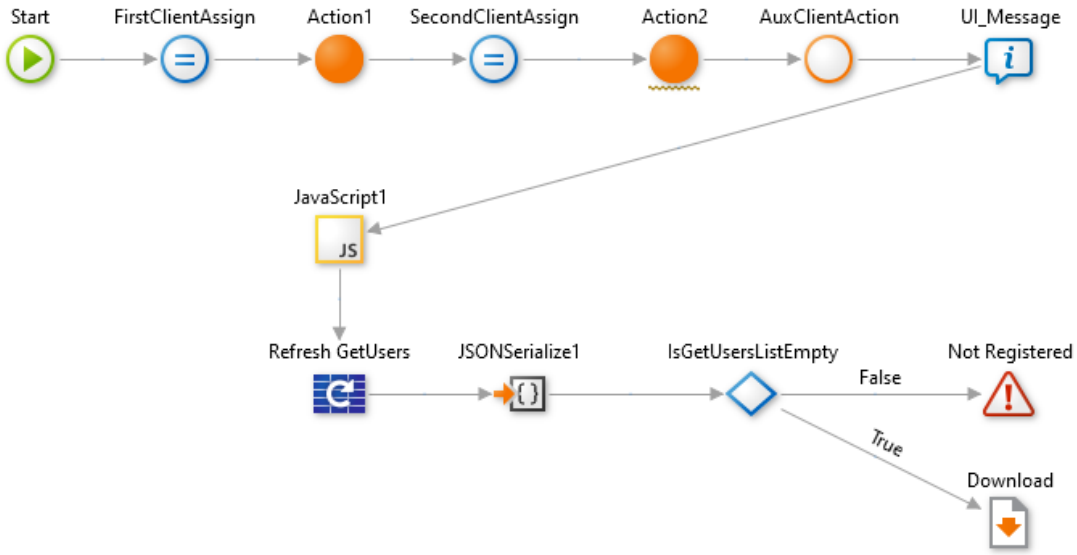


Figure 6.1 – *Service Studio* representation of *ButtonOnClick*, a complex *Client Action*.

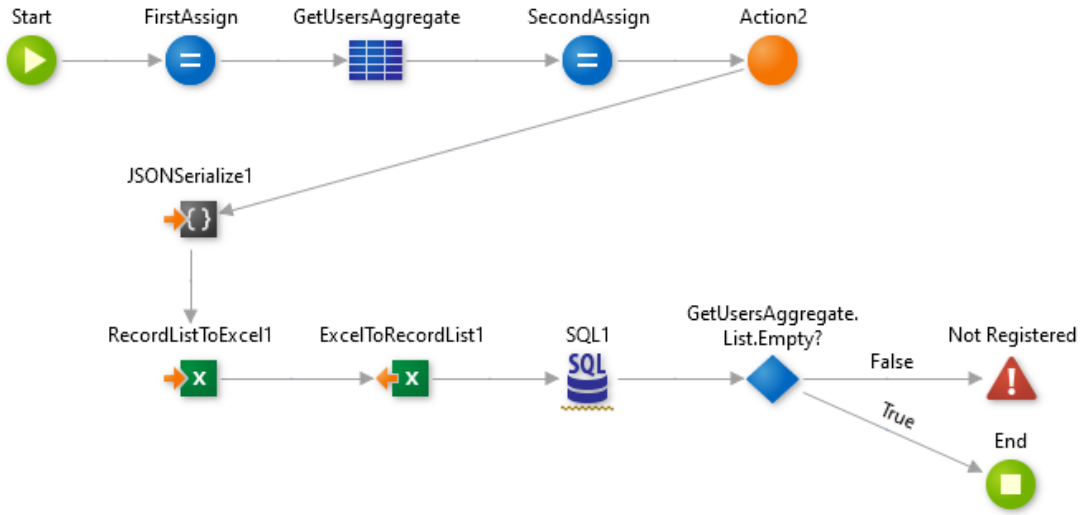


Figure 6.2 – *Service Studio*'s logical representation of *Action1*.

6.1.1.1 First Test Scenario - *CompletePurchase*

On the first scenario, users were presented with the trace depicted on 6.4c. This trace corresponded to a simple *Client Action*, with a few nodes and a call to a *Server Action*. The actual *OutSystems* code on those actions can be found, correspondingly, on figures 6.4a

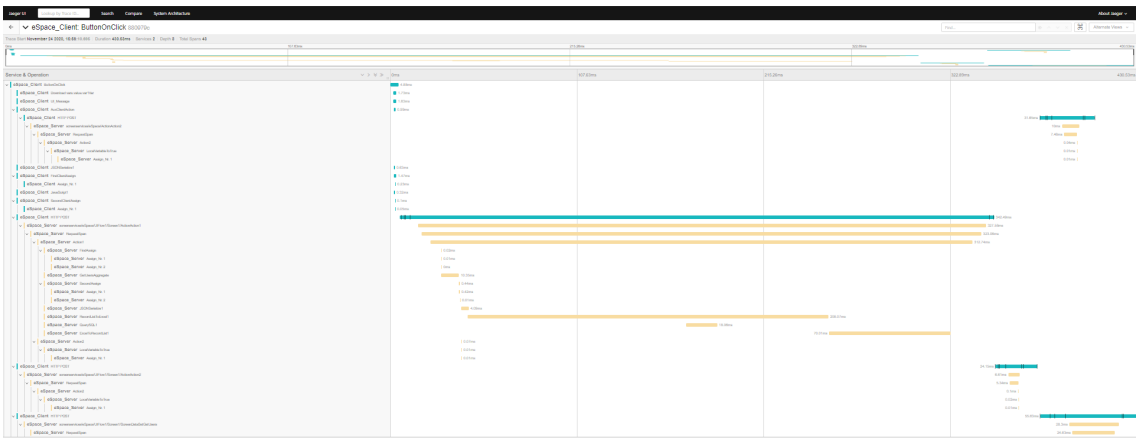


Figure 6.3 – *Jaeger* detailed presentation of a trace named *ButtonOnClick*, on service *eSpace_Client*.

and 6.4b. Being the first scenario, its simple goal was to assess the users' capability of infering the correct *OutSystems* flow that could be at the origin of such trace. This task was completed, across all ten users, with a success rate of 81,43%. Throughout all users, the hardest call to identify was the call to the server, as it encompassed many lines on *Jaeger*, i.e., multiple spans in from client to the server-side, instead of what all users were expection: a single span representing the call to the server itself (without references to *HTTP POST* or a specific URI). This failure rate was expected, as this difficulty had already been appointed as a caveat of the current solution.

In this scenario, another metric was retrieved: the success rate of identifying the total duration of the presented trace. Such rate achieved a result of 100%.

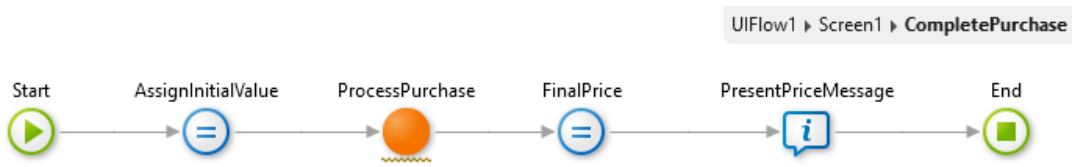
6.1.1.2 Second Test Scenario - *ProcessBillingInfo*

On the second scenario, users were presented with a trace, depicted on figure 6.5b, that contained the raise of an exception on the client flow. The actual code that lead to it is depicted on figure 6.5a.

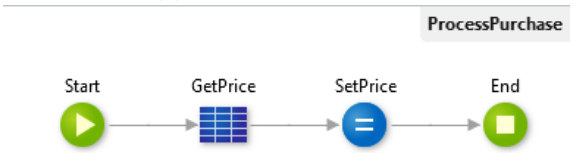
As on the first scenario, users were asked to identify what was the *OutSystems* flow, achieving a global success rate of 96.67%. Only one user (10%) failed to understand that an exception was being raised, as he believed that the trace showed one possible outcome (i.e., that an exception could be raised) rather that it actually happened.

Regarding the analysis of the exception raised, 70% of users successfully identified the exception's message (and verbally confirmed that that was a valuable information), 80% successfully identified the time spent on raising the actual exception and 70% correctly understood how much time it passed since the beginning of the action until the raising of the exception.

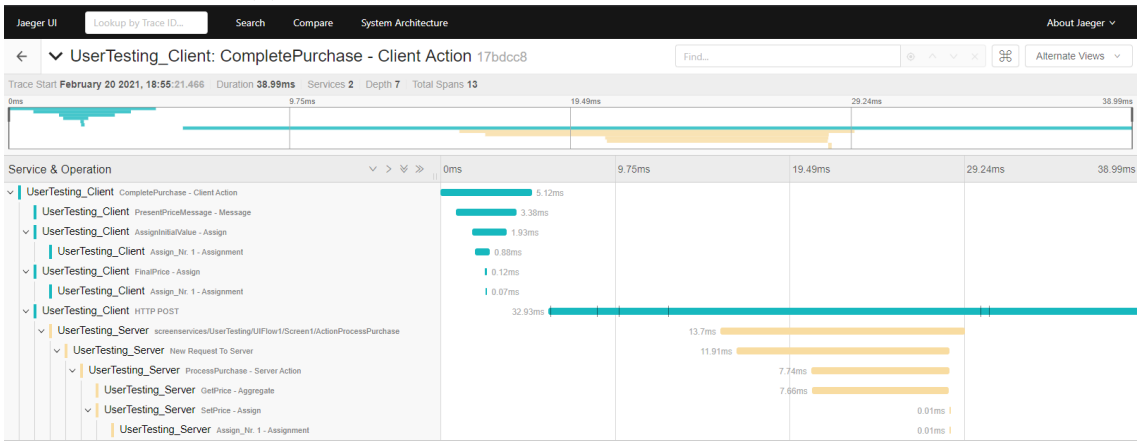
Again, the success rate of identifying the total duration of the presented trace was retrieved, with a result of 100%.



(a) Client Action flow on *Service Studio*



(b) Server Action flow on *Service Studio*



(c) Tracing representation on *Jaeger*

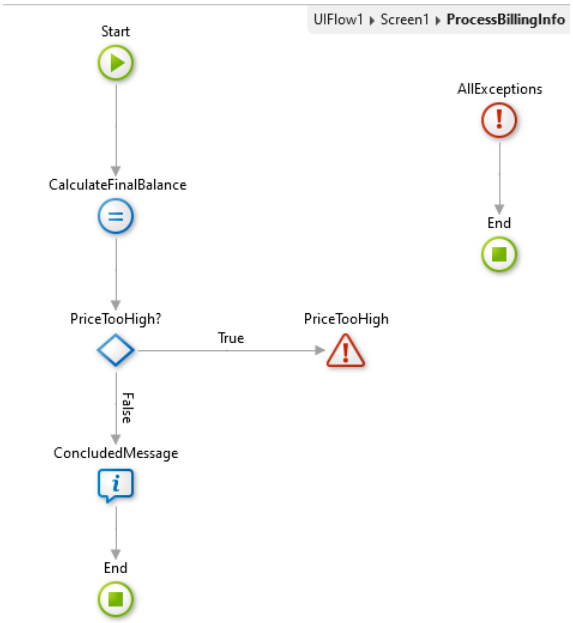
Figure 6.4 – Action flows and tracing representation of the first scenario's action.

6.1.1.3 Third Test Scenario - *BackupBillingData*

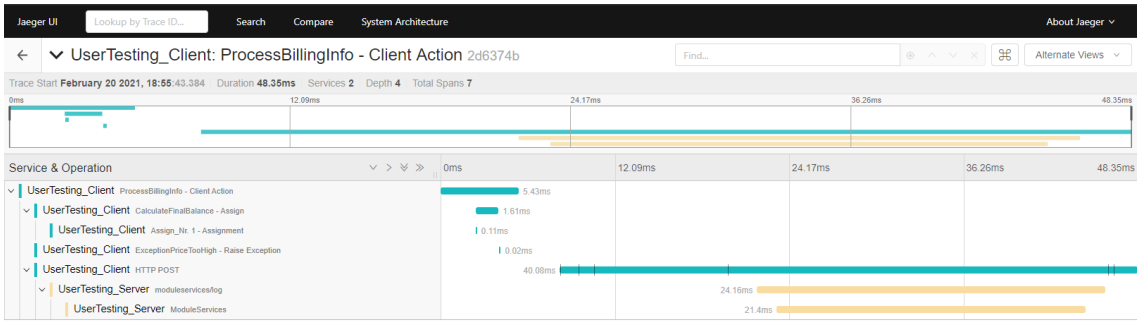
On the third and last scenario, the presented trace, depicted on figure 6.6c, referred to a client action, depicted on figure 6.6a, comprised solely of a call to a server action, depicted on figure 6.6b.

As with the other two scenarios, users were asked to identify the *OutSystems* flow that could lead to such a trace, with a success rate of 98%. Also as with the other two scenarios, users identified the total duration of the action with a success rate of 100%.

In this trace, a span was abnormaly longer than all the others, as it was related to an *OutSystems* node that performs a computationally heave operation and, as such, takes longer to execute: *Record List to Excel*. Users were asked to identified any problematic nodes, as if they were presented with this problematic trace by an hypothetical client, who was trying to figure out why was such an action so slow. 100% of users successfully identified that *Record List to Excel* node as the most problematic, as it was the longest, and 90% of these same users did so immediately by looking at the summary graph (on the right) and realizing that that node was represented by a longer bar, meaning a longer



(a) Action flow on *Service Studio*



(b) Tracing representation on *Jaeger*

Figure 6.5 – Action flow and tracing representation of the second scenario's action.

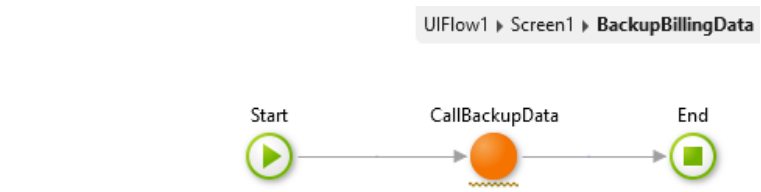
execution time (only one user reached such a conclusion by comparing the numeric values presented on the UI).

6.1.2 User Satisfaction Survey

At the end of the three test scenarios, and after providing verbal feedback, users were asked to fill an online form.

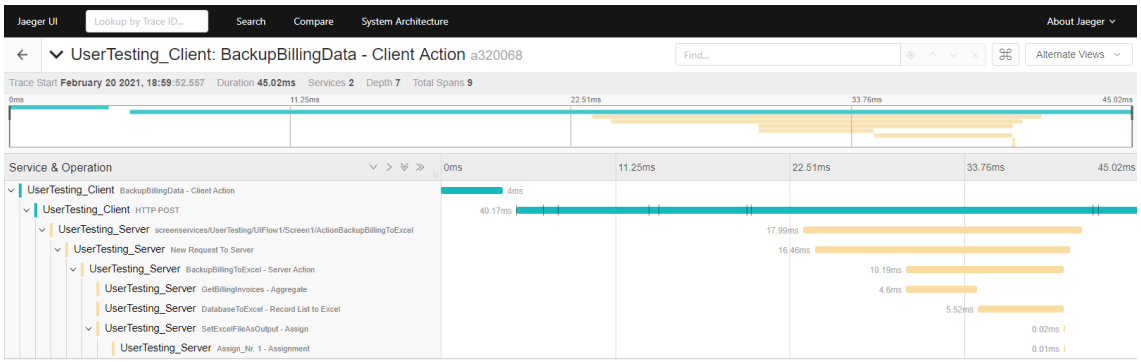
This form began by asking the users the ten questions that comprise the System Usability Scale (SUS) [83], on a scale of 1 to 5. The questions and results are shown on table 6.1.

Regarding this SUS, it is important to note that odd numbered questions are good (higher value is better), but even numbered questions are bad (lower value is better). The Scale provides a specific formula to calculate the final usability score [79], and the current solution achieved an average results (across all ten users) of 90,75 out of 100,



(a) Client Action flow on *Service Studio*

(b) Server Action flow on *Service Studio*



(c) Tracing representation on *Jaeger*

Figure 6.6 – Action flows and tracing representation of the third scenario's action.

with a Standard Deviation of 10. Given that the average result is 68, the System Usability Scale classifies this result as Excellent.

After this first part of the survey, users were asked to answer four more questions.

The first corresponded to a *Customer Satisfaction Score (CSAT)* [70], asking *"How would you rate your overall satisfaction with this low-code runtime tracing solution?"*. Users answered varying from 1 (Very unsatisfied) and 5 (Very satisfied), providing a result based on the percentage of satisfied users (ones that rated 4 or 5) divided by the number of survey responses. The results are depicted on figure 6.7, having the current solution achieved a total of 100% of satisfied users.

After that, a question corresponding to a *Net Promoter Score (NPS)* [73], asking *"How likely is it that you would recommend this low-code runtime tracing solution to a friend or colleague?"*. Users answered varying from 0 (Not at all likely) to 10 (Extremely likely), providing a result based on the difference of the percentage of *Promoters* (users that scored 9 or 10) and *Detractors* (users that scored 0 through 6). Users that score 7 or 8 are ignored, as *Passives*. Results are depicted on figure 6.8. Given the calculation (helped by [71]), the current solution achieved a *Net Promoter Score* of 90%.

Table 6.1 – Average and Standard Deviation of the System Usability Scale (SUS) results.

Question	Average	Standard Deviation
I think that I would like to use this system frequently.	4,9	0,32
I found the system unnecessarily complex.	1,4	0,52
I thought the system was easy to use.	4,8	0,42
I think that I would need the support of a technical person to be able to use this system.	1,5	0,97
I found the various functions in this system were well integrated.	4,4	0,52
I thought there was too much inconsistency in this system.	1,3	0,48
I would imagine that most people would learn to use this system very quickly.	4,7	0,95
I found the system very cumbersome to use.	1,6	1,26
I felt very confident using the system.	4,8	0,42
I needed to learn a lot of things before I could get going with this system.	1,5	0,71

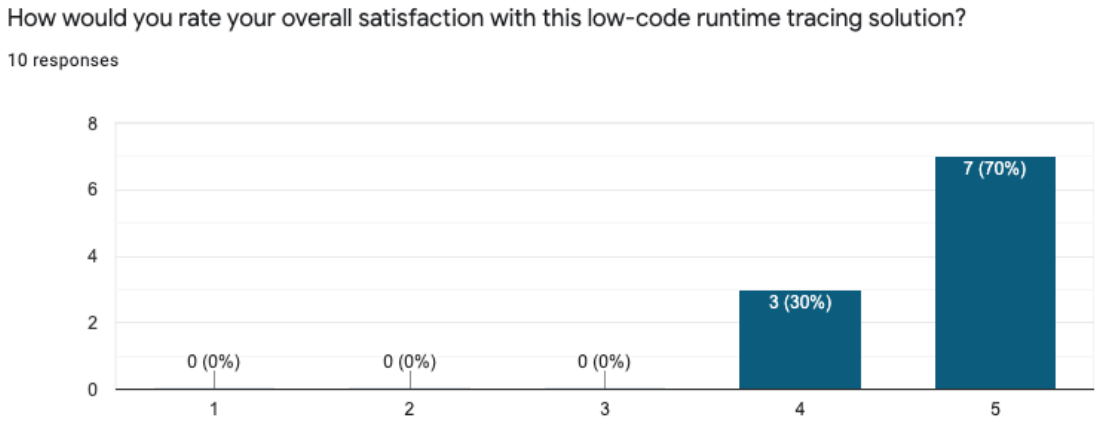


Figure 6.7 – *Customer Satisfaction Score (CSAT)* results.

Lastly, two custom questions - made by the author - were posed, with scores ranging from 1 to 5:

- *"Can you successfully identify low-code elements on the tracing tool?"* - rating 4,9 out of 5 (results depicted on figure 6.9a);
- *"How easy was it to identify the low-code nodes on the tracing tool?"* - rating 4,6 out of 5 (results depicted on figure 6.9b).

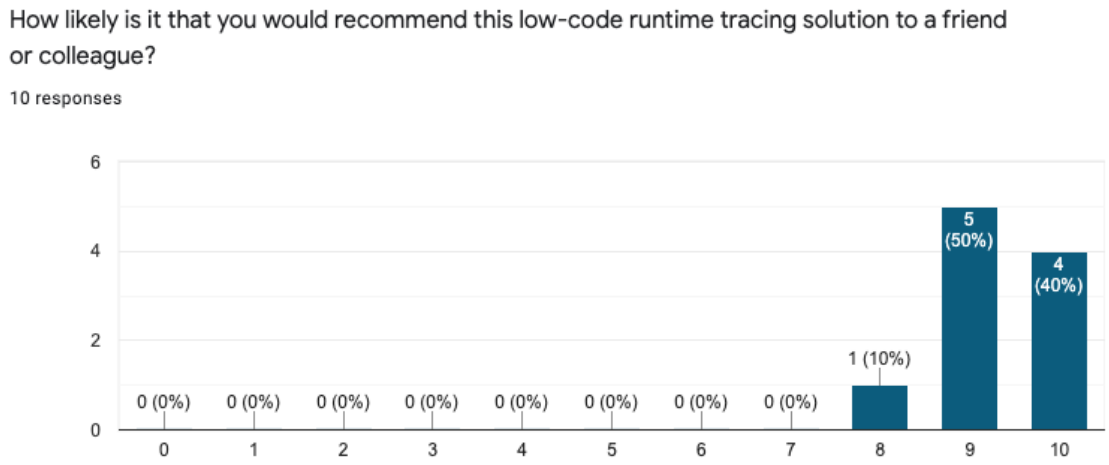


Figure 6.8 – *Net Promoter Score (NPS)* results.

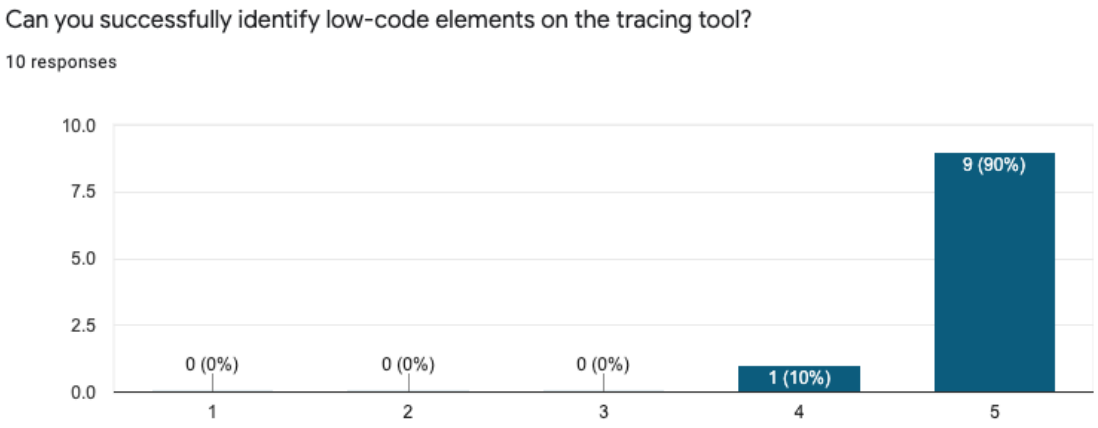
These last two questions were made in order to assess the effectiveness and efficiency (respectively) of the current solution, and both presented good results.

6.1.3 User Feedback Comments

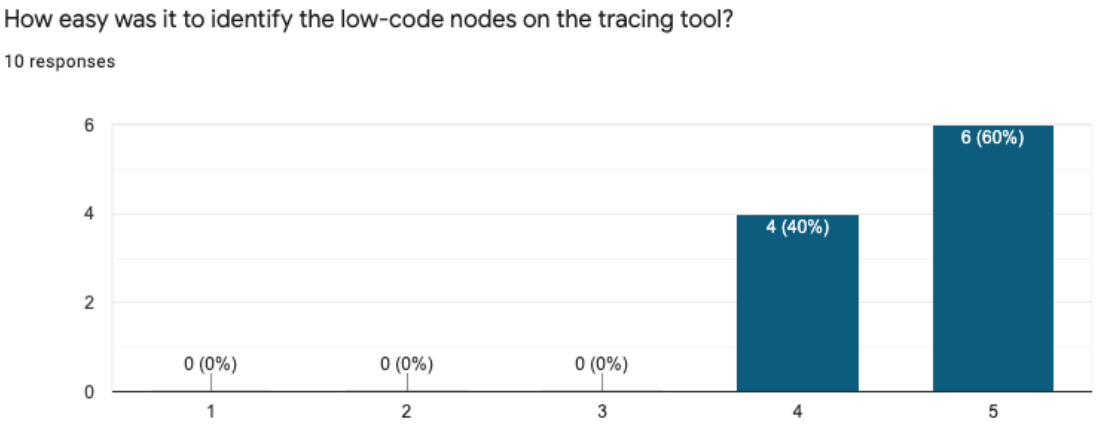
Users' feedback comments were noted, and analised, regarding their difficulties on the tests and the answers on the form. Comments focused on three major improvements:

- **Confusing calls to the server-side**: on *Service Studio* a user codes a *Client Action* to call a *Server Action* with a single node, i.e., this interaction is perceived as atomic, with the client-side just calling the server-side, and the server-side logic executing within that call. As such, the multiple spans presented on the proposed solution confused users, as they shown layers of client/server interaction not visible to the user on the *OutSystems* context. A future work improvement would be to process this information after being collecting, but before being presented, to filter out this additional (and automically put by the instrumentation library) spans;

- **Automatic calls to OutSystems services**: similarly to the previous remark, a concern was that as all requests were instrumented, a user would see every call made during the course of an action. This could lead to undisered observability to services not usually seen on the *OutSystems* context, as such call were not coded in low-code by the user, but rather generated automatically with the application code by the *OutSystems* platform. An example of such calls was present on the second test scenario, precisely to understand the real users' opinion under the context of this approach: unanimously, users thought that this information was excessive and should not be present in the context of a low-code observability approach. However, this was appointed to future work, as the current solution instruments all requests and such processing should be analised more thoroughly to understand where should this



(a) First custom question results



(b) Second custom question results

Figure 6.9 – Custom questions results.

line be drawn: which services to document and monitor and which to hide from the user, in this context;

- **Assign nodes with multiple assignments**: in the *OutSystems* language, a single *Assign* node can execute multiple *Assignments* (an *Assignment* being the pair of a variable to assign, and the value to be assigned to it). Some users did not remember this feature of the *OutSystems* language (and so did not understand why was an *Assignment* hierarchically under an *Assign* node), others did understand that but thought that would be more interesting to present immediately on the solution the name of the variable involved and its new value (rather than a list of *Assignments*, as *"Assign_Nr. 1"*, *"Assign_Nr. 2"*, etc.).

Besides those, a few minor improvements were also stated by some users. To clarify, "minor" is not a term applied to the relevance or the impact of such improvements, but rather to the lower percentage of mentions throughout user testing. Those improvements include:

- **UI component that started the trace**: traces are presented with the client-side action as their root. A user stated that it would be interesting to have information about the UI component that triggered the client action to execute (either as a root span or as a simple textual tag description somewhere). For example, in all three scenarios, client actions were triggered by the push of a button on the application's UI, but they could be triggered by a scroll to a certain part of a page, a slider, or some other UI component. The current solution does not inform about what component triggered the client action, as the client actions were chosen to be roots of all traces (as that is where the logical flow, designed by the user in low-code, begins);
- **The root span is too short**: some users were confused, as the duration of the root span (the one representing the *Client Action*, and that shall be parent of all the other spans in this trace) was shorter than expected. This is due to the placement of the instrumentation code that ends said span. The problem is that server-side call are generated as *Promises* and, as such, execute asynchronously even after the execution of the last line of the JavaScript code for said action. So, the last line of the JavaScript code (that contains the instrumentation to end that span) is being called before the server is. One possible solution to this problem would be to end this root span in the context of a promise as well, so that it is called only after all server calls return their responses (but this was not achieved successfully in the timeframe of this project, and is suggested as future work).

6.1.4 User Experience Results

A summary of the user experience results can be found on table 6.2.

Table 6.2 – User Experience Results: Summary.

Metric	Value
Success rate on identifying *OutSystems* flow (average of three scenarios)	92,03%
Success rate on identifying action's total duration	100%
Success rate on identifying a raised exception	90%
Success rate on identifying a slow node	90%
System Usability Scale (SUS) Score (average of all users)	90,75 out of 100
Customer Satisfaction Score (CSAT)	100%
Net Promoter Score (NPS)	90%
"Can you successfully identify low-code elements on the tracing tool?"	4,9 out of 5
"How easy was it to identify the low-code nodes on the tracing tool?"	4,6 out of 5

By the end of the analysis on all the data presented in this section, the author's perception was that most tested users were satisfied with the solution presented to them,

although the perception of many aspects that need improvement. The high results in all of the metrics leads to the conclusion that, although the solution is still a proof of concept, the overall approach was correct and should lead any further development.

6.2 Performance Overhead

In this section, the test cases used to assess the solution's performance impact will be presented. Nevertheless, these scenarios were also used to complement the correctness testing, as described on the beginning of the previous section: as they are comprised of real running applications using instrumented nodes, besides assessing the computational impact on the machines running instrumented applications, they were also used to validate that all the aforementioned correctness conditions were met (again, by the author, by checking if instrumented nodes were correctly displayed on *Jaeger*, after the end of each test). All test lead to absolute success, with all instrumented and executed nodes being correctly displayed.

Performance overhead was expected since the implemented technical entities and methods imply an additional computational burden to the machines running the applications being studied. Also, instrumentation gathered information that will be transmitted to a certain collector, leading to an increase in network activity. So, the impacts of the implemented solution are not only related to the application's running machine characteristics - reflecting an increase in time spent on answering a request - but also to the network activity of the environment in which the application is running.

A way of measuring these effects is to test a working version of a sample application. In this project, two test scenarios were used. One of the experimental scenarios had no instrumentation whatsoever put in place. This represented the expected application's baseline performance, equal to its performance before this project started. The second scenario had all the instrumentation, tracing configuration and services running and listening to the application's activity. This would represent the *new* application's performance: the one that translates to not only running the app, but also monitor its activity and enable its afterward presentation.

In both scenarios, a load testing tool was used: *WAPT* [78]. This tool simulated 20 concurrent users accessing an *OutSystems* application. The tested application was already presented in previous sections, comprised of a single screen containing a single button. That button, when pushed, calls the *ButtonOnClick Client Action* (figure 6.1), which executes every node that is ready to be instrumented in the current solution. This *Client Action* also calls *Server Actions* (such as seen on figure 6.2), which in turn calls every server node ready to be instrumented. This approach was put to practice so that a single push of a button could trigger the execution of every node that is ready to be instrumented, so that the overhead was measured in a scenario that encompassed all modified nodes and actions. As the same logic will be executed on either test, the difference between the two

will rely on the placement (or not) of the instrumentation, as well as its configuration. It is the impact of this difference that will be tested.

Both test scenarios were run three times, with three different durations: 10, 20 and 30 minutes. Having each duration fixed, metrics were collected from the server and client sides:

- On the Server side, CPU and RAM percentual usage was collected, leading to an absolute value: the average of each metric value in each run.
- On the Client side, more metrics were collected, namely Average Response Time, Successful Sessions, Successful Pages (these three to measure the application's request processing speed), KBytes Sent and KBytes received (to assess the impact on network activity) and, finnally, a Performance Degradation Factor, as presented by the testing tool *WAPT*.

To forestall any exhaustion of a certain machine resources, due to the demand of the testing environments (and the load they generate), two machines were using: one as a server, running the *OutSystems* platform, and other both as a client (running *WAPT* and its 20 simulated concurrent users) and as a tracing data repository (running *Jaeger*'s collector and UI).

The results of the conducted tests are displayed on figure 6.10. The table summarizes the collected metrics results for each test run (10 minutes, 20 minutes and 30 minutes). For each run, the applications results before and after applying the proposed solution are presented. The percentual impact is also presented. The last column of the table if the average of the percentual impact, for each metric, in the context of the three runs (10, 20 and 30 minutes).

By analysing the table, one can confirm the expected performance overhead, caused by the increase in computational burden that both the server and the client machines will have to process. Although the individual interpretation of most metrics is straightforward, a number of remarks is to be mentioned, regarding the overall analysis of the entire set of results:

- Although server side CPU and RAM were not heavily affected (with an increase of 35,62% and 7,98%, respectively), client-side has suffered a lot, pointing out an increase of +180% in CPU consumption and +57,62%. However, to remain in perspective, the absolute values only altered from roughly 4% to 12% in CPU and from 7% to 12% in RAM. So, although the percentual impact was significative, the absolute values remained low, not impacting nor the machine in which the test ran, nor the process running the tests.
- Average Response Time was affected heavily, averaging an increase of 133,33%. The was also due to the increase in computational burden, particularly with the need to execute additional instrumentation code. As test fixed the amount of time

Metric	10 minutes			20 minutes			30 minutes			Average Impact
	Before	After	Impact	Before	After	Impact	Before	After	Impact	
Average Server CPU %	10,051	11,828	**+17,68%**	10,635	17,444	**+64,02%**	10,390	13,005	**+25,17%**	**+35,62%**
Average Server RAM %	37,424	40,994	**+9,41%**	38,338	44,685	**+16,56%**	38,530	37,757	**-2,01%**	**+7,98%**
Average Client CPU %	4	12	**+200%**	5	12	**+140%**	4	12	**+200%**	**+180%**
Average Client RAM %	5	9	**+80%**	7	10	**+42,86%**	8	12	**+50%**	**+57,62%**
Average Response Time	0,01	0,01	**0%**	0,1	0,3	**+200%**	0,1	0,3	**+200%**	**+133,33%**
Successful Sessions	675	500	**-25,93%**	1358	1012	**-25,48%**	2048	1519	**-25,83%**	**-25,74%**
Successful Pages	7506	6653	**-11,36%**	15023	13280	**-11,60%**	22604	19905	**-11,94%**	**-11,64%**
KBytes Sent	4020	19708	**+390,25%**	8005	39773	**+396,85%**	12025	59660	**+396,13%**	**+394,41%**
KBytes Received	374699	354970	**-5,27%**	747430	705762	**-5,57%**	1123938	1056074	**-6,04%**	**-5,63%**
Performance Degradation Factor	0,05	0,12	**+140%**	0,05	0,26	**+420%**	0,05	0,21	**+320%**	**+293,33%**

Figure 6.10 – Test Results on the overhead impact of the current solution

running, this increase in response time lead to a slower overall execution speed, finally leading to a lower number of successfully loaded pages. This metric dropped by 11,64%. As a result, a lower number of sessions were successfully completed, suffering by 25,74%.

- In terms of network activity, it is important to understand that the results were taken by the perspective of the client. That being said, unexpectedly the number of KBytes Received (by the client, that is) dropped by 5,63%. This result was not correlated with the impact of the proposed solution, and was appointed as a result of noise in the test environment (as no modification was made that reduced the amount of information sent by the server to the client). On the other hand (and as expected), the number of KBytes Sent increased significantly, by 394,41%. This is easily appointed to the dumps of tracing information, by the client, sent to the *Jaeger* collector.

- Finally, *WAPT* output a Performance Degradation Factor, translating the degradation in several performance metrics throughout a test run. It is also easily noticed that the proposed solution impacted the system by increasing this Factor by 293,33%. This means that a system applying this solution has its performance degrading more quickly over time. This is specially concerning and should be addressed in future work, as it could lead to a major system failure, if not contained.

Remembering a detail from chapter 5, tracing configuration on the server-side was placed at the beginning of each request. This represents a major burden for each request to execute, resulting in an enormous repetition over the totality of requests made in the test. A probable improvement over the presented solution would be to place said configuration in a more efficient place (possibly, at the start of the application).

Chapter 7

Conclusion

This dissertation began by scrutinizing the problem that it had been faced with: enabling access to runtime tracing information to developers of a low-code application, specifically at the same abstraction level used on the development process, not of the generated code. This was followed by a period of extensive research, not only about the conceptual information to be dealt with, but also regarding related work done on the field, focusing on the various types of tools that could be used to solve the aforementioned problem. A set of these technologies and tools was chosen to use in the implementation of a proposed solution: a proof-of-concept (PoC), applied to the *OutSystems* product. Said solution was then evaluated and validated, regarding correctness and impact with performance overhead.

Observability is a concept always present throughout this dissertation. It translates to the need of being able to know how a system is behaving when it is running. This allows the system maintainers to know how well the system is behaving according to expectations, as well as pinpoint which part or section needs to be subject of optimization or correction. On the research done, several types of monitoring techniques and tracing technologies were analyzed, and all shared the goal of improving the observability of the systems they were put to practice. The technologies chosen for the implementation allied the contribution to achieve this goal with simplicity of use and great performance, as they are not only developed by a large and diverse community, but also are based on the leading investigations on the field, employing some of the most important concept and good practices that research shown to be key.

In the design stage, one aspect of the solution to implement became clear: the need to focus on manual instrumentation, rather than resorting to an automatic one. This could appear as counterintuitive, as the initial hope was that automatic instrumentation could be achieved. A clarification is, so, needed: the instrumentation should be automatic to

the developer of a low-code application. Meaning that, when a low-code developer is designing his application using a product such as *OutSystems*, it is not needed any regards towards instrumentation or tracing configuration, as it will be placed automatically when the application is published (i.e., enters the process of high-level code generation, compilation and deployment), just like the rest of the code to be run. However, as the rest of the code to be generated, its placing at the code generator level must be manual, at least in this stage. This means that the compiler's code that is responsible for generating the application code must be altered manually so that it generates the needed instrumentation on the final application code automatically. The need for these changes to remain manual for now is that the studied products that allowed for automatic instrumentation of applications all relied the analysis of the application's code to be ran. It became evident that such analysis by this product would present results that would appear at the abstraction level of the generated high-level code, since it would be the one that the product would analyze. Such results would fail to match the low-level abstraction that the developer knows from the development process, possibly becoming incomprehensible and, therefore, useless, to users with minor technological backgrounds (that only understand the abstracted code that they created, not its relation to the high-level generated code that is running). After some deliberation, it was realized that it would be necessary an extensive amount of time and effort to create some form of translation, that would pick up the tracing data exposed by automatic instrumentation and present results to the user at the low-code abstraction level. And that is why manual changes to the code's generator were put to practice, as this way all time was spent in instrumenting the code with the desired level of customization, always keeping the low-code abstraction level.

Implementing the PoC solution for the mentioned problem started as a challenge, having to deal with the complications related with technologies that are on their early stages of the development environment, so still growing. This led to some lack of available documentation, translating to a set of difficulties during the configuration process. However, an active community made these problems solvable and also ensures the growing tendency of quality and support of the technologies used. The challenge grew even bigger when faced with the difficulty to find all *OutSystems* platform's code region where each node is generated, as well as the correct way to place instrumentation in said regions. Also an important note regarding implementation is that a set of compromises were made when dealing with certain problems that diminish the direct placement of this solution to the production environment: in some cases, addressing very specific problems was left to Future Work, as it was considered out of the scope of the present work. Nonetheless, a solution was presented that accomplished the goal set to it, not only leading the way to future improvements, but already providing significant coverage regarding traceability, namely 19 out of 28 available nodes (across both Client and Server sides), as well as a set of already useful features in viewing runtime tracing data and identifying problems.

The evaluation of the proposed solution led to the conclusion that the results are successful. Regarding correctness, not only did the tracing data presented itself as expected,

but also executions were stable and results were consistent. Regarding performance, the conducted tests point out an impact, significant in some of the collected metrics. With the implementation of the proposed solution, the application is running more code (taking time to do so) and exporting more information over the network, so a certain degree of overhead was expected. Impacts were pointed out in this dissertation, hoping that future work can tackle them, enabling the application of this project's work on a real use-case scenario. Regarding user satisfaction, results point out that the overall approach was already usable and acted as an enabler for interesting and valuable insights on an *OutSystems* application runtime behaviour. Again, many possible improvements were pointed out and should be tackled in future work, so that the experience in using such a solution is the best possible, maintaining its value to real users. Overall, results were positive and the goal of this project was considered achieved, not overlooking the set of aforementioned limitations.

7.1 Contributions

As its main contribution, this dissertation produced a working PoC, in which it is possible to access runtime tracing information about a low-code application, namely an OutSystems one. Thus it presents a solution to the problem addressed in the dissertation. It does so while encompassing end-to-end information, regarding both sides of a working OutSystems' module, therefore achieving context-propagation and presenting precise and concise information. Both factors were determined as essential to the practicality of the solution, not only because of its great coverage of the OutSystems product as viewed by users, but also because it maintains the same level of abstraction that the users are used to when developing with this product, thus enabling its wide use by users with all kinds of technical backgrounds, even those with minor experience (also called 'citizen developers').

This contribution may, however, not be ideal, and that was initially expected, bearing in mind that this is not a final solution, and so it is not ready to enter the production environment of the *OutSystems* product, but rather a first try at solving this particular problem. The need to place manual instrumentation is considered to be its greatest flaw, from an *OutSystems* standpoint, as it forces engineers maintaining the product to worry about this necessity. However, as discussed above, it was a compromise taken to ensure the correctness, performance and suitability of the solution. Furthermore, a lot of future work could be done in order to improve its quality, right from the state as it is now. Suggestions to this work are described in detail in the next section.

Such a contribution was only made possible after the study of the techniques and approaches currently available, both in academic and industrial fields. The extensive research of such approaches is also appointed as a contribution of this dissertation, as it studied in detail several tools, techniques and standards. Ultimately, this converged on the choice of *OpenTelemetry* as the correct tool to use in this project, due to its prominence

on the fields, and foundations on previous successful projects, all backed-up by several research pieces and techniques proven to be industrially and academically relevant.

7.2 Future Work

As mentioned above, the designed solution had some limitations. Not only these limitations shortened the solution's performance, but some also made its implementation more difficult in future uses. Regarding future work on the solution designed and proposed on this dissertation, it should be considered that:

- The placement of instrumentation is placed manually on the *OutSystems*' platform code. This approach works but is rather time-consuming, namely regarding the fact that the person placing the instrumentation code has to find where, in the platform's code, is the code for each low-code node being generated.
- The *JavaScript* dependency code is still loaded as a bundle in each screen that the instrumentation is required. This is the limitation that makes the most expressive impact if an external user were to adopt the designed solution, as it requires manual placement of said code bundle in each screen of the application. An important task to take place if this solution were to be adopted is deliver said code bundle with each OutSystems application, so that it is accessible by all, making all instrumentation-enabled.
- Most nodes available on Service Studio were instrumented, and so, when executed on a running app, a span representing them appeared on the Jaeger UI, reporting their execution duration and context. However, said span could be documented with extra tags and labels, as shown in previous chapters. Some were already added, taking into account the author's needs, but any further implementation of this solution should take this feature into account, as measure well the level of detail it needs: as some use cases might profit from such extra span documentation to allow for a more complete picture of the runtime scenario, others might want to keep the information detail-level as low as possible (e.g.: while dealing with end-users with a weak technical background), but all should bear in mind that each extra detail represents an increase in performance overhead (mainly network usage, as more data needs to be sent to the tracing data repository).
- There are still nodes left without instrumentation, as the timespan of this project only allowed for a portion of all the available nodes to be considered.
- As stated on chapter 5 "*OpenTelemetry* code (namely, the *.NET* part) has not been updated since the beginning of the development of this work. At that time, the library was still on its *alpha* stage, so it was prone to changes that would affect severely the structure of the code. So, in order to avoid dealing with such problems

regarding this dependency, this code was left without updates". A future task to be done is to update to the most recent version of this dependency (as it should be, by the time of reading, at least more stable than what it was at the time of the beginning of the implementation) and correct any issues that may occur.

- In order for the context-propagation automation work between services, a CORS proxy is needed. In the future, this issue should be addressed, in order to avoid this necessity.

- Tracing configuration code (such as endpoints to be used by the Exporters and Services' names) are not configurable by the user, and were hard-coded on each side. An interesting following approach would be to fully integrate this solution with Service Studio, adding some sort of menu to enable such configurations for each OutSystems module. The current hard-coded settings used in this PoC could be used as defaults, as they have already proven their usefulness.

- Performance overhead is an issue, and possibly a deeper analysis of the *OutSystems* platform could help find better ways of instrumenting it. Updating the *OpenTelemetry* libraries to newer versions, as it is still an in-progress project, would help to mitigate this problem aswell (as newer releases strive to be more stable and less impactful).

- There are still several improvements to be made, regarding user experience. Those were described in detail in the previous chapter 6, but focused on some gaps in abstraction levels (such as the display of verbose server calls otherwise unseen by an *OutSystems* user). Some of this gaps were not related with the collection of tracing information (which was the main focus of this dissertation), but rather its presentation. An important future work would be to design an interface, inside the *OutSystems* "ecosystem", that presents the information collected using this approach, in the best manner possible (focusing on user experience and maintaining the *OutSystems*' line of design and interface).

Bibliography

[1] A. Aghabayli et al. "Integrating Runtime Data with Development Data to Monitor External Quality: Challenges from Practice". In: *Proceedings of the 2nd ACM SIGSOFT International Workshop on Software Qualities and Their Dependencies*. SQUADE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 20–26. ISBN: 9781450368575. DOI: 10.1145/3340495.3342752. URL: https://doi.org/10.1145/3340495.3342752.

[2] E. Ates et al. "An Automated, Cross-Layer Instrumentation Framework for Diagnosing Performance Problems in Distributed Applications". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 165–170. ISBN: 9781450369732. DOI: 10.1145/3357223.3362704. URL: https://doi.org/10.1145/3357223.3362704.

[3] B. Beyer et al. *Site Reliability Engineering: How Google Runs Production Systems*. 1st. O'Reilly Media, Inc., 2016. ISBN: 149192912X.

[4] P. C. Brebner. "Automatic Performance Modelling from Application Performance Management (APM) Data: An Experience Report". In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE '16. Delft, The Netherlands: Association for Computing Machinery, 2016, pp. 55–61. ISBN: 9781450340809. DOI: 10.1145/2851553.2851560. URL: https://doi.org/10.1145/2851553.2851560.

[5] J. Cito, G. Mazlami, and P. Leitner. "TemPerf: Temporal Correlation between Performance Metrics and Source Code". In: *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*. QUDOS 2016. Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 46–47. ISBN: 9781450344111. DOI: 10.1145/2945408.2945420. URL: https://doi.org/10.1145/2945408.2945420.

[6] J. Cito et al. "Context-Based Analytics: Establishing Explicit Links between Runtime Traces and Source Code". In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. ICSE-SEIP '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 193–202. ISBN: 9781538627174. DOI: 10.1109/ICSE-SEIP.2017.1. URL: https://doi.org/10.1109/ICSE-SEIP.2017.1.

[7] J. Cito et al. "Interactive Production Performance Feedback in the IDE". In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 971–981. DOI: 10.1109/ICSE.2019.00102. URL: https://doi.org/10.1109/ICSE.2019.00102.

[8] R. Fonseca et al. "X-Trace: A Pervasive Network Tracing Framework". In: *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. Cambridge, MA: USENIX Association, Apr. 2007. URL: https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework.

[9] C. Heger et al. "Application Performance Management: State of the Art and Challenges for the Future". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. LAquila, Italy: Association for Computing Machinery, 2017, pp. 429–432. ISBN: 9781450344043. DOI: 10.1145/3030207.3053674. URL: https://doi.org/10.1145/3030207.3053674.

[10] P. Huang et al. "Capturing and Enhancing in Situ System Observability for Failure Detection". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 1–16. ISBN: 9781931971478.

[11] M. Julian. *Practical Monitoring: Effective Strategies for the Real World*. O'Reilly Media, Incorporated, 2017. ISBN: 9781491957356. URL: https://books.google.pt/books?id=d1fHjwEACAAJ.

[12] A. Lahmadi and F. Beck. *Powering Monitoring Analytics with ELK stack*. 9th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2015). June 2015. URL: https://hal.inria.fr/hal-01212015.

[13] J. Mace and R. Fonseca. "Universal Context Propagation for Distributed System Instrumentation". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190526. URL: https://doi.org/10.1145/3190508.3190526.

[14] D. A. Meedeniya, I. D. Rubasinghe, and I. Perera. "Traceability Establishment and Visualization of Software Artefacts in DevOps Practice: A Survey". In: *International Journal of Advanced Computer Science and Applications* 10.7 (2019). DOI: 10.14569/IJACSA.2019.0100711. URL: http://dx.doi.org/10.14569/IJACSA.2019.0100711.

[15] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013. ISBN: 9781680505009. URL: https://books.google.pt/books?id=gA9QDwAAQBAJ.

[16] B. H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. URL: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[17] C. Sridharan. *Distributed Systems Observability: A Guide to Building Robust Systems*. O'Reilly Media, 2018. URL: https://books.google.pt/books?id=wwzpuQEACAAJ.

[18] H. Wang et al. "GRANO: Interactive Graph-Based Root Cause Analysis for Cloud-Native Distributed Data Platform". In: *Proc. VLDB Endow.* 12.12 (Aug. 2019), pp. 1942–1945. ISSN: 2150-8097. DOI: 10.14778/3352063.3352105. URL: https://doi.org/10.14778/3352063.3352105.

[19] P. Wang et al. "CloudRanger: Root Cause Identification for Cloud Native Systems". In: *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGrid '18. Washington, District of Columbia: IEEE Press, 2018, pp. 492–502. ISBN: 9781538658154. DOI: 10.1109/CCGRID.2018.00076. URL: https://doi.org/10.1109/CCGRID.2018.00076.

Webography

[21] F. Alexander. *OutSystems Platform Public Cloud*. 2020. URL: https://www.outsystems.com/blog/posts/application-platform-as-a-service/ (visited on 02/18/2020).

[22] E. B.V. *APM .NET Agent Reference - Introduction*. 2020. URL: https://www.elastic.co/guide/en/apm/agent/dotnet/1.x/intro.html (visited on 01/27/2020).

[23] E. B.V. *Beats: Data Shippers for Elasticsearch*. 2020. URL: https://www.elastic.co/beats (visited on 01/27/2020).

[24] E. B.V. *Elastic APM - Components and Documentation*. 2020. URL: https://www.elastic.co/guide/en/apm/get-started/current/components.html (visited on 01/27/2020).

[25] E. B.V. *Elastic APM Overview*. 2020. URL: https://www.elastic.co/guide/en/apm/get-started/current/overview.html (visited on 01/27/2020).

[26] E. B.V. *Elastic Stack*. 2020. URL: https://www.elastic.co/products/elastic-stack (visited on 01/21/2020).

[27] E. B.V. *Elasticsearch Introduction*. 2020. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html (visited on 01/27/2020).

[28] E. B.V. *Kibana Introduction*. 2020. URL: https://www.elastic.co/guide/en/kibana/current/introduction.html (visited on 01/27/2020).

[29] E. B.V. *Logstash Introduction*. 2020. URL: https://www.elastic.co/guide/en/logstash/current/introduction.html (visited on 01/27/2020).

[30] E. B.V. *Observability with the Elastic Stack*. 2020. URL: https://www.elastic.co/blog/observability-with-the-elastic-stack (visited on 01/27/2020).

[31] E. B.V. *Observability with the Elastic Stack*. 2020. URL: https://www.elastic.co/guide/en/ecs/current/ecs-reference.html (visited on 01/27/2020).

[32] E. B.V. *Use Cases - Elastic Stack Success Stories*. 2020. URL: https://www.elastic.co/customers/?usecase=elastic-observability (visited on 01/28/2020).

[33] Browserify. *Browserify*. 2020. URL: http://browserify.org/ (visited on 11/20/2020).

[34] J. Cito et al. *Context Based Analytics*. 2020. URL: https://github.com/sealuzh/ContextBasedAnalytics (visited on 01/28/2020).

[35] Datadog. *Cloud Monitoring as a Service | Datadog*. 2020. URL: https://www.datadoghq.com/ (visited on 01/28/2020).

[36] O. U. P. Dictionary.com. *English Dictionary, Thesaurus, & Grammar Help | Lexico.com*. 2020. URL: https://www.lexico.com/ (visited on 01/09/2020).

[37] Epsagon. *Automated Tracing for Cloud Microservices | Epsagon*. 2020. URL: https://epsagon.com/ (visited on 02/03/2020).

[38] Epsagon. *Zipkin or Jaeger? The Best Open Source Tools for Distributed Tracing*. 2019. URL: https://epsagon.com/blog/zipkin-or-jaeger-the-best-open-source-tools-for-distributed-tracing/ (visited on 02/03/2020).

[39] C. N. C. Foundation. *Cloud Native Computing Foundation*. URL: https://www.cncf.io/ (visited on 02/07/2020).

[40] Graphite. *Graphite*. URL: https://graphiteapp.org/ (visited on 02/10/2020).

[41] Jaeger. *Jaeger: open source, end-to-end distributed tracing*. 2020. URL: https://www.jaegertracing.io/ (visited on 02/03/2020).

[42] R. F. Jonathan Mace. *Java implementation of the Tracing Plane – Baggage Contexts and Execution-Flow Scoped Variables*. 2017. URL: https://github.com/tracingplane/tracingplane-java (visited on 02/09/2020).

[43] G. Labs. *Grafana: The open observability platform | Grafana Labs*. 2020. URL: https://grafana.com/ (visited on 02/03/2020).

[44] W. Li. *Anomaly Detection in Zipkin Trace Data*. URL: https://engineering.salesforce.com/anomaly-detection-in-zipkin-trace-data-87c8a2ded8a1 (visited on 02/07/2020).

[45] Logz.io. *Log Analysis Software That Combines ELK + Machine Learning*. 2020. URL: https://logz.io/ (visited on 02/03/2020).

[46] Logz.io. *Zipkin vs Jaeger: Getting Started With Tracing | Logz.io*. 2018. URL: https://logz.io/blog/zipkin-vs-jaeger/ (visited on 02/03/2020).

[47] Merriam-Webster. *Dictionary by Merriam-Webster*. 2020. URL: https://www.merriam-webster.com/ (visited on 01/09/2020).

[48] OpenCensus. *OpenCensus*. URL: https://opencensus.io/ (visited on 02/04/2020).

[49] OpenTelemetry. *OpenTelemetry .NET SDK - distributed tracing and stats collection framework*. 2020. URL: https://github.com/open-telemetry/opentelemetry-dotnet/tree/32829d40437e403c691f9474e5ec428ab8a47688 (visited on 11/20/2020).

[50] OpenTelemetry. *OpenTelemetry .NET SDK - distributed tracing and stats collection framework | Configuration*. 2020. URL: https://github.com/open-telemetry/opentelemetry-dotnet/tree/32829d40437e403c691f9474e5ec428ab8a47688#configuration (visited on 11/20/2020).

[51] OpenTelemetry. *OpenTelemetry | Effective observability requires high-quality telemetry*. 2020. URL: https://opentelemetry.io/ (visited on 02/03/2020).

[52] OpenTelemetry. *OpenTelemetry JavaScript*. 2020. URL: https://github.com/open-telemetry/opentelemetry-js (visited on 11/20/2020).

[53] OpenTelemetry. *OpenTelemetry XMLHttpRequest Instrumentation for web*. 2020. URL: https://www.npmjs.com/package/@opentelemetry/plugin-xml-http-request (visited on 11/20/2020).

[54] OpenTracing. *The OpenTracing project*. URL: https://opentracing.io/ (visited on 02/04/2020).

[55] OutSystems. *Choose the Right App for Your Project*. URL: https://success.outsystems.com/Documentation/11/Getting_Started/Choose_the_Right_App_for_Your_Project (visited on 02/18/2020).

[56] OutSystems. *How Application Performance Is Measured*. URL: https://success.outsystems.com/Documentation/11/Managing_the_Applications_Lifecycle/Monitor_and_Troubleshoot/How_Application_Performance_Is_Measured (visited on 02/17/2020).

[57] OutSystems. *How do I instrument my apps for performance metrics?* URL: https://www.outsystems.com/evaluation-guide/how-do-i-instrument-my-apps-for-performance-metrics/ (visited on 02/12/2020).

[58] OutSystems. *Logging database and architecture*. URL: https://success.outsystems.com/Documentation/11/Managing_the_Applications_Lifecycle/Monitor_and_Troubleshoot/Logging_database_and_architecture (visited on 02/17/2020).

[59] OutSystems. *OutSystems tools and components*. URL: https://www.outsystems.com/evaluation-guide/outsystems-tools-and-components/ (visited on 02/18/2020).

[60] OutSystems. *PerformanceMonitoring API*. URL: https://success.outsystems.com/Documentation/11/Reference/OutSystems_APIs/PerformanceMonitoring_API (visited on 02/17/2020).

[61] OutSystems. *Service Studio Overview*. URL: https://success.outsystems.com/Documentation/11/Getting_Started/Service_Studio_Overview (visited on 02/18/2020).

[62] OutSystems. *The APDEX Performance Score*. URL: https://success.outsystems.com/Documentation/11/Managing_the_Applications_Lifecycle/Monitor_and_Troubleshoot/The_APDEX_Performance_Score (visited on 02/17/2020).

[63] OutSystems. *Troubleshoot the Performance of an Application*. 2020. URL: https://success.outsystems.com/Documentation/11/Managing_the_Applications_Lifecycle/Monitor_and_Troubleshoot/Troubleshoot_the_Performance_of_an_Application (visited on 11/15/2020).

[64] OutSystems. *Troubleshoot the Performance of an Application*. URL: https://success.outsystems.com/Documentation/11/Managing_the_Applications_Lifecycle/Monitor_and_Troubleshoot/Troubleshoot_the_Performance_of_an_Application (visited on 02/17/2020).

[65] OutSystems. *Use Processes (BPT)*. URL: https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Processes_(BPT) (visited on 02/18/2020).

[66] OutSystems. *View the Environment Logs and Status*. 2020. URL: https://success.outsystems.com/Documentation/11/Managing_the_Applications_Lifecycle/Monitor_and_Troubleshoot/View_the_Environment_Logs_and_Status (visited on 11/15/2020).

[67] OutSystems. *View the Environment Logs and Status*. URL: https://success.outsystems.com/Documentation/11/Managing_the_Applications_Lifecycle/Monitor_and_Troubleshoot/View_the_Environment_Logs_and_Status (visited on 02/17/2020).

[68] OutSystems. *What kind of monitoring and analytics does OutSystems offer?* URL: https://www.outsystems.com/evaluation-guide/what-kind-of-monitoring-and-analytics-does-outsystems-offer/ (visited on 11/27/2020).

[69] Prometheus. *Prometheus - Monitoring system & time series database*. 2020. URL: https://prometheus.io/ (visited on 02/10/2020).

[70] Qualtrics. *What is CSAT (customer satisfaction score)?* 2021. URL: https://www.qualtrics.com/uk/experience-management/customer/what-is-csat/ (visited on 02/20/2021).

[71] Quicksearch. *NPS Calculator*. 2021. URL: http://www.npscalculator.com/en (visited on 02/20/2021).

[72] M. Revell. *What Is Low-Code? [2020 Update]*. 2020. URL: https://www.outsystems.com/blog/what-is-low-code.html (visited on 02/18/2020).

[73] N. Satmetrix. *What Is Net Promoter? A TRUSTED ANCHOR FOR YOUR CUSTOMER EXPERIENCE MANAGEMENT PROGRAM*. 2021. URL: https://www.netpromoter.com/know/ (visited on 02/20/2021).

[74] Y. Shkuro. *Embracing context propagation*. URL: https://medium.com/jaegertracing/embracing-context-propagation-7100b9b6029a (visited on 02/07/2020).

[75] Y. Shkuro. *Jaeger and OpenTelemetry*. URL: https://medium.com/jaegertracing/jaeger-and-opentelemetry-1846f701d9f2 (visited on 02/07/2020).

[76] B. Sigelman. *Merging OpenTracing and OpenCensus: Goals and Non-Goals*. 2019. URL: https://medium.com/opentracing/merging-opentracing-and-opencensus-f0fe9c7ca6f0 (visited on 02/07/2020).

[77] T. Simões. *Reactive Web: The Next Generation of Web Apps*. 2020. URL: https://www.outsystems.com/forums/discussion/52761/reactive-web-the-next-generation-of-web-apps/ (visited on 02/18/2020).

[78] SoftLogica. *WAPT 10: Performance testing tool for web and mobile applications*. 2020. URL: https://www.loadtestingtool.com/product.shtml (visited on 11/15/2020).

[79] W. T. *Measuring and Interpreting System Usability Scale (SUS)*. 2021. URL: https://uiuxtrend.com/measuring-system-usability-scale-sus/ (visited on 02/20/2021).

[80] O. S. D. team. *MonitorProbe*. URL: https://www.outsystems.com/forge/component-overview/4559/monitorprobe (visited on 02/17/2020).

[81] V. Technologies. *A Comprehensive Tutorial to Implementing OpenTracing With Jaeger*. 2019. URL: https://medium.com/velotio-perspectives/a-comprehensive-tutorial-to-implementing-opentracing-with-jaeger-a01752e1a8ce (visited on 02/07/2020).

[82] B. University. *Brown University Tracing Framework*. URL: http://brownsys.github.io/tracing-framework/ (visited on 02/09/2020).

[83] usability.gov. *System Usability Scale (SUS)*. 2021. URL: https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html (visited on 02/20/2021).

[84] W3Schools. *JavaScript Window - The Browser Object Model*. 2020. URL: http://browserify.org/ (visited on 11/20/2020).

[85] D. Wild. *Observability for front-end web clients with OpenTelemetry and Jaeger in 5 minutes*. 2020. URL: https://medium.com/@danwild.io/observability-for-front-end-web-clients-with-opentelemetry-and-jaeger-in-5-minutes-343f719fbf5a (visited on 11/20/2020).

[86] D. Wild. *tracer-web-jaeger*. 2020. URL: https://github.com/danwild/opentelemetry-js/tree/master/examples/tracer-web-jaeger (visited on 11/20/2020).

[87] M. Wutzke. *Elastic Stack and Elasticsearch*. 2020. URL: https://www.michael-wutzke.com/elastic-stack-and-elasticsearch-basics-and-tips/ (visited on 01/27/2020).

[88] A. Yigal. *Grafana vs. Kibana: The Key Differences to Know*. 2018. URL: https://logz.io/blog/grafana-vs-kibana/ (visited on 02/04/2020).

[89] T. Young. *Merging OpenTracing and OpenCensus: A Roadmap to Convergence*. 2019. URL: https://medium.com/opentracing/a-roadmap-to-convergence-b074e5815289 (visited on 02/07/2020).

[90] Zipkin. *OpenZipkin : A Distributed Tracing System*. 2020. URL: https://zipkin.io/ (visited on 02/03/2020).