INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Testes End to End em Microserviços

DANIEL CANASTRO DIAS

Outubro de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

End to End Testing in Microservices

Daniel Canastro Dias

Dissertation to obtain the Master's degree in Informatics Engineering, Specialization in Software Engineering

Advisor: Alberto Sampaio

Co-advisor: Isabel Sampaio

Porto, 2021

Dedication

In dedication to all student workers who build their own future with extra sweat and blood and tears. You are all almost free too.

Abstract

The microservices architecture is a recent trend in the software engineering community, with the number of research articles in the field increasing, and more companies adopting the architectural style every year. However, the testing aspect of this architecture can sometimes be overlooked, with a lack of guidelines for its execution. Also, microservices testing introduce a lot of different challenges that are not faced when following a monolithic architecture.

This work aims to fill some gaps in current microservices testing research by providing a study of five existing service virtualization tools, implementing them in a company system and extracting the results of the tool properties identified through a survey delivered to a convenience sample of software development experts.

Finally, the mentioned experts in the microservices field validated the results of the research and the tool identified as the best and provided insights regarding the value of this work.

Key Words: End-to-End Testing, Microservice Architecture, Service Virtualization

Resumo

A arquitetura de micros serviços é uma tendência recente na comunidade de engenharia de *software*, com o número de artigos na área a aumentar e mais empresas a adotar o estilo arquitetónico a cada ano. No entanto, a área dos testes end-to-end desta arquitetura pode às vezes ser esquecido. Além disso, os testes de micros serviços apresentam muitos desafios diferentes que não são enfrentados ao seguir uma arquitetura monolítica.

Este trabalho visa preencher algumas lacunas na pesquisa de teste de micros serviços atuais, fornecendo um estudo de cinco ferramentas de virtualização de serviço existentes, implementando-as num sistema empresarial e extraindo os resultados das propriedades da ferramenta identificadas por meio de um questionário entregue a uma amostra de conveniência de especialistas em desenvolvimento de *software*.

Por fim, os citados especialistas na área de micros serviços validaram os resultados da pesquisa e a ferramenta identificada como a melhor e forneceram a sua visão geral sobre o valor deste trabalho.

Palavras-chave: Testes End-to-End, Arquitetura de Micro Serviços, Virtualização de Serviços

Acknowledgements

First, I must thank with the deepest love to my mother and father, who showed me the value of hard work for the things and people we love and always managed to provide everything I ever needed to achieve all my objectives while improving as a human being.

I also want to especially thank my friends Carlos Vicente, José Gonçalves, Tiago Leite, Henrique Costa, Francisco Azevedo, and many others whose paths crossed with mine during this year at ISEP. It's thanks to all that this journey was bearable and enjoyable.

Furthermore, I must thank my thesis advisors Alberto Sampaio and Isabel Sampaio of ISEP, who were kind enough to accept my thesis proposal, who instantly answered emails with all the help anyone would ever need and were always ready for a meeting to discuss the work. Without their availability to help when I ran into trouble, this accomplishment would probably not be achieved.

Finally, I must thank the experts who were involved in the surveys for this project. Without their passionate participation and input, the surveys could not have been successfully conducted.

Thanks to all of you, who made this accomplishment possible.

Table of Contents

Table of Figures

Table of Tables

Acronyms and Symbols

Acronyms

E2E	End-to-End
SOA	Service Oriented Architecture
ERP	Enterprise resource planning
CI	Continuous Integration
GUI	Graphical User Interface
IDE	Integrated development environment

1 Introduction

In this chapter, the problem, objective, and context of the theme of this report are presented alongside the document structure.

1.1 Context

The literature defines microservice as a small application (generally less than a couple of thousand lines of code) with a single responsibility (a functional, non-functional, or cross functional requirement) that can be independently deployed, scaled, and tested (1). It must be cohesive and independent of other processes, interacting with them via messages using a clearly defined interface (2).

A microservice architecture was defined as "a distributed application where all its modules are microservices" (2). On the microservices architectural style, each module of the system must be identified and isolated on a single microservice. Therefore, the functionality must be divided through the services using the appropriate granularity, to achieve high cohesion inwards and loose coupling outwards (3).

Microservices oriented architecture has been regarded as a promising solution (1) that conjugates scalability, maintainability, ease of deployment, reduced infrastructure costs, technology heterogeneity, resilience, reusability, among others (4).

The microservices architectural style is a recent trend in the software engineering community since it was first publicly proposed by Fowler and Lewis in 2014 (3). However, the term was first discussed at a workshop near Venice in 2011 where different authors and experts of the field debated some techniques like the microservices architecture, including some of the SOA principles (5). These techniques and principles were consolidated in the microservices architectural style, and that definition started the referred trend.

Each microservice is developed, deployed, and managed independently; new features and updates are delivered continuously (6), possibly hundreds of times a day, making the applications extremely dynamic. Microservice applications are typically polyglot: developers write individual microservices in the programming language of their choice, and the communication between services happens using remote API calls.

Microservices are designed to withstand infrastructure failures and outages yet struggle to remain available when deployed. The *postmortem* reports point to missing or faulty failure-recovery logic, indicating that unit and integration tests are insufficient to catch such defects. The application deployment needs to be subjected to end-to-end testing.

End-to-end testing is a technique that tests the entire software product from beginning to end to ensure the application flows behave as expected (7). It defines the product's system dependencies and ensures all integrated pieces work together as expected. Software systems nowadays are complex and interconnected with numerous subsystems. If any of the subsystems fails, the whole software system could crash. This is a major risk and can be avoided by end-to-end testing (8).

Microservices and their development model, however, pose new challenges for systematic resiliency testing:

- Microservices' polyglot nature requires the testing tool to be agnostic to each service's language and runtime.
- Rapidly evolving code requires tests that are fast and focus on the failure-recovery logic, not business logic.

Existing works on end-to-end of general distributed systems and service-oriented architectures (SOAs) are unsuitable for microservice applications, since they do not address the challenges mentioned above (9). In contrast, in this project the objective is to provide systematic, application-agnostic testing on live services by analyzing a series of tools and propose one as the best approach.

1.2 Problem

In the context of most development environments in most companies, the microservice architecture is being used or adapted into (10). Although this type of architecture comes with an array of advantages, a new problem arose with their test methods, due to the added complexity of network communication between the collaborating services, this pattern needs an infrastructure to establish communication between microservices, possibly a cloud infrastructure (3), databases, third-party services, or other external components. The network connections used in this infrastructure add complexity to testing the application making it more difficult to perform end-to-end testing when compared with a monolithic system. Hence, it is necessary to test it at runtime to monitor the behaviour of the different components in a production environment.

Therefore, there is a need to analyse the tools available in the market and extract the features most suited to a business environment to facilitate the testing of services developed in a microservice architecture.

1.3 Goal

The goal of this project is to analyse some of the currently available tools for service virtualization and their role in End-to-End testing with focus on microservices created withing a certain company environment with specific criteria, such as the framework and message brokers used by the company. This kind of tests will focus on the integrated software system because the interconnected systems are mocked with the service virtualization tool.

To achieve this, it is necessary to first analyse the tools and their documentation and identify their challenges. After identifying the challenges, it will be necessary to identify the best features and practices of each one of them by implementing them in a controlled environment identical to the one used in a specific company, where the development language is C#, the development framework is .Net, the services use Kafka as the message broker of choice and communicate via REST API as well. Finally, with data obtained and complied, it will be necessary to pinpoint the better tool of the bunch and its implementation practice that is best suited for the environment in which the tools are being tested on.

The findings of this report will them be evaluated by a selected group of professionals within the company in question.

1.4 Contributions of this work

The first main contribution of this work is the analysis and comparison of the available literature regarding end-to-end testing and service virtualization tools. This work tries to clarify what are the current most common challenges while performing end-to-end test in a microservice architecture, guiding future researchers, and helping the industry to avoid the identified issues, while providing possible tools and best practices also identified in the research. Furthermore, an article was developed based on the output of this work and publicly published on Computer Engineering Symposium (11) where a different perspective of reviews was gathered.

In the future, the tool identified as the better one among the selection can be applied in projects or put in practice by other interested companies which can then provide their testimony on the value of this project.

This document is written in English so that this work can reach a larger number of professionals or companies.

1.5 Work methodology

The development of this work will consist of different phases.

First, a narrative literature review will be performed to validate the problem. With that information, the objectives of this work will be defined, and therefore, this work aims to reduce the gap of missing research regarding challenges of end-to-end testing in a microservice architecture by identifying the most reported problems. Also, another objective is to support the use of service virtualization in end-to-end testing by further detailing the issue and exploring possible solutions. This first literature review helps define the context of the work and the problem to be solved. Also, the gathered information will help conclude the value analysis of this project.

Once the problem context and the objectives of this work were defined, literature research was be performed through a systematic mapping study that helped identify the most common challenges of end-to-end testing in a microservice architecture.

Regarding service virtualization, a set of existent tools and approaches will be analysed, and the most adequate to the context of this work will be chosen, to propose a solution for E2E of microservices architectures.

Finally, the research findings will be presented to the industry and evaluated through a survey in which expert professionals of the field can assess it and provide insights regarding the work developed.

1.6 Document structure

This document is divided into seven different chapters, which are followed by references and appendix sections.

1 **Introduction** - This first chapter introduces the reader to the developed work by presenting the motivation context and the document structure.
2 **Value analysis** - presents the value analysis of this work, containing the different steps of the new concept development model of Peter Koen and a business model canvas of the project.
3 **State of the art** - describes different crucial concepts related to this work that may help the readers understand the following chapters. The most recent stage in microservices end to end testing research is described and compared, along with related technologies.

4 **Study of Service Virtualization Tools in E2E Testing** - the design of the performed research is defined and justified. Also, the systematic mapping study and participant observation study are described, and the results analysed.

5 **Experts' Evaluation** - evaluates the quality of the final work using an industry questionnaire answered by experienced professionals of the field and hypothesis testing using the questionnaire provided data.

6 **Conclusions** – describes the conclusions obtained with the outputs of this work. In this chapter, the achieved objectives are described along with the difficulties faced during this project, contributions of the accomplished work and future work that can be done or continued in this topic.

2 Value Analysis

This chapter describes the value analysis of this work. On the following sections, the value analysis will be supported using the New Concept Development (NCD) model of Peter Koen (12). Furthermore, the value proposition will also be described and illustrated by a business model canvas.

2.1 Introduction

Value analysis is a systematic, formal, and organised process of analysis and evaluation (13) of possible solutions to a specific problem with the purpose of improving the value of a product. Therefore, this chapter has the objective of analysing the value for the customer this work creates.

To create value, this analysis verifies if the product meets the needs of the customer and increases the product value by reducing the costs and/or improving product performance. Reducing costs that bring no benefit to the customer and that do not have any impact on the product performance naturally increase the profit and therefore the value provided by the product.

2.2 New Concept Development Model

The NCD model was developed to define best practices in the innovation process of creating or establishing a product. This model provides a method to improve this process by defining a universal language that distinguishes the different stages of an iterative process of innovation (12).

The model consists of three key components:

- Five controllable key activity elements

    - Opportunity identification.
    - Opportunity analysis.
    - Idea generation and enrichment.
    - Idea selection.
    - Concept definition
- The engine that powers the elements (leadership, culture, and business strategy).
- Influencing factors, which affect the innovation process and cannot be controlled by the corporation (organisational capabilities, the outside world, and the enabling sciences).

This model is a relationship model and not a linear process. This means that ideas and concepts can iterate and move back and forwards across the five key elements, like the circular shape and the arrows between the key elements suggest.

Furthermore, the engine "represents senior and executive-level management support and powers the five elements of the NCD model" (12). The engine and the five key elements are influenced by the base of the circle, the influencing factors. Finally, the arrows indicate that projects begin at *Opportunity Identification* or *Idea Generation & Enrichment* but only leave the model after the *Concept Definition*.

"The influencing factors are the corporation's organisational capabilities, customer and competitor influences, the outside world's influences, and the depth and strength of enabling sciences and technology" (12).

The engine consists of leadership, culture, and business strategy and "sets the environment for successful innovation" (12).

2.3 Opportunity identification

This element has the objective of identifying opportunities that might be pursued. They can be a possibility to capture competitive advantage, or a means to simplify operations, speed them up, or reduce their cost (12). Opportunity identification may come from an individual that recognises an unmet customer need or a problem to be solved.

One of the main techniques used to identify opportunities is technology trend analysis, which consists of gathering information regarding technological trends and defining opportunities of process or product improvement that may arise from it. This was the technique used in this work.

Like defined in the context section of the previous chapter, the technology trend in using the microservices architecture as lead to an increase in the difficult and yet an increase in the demand for reliable end-to-end tests. This happens because since must systems rely on message brokers to communicate there is a need to ensure that the messages passed on through maintain a quality status and satisfy requirements. This mainly affects the maintainability and scalability of the system but also has an impact on the software development lifecycle and on organisational flexibility as the time needed to release new features increases.

Therefore, an opportunity is identified in this testing process. If the costs of the microservices architecture testing are reduced, then the microservices architectural testing style becomes a more appealing solution for companies. It enables them to increase the maintainability and scalability of their systems while reducing the time-to-market of new features implemented in their software systems, at a reduced cost.

2.4 Opportunity analysis

This stage of the NCD model has the objective of analysing the identified opportunity to confirm its viability. For that, additional information is required so that the opportunity identified can be defined as a specific business and technology opportunity. This involves making early and often uncertain technology and market assessments. The technique used may be the same used on the opportunity identification stage, but while it was used with the objective to determine if an opportunity existed, now more resources are expended so that the opportunity is defined with further detail to verify its appropriateness and attractiveness (12). The opportunity identified in the previous section is therefore analysed so that it is possible to understand it better and the possibilities of value it may provide.

Since the official definition of microservices, there are multiple reports of end-to-end testing processes, systematic literature reviews regarding the subject and studies of best practices and patterns for the microservices architecture testing styles and for the end-to-end testing process. These documents report multiple common problems that still have no explicit or linear solution. They include fundamental intrinsic issues of the microservices architecture style, like dealing with distributed transactions across microservices, data synchronisation and consistency across multiple databases and the loss of messages regarding the messages broker.

Issues of distributed systems, for instance, network-related problems, are another inherent problem of the microservices field.

This analysis defines multiple technical challenges that may be addressed regarding the microservices architecture testing. All of them constitute an opportunity that can bring value to the customer.

2.5 Idea Generation and Enrichment

This key element of the NCD model may be a formal process with the objective of generating new or modified ideas for the identified opportunity. It consists of the birth, development, and maturation of a concrete idea (12).

On this work, the individual brainstorming technique (14) was used to generate and enrich ideas for the identified opportunity. From the brainstorming sessions, the following enumerated ideas were made:

1. **Implement the end-to-end tests of a specific microservices architecture project.** The objective of this idea is to identify the problems of this kind of implementation through practical experience, defining solutions for the challenges faced.
2. **Define a technical guide with best practices, conventions, and guidelines for microservices end to end tests**. This idea has the purpose of defining technical guidelines to avoid some of the common problems of the microservices architecture testing process, or at least reduce their impact and costs.
3. **Use static analysis to inspect the existent microservice system and generate suggestions for the tests of each of the microservices**. This idea provides a tool to automatically define the tests of each one of the components of a microservices architecture, based on an existent system.
4. **Use model-driven software engineering (MDSE) to create the microservices architecture system end to end tests based on a defined metamodel**. This idea uses the MDSE approach to generate a skeleton of the microservices architecture end to end tests, providing a typical structure for all the components.
5. **Analyse existing market solutions, identify problems and advantages, and implement a solution with the most suited one**. The framework or tool chosen has the objective of simplifying the process of developing automated end to end tests in the microservices architecture.

2.6 Idea Selection

Idea Selection is the element of NCD where the idea with the most value is selected. This process is affected by insights from the influencing factors and directives from the engine (12).

The selection process can be just an individual choice between many self-generated options. Usually, this stage is sustained by early personal judgements, with only the idea itself to consider and without more information. Some techniques traditionally used on this process and applied on this work are technical success probability and the strategic fit.

Analytic Hierarchy Process (AHP) is a method to help on the decision-making process. To explain complex decision-making problems, the method models the problem into hierarchical elements. The hierarchy levels are the primary objective, the criteria that define a right decision, and the alternatives that are being considered (15). Therefore, this method was chosen to select the idea that brings the most value from the alternatives described in the Idea Generation and Enrichment section.

Following the AHP method, the hierarchy tree presented in Figure 1 was developed. The first layer defines the main objective of this work that the selected idea should help achieve.

Furthermore, the middle layer consists of the following criteria used to evaluate each one of the ideas and choose the best one accordingly.

- **Time Restrictions** – If the idea presents time restrictions as this work has a pre-defined due date and is limited by it.
- **Infrastructure Restrictions** – Mandatory Infrastructure requirements for the idea success. The infrastructure available for this work is limited.
- **Current Relevancy** – Current value for the stakeholder.
- **Technical success probability** – If the idea is achievable with the restrictions of this work with a high success probability.

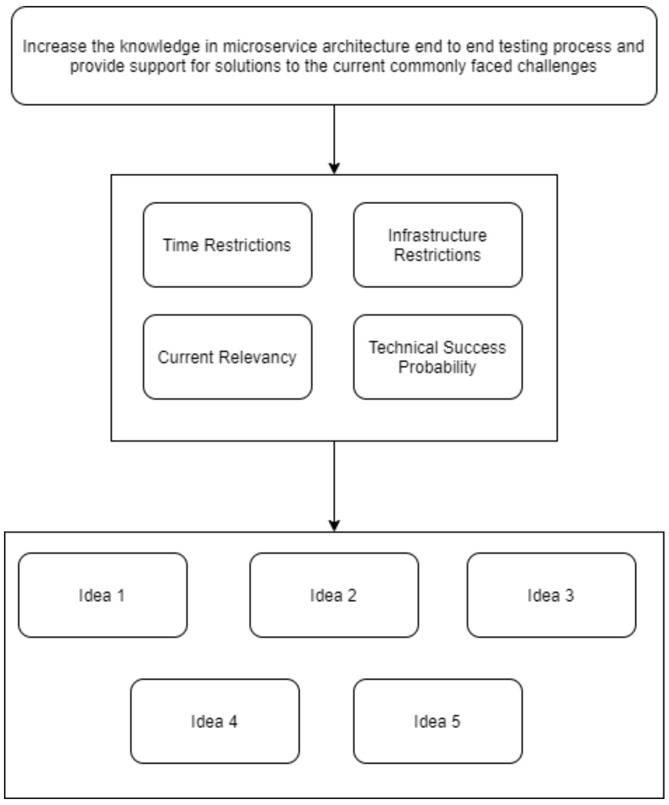

Increase the knowledge in microservice architecture end to end testing process and provide support for solutions to the current commonly faced challenges
Time Restrictions
Infrastructure Restrictions
Current Relevancy
Technical Success Probability
Idea 1
Idea 2
Idea 3
Idea 4
Idea 5

Figure 1 – AHP hierarchical model tree.

Finally, on the lowest hierarchical level, the five ideas described in the Idea Generation and Enrichment section are presented.

Based on these criteria, it is possible to evaluate and select the best idea to achieve the main objective. Table 1 below describes the considered weight for each of the criterions following the AHP scale.

To accomplish the primary goal of this work, it is essential that the idea is currently relevant and presents the current value. Given the type of project described in this document, there are also some time and infrastructure restrictions that should be considered, as well as the probability of technical success of the chosen idea.

Table 1 – AHP evaluation table.

Evaluation Criteria	**Time Restrictions**	**Infrastructure Restrictions**	**Current Relevancy**	**Technical success probability**
Time Restrictions	1	2	0.33	0.50
Infrastructure Restrictions	0.50	1	0.25	0.33
Current Relevancy	3	4	1	3
Technical success probability	2	3	0.33	1
Sum	6.5	10	1.91	4.83

After defining the weight of each criteria using a pairwise comparison on the table above, the matrix must be normalised to retrieve the priorities of each measure by calculating the mean value of each row. To generate the normalised matrix each cell should be divided the total of the correspondent column. This is presented in Table 2.

Table 2 – AHP normalized matrix.

Evaluation Criteria	**Time Restrictions**	**Infrastructure Restrictions**	**Current Relevancy**	**Technical success probability**	**Mean**
Time Restrictions	0.154	0.2	0.173	0.104	0.158
Infrastructure Restrictions	0.077	0.1	0.131	0.068	0.094
Current Relevancy	0.462	0.4	0.524	0.621	0.501
Technical success probability	0.308	0.3	0.173	0.207	0.247
Sum	1	1	1	1	1

Calculated the normalised matrix, it is possible to define the priorities of each criterion for the process of idea selection. This is defined in Table 3 below.

Table 3 – AHP criteria priorities

Priority	Criterion	Rate
1	Current Relevancy	50.1%
2	Technical Success Probability	24.7%
3	Time Restrictions	15.8%
4	Infrastructure Restrictions	9.4%

The AHP method concludes that the current relevancy is the most important criterion to apply while selecting the idea with the most value. Technical success probability comes after, followed by time restrictions and then infrastructure restrictions. Therefore, we can now analyse the described ideas based on these priorities to select one.

1. **Implement the end-to-end tests of a specific microservices architecture project.** this idea is not considered currently relevant because it would require high infrastructural resources which are not available for this work.
2. **Define a technical guide with best practices, conventions, and guidelines for microservices end to end tests**. This idea may be considered currently relevant as there is an evident lack of guidelines on microservices end to end tests reported on the literature. However, it may not respect the time restrictions of this work. Furthermore, this is a highly sophisticated solution that may not be able to achieve high technical success.
3. **Use static analysis to inspect the existent microservice system and generate suggestions for the tests of each of the microservices**. There are multiple solutions and studies regarding this topic. Therefore, this idea is not considered currently relevant as it has already similar solutions on the market.
4. **Use model-driven software engineering (MDSE) to create the microservices architecture system end to end tests based on a defined metamodel**. This idea is currently relevant as it solves some of the reported problems using a different approach. However, the technical success probability of this idea may be hard to measure as it is a disruptive idea with some uncertainty level.
5. **Analyse existing market solutions, identify problems and advantages, and implement a solution with the most suited one**. This idea is currently relevant as it solves a recently reported issue and can be designed to achieve a high technical probability of success and respect the time restrictions. It may present some difficulties regarding infrastructure restrictions, but they can be surpassed.

Therefore, following the analysis and comparison of each one of the ideas, the selected plan to achieve the objectives of this work is idea 5.

2.7 Concept Definition

This project has the purpose of identifying the current state of end-to-end testing in a microservice architecture and find the best practices and solutions to use in a work environment.

The research can use methods like literature review, and industry surveys or interviews. Therefore, the main requirements are an increased knowledge of end-to-end testing and microservice architecture as well as their must common challenges and best practices. Also, the implementation of a solution to manage end-to-end testing should be provided. It should be reusable by multiple teams, providing a generic and abstract approach that can be adapted to any microservices oriented system with reduced costs.

The value of the concept defined above will be described in more detail in the following sections where a business model canvas of the solution is presented.

As mentioned in the previous sections and chapters, based on various public documents, there is a trend of companies migrating their systems to a microservices oriented architecture to be more flexible. This flexibility is related to their capacity of adapting to environmental changes or business needs (organisational agility) with inferior costs, which can be achieved with a microservices oriented architecture as it improves the maintainability of the system, as explained before. Furthermore, one of the advantages of microservices is having more flexible scalability and optimised infrastructural costs.

However, this migration makes surface other problems like the ones regarding an effective end-to-end testing process as related on public documentation.

This work intends to help solve this problem by analysing and compiling all the issues reported, identifying the most common ones, and finding the best solutions for them.

Therefore, with this work, companies will be able to test microservices with reduced costs. Also, the final system may be better engineered due to a better-quality assurance that this work may bring, and for that reason, this work can improve the maintainability, performance, reusability, and other characteristics of the system, which leads to a more resilient system.

To present this idea in a more structured way, the following Canvas model was developed in Table 4. The cost structure and revenue streams sections of Table 4 show the main reason for the value of this solution. There are almost no costs on using the developed solution, but there are many benefits like providing more resilient systems with higher maintainability, reusability, and quality assurance while reducing the overall infrastructural costs of the system.

Table 4 – Business model canvas

Key Partners -Google Scholar, ACM, IEEE, and other digital libraries. -Companies willing to participate in the study, providing support to experiments.	**Key Activities** -Systematic Literature Review to Identify common problems and solutions. -Design and implementation of a solution for the end-to-end testing with service virtualization	**Value Propositions** -Identification of the currently most common challenges faced with testing microservices architecture. This allows companies to avoid or at least be aware of these problems, leading to an overall better microservice oriented final system. -The solution provided to the distributed transactions challenge will also reduce the costs of microservice architecture adoption as there will be fewer problems to address.	**Customer Relationships** -Implementation of the final solution in an interested company.	**Customer Segments** -Companies that are interested in performing a microservice migration or solving problems that they are currently facing on microservice oriented system tests.
	Key Resources - Public documentation regarding microservices architecture and testing. - Industry knowledge of the field		**Channels** Digital Libraries, Technology blogs, Technology conferences, Companies' presentation	
Cost Structure -The knowledge provided by the study has no costs. -The solution developed may require some infrastructural costs to be used.			**Revenue Streams** - Reduced technical debt. - Reduced infrastructure costs. - Possibly faster development of new features. -Solution to some of the most common problems;	

3 State of the Art

In this chapter, the background and state of the art are presented. The first section introduces the microservice architecture which is prevalent to the case in study. The second section explains a little about service messaging, specifically about the Kafka message broker which is the tool mostly used in the environment in which the solution is being evaluated on. The third section gives more information about concepts of software testing in a general matter with a later focus on end-to-end testing follow by the fourth section which explains the concept of service virtualization which is used in end-to-end testing. The fifth section presents the possible tools used for service virtualization, which will be the focus of study in this report.

3.1 Microservice Architecture

When talking about Microservices, it is also important to explain the monolith term as well. Monolith was formally defined (2) *"A monolith is a software application whose modules cannot be executed independently."*.

In general, the "monolith" and "monolithic" terms are used for lack of a better designation to refer to a system in which the different architectural elements are together in a single executable, unit, or block (16).

Some authors still suggest that software development should begin with a monolith, but over time and with better knowledge of the system complexities it should be migrated to a Microservices oriented architecture to avoid the limitations of a monolithic architecture (17). This pattern is usually called "Monolith First".

According to Sam Newman in (18), Microservices are independently deployable services modelled around a business domain. They communicate with each other via networks, and as an architecture choice offer many options for solving the problems a developer might be facing. It follows that a microservice architecture is based on multiple collaborating microservices. They are a type of service-oriented architecture (SOA), albeit one that is opinionated about how service boundaries should be drawn, and that independent deployability is key. Microservices also have the advantage of being technology agnostic. Microservices make a form of a distributed system as they expose their information via on or more network endpoints.

As said in the previous paragraph, that independent deployability is key point of the microservice architecture. It is the idea a change can be made to a microservice and deploy it into a production environment without having to utilize any other services. To guarantee independent deployability, there is a need to ensure that the services are loosely coupled—in other words, the ability to change one service without having to change anything else.

Although a microservice is an independent component that can be deployed in isolation a microservice alone presents no value which leads to the concept of microservices architecture (2) “A microservice architecture is a distributed application where all its modules are microservices”. Therefore, a microservice architecture is defined as a distributed application in which its behaviour depends on the communication, composition, and coordination of its microservices via messages (2).

3.1.1 Microservices Benefits

Microservices independence emphasizes loose coupling and high cohesion concepts, offering different benefits. Some key features of using a microservices architecture are (18):

- **Technology Heterogeneity:** With the application composed as a set of independent and loosely coupled services we have the freedom to pick the best tool that fulfill the needs. That can be the full application stack, from the programming language to the application server. Also, since services are small, we can replace a technology with a low risk.

- **Resilience:** The key concept of resilience is to isolate failures, so it does not affect the whole system. The natural independence of the services in a microservices architecture allow to isolate key services in its own infrastructure to keep them working even with failures in other services.

- **Scalability:** Each service boundary is well defined and organized around the business capabilities. This sort of granularity allows to scale each one of the services as needed, reducing costs, and providing only the necessary resources as opposed to monolithic applications where we must scale the whole application.

- **Ease of Deployment:** In a microservices architecture we can change a single service and deploy it independently in a short time.

3.1.2 Possible Setbacks

According to (18) one of the main challenges comes from the networks which is the way different systems can communicate with each other. Communication between computers over networks is not instantaneous which causes latency to be a worry. Things get worse when we consider that these latencies will vary, which can make system behaviour unpredictable. And we also must address the fact that networks sometimes fail—packets get lost; network cables are disconnected.

Dealing with the fact that any network call can fail becomes a recurring problem, as the fact that the services could go offline for whatever reason or otherwise start behaving oddly, there is also the need to work out how to get a consistent view of data across multiple machines.

A single-process application likely reads data from a database that runs on a different machine and presents data on to a web browser. That is at least three computers in the mix there, with communication between them over networks. The difference is the extent to which monolithic systems are distributed compared to microservice architectures. As you have more computers in the mix, communicating over more networks, it is more likely to exist problems associated with distributed systems. These problems may not appear initially, but over time, as a system grows, it will likely hit most, if not all, of them.

As a possible solution for the setbacks presented relating to a microservice architecture, message brokers are utilized (7) to maintain the messaging flux between services and not lose messages when services are down. The topic of message brokers is explorer in more detail in the next section.

3.2 Kafka Messaging

As was mentioned in the first chapter, section 1.1 and 1.3 of this work, Kafka is the message broker by choice for the environment in which the solution should be tested on.

According to Apache Kafka's website (19) event streaming is the digital equivalent of the human body's central nervous system. It is the technological foundation for the 'always-on' world where businesses are increasingly software-defined and automated, and where the user of software is more software.

Technically speaking, event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

In an overview in (20) Apache Kafka provides a publish-subscribe messaging service, where a producer (publisher) sends messages to a Kafka topic in the Kafka cluster (message brokers), and a consumer (subscriber) reads messages from the subscribed topic. A topic is a logical category of messages, for instance the producer will send the logs of a web server access records to the *serverLogs* topic, while the records of the querying records on a website will be sent to the search's topic. As depicted in Figure 2, we observe 2 producers sending messages to 2 topics in this Kafka cluster, topic A and topic B, respectively.

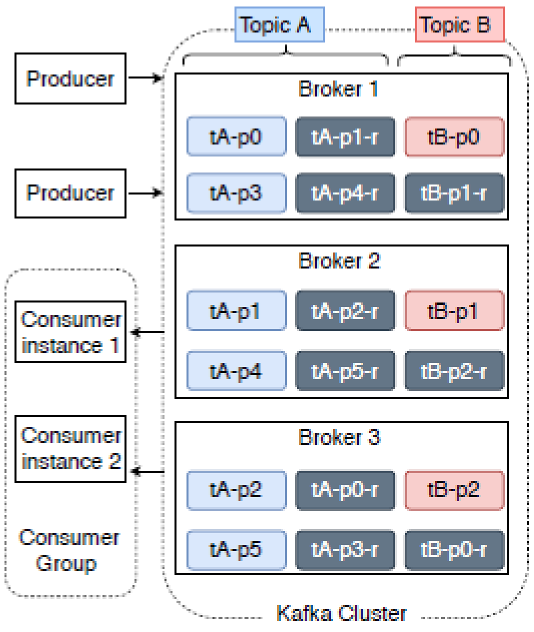


Figure 2 – Overview of Apache Kafka (20).

A topic may be stored in one or more partitions, which are the physical storage of messages in the Kafka cluster. The Kafka cluster consists of several brokers (Kafka servers), and all the partitions of the same topic are distributed among the brokers. In the example in Figure 2 there are 3 brokers, and we see topic A consisting of 6 partitions while topic B with 3 partitions, which are denoted by tA-pf0-5g and tB-pf0-2g, respectively.

Each partition is physically stored on disks as a series of segment files that are written in an append-only manner, and it is replicated across the Kafka broker nodes for fault tolerance, and we denote the replica of a partition with the suffix -r in Figure 2. The messages of the leader partition tA-p0 on broker1 are replicated to the partition tA-p0-r on broker3, so once the broker1 server fails, tA-p0-r is chosen as the leader partition, which is similar for other partitions on broker1. Only the leader partition handles all reads and writes of messages with producer and consumer, which is performed in FIFO manner. Kafka uses partitions to scale a topic across many servers for producers to write messages in parallel, and to facilitate parallel reading of consumers.

We see 2 consumer instances in our example, and they belong to the same consumer group. When the consumer group is subscribed to a topic, each consumer instance in the group will in parallel fetch messages from a different subset of the partitions in the topic. A consumer instance can fetch messages from multiple partitions, while one partition must be consumed by only one consumer instance within the same consumer group. However, different consumer groups can independently consume the same messages and no coordination among consumer groups is necessary. As a result, the number of partitions controls the maximum parallelism of the consumer group, and it is obvious that the number of consumer instances in the consumer group should not exceed the number of partitions in the subscribed topic, otherwise there will be idle consumer instances.

Cloud service vendors provide Apache Kafka as a service mainly in two different modes, the full-service mode and cluster-service mode (21). The VPC provided by AWS offers full-service mode where a user can create producer clients, consumer clients and a Kafka cluster on the cloud servers, and the user only needs a laptop connecting to the console for sending commands, as depicted in Figure 3a. This mode is applicable when the upstream and downstream applications that cooperate with Kafka are also deployed on cloud services. Under many circumstances data are generated or consumed locally, and users will run Kafka producer or consumer clients on their own machines. Then the choice should be cluster-service mode where cloud vendors like Confluent and Aiven provide a Kafka cluster as a service, as shown in Figure 3b. In this mode producer clients send messages with local machines, then cloud servers distribute and store those messages for consumer clients to fetch. Normally users must run test cases on a Kafka cluster and compare the performance results under different configuration parameters. However, the Kafka cluster needs a restart every time the configuration parameter is changed and running test cases on cloud servers is time consuming and expensive (21).

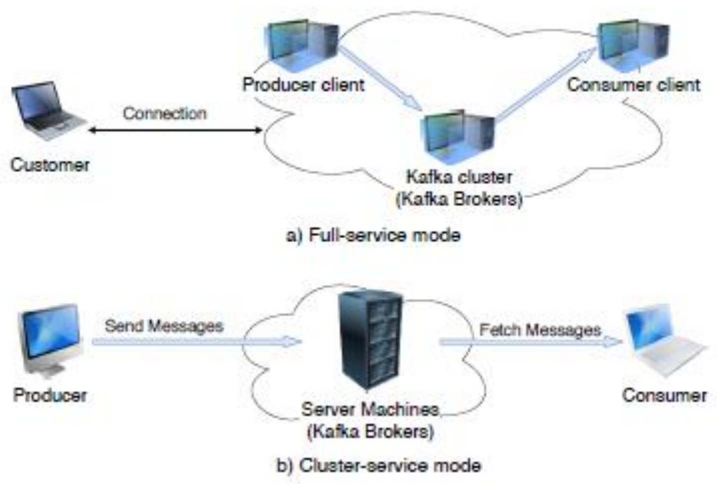


Figure 3 – Kafka as a Service (21).

3.3 Software End-to-End Tests

Software testing is the process of executing a program with the intent of finding errors (22). Software testing is mainly divided into two methods of tests: white-box and black-box. Black-box method consists of disregard the system internal behaviour, instead, the test is made considering only the input and output data. On the other hand, White-box method consider the system internal structure causing each statement of the program to be executed at least once. In software testing area there are different test levels or stages (22):

- **Unit testing:** The objective of this phase is to test each component or software unit individually and independently without considering other parts of the application.
- **Integration testing**: This level focus on testing if the components work well after they are integrated. For this test, the application components are bound together and assembled in a single artifact.
- **System testing:** The objective of this test is to use the requirements that were raised during the analysis and check if all requirements are being attended by the application.
- **Acceptance testing:** At this level, final users perform the test in the system to check if the solution delivered meets their needs.

End-to-end testing (E2E testing) refers to a software testing method that involves testing an application's workflow from beginning to end. This method basically aims to replicate real user scenarios so that the system can be validated for integration and data integrity (23).

Essentially, the test goes through every operation the application can perform to test how the application communicates with hardware, network connectivity, external dependencies, databases, and other applications. Usually, E2E testing is executed after functional and system testing is complete (22).

The diagram in the Figure 4 gives an overview of the End-to-End testing process.

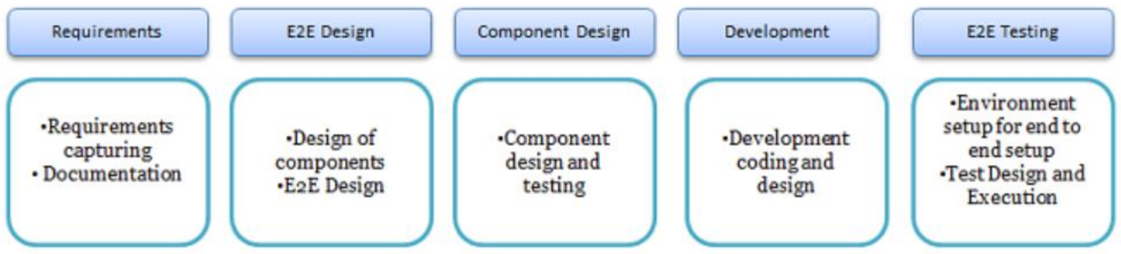


Figure 4 – Overview of the End-to-End testing process (23).

End-to-end testing has been more reliable and widely adopted because of these following benefits (23):

- Expand test coverage
- Ensure the quality of the application
- Reduce time to market
- Reduce cost
- Detect defects

Modern software systems allow subsystem interactions through advancements in technology. Whether the subsystem is the same or different from the main system, within or outside the organization, subsystem failures can cause adverse effects throughout the entire system (22).

System risks can be avoided by performing the following:

- Verifying the system flow
- Increasing test coverage areas
- Detecting issues associated with the subsystem

E2E testing is regularly conducted on finished products and systems, making each review a test of the completed system. A second test will take place if the system does not output what is expected or if a problem is found. In this case, the team will have to record and analyze the data to determine the issue's origin; then fix and re-test them (23).

Detecting defects in a complex workflow entails challenges. The two major ones are explained below (6):

- **Creating workflows**: To examine an app's workflow, test cases in an E2E test suite must be run in a particular sequence. This sequence must match the path of the end-user as they navigate through the app. Creating test suites to match this workflow can be taxing, especially since they usually involve creating and running thousands of tests.
- **Accessing Test Environment:** It is easy enough to test apps in dev environments. However, every application must be tested in client or production environments. Chances are, that production environments are not always available for testing. Even when they are, testers must install local agents and log into virtual machines. Testers also must prepare for and prevent issues like system updates that might interrupt test execution. The best way to access an ideal test environment is to test on a real device cloud.

E2E can be divided into two different methods (23):

- **Horizontal E2E testing:** A commonly used method occurring horizontally across the context of multiple applications and easily takes place in a single ERP (Enterprise Resource Planning) application like for example a Web-based application of an e-commerce system includes accounts, product inventory status, and shipping details.
- **Vertical E2E testing:** This method refers to testing in layers, meaning that tests happen in sequential, hierarchical order. To ensure quality, each component of a system or product is tested from start to finish. Vertical testing is often used to test critical components of a complex computing system which does not typically involve users or interfaces.

3.4 Service Virtualization

Service virtualization involves the creation and deployment of “virtual services” that emulate the specific behaviour of the dependent components or services and facilitate the testing of the SUT without requiring access to the actual services. The formulation of such service models in the service virtualization environment is relatively more manageable than in mocking objects where its internal components require re-implementation for every testing scenario. The creation of virtual services can be done in two ways: manually defining service models by an expert with the required knowledge of underlying services (24); and automatically infer service models through extracting the relevant knowledge from the service interaction traces and utilizing them in generating responses. The behaviour/characteristics of a virtual service depends on the dependent/connected services of an enterprise system, and it emulates the specific behaviour of the dependent service, which is required to execute the development and testing tasks. For example, virtualizing a web service requires listening for a request message over HTTP, JMS, or MQ and then returning a response message. Virtualizing a database application means that the service model can parse an SQL query and then return the data source rows according to the query request as the response message (25).

Service virtualization creates an asset known as a Virtual Service (VS), which is a system-generated software object that contains the instructions for a plausible “conversation” between any two systems (26). The fundamental process works this way (Figure 5):

1. **Capture**: a “listener” is deployed wherever there is traffic or messages flowing between any two systems. Generally, the listener records data between the current version of the application under development and a downstream system that we seek to simulate.

2. **Model**: Here the service virtualization solution takes the captured data and correlates it into a VS, which is a “conversation” of appropriate requests and responses that is plausible enough for use in development and testing. Sophisticated algorithms are employed to do this correctly.

3. **Simulate**: the development team can now use the deployed virtual services on-demand as a stand-in for the downstream systems, which will respond to requests with appropriate data just as the real thing would, except with more predictable behaviors and much lower setup/teardown cost.

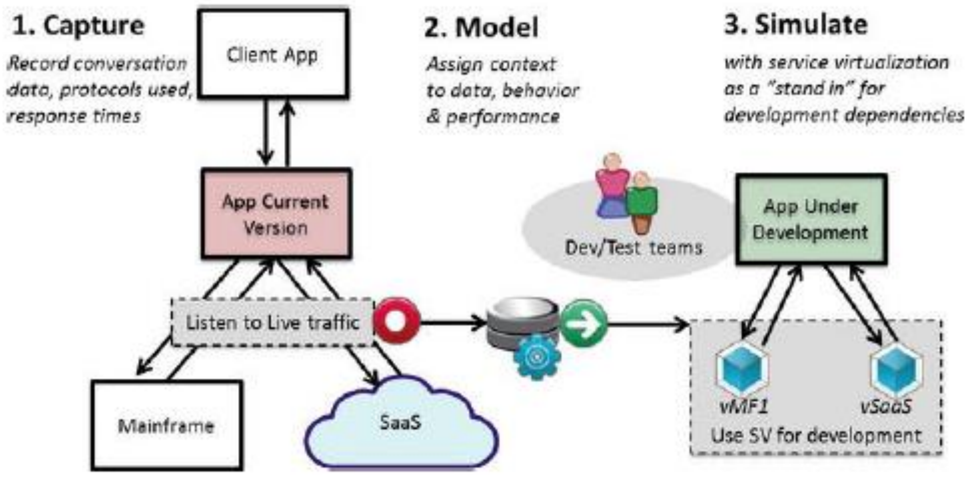

1. Capture
Record conversation data, protocols used, response times
Client App
2. Model
Assign context to data, behavior & performance
3. Simulate
with service virtualization as a "stand in" for development dependencies
App Current Version
Dev/Test teams
App Under Development
Listen to Live traffic
Mainframe
SaaS
vMF1
vSaaS
Use SV for development

Figure 5 – Basic process for creating a Virtual Service (26).

3.5 Service Virtualization and E2E Tools

Over the next sections of the report, a group of selected tools of service virtualization, used in software end to end testing, will be presented in greater detail, starting with Gremlin and Hoverfly, which were mention in studies (27) that are present in section 3.6 of this report and also WireMock, Traffic Parrot and MockLab, which are relevant tools for the environment in which the work of this report will be evaluated on.

There are more existing tools, but to better improved the focus of the work and due to time restrictions, only the ones present in this section will be mention and analysed. Also, it is important to note that some the tools presented are also present in the literature analysed in the 3.6 section.

3.5.1 Gremlin

Gremlin is a framework for systematically testing the failure-handling capabilities of microservices (28). Gremlin is based on the observation that microservices are loosely coupled and thus rely on standard message-exchange patterns over the network. Gremlin allows the operator to easily design tests and executes them by manipulating interservice messages at the network layer. Gremlin supports emulation of fail-stop/crash failures, performance/omission failures, and crash-recovery failures, the most common types of failures encountered in modern-day cloud deployments.

In Gremlin, the human operator (e.g., developer or tester) writes a Gremlin recipe: a test description, written in Python, which consists of the outage scenario to be created and assertions to be checked. Assertions specify expected behaviour of microservices during the outage. An operator can orchestrate elaborate failure scenarios and validate complex application behaviours in short and easy-to-understand recipes.

Consider a simple application consisting of two HTTPbased microservices, namely ServiceA and ServiceB, where ServiceA makes API calls to ServiceB. An operator might wish to test the resiliency of ServiceA against any degradation of ServiceB, with the expectation that ServiceA would retry failed API calls no more than five times. With Gremlin, this resiliency test can be conducted using the following recipe (Code 1):

```
Overload(ServiceB)
HasBoundedRetries(ServiceA, ServiceB, 5)
```

Code 1 – Overload Test (28).

In line 1, Gremlin emulates the overloaded state of ServiceB, without impacting ServiceB. When traffic is injected into the application, ServiceA would experience delayed responses from ServiceB, or receive an HTTP error code (503 Service unavailable). The operator's expectation (when ServiceA encounters such behavior, it should restrict the number of retries to five attempts) is expressed using the assertion in line 2.

A high-level view of Gremlin architecture is shown in Figure 6. Broadly, the framework is divided into data plane and a control plane (29).

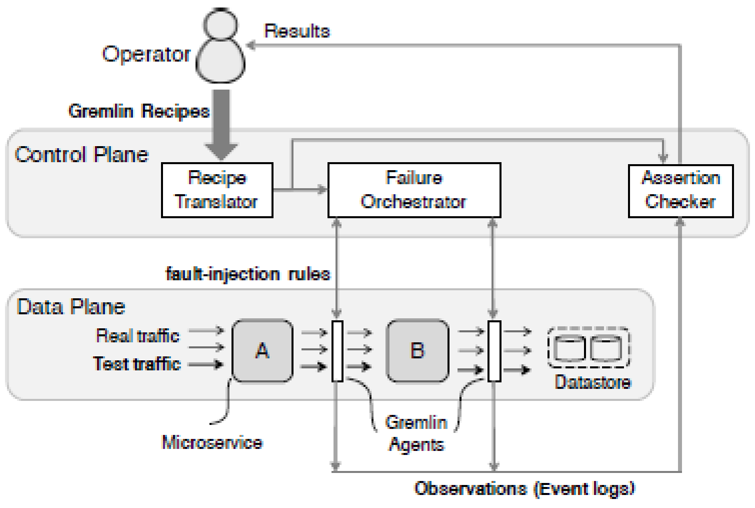


Figure 6 – High-level overview of the Gremlin framework (29).

The data plane consists of network proxies, called Gremlin agents. Microservices are configured to communicate with each other via these agents. In addition to proxying the API calls, Gremlin agents can manipulate the arguments, return values, and timing of the calls, thus acting as fault injectors. As shown in Table 5 (26), the data plane supports three primitive fault injection actions: Abort, Delay, and Modify. Using these primitives, complex failure scenarios that emulate real-world outages can be constructed. Like software defined network switches, Gremlin agents expose a well-defined interface to the control plane. The control plane uses the interface to send rules to the agents, instructing them to inspect the messages and perform fault-injection actions if a message matches a given criteria (25).

Table 5 – Interface exposed by the data plane to the control plane (29).

Interface	Mandatory Parameters	Description
Abort	Src, Dst, Error, Pattern	Abort messages from Src to Dst, where messages match pattern Pattern. Return an application-level Error code to Src
Delay	Src, Dst, Interval, Pattern	Delay forwarding of messages from Src to Dst, that match pattern Pattern, by specified Interval
Modify	Src, Dst, ReplaceBytes, Pattern	Rewrite messages from Src to Dst, that match pattern Pattern and replace matched bytes with ReplaceBytes

The control plane has three components (29):

- **Recipe Translator** - exposes a Python interface to the operator, which enables it to compose high-level failure scenarios and assertions from pre-existing recipes or directly from low-level primitives for fault injection (shown in Table 5) and assertions. The operator is also expected to provide a logical application graph: a directed graph describing the caller/callee relationship between different microservices. Internally, the translator breaks down the recipe into a set of fault-injection rules to be executed on the application's logical graph.

- **Failure Orchestrator** - sends fault-injection actions to the Gremlin data plane agents through an out-of-band control channel. Since an application might have multiple instances of any given service, the Failure Orchestrator locates and configures all physical instances of the Gremlin agents.

- **Assertion Checker** - is responsible for validating the assertions provided in the recipe. It does so by querying a centralized data store that contains event logs collected from the data and performing a variety of processing steps. To aid the operator in querying the event logs, Gremlin provides abstractions for fetching and analysing the data. The queries return a filtered list of observations from the Gremlin agents, sorted by time. Basic statistics on the requests (or replies) can be computed without requiring knowledge of the log-record structure.

3.5.2 Hoverfly

Hoverfly is an open-source product developed by SpectoLabs and written in Go whose core functionality of Hoverfly is to capture HTTP(S) traffic to create API simulations which can be used in testing (30). A Hoverfly instance can work according to two schemes (30):

1. **Proxy server**: it receives an incoming request and forwards it to the destination service. In the meantime, it waits for the response from the destination. When it arrives, the instance stores the array of request-response pairs in a JSON objects, called simulations, and forward the response to the source that made the original request.

2. **Webserver**: it receives an incoming request and looks among previously captured simulations for a pair having an analogous request. If the pair is found, the instance forwards it to the original source, otherwise it can behave differently according to the working mode of Hoverfly that has been set.

These schemes form the basis for service virtualization (30). Each time Hoverfly receives a request, rather than forwarding it on to the real API, it will respond instead. To make the service virtualization more realistic, it is also possible to simulate network latency, by applying delays to responses based on URL pattern matching or HTTP method.

Hoverfly stores in-memory captured traffic as a JSON object, called simulations, that follows the Hoverfly Simulation schema (31). Simulations consists essentially of request-response pairs.

Hoverfly can work in different way, accordingly its mode (30):

- **Capture Mode:** It is an "as a proxy" mode and the Hoverfly instance is placed in the middle of a client-server application, saving simulations. When in this mode, it is checked the binary flag stateful, which determines the instance behavior when there is an incoming request-response pair (31):
    - if stateful is true, all pairs are saved, without further checks.
    - if stateful is false, for each pair it is checked whether the request is not already present. It is not, the pair is saved, otherwise is checked a further binary flag, called *overwriteDuplicate*. If it is true, the existing pair is replaced by the new one, otherwise nothing happen.
- **Simulate Mode:** It is an "as a webserver" mode and the Hoverfly instance represent the destination server. When a new request arrives, the instance searches for a simulation matching the request. If it is not found, an error is returned with status code 502. For choosing the simulations to return, Hoverfly must use a matching strategy. There is only one valid matching strategy, called strongest match, which consists of calculating and assigning a matching score to each simulation, and then the one with the highest score is selected.
- **Spy Mode:** It extends the behavior of the simulate mode, with the difference that, if a matching is not found, a request to the real service is made and its response is returned to the client. When it happens, the simulation is not persisted.

3.5.3 WireMock

WireMock is an HTTP mock server (32). At its core it is web server that can be primed to serve canned responses to requests (stubbing) and that captures incoming requests so that they can be checked later (verification).

It also has an assortment of other useful features including record/playback of interactions with other APIs, injection of faults and delays, simulation of stateful behavior

It can be used as a library by any JVM application or run as a standalone process either on the same host as the system under test or a remote server.

All WireMock's features are accessible via its REST (JSON) interface and its Java API. Additionally, stubs can be configured via JSON files.

WireMock is distributed in two ways - a standard JAR containing just WireMock, and a standalone fat JAR containing WireMock plus all its dependencies.

Most of the standalone JAR's dependencies are shaded which is, they are hidden in alternative packages. This allows WireMock to be used in projects with conflicting versions of its dependencies. The standalone JAR is also runnable (32).

Additionally, versions of these JARs are distributed for both Java 7 and Java 8+.

The Java 7 distribution is aimed primarily at Android developers and enterprise Java teams still using JRE7. Some of its dependencies are not set to the latest versions (32).

The Java 8+ build endeavors to track the latest version of all its major dependencies. This is usually the version you should choose by default.

As mentioned before, the WireMock server can be run in its own process, and configured via the Java API, JSON over HTTP or JSON files. Once the standalone JAR has been downloaded, it can run it simply by execution the command in Code 2.

```
$ java -jar wiremock-standalone-2.27.2.jar
```

Code 2 - Standalone execution command (32).

To create a stub via the JSON API, the Code 3 document can either be posted to http://<host>:<port>/__admin/mappings or placed in a file with a .json extension under the mappings directory (32).

```
{

    "request": {

        "method": "GET",
        "url": "/some/thing"
    },
    "response": {
        "status": 200,
        "body": "Hello world!",
        "headers": {
            "Content-Type": "text/plain"
        }
    }
}
```

Code 3 – Json File with Stub Code (32).

3.5.4 MockLab

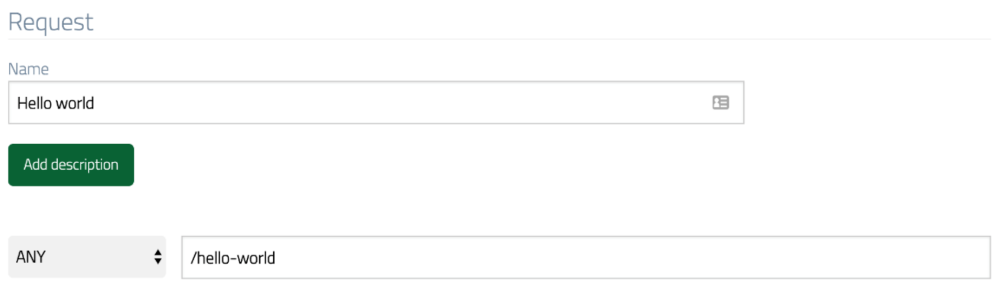
To continue from the previous section where WireMock is presented, this one will follow up with MockLab. MockLab is an API simulator built on WireMock (33). It possesses a user interface to facilitate the user's work and provides several ways to set up a mock API:

- **Manually via the web UI** - After signup, an example mock API is created, showcasing different MockLab features. This can be edited or added more stubs to experiment with MockLab's capabilities.
- **Swagger or OpenAPI specification import** - Swaggerhub users can integrate with MockLab via a webhook, so that the mock API will be updated each time a change is saved.
- **Record traffic to and from another API** - a mock of an existing API which is accessible over the internet can be configured in MockLab to proxy (forward) traffic to it and record requests as stubs.
- **Import an existing project from WireMock -** MockLab uses WireMock as its underlying engine, so mock APIs created within WireMock can be directly imported into MockLab (and vice versa). This can be useful when there is a need to record APIs that are only accessible inside an organisation or from a private network, or if there are existing projects utilising WireMock that must be hosted in the cloud.
- **Automate via the REST API or WireMock -** MockLab enables this approach by making all features available via its REST APIs. The provisioning API supports the creation,

querying and deletion of mock APIs. The mocking API supports configuration of an individual mock API, including stub create/update/delete, request log querying and verification and more. It is 100% compatible with WireMock's API and can therefore be used with any WireMock client library.

MockLab at its core, like WireMock which underpins it is an HTTP stubbing tool. This means that it can be configured to return specific canned responses depending on the request. This can be a simple as just matching the URL, right up to a combination of URL, header and body matches using regexes, JSONPath, XPath and others (33).

To create a basic Stub the mock API to work in must be select, then navigate to the Stubs page and click on the button name "+New". The URL field from the default value to the one which it will be worked on (Figure 7).


Request

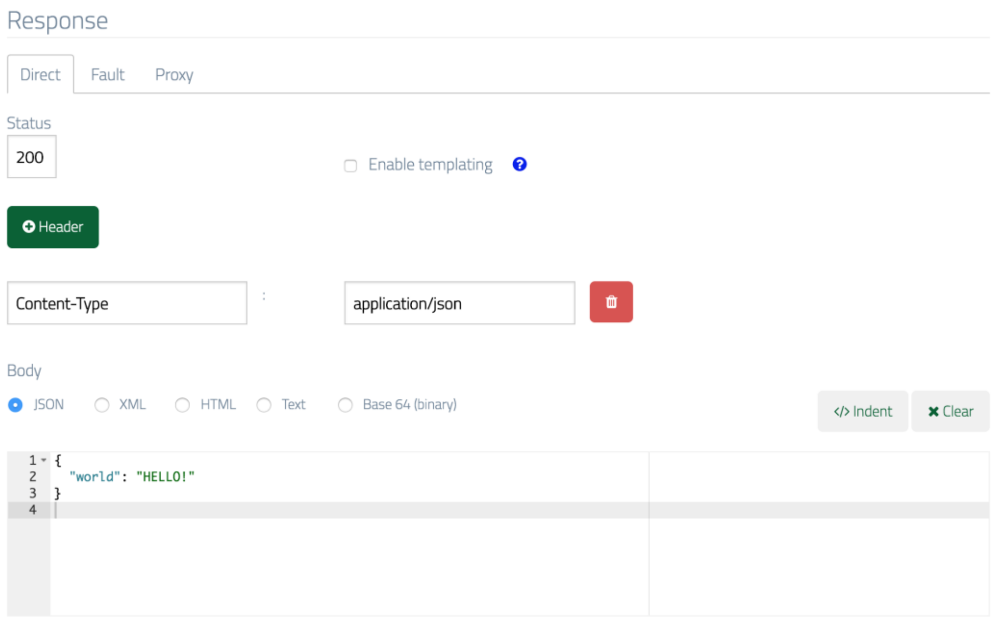
Name: Hello world

Add description

ANY /hello-world

Figure 7 – MockLab Request Section (33).

In the Response section (Figure 8), HTTP status must be settled, headers and body text as well. Typically, it is a good idea to send a Content-Type header in HTTP responses, so one can be added by clicking the "+Header" button and setting "Content-Type" to "application/json".


Response

Direct Fault Proxy

Status: 200

Enable templating

Header

Content-Type : application/json

Body: [✓ JSON] XML HTML Text Base 64 (binary)

Indent Clear

```
{
  "world": "HELLO!"
}
```

Figure 8 – MockLab Response Section (33).

The request must be saved and then the stub should be ready for testing (33). In a browser, at for example *http://<your-subdomain>.mocklab.io/hello-world* the body text “Hello World!!!” that was entered into the body text box should be seen.

3.5.5 Traffic Parrot

Traffic Parrot is an API mocking and service virtualization tool (34). It simulates APIs and services so that you can test your microservice without having to worry about test data set up or environment availability.

Traffic Parrot is specifically designed to maximize developer and tester productivity when working in autonomous or cross-functional product teams. Small footprint (less than 50MB of disk space), lightweight but powerful, supporting HTTP, JMS, IBM MQ, File transfers, gRPC and more.

Traffic Parrot supports a method for grouping together virtual service mappings to form a scenario. For example, to group together mappings used for different types of testing like integration, manual and performance tests.

The default scenario stores mappings on the filesystem under the directory specified in *trafficparrot.properties* by the *trafficparrot.virtualservice.trafficFilesRootUrl* property. For example, the mappings and __files directories which are used for HTTP mappings in the default scenario.

All other scenarios are placed in the scenarios directory, which is also found under the configured root folder. For example, the *scenarios/Example/mappings* and *scenarios/Example/__files* which are used for HTTP mappings in the scenario named Example.

Once some files are placed under a scenario directory, they can be seeming in the navigation bar dropdown (Figure 9) (34).

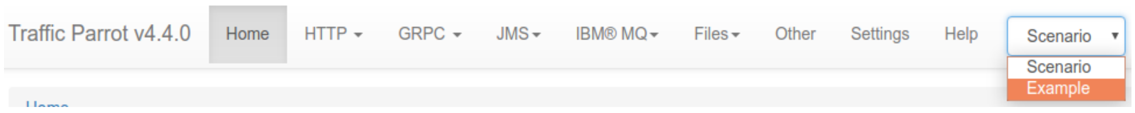


Figure 9 – Navigation Bar dropdown (34)

Clicking on a scenario name will activate that scenario. The scenario directory on the file system will be used as the source of mappings until Traffic Parrot is shut down. The default scenario is activated when Traffic Parrot starts up.

The Traffic Parrot web console and virtual service can be configured by editing property values in *trafficparrot.properties* file or by passing parameters to the start script (34).

The *trafficparrot.properties* is located in the main Traffic Parrot directory. To edit the file, it must be opened in a text editor and update the property values. The names of properties should be self-explanatory.

Traffic Parrot can also be configured by passing arguments to the start script. This will override properties defined in the *trafficparrot.properties* file (Code 4).

```
start.cmd trafficparrot.gui.http.port=20000
trafficparrot.virtualservice.http.port=20001
```

Code 4 – Passing values to the start script (34).

The list of all available properties that can be configured is available in the *trafficparrot.properties* file which is located in the main Traffic Parrot directory.

Some properties may only be changed by passing values to the start script. For example, outbound HTTPS certificate configuration and outbound HTTP proxy configuration may only be specified as start script values (34).

3.6 Microservice E2E Testing Automation Studies

In this section of the report, some existing studies related to the concept of testing microservice architecture systems will be presented.

3.6.1 Comparison of runtime testing tools for microservices

According to (27) the testing of microservices applications continues to be challenging due to the added complexity of network communication between the collaborating services. That paper provides a comparison of several open-source tools used to support the testing of microservices, which can be found in Table 6.

Table 6 - Tools used to support the testing of Microservices (27).

Name	**Interface Method**	**Implementation**	**Platform**	**Test Case Language(s)**	**Testing Objectives/Support**	**Testing Strategies**
Docker Compose Rule	HTTP Requests	Library	JVM	Java	Regression	Integration, System
Gremlin	HTTP Requests	Side-car + Proxy	Linux, Python Runtime Environment (PRE)	Linux Scripting Language, Python	End-to-End	Component, Integration, System
Hoverfly	HTTP Requests	Side-car + Proxy	JVM, PRE, MacOS, Windows, Linux	Java, Python, Scripting Language	End-to-End	Component, Integration
Minikube	CLI	Container's Orchestrator	Linux, Windows, MacOS	Scripting Language	Test Harness	Component, Integration
Telepresence	Two-way proxy	CLI	Windows, Linux	Any	Test Harness	Component

The polyglot testbed microservices system called *Rideshare*, was used in the paper to evaluate the tools mention on Table 6. The system implements a cluster of microservices for a ridesharing application that allows passengers to request a ride from a list of available drivers going from a pickup address to a destination address. The description of the application includes a high-level architecture of the system and a component diagram of the system.

The Rideshare application uses the microservices architectural pattern consisting of a front-end, backing services and domain services. The high-level architecture of the Rideshare application is shown in Figure 10.

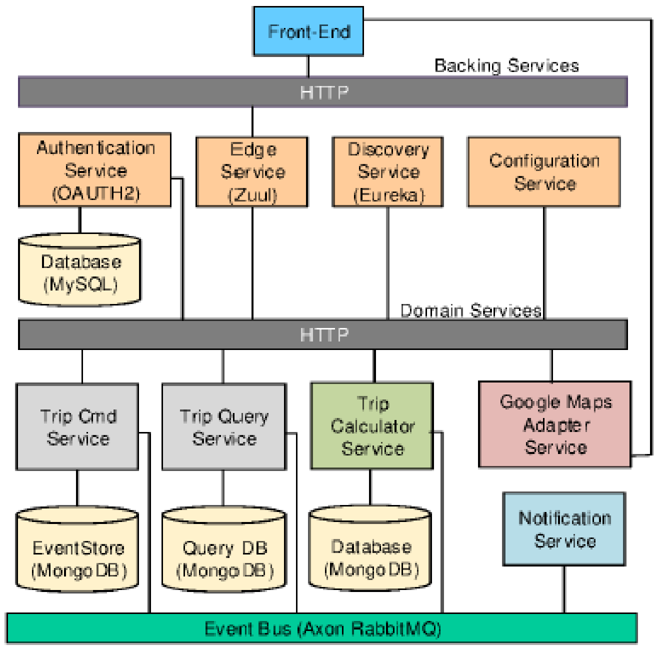


Figure 10 – Ridesharing testbed application (27).

The front-end is a web user interface that provides user interfaces for both the passenger and the driver. These interfaces include screens for the requirements stated in the previous paragraph. The back-end services are the generic services to support running a microservices application and include the following:

- Authentication - validates user credentials.
- Edge - is a proxy that provides communication between the front-end and the domain services.
- Discovery - provides a directory and lookup functionality of the active microservices.
- Configuration Services - provides a centralized repository of files to support configuration of the domain services.

The Front-end of the Rideshare application is implemented using AngularJS. The backing services were developed with Java Spring Boot framework. The Edge service is an instance of Zuul that performs proxy communication as previously mentioned, in addition to the proxy tasks the Edge service also provides load balancing and fault tolerance. The Discovery service implements Eureka, which is a REST (Representational State Transfer) based service that provides support for locating other active services in the network. The Configuration service provides remote configuration capabilities to Spring Boot applications and finally the Authentication service is an OAuth2 implementation that authorizes applications on behalf of an authenticated user.

The domain services are implemented as follows: Notification uses RabbitMQ as a message broker that facilitates asynchronous communication between all the services. Google Maps Adapter connects to the Google Maps Platform to retrieve details of locations. The Trip Calculator is implemented in Golang and uses a MongoDB database. The other services implement Command and Query Responsibility Segregation (CQRS), that is, every service is split into two smaller services, one that takes care of the command executions using an event bus implemented with RabbitMQ, and another one for queries.

The results of the tests made in (27) can be found in Table 7.

Table 7 – Execution times to run 5 system test cases each iterated 5 times (27).

Tool	**Total Time (sec.)**	**Mean (sec.)**	**Std. Dev.**	**Max. (sec.)**	**Min. (sec.)**
Docker CE	5.30	0.11	0.23	0.88	< 0.01
Gremlin	6.36	0.13	0.28	1.19	< 0.01
Hoverfly	5.57	0.11	0.27	1.15	< 0.01
Minikube	14.85	0.30	0.43	2.17	< 0.01
Telepresence	1003.22	20.06	40.92	123.13	0.05

The columns in the table from left to right are the total time, the arithmetic mean, standard deviation, maximum time, and the minimum time for all the iterations. Since the Rideshare application runs on Docker the execution times in the first row of the table can be considered the baseline for running the Rideshare application. Telepresence takes the longest time to run all iterations of the system test cases with a total of 1,003.22 seconds. The percentage increase of execution time of Telepresence over Docker CE is 18,842% while Hoverfly has the least percentage increase over Docker CE, 5.1%. Gremlin has an increase of 20.11% and Minikube 180.35% over Docker CE.

The tools used in the study represent test harness tools (Minikube and Telepresence) and end-to-end testing tools (Gremlin and Hoverfly), on which, special attention should be provided for the last ones. The authors in (27) mention that the particularly the high execution times observed for Minikube and Telepresence were unexpected, since both are test harnesses. It is inferred that these high executions times are related to the communication setup with the services to be tested. This thesis is supported by the fact that Hoverfly has execution times close to Docker CE and requires manual changes to the system test cases to use the proxy service provided by Hoverfly.

3.6.2 Automated Testing for Provisioning Systems of Complex Cloud Products

The goals of the work in (9) were to design an efficient testing strategy that considers a microservices architecture with infrastructure provisioning capabilities while integrating it in a Continuous Integration (CI)/Continuous Deployment (CD) pipeline.

The solution devised in this dissertation was tested against a set of prototypes, developed alongside the strategy definition, that emulate the functionality and limitations of the orchestrator system to prove its applicability in the *OutSystems* context. The solution was able to reduce the exacerbated feedback loop by applying a pyramid distribution to the implemented test levels and by using virtualization to replace some dependencies in the testing stage.

The created strategy focused on having a clear architectural notion of the components under test to provide a better mapping between the components and the errors to detect at each level. The strategy encompasses multiple types of tests to guarantee coverage of the system, and priorities using lower-level tests whenever possible to provide fast, focused, and reliable feedback.

3.6.2.1 Microservices Testing Strategy

The prototype used in (9) exercised the most critical dependencies of the orchestrator system. The design of the prototype revolved around defining microservices that performed cloud operations commonly used in the orchestrator system. The microservices were developed using Spring Boot and Java.

Understanding proper responsibilities division, and how the design of the service influenced the definition of the tests, was a clear indicator of the need to design the microservices following a layered architecture, with clear boundaries and well-defined responsibilities (9).

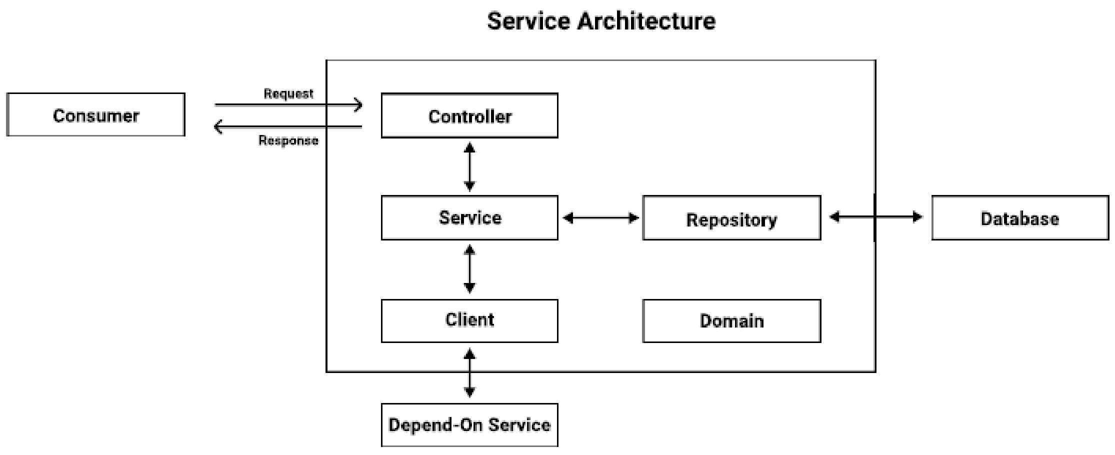


Figure 11 – Microservice architecture schematic (9).

Figure 11 represents the general architecture of the implemented microservices. The controller classes are the entry point of the service are responsible for validating and routing the incoming requests. The service layer encompasses the business logic specific to the microservice and communicates with the clients and existing repositories. The client classes create an abstraction layer for all the communications with other services of which we are dependent, while the repositories make the bridge to a persistence solution. The domain encapsulates the entities used across the service (9).

This architecture allows the better separation of concerns and is a better fit for the testing needs. Most implemented services follow this architecture model, but not all are required to have all the presented components. Stateless services do not need repositories and therefore databases, while some services contain trivial business logic that makes the service layer just boilerplate. When that is the case, the controller can communicate directly with the client class, for example (9).

3.6.2.2 End-to-End Testing

The goal of the end-to-end tests is to understand if the system delivers business-critical functionality. The system is the composition of the microservices with all the required dependencies.

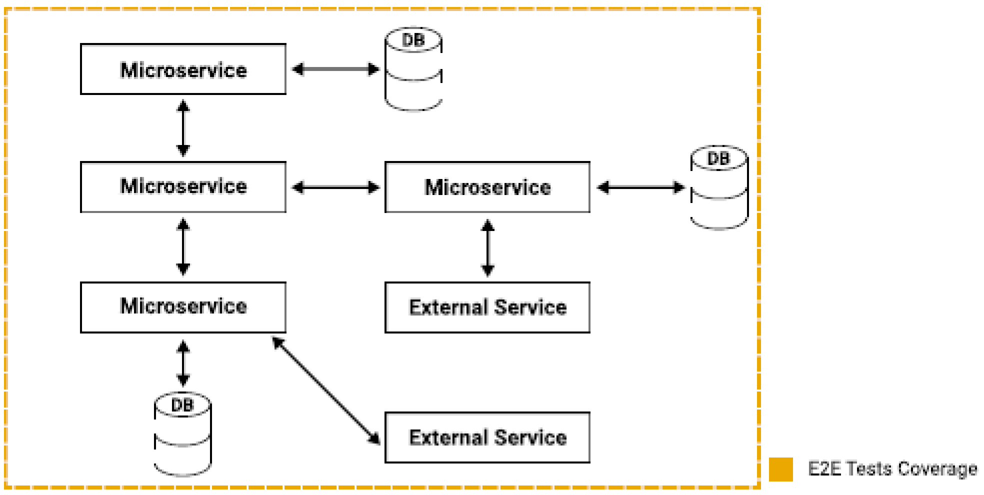


Figure 12 – End-to-End tests coverage (9).

Figure 12 portrays the coverage attained with the definition of the end-to-end tests.

End-to-end tests are usually flaky due to their complex nature as they have many moving parts (9). Beyond their flaky nature, the provided feedback by this type of tests is usually not very accurate or precise, which means a failure in this type of tests demands a much bigger debugging effort.

Despite their flaws, this type of tests is beneficial to validate that the system can deliver the expected value. The definition of end-to-end tests aimed to minimise the flakiness while ensuring the critical business paths worked as expected.

This translated into grouping microservices together into a specific domain which means reducing the amount of moving parts and reducing the possible points of failure that are usually present in highly distributed systems.

The other high value of end-to-end tests is that they are much more expressive in the sense that they can be shown to managers and decision-makers as a representation of critical user journeys (9).

Because this is a significant advantage of end-to-end tests, the solution defined the critical user journeys and tested the commonly denominated happy paths. The definition of the tests ensures that the different pieces can work together to deliver the needed functionality. Just like with component tests, the remainder non-happy paths were already tested in the lower levels of the pyramid with much narrower focus. Covering those cases in the lower levels instead means the coverage of those cases is much cheaper.

By grouping multiple microservices to define end-to-end tests and covering only the most critical paths, we reduce the number of tests of this type and address the considerable time cost usually associated with this type of testing (9).

3.6.3 End-to-End Regression Testing for Distributed Systems

The study in (35) describes a framework for regression testing that bridges a gap between local ad-hoc experiments and end-to-end stress testing, potentially lowering the recurrence of critical defects.

The motivating example (Figure 13) described an actual faulty behavior observed in *OpenSimulator*, a 3D distributed virtual environment application. The work with *OpenSimulator* includes a more complicated case study of user login behavior, also exhibiting a long manual event chain and an unpleasant user-facing experience. Even when faults like this resolve eventually (the assets in question were downloaded after a long period of time), they are problematic for user-facing application (35).

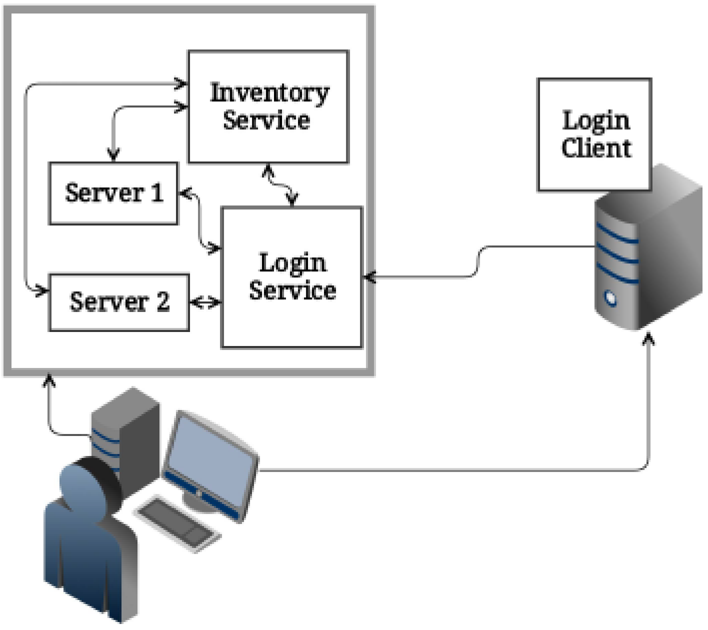


Figure 13 – Reproducing an application server defect with a small, distributed setup (35).

The testing framework was designed and implemented in C# and it was named *FlowTesting* that utilizes aspect-oriented weaving techniques to provide a controlled experiment setting for end-to-end regression testing in distributed systems. Replication, control, and automation of distributed systems experiments are critical to reducing the reappearance of previously known defect. Once a defect is understood and fixed, software system developers should have the ability to rapidly codify the minimum operation conditions that replicate that bug without having to deploy the system into production. The two-primary user-facing components are the *FlowTestRuntime* and the *WeavePoint* (35).

The *FlowTestRuntime* handles the injection of instrumentation into target libraries and executables, the execution of external test components, and messaging of target system internal state information for debugging. Its internal instrumentation engine is based on *Mono.Cecil*, an open source, bytecode weaving library for C#. The *FlowTestRuntime* instrumentation API adds a layer on top of *Mono.Cecil* to support clean, user-friendly weaving composition and to avoid poorly formed bytecode weaving. A user can instrument target code with functionality from the core C# system libraries or compose complex custom methods.

The *WeavePoint* component provides the main API for test specification. Each weave point is matched to a method in the target software system and provides an API for instrumentation on entering and exiting that method. A user can attach a debug message that they would otherwise enter manually, or an extra sleep call to trigger a time-out, all without permanently modifying the source code. Each *WeavePoint* is attached to a single *FlowTestRuntime* for the duration of a regression test.

To replicate a fault from the motivating scenario for system P and P' the following high-level steps in a unit test suite can be taken (35):

1. Initialize a *FlowTestRuntime* F for P’ with config data.
2. Add a *WeavePoint* W in *InventoryService*.
3. Specify a small sleep call for the start W.
4. *F.Write()*, to write instrumented P’ to a staging directory.
5. *F.Start()*, to launch the configured and instrumented P’.
6. Specify and start a third-party client for F.
7. Assert that the inventory progress is not complete after 15 seconds in instrumented P’ when executed from the staging directory.
8. Clean up with *F.Stop()*.

3.6.4 Efficient test execution in End-to-End testing

The study in (36) proposes a research problem and a feasible solution that looks to improve resource usage in the E2E tests, towards smart resource identification and a proper organization of its execution to achieve efficient and effective resource usage. The resources are characterized by a series of attributes that provide information about the resource and its usage during the E2E testing phase. The test cases are grouped and scheduled with the resources (i.e., parallelized in the same machine or executed in a fixed arrangement), achieving an efficient test suite execution, and reducing its total cost/time.

The proposal is composed of three processes that are the basis of the orchestration technique (36):

1. The resource identification process characterizes the resources required.
2. The grouping process aims to reduce the costs, grouping together the test cases with similar resource requirements to allow sharing and avoid unnecessary deployments.
3. The scheduling process organizes the test cases focused on achieving savings in execution time.

The proposed solution rest on the following concepts promoting efficient of E2E (36):

- **Resource Identification** characterizes the resources employed by each test case to detect which require similar resources. These resources are classified into different categories and characterized by static attributes, that describe how the resource is used and made available for the test. These include elasticity that refers to the possibility of making available the resource on the fly, the resource hierarchy if the resource can be replaced by a mock during the test phase and a Lifecycle with different phases such as set-up, execution, or disposal. Resources are also characterized by several access modes such as read, read-write, write-only or dynamic. These modes are related to how safe and idempotent are the operations performed by the test cases over it are, allowing sharing between the test cases that perform a compatible use of those resources. Resources also have dynamic attributes that change during the resource usage such as: measurable to refer to the fact that it has indicators to measure its performance, traceable that allows knowing the phase of the lifecycle, or availability according to how and the number of times that it may be instantiated, to specify how and the number of resources that would be deployed.

- **Grouping Process** optimizes the usage of resources through an aggrupation of test cases according to the way they use such resources. These test groups, called *T-Groups*, include all the required execution scaffolding, and focus on avoiding oversubscription and reducing the cost of performing the test suite.

- **Scheduling Process** in which the *T-Groups* are optimized to achieve resource savings in terms of time. The T-Groups are divided, composed by several test cases and the scaffolding required (*T-Jobs*). The *T-Jobs* test cases are parallelized and arranged according to the execution constraints ensuring that the resources are deployed in an optimal way.

3.6.5 Web Test Automation Insights from Grey Literature

The paper in (37) provides the results of a survey of the grey literature concerning best practices for end-to-end web test automation focusing on literature for functional testing of web applications with the goals to understand what best practices are suggested by practitioners and, to structure, curate, and unify the grey literature.

The study (37) grouped 706 occurrences of best practices into two main categories, namely technical aspects (80%) and business-level aspects (20%).

3.6.5.1 Technical Best Practices

Technical best practices refer to the development, maintenance, and execution of web tests. That is, testers are already equipped with test requirements, and their task is to translate such requirements into actual test code, with appropriate oracles, or to adapt existing test code to changes and extensions of such requirements, or applications' functionalities.

The most represented subcategory pertains to guidelines on how to achieve a high structural quality of the test code (29%). Particularly, the most mentioned tip is careful handling of the synchronization between the web app and the test code. Modern web applications are developed using front-end technologies in which the Document Object Model (DOM) elements are loaded dynamically by the browser and may be ready for interaction at unpredictable time intervals (37). This is a huge problem for automated testing since there is no universal mechanism to understand when a page is fully loaded and when it is possible to perform actions.

If one fails to place appropriate wait commands in the test code, the associated risks span from having pointless lengthy delays in tests' execution, to having flaky checks due to the waits being non-deterministic (37). Other best practices pertain to keep the test scripts atomic (each test method should concern only one single test scenario), using test naming conventions as well as coding rules, and focusing on reusable test code.

The second most mentioned subcategories pertain to test development and reporting of the test results (37). Concerning the former, it is suggested to implement deterministic tests by removing uncertainties that may cause tests to pass/fail non-deterministically, as well as implementing GUI-resilient tests, both positive and negative tests, and mock external services to keep the testing environment under full control. Related to the latter, developers suggest providing detailed reporting, making use of screenshots to help visually assess the bugs, and of continuous integration (CI) environments (37).

Design patterns are suggested as an effective mechanism to isolate the code's functionalities into reusable methods (12%). Developers suggest different design patterns: most of our references mention the Page Object (37), whereas lower occurrences pertain to other patterns such as Bot Pattern, AAA Pattern, and Screenplay Pattern which do not seem yet consolidated within test development.

Finally, a test suite is cost-effective if test data are of high quality (9%). In a way, a test suite is as weak as the test data it uses, which denes the overall faultfinding capability and hence cost-effectiveness of running it. Among the tips, developers suggest adopting data-driven testing techniques by parameterizing the test cases and using realistic inputs, as well as meaningful real-world combinations that the users may experience.

3.6.5.2 Business-level Best Practices

Business-level aspects are related to the practices of establishing a process that ensures the final quality of the software product and satisfies the customers as well as users. Also, it concerns aspects like resource optimization, communication, cost management, and team building (37).

The most represented subcategory pertains to Planning the process of test code development (39%). The main guidelines in this subcategory are not considering automation as a replacement for manual testing, choosing the correct/right testing tool/framework for your organization and, hiring a team of experts or a skilled automation engineer.

The second most mentioned subcategory is Design (33%), which pertains to guidelines on how design test cases and how to transform them into test code. In this subcategory, the most mentioned tips are the focusing on key user flows during test code development, that means to test mainly “happy paths” capturing typical use scenarios and so limit exception testing; creating scenarios and test cases in advance before automating test cases, i.e., having a clear understanding of what test cases to automate, indeed diving straight into automation without a proper test design can be dangerous; conducting testing from the users' perspective, e.g., by getting into the mindset of novice users (37).

Finally, it is also worth mentioning the best practice of not relying entirely on GUI test automation belonging to the Process subcategory. This is one of the main best practices a testing team should consider at first. Ideally, a test suite should be constituted by more low-level unit tests and integration tests than E2E tests running through a GUI (the practical test pyramid). Another best practice that is gaining momentum concerns reviewing the test code, similarly to production code. Test code review aims to analyze its quality and to find mismatches or bad practices (37).

3.6.5.3 Top 10 Best Practices

Based the analysis in (37), ten best practices emerged as essential for obtaining high-quality test code (Table 8). Nine of them are related to technical aspects, and only one to business-level best practices. This suggests that most sources of grey literature in this domain are predominantly of technological nature. However, according to (37), in the literature, no solutions and tools have been proposed to detect and solve flaky web tests, even less to tackle the synchronization problem.

Table 8 - Top 10 best practices (listed in descending order of references) (37).

Rank	Best Practice	Technical	Business
1	Manage the synchronization w/ the web app	X	
2	Use the Page Object Pattern (also Page Factory)	X	
3	Create robust/proper locators/selectors	X	
4	Keep the tests atomic and short	X	
5	Produce detailed reports	X	
6	Make tests independent from each other	X	
7	Use data-driven testing	X	
8	Use appropriate naming and code conventions	X	
9	Do not limit to only GUI testing (the testing pyramid)		X
10	Remove sources of uncertainty (no flakiness)	X	

3.7 Conclusions

The related work presented, helped to shape the solution. Understanding the strategies used to tackle each of the identified problems enabled the appliance of some of the solutions presented to our specific use case. The related work also highlighted some lessons and pitfalls to be mindful of when designing the solution.

The study in (36) highlighted that the efficient usage of resources in E2E testing is a challenging and promising field and that with smart resource identification and organization of the test cases with the resources required, we can achieve savings in terms of time and reductions in the total cost of the test suite. One can say that in a way the study in (35) is related to (36) in the way that the described *FlowTesting* goal is to smooth the testing process for complex bugs that resurface between patches of distributed systems.

The taxonomy provided by (37) of best practices for E2E web test automation derived from an analysis of the grey literature serves as proof of the increasing interest of developers and industry around web test automation which has fostered a large amount of software engineering research. Novel analysis and testing techniques are being proposed every year, however, without a centralized knowledge base of best practices by professionals, it is difficult to fairly design and implement solutions, or to assess research advancements, so with that in mind, the solutions explorer is this study should always verify that the ten concepts presented are being follow in each step.

The dissertation in (9) testing strategy should allow writing and running tests efficiently while allowing to detect errors sooner to shorten the feedback loop. It also aimed to provide support for testing commonly found errors in distributed systems in a deterministic way and be integrated in a CI/CD pipeline, which although this last aspect of a pipeline is not to be contemplated in this study, it serves as an example of future work and how the topics explored can also have ramifications outside of E2E testing.

Out of the studies exposed and analyse, the (27) “Comparison of runtime testing tools for microservices” has a prominent placement as it is not only the one which correlates in a higher percentage with the objectives of this study but it also is a study that was referenced in several other mentioned, such has (9) and (37). Most of all, (27) brought to light two tools of service virtualization, that will be part of the study group of this dissertation which were Hoverfly and Gremlin.

4 Study of Service Virtualization Tools in E2E Testing

In this chapter the thesis proposal is presented and elaborated. The system in which the research is implemented, and the company system is described using a series of UML diagrams and the test cases specified. The tests implementations are demonstrated and explained, and the results listed.

4.1 Introduction

The study is divided into two phases, first the implementation of an E2E test is done using each of the tools studied in the previous chapter on a prototype system and them on a real company system. The second part consists of the evaluation of the results related with the test implementation.

4.1.1 Test Implementation

To achieve the objective proposed in 1.3 an End-to-End test must be replicated in a testing environment that aims to represent a real system used in a company environment to obtain standard data for comparison purposes.

That test is than recreated using the five tools proposed and analysed on section 3.5 to simulate the API responses of the services the main service depends on, retrieving the details and challenges of the implementation. Since the service virtualization is used to create API responses, the pattern proposed can then be used regardless of the system language.

The test cases are then repeated on a system provided by a company on which the protype is based on, sharing the same concepts. The results on this system are then compared with the ones obtained in the prototype.

4.1.2 Test Evaluation

The evaluation of the tools implementation that will help in the reach of conclusions and lessons learned will focus on two criteria, the test running time and the ease of implementation. The test running time will be measured against a standard time provided by a test using the Moq framework (38) as a substitute of a virtualization tool.

The ease of implementation can be measure by how easy it is to install the tool, as in, if the tool may be installed as a NuGet (39) or if it must run as a separate standalone service. Also, the availability of the tool contributes as a factor to the ease of implementation, for example, if the tool is open-source or it requires a subscription. The final characteristic to evaluate the tool ease of installation is the quality of the documentation, has in, if the documentation is updated and/or provides enough examples or guides on how the tool must be used and configured.

The choice of attributes to be considered was presented to the professionals using a questionnaire. The professionals will later participate in the evaluation of the results to validate if the attributes evaluated are considered valuable and important in workplace project.

The questionnaire the group of professionals answered to obtain evaluation criteria is available in **Appendix A** and a summary of the answers is present in Figure 14.

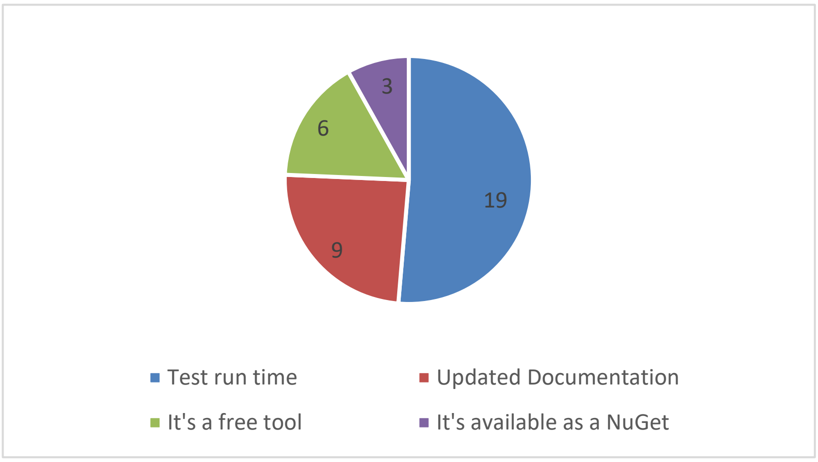


Figure 14 – Test Attribute Questionnaire Results.

There was a total of four properties named by the participants, after the answers were analysed and edit to be more easily presented in the chart in Figure 14, due to the fact that the original answers were given in the form of a loose paragraph, not all participants wrote the same property in the same way, for example “It’s a free tool” was written also as “It must be free” or simply “Free”.

One observation that can be made is that the sum of all the properties mentions equals to 37, that is since the participants could give more than one property as an answer.

Looking at the chart in Figure 14, the property that had the must focus was the time it took for the test to run, with a total of 19 mentions, while the property least mention was if the tool is available as a NuGet. It makes sense that the test run time is seem like a valuable property, since the service performance takes a higher priority when implementing a service, but it is also one of the more easily perceivable properties on a day-to-day basis.

The tool with the better result will be proposed as the best one to use in a company working environment.

The results of the tools experimentation will be grouped into a questionary form and given to a group of professionals to answer. The questionary format was chosen to facilitate the professionals to express their opinion on the results, but they will also have access to the repository to experiment on their own and provide a more bolstered opinion.

4.2 Design of the Validation Systems

The system will try to simulate a plausible company environment focused on report generation of an e-commerce platform. The architecture is composed of three services, *OrderService*, *JournalService* and *TransactionService*. *OrderService* is responsible for the registry of the customer orders, and it is responsible for the start of the test process as it sends an order event to the *JournalService* with the required information to create a Journal. A Journal can be described as a Transaction Bag, in which it holds all the transactions related with an Order and a Transaction is a financial movement, represented by a name, a currency and an amount and a report type. The *JournalService* is also responsible to split a Journal according to the Transaction types it has. The *TransactionService* is responsible to provide the Transactions related to one Journal according to its Id.

After considering the architecture, the two Use Cases on Figure 15 can be ascertained.

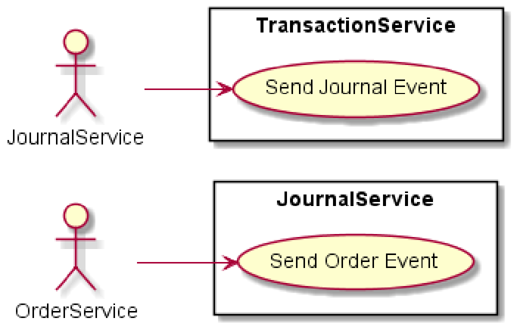


Figure 15 – Use Case Diagram.

The diagram in Figure 16 demonstrates a more detailed sequence of events of Use Case 1 and the same can be observed in Figure 17 for Use Case 2.

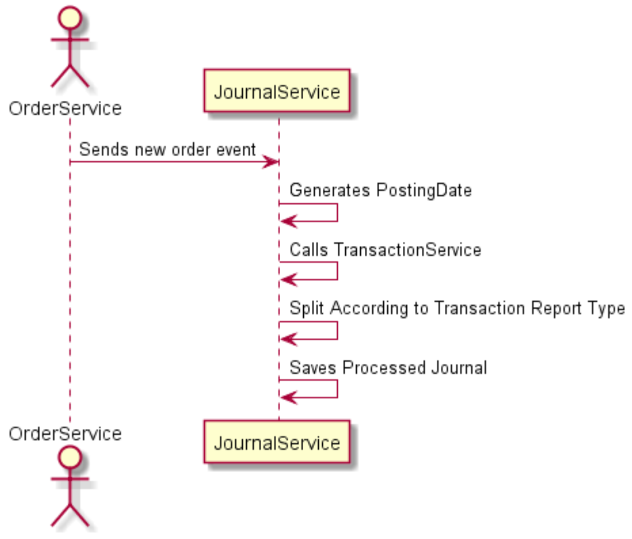


Figure 16 – Use Case 1.

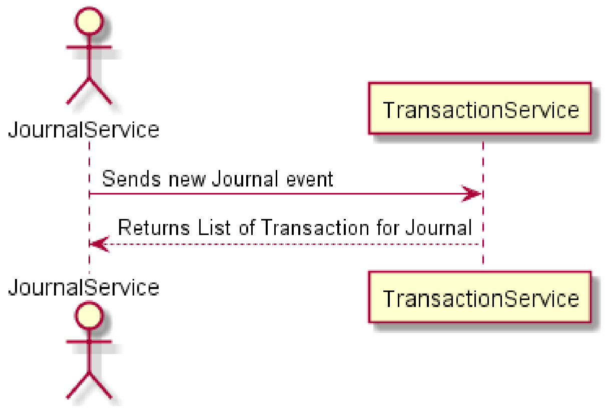


Figure 17 – Use Case 2.

In the Figure 18 there is a representation of the Domain Model with all the objects present and used in all three services and which all are present in the domain of the *JournalService*.

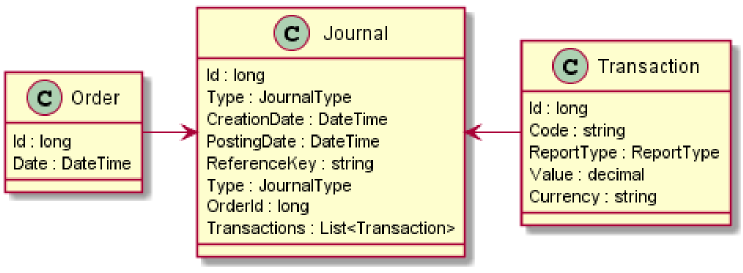


Figure 18 – Domain Model.

After defining the Domain Model and Use Cases, the overall architecture of the environment in test can be described in the component diagram of Figure 19.

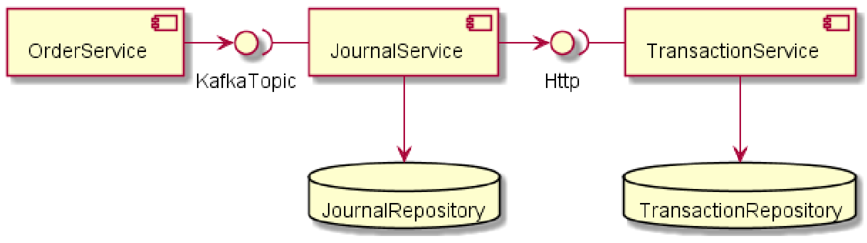


Figure 19 – Component Diagram.

Now that the components are defined, to facilitate the research process, only the main process of the *JournalService* should be the focus of the test cases, as it is the only service in which all the domain model objects are present and it is the service that communicates with the other two, giving it a wider range of possible functions and increased complexity. The sequence of actions of the main process of the *JournalService* is described in the sequence diagram of Figure 20.

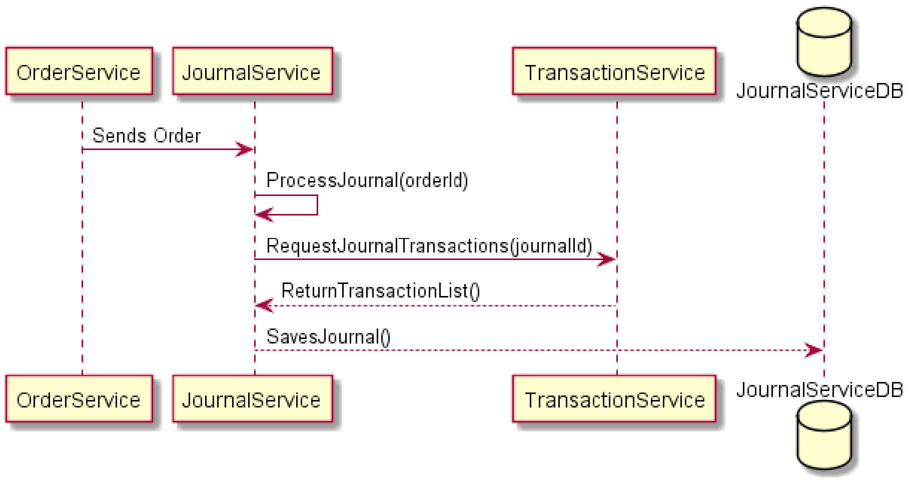


Figure 20 – Sequence Diagram.

The company system in which the tools will also be tested served as the base for the prototype, therefore there are similar concepts in their domain models and in their sequence diagrams as it can be seen in Figure 21. More details about the company system cannot be provided due to confidentiality.

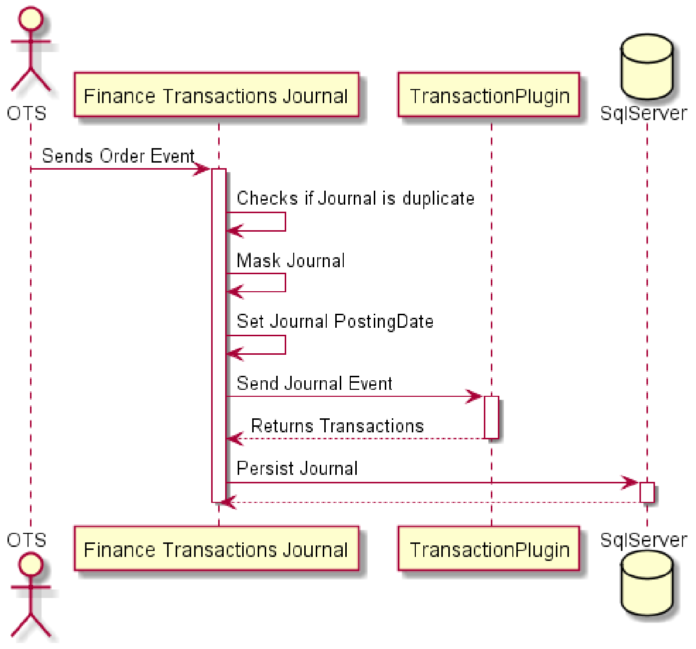


Figure 21 – Company System Sequence Diagram.

4.3 End-to-End Test Cases

In this section, the E2E test cases are defined according to process of *JournalService* (see Figure 20). E2E Testing must include the following three categories of activity (40):

- **User functions** - list down the software system’s features and the subsystems that are interconnected, keep track of the actions performed for any functionality along with the input data and output results, between different functionalities performed from the user’s end find out if there is any relation, check whether the user functions are independent or can be reused.
- **Conditions** - for each user function, build a set of conditions, conditions should include parameters like timing, data conditions, and sequence.
- **Test Cases** - write more than one test cases for every scenario and every functionality, enlist each condition as separate test cases.

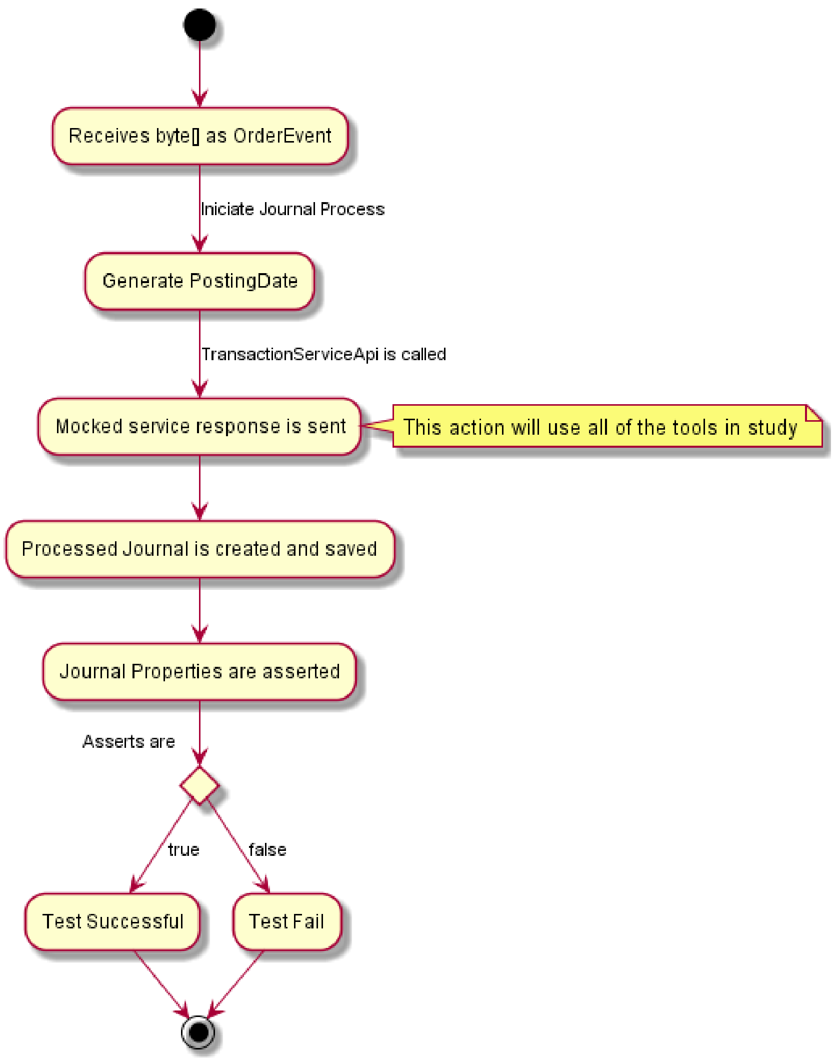
4.3.1 User Functions

To build user functions (40), the following list must be pursued:

- List the features of the software and its interconnected sub-systems.

- For each function, track and record all actions performed. Do the same for all input and output data.
- Identify all relations between user functions.
- Establish if each user function is independent or reusable.

In this case the focus will be the use case 1 (Figure 16), whose objective is to create a new Journal, therefore the user function defined is "Process Order Event to obtain a new Journal". For this user function the following steps and actions are ascertained in Figure 22.



Receives byte[] as OrderEvent
Iniciate Journal Process
Generate PostingDate
TransactionServiceApi is called
Mocked service response is sent
This action will use all of the tools in study
Processed Journal is created and saved
Journal Properties are asserted
Asserts are
true
false
Test Successful
Test Fail

Figure 22 – User Function Activity Diagram.

4.3.2 Conditions

To build conditions based on user functions, a set of conditions for every user function must be decided (40). This could include timing, data conditions, etc. – essentially any factor that can affect user functions.

For the user function defined the following set of condition can be ascertained:

- Check if the array of bytes sent by *OrderService* is correctly deserialiased into an *OrderEvent*
- Verify that the Posting Date generated for the Journal is higher than the Creation Date on the Order Event
- Verify that the *TransactionService* returns a response with code 200, meaning a successful request
- Check if the Transaction list sent by the *TransactionService* is not null or empty.
- Check the report type groups of the transactions.
- Assert that the Journal is persisted successfully in the database.

4.3.3 Test Cases

To build test cases for E2E Testing it is necessary to create multiple test cases to test every functionality of the user functions and assign at least a single, separate test case to every condition (40).

For this study, the focus is on how the tools will work to mock the response of the *TransactionsService*, therefore each test case will follow the Journal process *happy path*. A test *happy path* is a well-defined test case using known input, which executes without exception and produces an expected output. *Happy path* testing can show that a system meets its functional requirements, but it does not guarantee a graceful handling of error conditions or aid in finding hidden bugs (41).

In the case of the tests to be implemented in the context of the *JournalService* the *happy path* will follow the structure presented in Figure 22, where the test starts with a message as an array of bytes which is deserialize by *JournalService* and the success of the test is determined by the success of the asserts on the processed Journal properties and if they match the expected ones configured in the test.

The tests will vary among themselves according to the tool in use:

- ProcessJournal_WithoutUsingATestTool

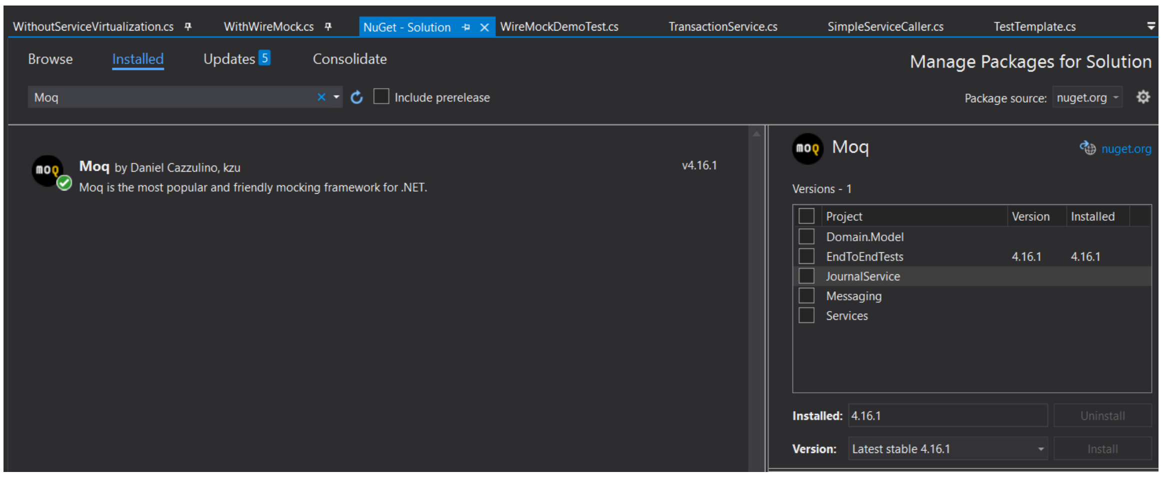
- ProcessJournal_UsiginWireMock
- ProcessJournal_UsingMockLab
- ProcessJournal_UsingGremlin
- ProcessJournal_UsingHoverfly
- ProcessJournal_UsingTrafficParot

4.4 Test Cases Implementation

4.4.1 Test Without Using a Test Tool

The Moq framework (38) is mostly used in a context of unit tests. In a similar fashion to the service virtualization tools, it returns a mocked response to a method of a class using that class interface without executing the method itself, therefore the Moq framework does not actually produce an API response or does it capture an API call. If a method is not mocked, test execution fails.

The framework can be installed in a NuGet by using the Integrated Development Environment (IDE), which in this case is the Visual Studio, NuGet Manager as seem in Figure 23.



Moq

Moq by Daniel Cazzulino, kzu v4.16.1
Moq is the most popular and friendly mocking framework for .NET.

Versions - 1

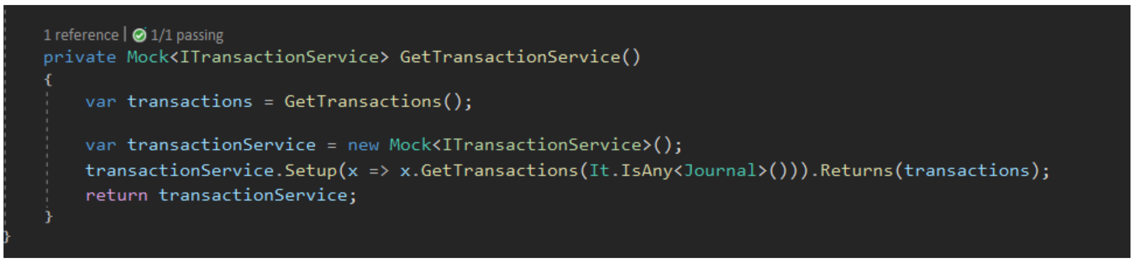
Project	Version	Installed
Domain.Model		
EndToEndTests	4.16.1	4.16.1
JournalService		
Messaging		
Services		

Installed: 4.16.1

Version: Latest stable 4.16.1

Figure 23 – Moq Framework NuGet.

After installing the NuGet, a mocked object can be created in the context of the test, the mock object uses a class interface as a base and with that its methods responses and calls can be configured using the *Setup* method Figure 24.

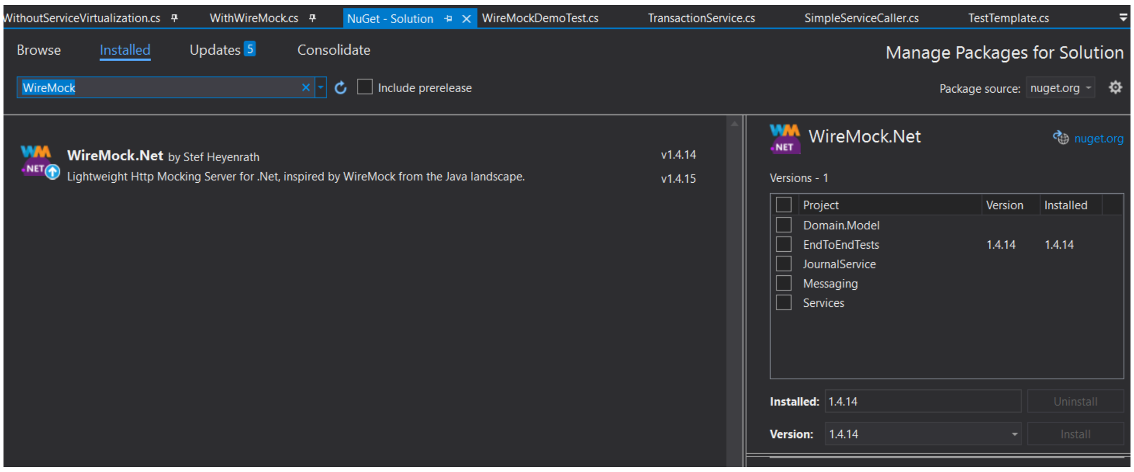


```
1 reference | 1/1 passing
private Mock<ITransactionService> GetTransactionService()
{
    var transactions = GetTransactions();

    var transactionService = new Mock<ITransactionService>();
    transactionService.Setup(x => x.GetTransactions(It.IsAny<Journal>())).Returns(transactions);
    return transactionService;
}
}
```

Figure 24 – Setting up a mock object with Moq.

4.4.2 Test using WireMock

The WireMock framework is installed on the test project as NuGet (Figure 25). The NuGet itself has a very complete documentation on its Github page (42), but it lacks an actual test implementation example but those can be found in other sources like in (43), although the examples provided use an old version of WireMock in which the name of some methods have been updated between versions so that is a fact to keep in mind when reading the documentation.



WithoutServiceVirtualization.cs | WithWireMock.cs | NuGet - Solution | WireMockDemoTest.cs | TransactionService.cs | SimpleServiceCaller.cs | TestTemplate.cs

Browse | Installed | Updates 5 | Consolidate | Manage Packages for Solution

WireMock | Include prerelease | Package source: nuget.org

WireMock.Net by Stef Heyenrath v1.4.14
Lightweight Http Mocking Server for .Net, inspired by WireMock from the Java landscape. v1.4.15

WireMock.Net nuget.org

Versions - 1

Project	Version	Installed
Domain.Model		
EndToEndTests	1.4.14	1.4.14
JournalService		
Messaging		
Services		

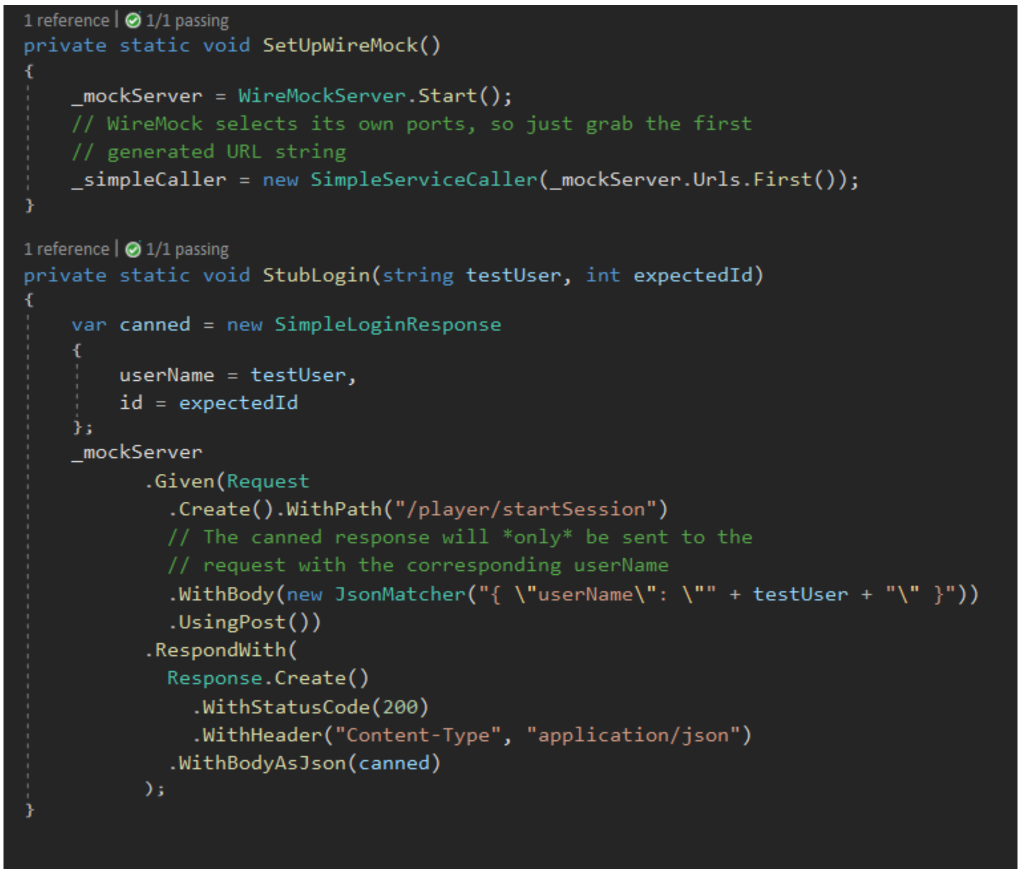
Installed: 1.4.14 Uninstall

Version: 1.4.14 Install

Figure 25 – WireMock Framework NuGet.

Since WireMock can receive an API request because it creates an instance of a web service it is necessary to first pass the WireMock server URL to the method that is sending the request. The mocked server is programmed to catch a specific request from a path and http method, which has a query parameter of the OrderId in this context.

The response is also configured with a chosen http status code and a body content which is the list of Transactions object which is converted into a json with the "BodyAsJson()" method, according to what is necessary for the test (Figure 26).



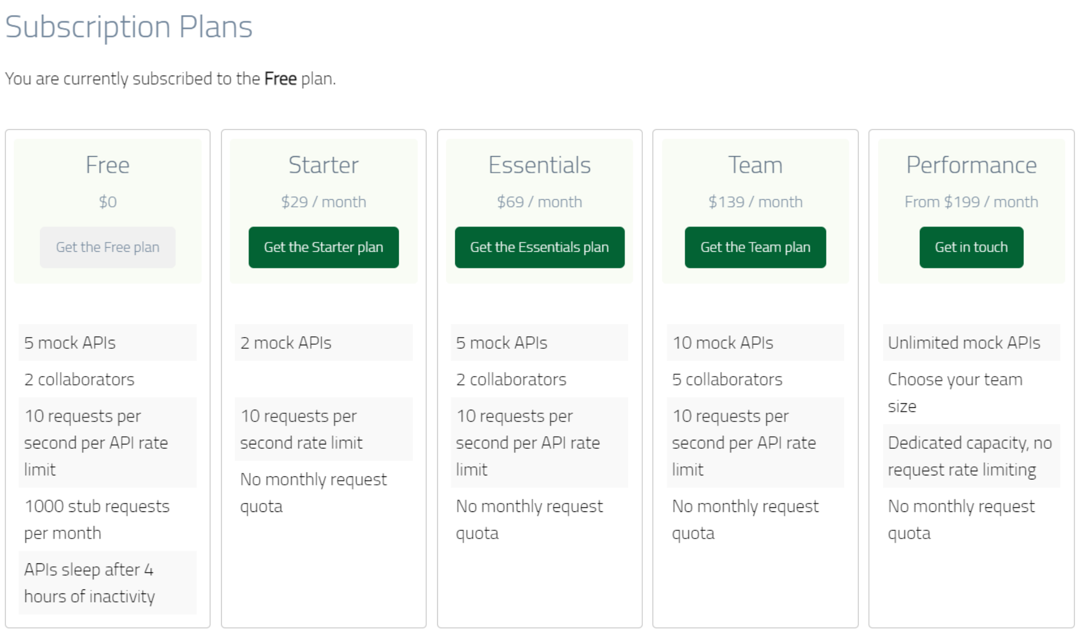
```
1 reference | 1/1 passing
private static void SetUpWireMock()
{
    _mockServer = WireMockServer.Start();
    // WireMock selects its own ports, so just grab the first
    // generated URL string
    _simpleCaller = new SimpleServiceCaller(_mockServer.Urls.First());
}

1 reference | 1/1 passing
private static void StubLogin(string testUser, int expectedId)
{
    var canned = new SimpleLoginResponse
    {
        userName = testUser,
        id = expectedId
    };
    _mockServer
          .Given(Request
            .Create().WithPath("/player/startSession")
            // The canned response will *only* be sent to the
            // request with the corresponding userName
            .WithBody(new JsonMatcher("{ \"userName\": \"" + testUser + "\" }"))
            .UsingPost())
          .RespondWith(
            Response.Create()
              .WithStatusCode(200)
              .WithHeader("Content-Type", "application/json")
            .WithBodyAsJson(canned)
          );
}
```

Figure 26 – Setting up a WireMock stub.

4.4.3 Test using MockLab

MockLab, unlike WireMock as seem in 4.4.2, does not have a NuGet to be installed but instead depends entirely on an external service which can be accessed through MockLab main website after creating an account and logging in, not actually needing any processing or memory to be used by the user system.

Although the service has a free option, it comes with a few restrictions in the number of API's mocked and mocked request that can be made, as seem in the welcome message in Figure 28. A more detailed view of the packages offered by MockLab service can be seen in Figure 27.

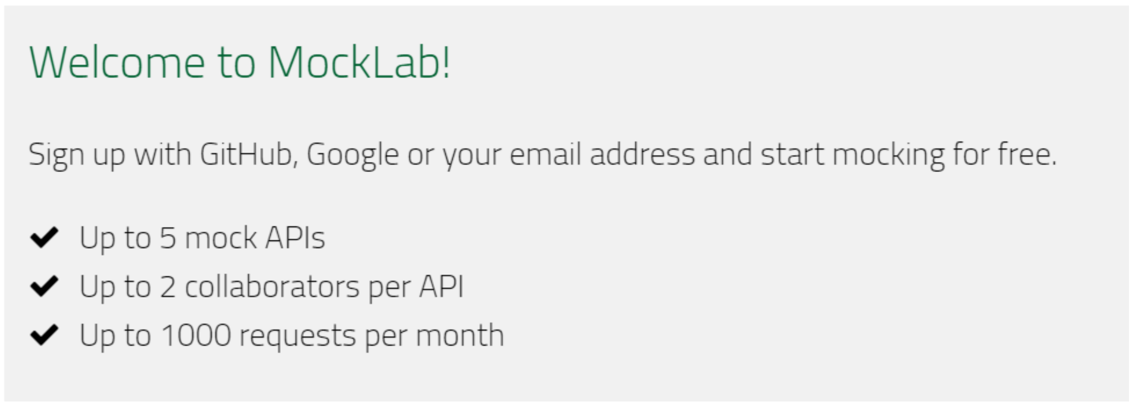

Subscription Plans

You are currently subscribed to the **Free** plan.

Free	Starter	Essentials	Team	Performance
$0	$29 / month	$69 / month	$139 / month	From $199 / month
Get the Free plan	Get the Starter plan	Get the Essentials plan	Get the Team plan	Get in touch
5 mock APIs	2 mock APIs	5 mock APIs	10 mock APIs	Unlimited mock APIs
2 collaborators		2 collaborators	5 collaborators	Choose your team size
10 requests per second per API rate limit	10 requests per second rate limit	10 requests per second per API rate limit	10 requests per second per API rate limit	Dedicated capacity, no request rate limiting
1000 stub requests per month	No monthly request quota	No monthly request quota	No monthly request quota	No monthly request quota
APIs sleep after 4 hours of inactivity				

Figure 27 – MockLab Subscription Plans.

The free account already comes with several examples of mocked requests, using the four main Http verbs (44), GET, POST, PUT and DELETE, therefore it is simple to create a new Mock API by following the provided examples.

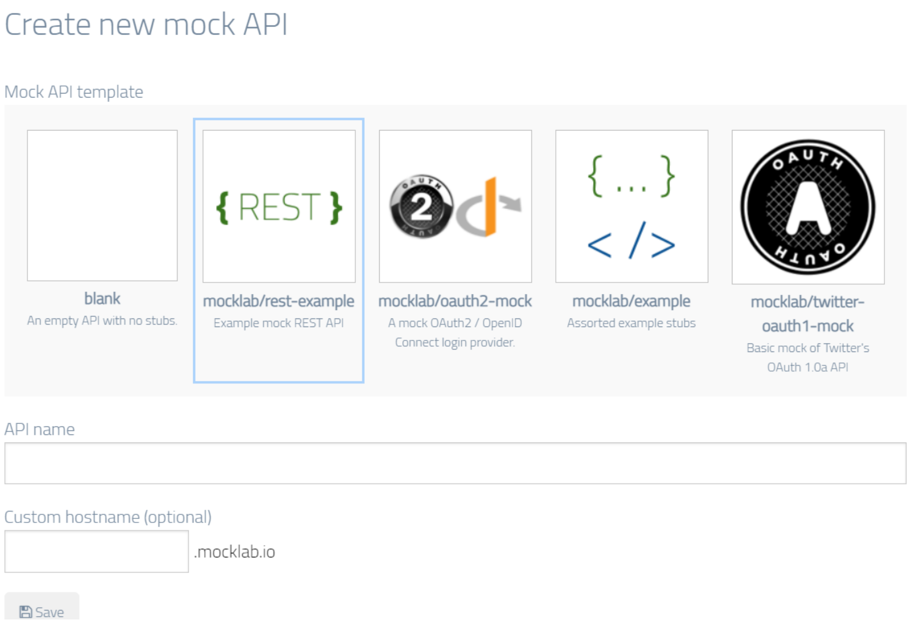

Welcome to MockLab!

Sign up with GitHub, Google or your email address and start mocking for free.

- ✓ Up to 5 mock APIs
- ✓ Up to 2 collaborators per API
- ✓ Up to 1000 requests per month

Figure 28 – MockLab login information message with exposed limits of a free account.

The new mock API can also follow an example REST structure, but MockLab also provides three more build sets for a mock API, depending on what the user might need (Figure 29). In this case the mocklab/rest-example template was chosen.


Create new mock API

Mock API template

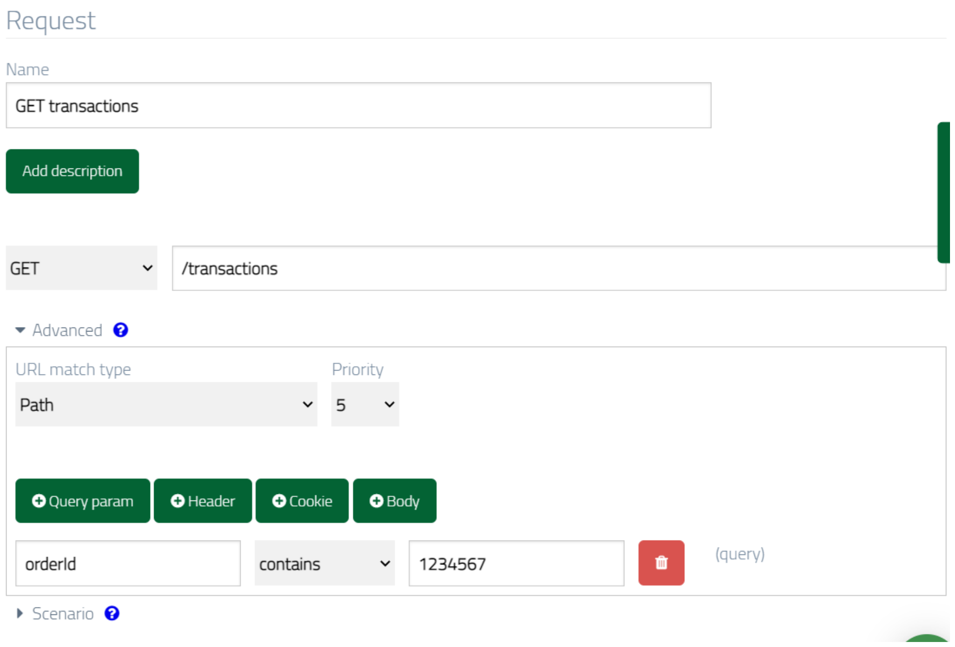
blank
An empty API with no stubs.
mocklab/rest-example
Example mock REST API
mocklab/oauth2-mock
A mock OAuth2 / OpenID Connect login provider.
mocklab/example
Assorted example stubs
mocklab/twitter-oauth1-mock
Basic mock of Twitter's OAuth 1.0a API

API name

Custom hostname (optional) .mocklab.io

Figure 29 – Start-up screen for creating a new mock API.

After having the new mock API created, a new request can be specified with the necessary parameters, which in the case of the request to the *TransactionService*, the OrderId must be passed as a query parameter (Figure 30). The route of the request must also be specified, which in this case was "/transactions".


Request

Name: GET transactions

GET /transactions

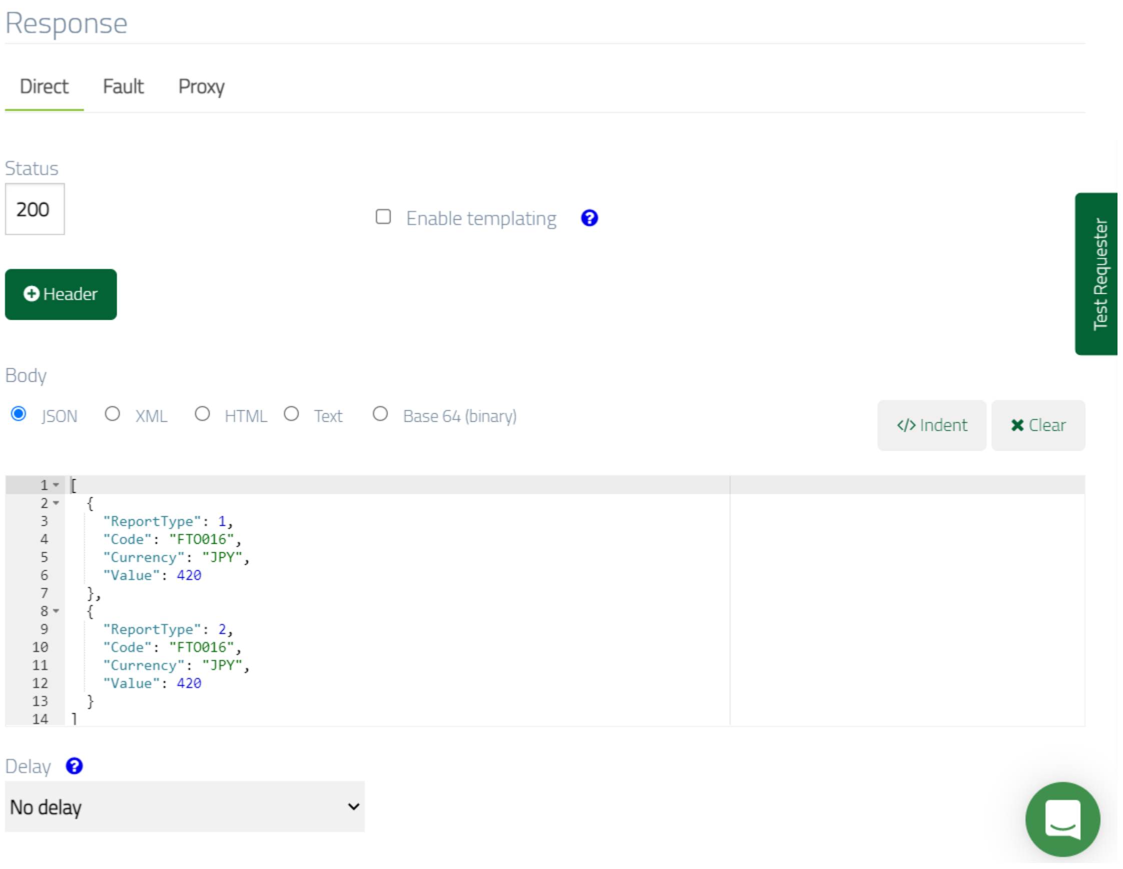
Advanced

URL match type: Path
Priority: 5

orderId contains 1234567 (query)

Figure 30 – Creating a new request on a mock API.

The status code and the response body are also defined when creating a new request, in this case the necessary status code is 200 and the body is composed of a Json with a list of Transaction objects necessary for the test (Figure 31).



Status: 200

☐ Enable templating

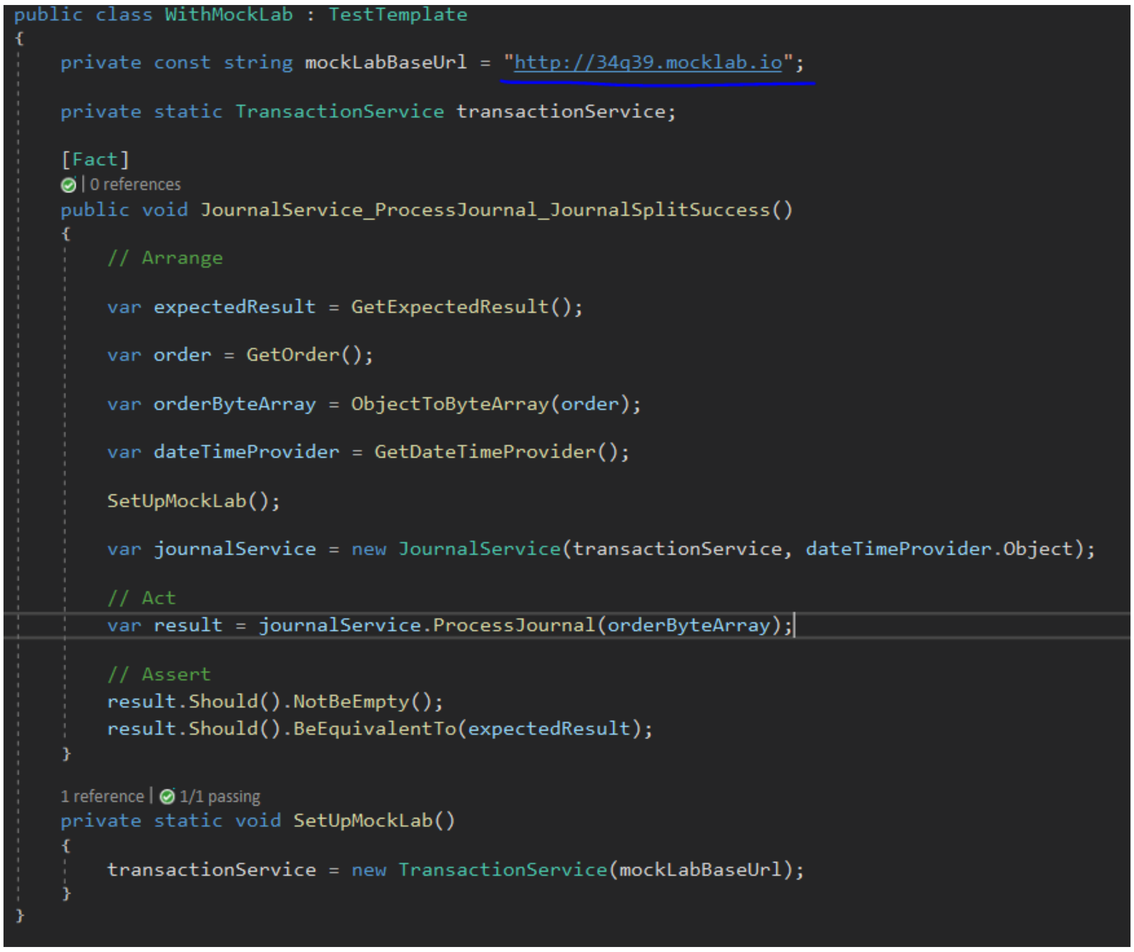
Body: [✓ JSON] XML HTML Text Base 64 (binary)

```
[
  {
    "ReportType": 1,
    "Code": "FT0016",
    "Currency": "JPY",
    "Value": 420
  },
  {
    "ReportType": 2,
    "Code": "FT0016",
    "Currency": "JPY",
    "Value": 420
  }
]
```

Delay: No delay

Figure 31 – Creating a request response on a mock API.

The newly created mock API creates an URL which is used on the required tests for the service to call the mock API created in MockLab (Figure 32), in this case the URL created was "http://34q39.mocklab.io".

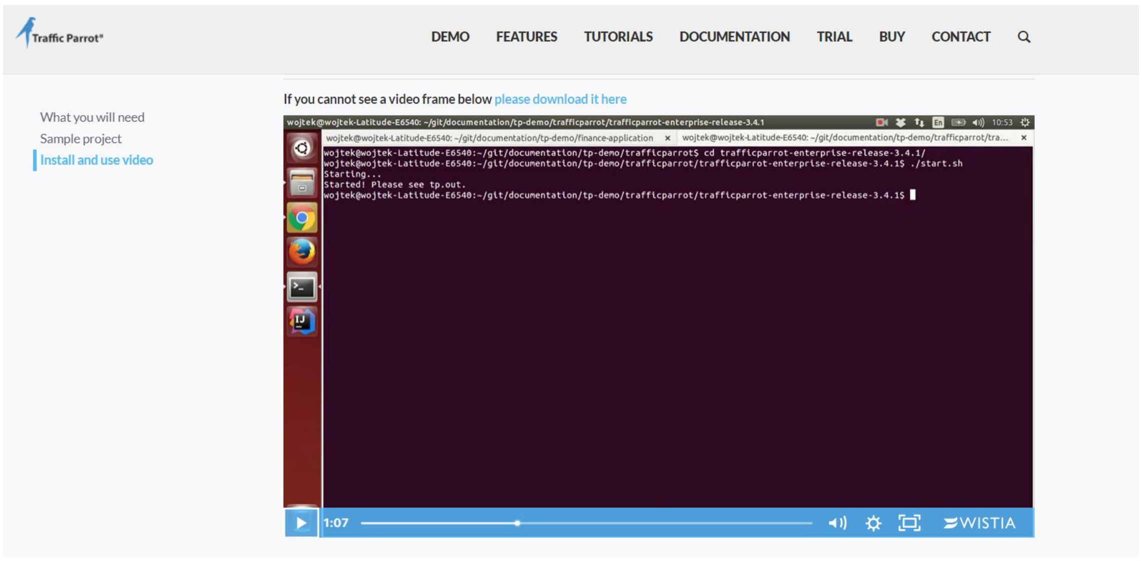


```
public class WithMockLab : TestTemplate
{
    private const string mockLabBaseUrl = "http://34q39.mocklab.io";

    private static TransactionService transactionService;

    [Fact]
    | 0 references
    public void JournalService_ProcessJournal_JournalSplitSuccess()
    {
        // Arrange

        var expectedResult = GetExpectedResult();

        var order = GetOrder();

        var orderByteArray = ObjectToByteArray(order);

        var dateTimeProvider = GetDateTimeProvider();

        SetUpMockLab();

        var journalService = new JournalService(transactionService, dateTimeProvider.Object);

        // Act
        var result = journalService.ProcessJournal(orderByteArray);

        // Assert
        result.Should().NotBeEmpty();
        result.Should().BeEquivalentTo(expectedResult);
    }

    1 reference | 1/1 passing
    private static void SetUpMockLab()
    {
        transactionService = new TransactionService(mockLabBaseUrl);
    }
}
```

Figure 32 – E2E test structure that used the MockLab service URL.

4.4.4 Test using Traffic Parrot

The Traffic Parrot main website (34) already comes with a very complete documentation of the service and even comes with a video explaining the process of how to use the service as seem in Figure 33.



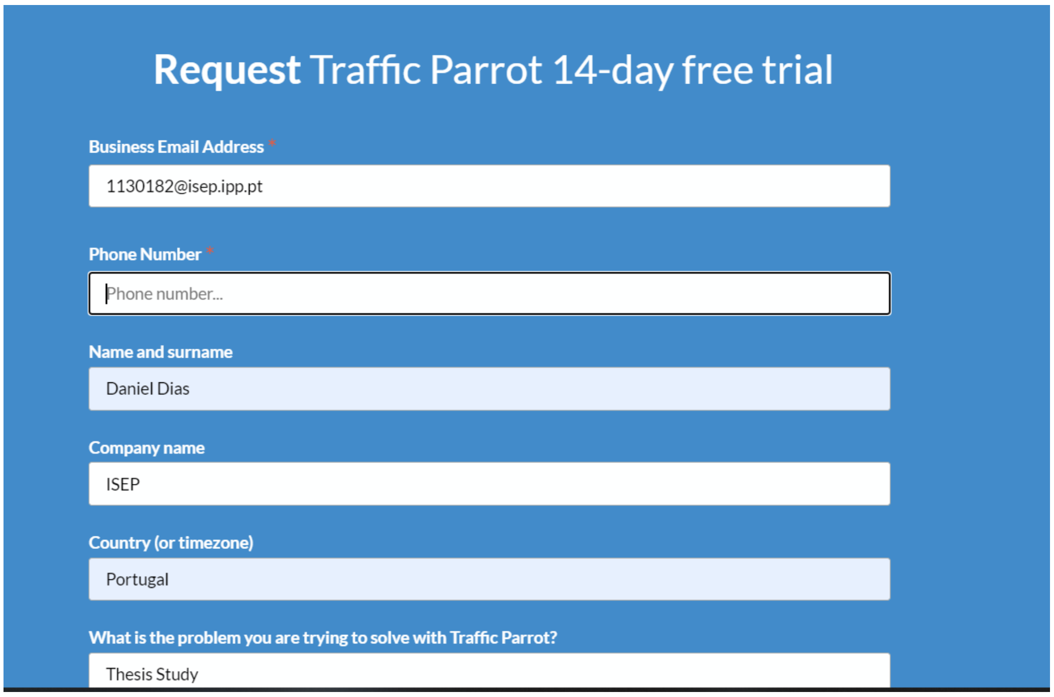
If you cannot see a video frame below please download it here

```
wojtek@wojtek-Latitude-E6540:~/git/documentation/tp-demo/trafficparrot$ cd trafficparrot-enterprise-release-3.4.1/
wojtek@wojtek-Latitude-E6540:~/git/documentation/tp-demo/trafficparrot/trafficparrot-enterprise-release-3.4.1$ ./start.sh
Starting...
Started! Please see tp.out.
wojtek@wojtek-Latitude-E6540:~/git/documentation/tp-demo/trafficparrot/trafficparrot-enterprise-release-3.4.1$
```

Figure 33 – Traffic Parrot introductory video.

In the video it is explained that the service runs on the users' machine as a service and uses a console window as an interface to configure the necessary mock requests. Even though the video is very explicit and the documentation on the website extensive, it has not been updated since 2017.

The main obstacle faced with Traffic Parrot appeared even before the service was downloaded into a user machine. Traffic Parrot is not an open-source service and to gain access to a fourteen-day trial it is necessary to fill a form (Figure 34) with several information of the user and the reason and other services that the user might compare with or has already used prior to Traffic Parrot.


Request Traffic Parrot 14-day free trial

Business Email Address *	1130182@isep.ipp.pt
Phone Number *	Phone number...
Name and surname	Daniel Dias
Company name	ISEP
Country (or timezone)	Portugal
What is the problem you are trying to solve with Traffic Parrot?	Thesis Study

Figure 34 – Traffic Parrot fourteen-day free trial request form.

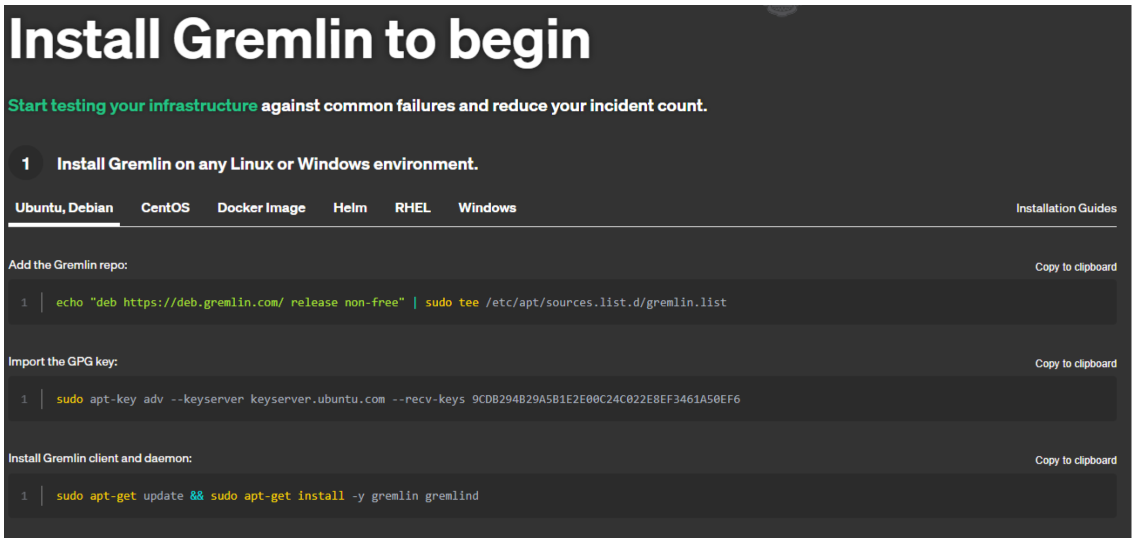
It was detailed on the mentioned form that during those fourteen days Traffic Parrot was going to be used in E2E test to mock a service response in the context of a thesis report and that the data obtained with those tests would be compared with four other services which were listed by name in the form. After the form is sent, a response from the company should arrive within one business day.

Unfortunately, the trial was denied by the company due to the lack of business prospects and lack of academic packages and support by the Traffic Parrot owner company, therefore the time of execution of the test could not be obtained as the implementation of the test could not continue after this point.

4.4.5 Test using Gremlin

Gremlin, much like Traffic Parrot, runs as a sperate service. To start using Gremlin, first it is necessary to create an account on Gremlin main website (29), from there the documentation is very much complete throughout the whole installation and mocking process.

Although Gremlin is a separate service, it comes with a plethora of options of installation (Figure 35), since in this context the services run in a Windows machine, the Windows installation guide could be followed, but the Docker installation was chosen has having a Docker image of Gremlin would them be easier to install in different machines. To install using Docker the command “docker pull gremlin/gremlin” must run in a console window.



Install Gremlin to begin

Start testing your infrastructure against common failures and reduce your incident count.

1 Install Gremlin on any Linux or Windows environment.

Add the Gremlin repo:

```
echo "deb https://deb.gremlin.com/ release non-free" | sudo tee /etc/apt/sources.list.d/gremlin.list
```

Import the GPG key:

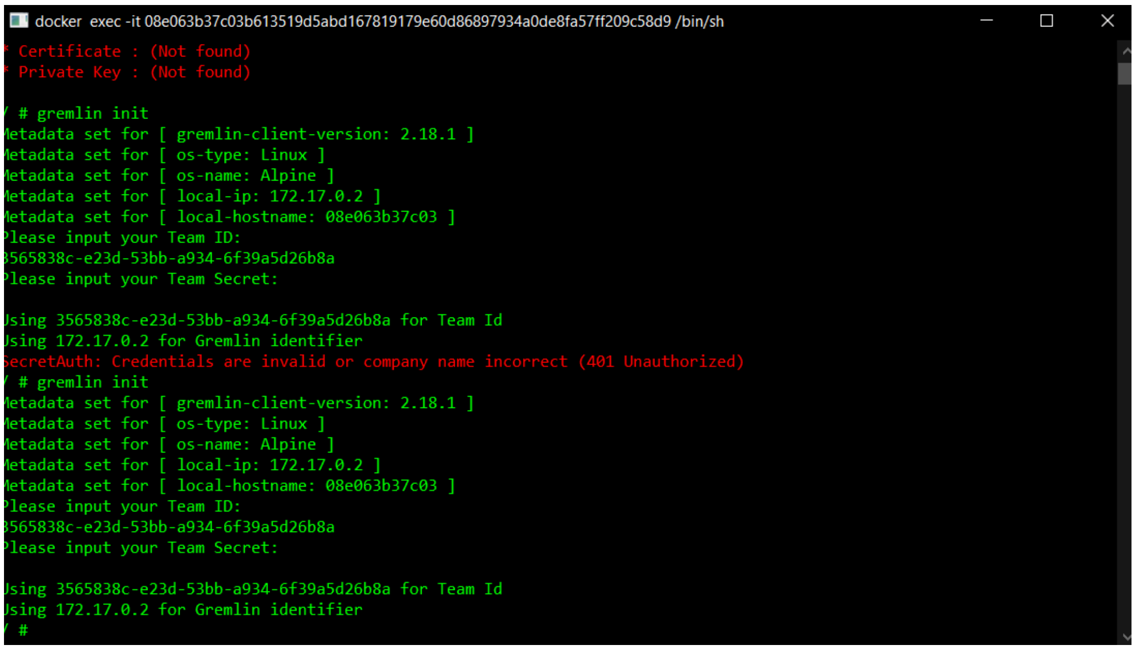
```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 9CDB294B29A5B1E2E00C24C022E8EF3461A50EF6
```

Install Gremlin client and daemon:

```
sudo apt-get update && sudo apt-get install -y gremlin gremlind
```

Figure 35 – Gremlin installation options.

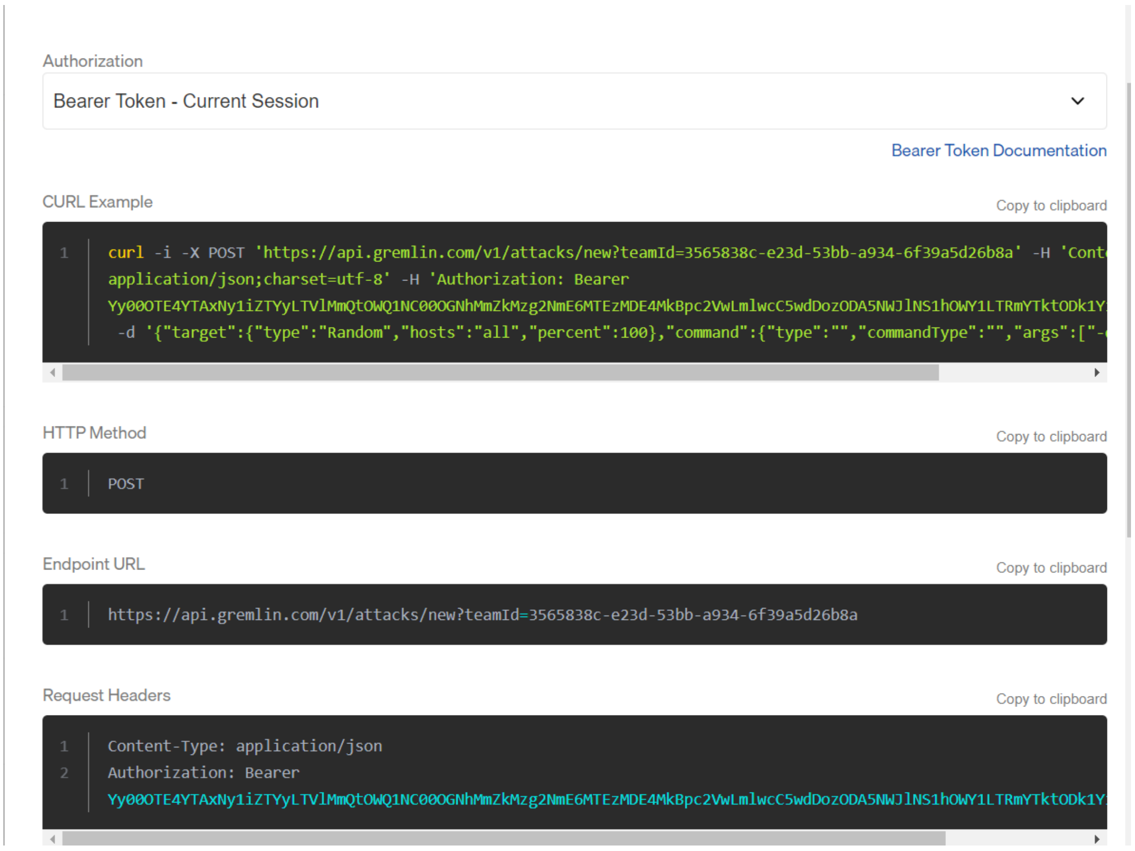
After downloading and starting the image, it is necessary to configure the team id and secret that are provided on the previously created Gremlin account, as seen in Figure 36. As it happens with many passwords when working with a console, the characters for the secret are not visible, therefore there is a need to be precise when inserting the team secret, but if it fails or if it is lost, the secret can be generated as many times as necessary from the account interface in (29).



```
docker exec -it 08e063b37c03b613519d5abd167819179e60d86897934a0de8fa57ff209c58d9 /bin/sh
 Certificate : (Not found)
 Private Key : (Not found)

/ # gremlin init
etadata set for [ gremlin-client-version: 2.18.1 ]
etadata set for [ os-type: Linux ]
etadata set for [ os-name: Alpine ]
etadata set for [ local-ip: 172.17.0.2 ]
etadata set for [ local-hostname: 08e063b37c03 ]
lease input your Team ID:
565838c-e23d-53bb-a934-6f39a5d26b8a
lease input your Team Secret:

sing 3565838c-e23d-53bb-a934-6f39a5d26b8a for Team Id
sing 172.17.0.2 for Gremlin identifier
ecretAuth: Credentials are invalid or company name incorrect (401 Unauthorized)
/ # gremlin init
etadata set for [ gremlin-client-version: 2.18.1 ]
etadata set for [ os-type: Linux ]
etadata set for [ os-name: Alpine ]
etadata set for [ local-ip: 172.17.0.2 ]
etadata set for [ local-hostname: 08e063b37c03 ]
lease input your Team ID:
565838c-e23d-53bb-a934-6f39a5d26b8a
lease input your Team Secret:

sing 3565838c-e23d-53bb-a934-6f39a5d26b8a for Team Id
sing 172.17.0.2 for Gremlin identifier
/ #
```

Figure 36 – Initial configuration of Gremlin.

Now that the Gremlin account has been linked with the Gremlin service, the necessary requests and responses for the test can be configure, much like what was described in 4.4.3 with MockLab. The Http method is chosen, then the endpoint must be defined as well as the body of the response (Figure 37).



```
curl -i -X POST 'https://api.gremlin.com/v1/attacks/new?teamId=3565838c-e23d-53bb-a934-6f39a5d26b8a' -H 'Cont
application/json;charset=utf-8' -H 'Authorization: Bearer
Yy00OTE4YTAxNy1iZTYyLTVlMmQtOWQ1NC00OGNhMmZkMzg2NmE6MTEzMDE4MkBpc2VwLmlwcC5wdDozODA5NWJlNS1hOWY1LTRmYTktODk1Y
 -d '{"target":{"type":"Random","hosts":"all","percent":100},"command":{"type":"","commandType":"","args":["-
```

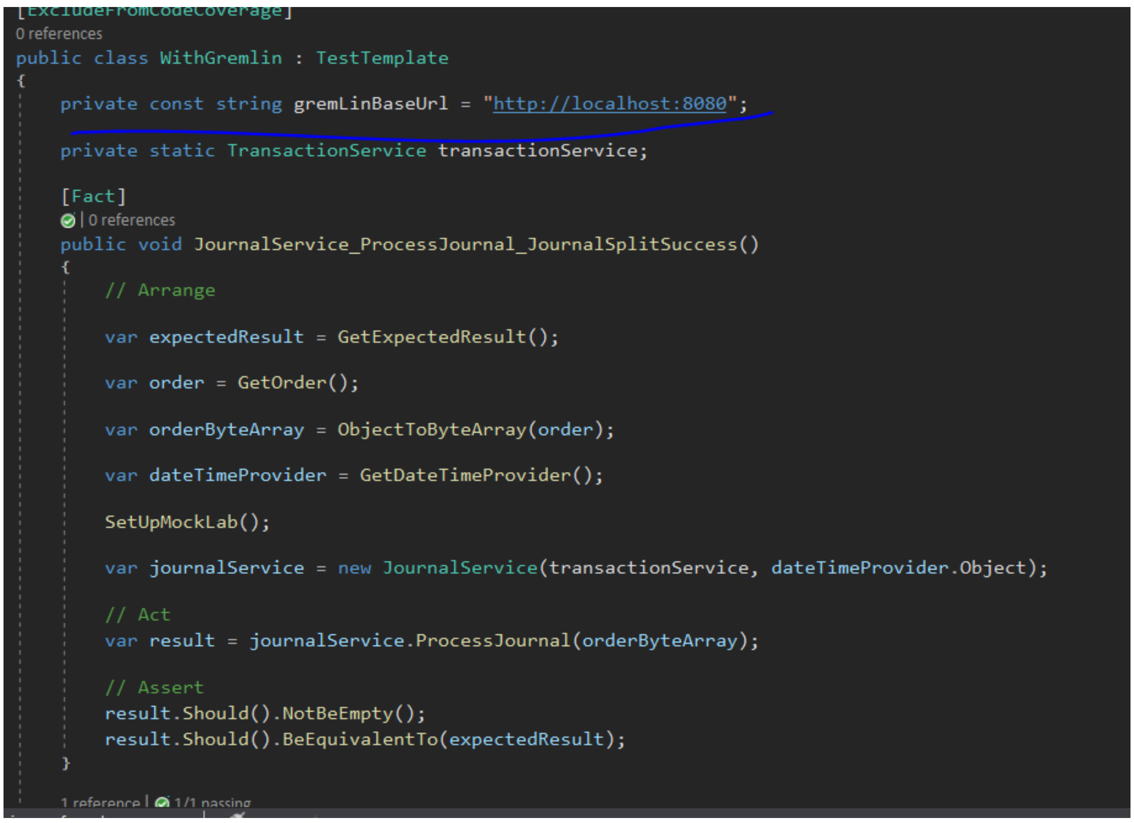
```
POST
```

```
https://api.gremlin.com/v1/attacks/new?teamId=3565838c-e23d-53bb-a934-6f39a5d26b8a
```

```
Content-Type: application/json
Authorization: Bearer
Yy00OTE4YTAxNy1iZTYyLTVlMmQtOWQ1NC00OGNhMmZkMzg2NmE6MTEzMDE4MkBpc2VwLmlwcC5wdDozODA5NWJlNS1hOWY1LTRmYTktODk1Y
```

Figure 37 – Configure call to Gremlin mock API.

The final step is to pass the Gremlin service URL on the test for the service to call it (Figure 38).



```
[ExcludeFromCodeCoverage]
0 references
public class WithGremlin : TestTemplate
{
    private const string gremLinBaseUrl = "http://localhost:8080";

    private static TransactionService transactionService;

    [Fact]
    | 0 references
    public void JournalService_ProcessJournal_JournalSplitSuccess()
    {
        // Arrange

        var expectedResult = GetExpectedResult();

        var order = GetOrder();

        var orderByteArray = ObjectToByteArray(order);

        var dateTimeProvider = GetDateTimeProvider();

        SetUpMockLab();

        var journalService = new JournalService(transactionService, dateTimeProvider.Object);

        // Act
        var result = journalService.ProcessJournal(orderByteArray);

        // Assert
        result.Should().NotBeEmpty();
        result.Should().BeEquivalentTo(expectedResult);
    }

    1 reference | 1/1 passing
```

Figure 38 – E2E test structure that used the Gremlin service URL.

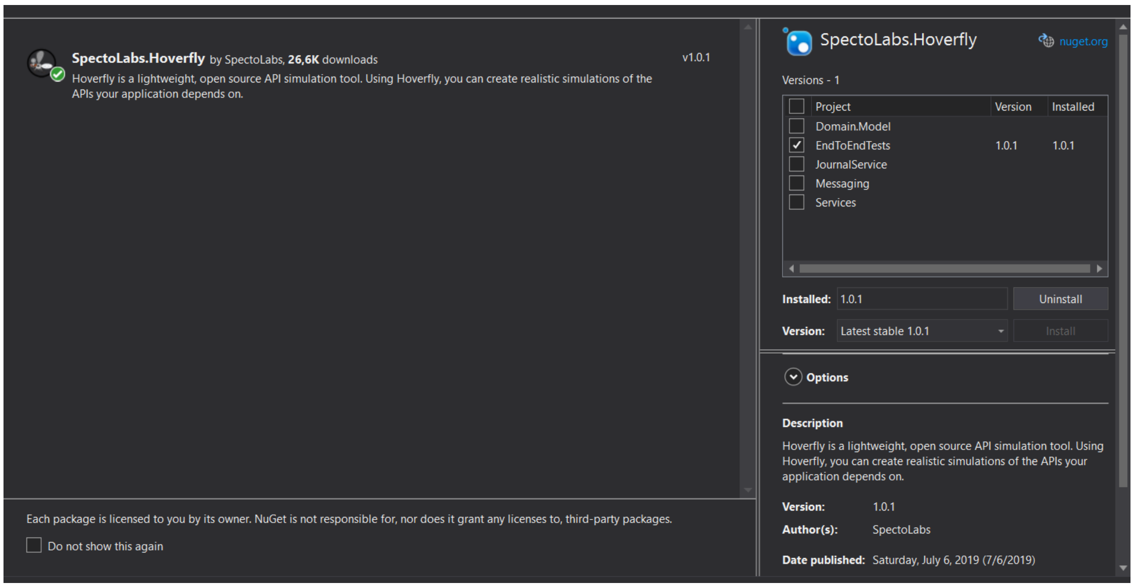
4.4.6 Test using HoverFly

HoverFly main website (30) has a very complete documentation section with examples, although not focused on any language for test development but very detailed on how to setup the service. With further research on the Visual Studio Marketplace, two NuGet packages were also found, even though they are not mentioned on the official documentation.

One of those NuGet packages (45), would allow for a service configuration without needing for HoverFly to run separately, similarly to WireMock in 4.4.2, but this package only runs in a .Net Framework Standard project and not on a .Net Core project has it is the case with JournalService, so it wasn't possible to use this package.

The second package (Figure 39), which is the one supported officially by the HoverFly company (SpectoLabs), does not provide methods and metadata to implement and run tests with. What this NuGet provides instead are the necessary files to run the HoverFly service in a machine into the project folder.

To run HoverFly locally, it is necessary to use a console window and navigate to the folder where hoverfly.exe and hoverctrl.exe are and run the command "hoverctrl start", with this the HoverFly service starts to run.


SpectoLabs.Hoverfly by SpectoLabs, 26,6K downloads v1.0.1

Hoverfly is a lightweight, open source API simulation tool. Using Hoverfly, you can create realistic simulations of the APIs your application depends on.

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

☐ Do not show this again

SpectoLabs.Hoverfly nuget.org

Versions - 1

	Project	Version	Installed
☐	Domain.Model		
✓	EndToEndTests	1.0.1	1.0.1
☐	JournalService		
☐	Messaging		
☐	Services		

Installed: 1.0.1 Uninstall

Version: Latest stable 1.0.1 Install

Options

Description

Hoverfly is a lightweight, open source API simulation tool. Using Hoverfly, you can create realistic simulations of the APIs your application depends on.

Version: 1.0.1

Author(s): SpectoLabs

Date published: Saturday, July 6, 2019 (7/6/2019)

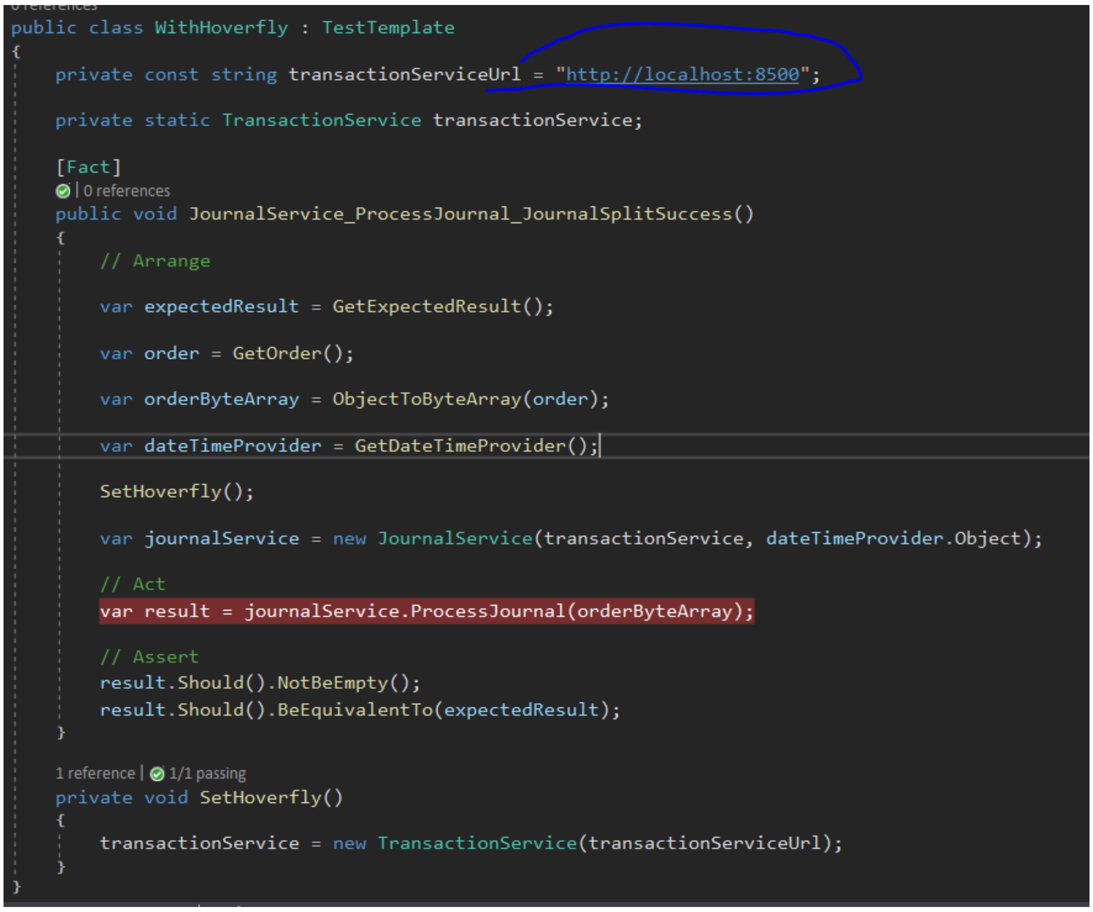
Figure 39 – HoverFly NuGet Package.

HoverFly uses a .json file to obtain the configuration of the requests and responses necessary for the tests. The json file must be saved in the same folder has the hoverfly.exe and hoverctrl.exe files and must follow a similar structure to Code 5.

```
{
    "data": {
        "pairs": [
            {
                "request": {
                    "path": [
                        {
                            "matcher": "exact",
                            "value": "/transactions"
                        }
                    ],
                },
                "response": {
                    "status": 200,
                    "body": "[{\"ReportType\":1,\"Code\":\"FT0016\",\"Cur
rency\":\"JPY\",\"Value\":420},{\"ReportType\":2,\"Code\":\"FT0016\",\"Cu
rrency\":\"JPY\",\"Value\":420}]",
            }
        ]
}
```

Code 5 – HoverFly configuration file.

After having the json simulation file ready, it is necessary to run a chain of commands in the previously mention console window. The first command is “hoverctl start webserver”, which starts HoverFly as a webservice, then the simulation file is imported with “hoverctl import simulation.json”. After running these two commands successfully, the service is ready to be use for testing, all that is left is to set the correct URL for the test to call on, as seem in Figure 40.



```
public class WithHoverfly : TestTemplate
{
    private const string transactionServiceUrl = "http://localhost:8500";

    private static TransactionService transactionService;

    [Fact]
    0 references
    public void JournalService_ProcessJournal_JournalSplitSuccess()
    {
        // Arrange

        var expectedResult = GetExpectedResult();

        var order = GetOrder();

        var orderByteArray = ObjectToByteArray(order);

        var dateTimeProvider = GetDateTimeProvider();

        SetHoverfly();

        var journalService = new JournalService(transactionService, dateTimeProvider.Object);

        // Act
        var result = journalService.ProcessJournal(orderByteArray);

        // Assert
        result.Should().NotBeEmpty();
        result.Should().BeEquivalentTo(expectedResult);
    }

    1 reference | 1/1 passing
    private void SetHoverfly()
    {
        transactionService = new TransactionService(transactionServiceUrl);
    }
}
```

Figure 40 – E2E test structure that uses the HoverFly service URL.

When all tests have ended their run and the user no longer needs to use HoverFly, the command “hoverctl stop” must be used to terminate the HoverFly service safely.

4.5 Discussion of The Results

In section 4.1.2 it was presented that the properties to evaluate the tools were obtained using a questionnaire. In this section the answers are stated and discussed.

Based on the data collected in Figure 14, Table 9 and Table 10 were constructed. In both tables it is present a summary of the implementation results of the tests detailed in 4.3, in both systems in which they were implemented. These tables summarize the results obtained during the implementation of the tests on two criteria, the test running time, whose standard is defined by a test using the Moq framework (38) and measure in milliseconds, and the ease of implementation, measured by the tool installation method, the tool availability, and the tool documentation status.

In Table 10 it is possible to observe the results of the test run times implemented in both the prototype and the company more complex service.

Table 9 – Test Result Summary.

Tool Name	**Tool Installation**	**Tool Availability**	**Tool Documentation**
Moq (standard test)	NuGet	Open source	Updated
WireMock	NuGet	Open source	Updated
MockLab	External Service	Free with limitations	Updated
TrafficParrot	External Service	Paid	Last updated in 2017
Gremlin	External Service	Free	Updated
HoverFly	External Service	Open source	Last updated in 2017

Table 10 – Test Runtime Results in milliseconds.

Tool / **System**	**Moq (standard test)**	**WireMock**	**MockLab**	**TrafficParrot**	**Gremlin**	**HoverFly**
Prototype	329	490	562	-	564	2300
Company service	527	784	899,2	-	902,4	3680

Analysing the tables, it is possible to observe that none of the tools could reach an execution time lower than the standard time set by the Moq framework, but it was something expected because Moq does not go has deep as a service virtualization tool.

Regarding Table 10, the time of each tool increase proportionally when implementing the tests in the company service and no tool reached a faster time when changing services nor did the order of the fastest to the slowest. The higher run time when implementing in the company service was expected, since the service has a higher complexity when comparing with the prototype service.

The tool with the fastest run time in both systems was WireMock with 490 milliseconds in the prototype and 784 milliseconds in the company system, although the difference when compared with MockLab and Gremlin is not very high. In theory, the time difference might be since WireMock is being run as NuGet package in the project as opposite to all the other tools that run as an external service, making it more easily accessed by the system.

The TrafficParrot run time could not be measured due to the inaccessibility to the tool by the parent company so it will not be possible to know if it would have the lowest speed.

Continuing with the subject of the paragraph before the last one, WireMock was also the only tool that could be installed as a NuGet service, at least, when searching in the NuGet.org library, which is the one used when developing in .Net. The other tools might be available in NuGet or similar notion in other language which is something to discuss in future developments of this research.

The fact that a tool must run as an external service, makes it have extras steps for the test to run and for test development, delaying the completion of a test case and also increasing the complexity for the developers as they must work with various environments instead of just focusing on one IDE.

Looking at the fourth column of Table 9 we can see that only Traffic Parrot has its use barred by a paid subscription. One of the aspects of a tool mentioned in answers of the questionnaire was that a tool must be either free or open source, meaning that if a tool is paid, normally it is automatically discarded. The same can be said for the MockLab tool, even though it has a free subscription, due to its limitations it could also be discarded.

The last property to be evaluated is related to the documentation. All the tools possess updated documentation and very well organized in each of their company's website, except for WireMock where the official documentation is present in a Github wiki. Hoverfly and Traffic Parrot were the only ones with a slight delay in their documentation as their last update was in 2017.

Having these analyses and Table 9 and 10 in mind the WireMock tool presents the best results and characteristics and is therefore the tool of choice according to this study to implement E2E tests in services with features like the ones presented.

5 Experts' Evaluation

This chapter has the objective of evaluating the tool study that was implemented. It defines the measures to be used, the hypotheses to be tested, the test methodology and the results of the tests. It also describes the approach used to evaluate the solutions.

5.1 Evaluation Objectives

The goal of this project is to provide a way to consistently implement end-to-end tests with resource to service virtualization that is possible to use within a working/company environment. Therefore, it is necessary to evaluate not only the solution implemented but also the challenges faced and ascertain the validity of the reasons why a certain tool may be used over another.

Since this work focus on real life applications of the tools, it was decided to use questionaries that can be answered by experts of the field. The opinion of the professionals is of high importance as they have extensive experience in microservices implementations. The complete questionnaire will be found in the appendix of this report.

5.2 Methodology

The questions presented to the experts are divided into sections to provide a better adjustment for the professionals' evaluation. The groups are:

A. Questions regarding the experience of the questionees with microservices and E2E in general.

B. Questions regarding the choice of characteristics chosen to evaluate the service virtualization tools to validate their choice.

C. Questions regarding the results of each tool and the tool chosen as the best one.

All the questions are closed-ended, and the answers are provided using values of the Likert scale (Table 11).

Table 11 – Likert scale (46).

Strongly Disapprove	Disapprove	Undecided	Approve	Strongly Approve
1	2	3	4	5

To analyse the results of the questionnaire regarding the groups of questions 1 and 2, each one will have specified intervals which will be described to the professionals before they answer the survey. These intervals will be used to better exemplify the meaning of each value of the Likert scale and provide the participants with as much information as possible so that they may make a conscientious decision when answering the questionary.

The intervals defined for the answers to group B are presented in Table 12 below.

Table 12 – Mean intervals for the evaluation of the solutions and patterns identified.

Interval	Description
[1-2]	The features and patterns identified in each service virtualization tool have no relation to end-to-end testing in a microservice architecture.
[2-3]	Some of the features and patterns identified in each service virtualization tool are related to end-to-end testing in a microservice architecture, but there are essential issues missing.
[3-4]	The list of features and patterns in each service virtualization tool is complete and clear. The methods are related to end-to-end testing a microservice architecture, but some are not relevant.
[4-5]	The research identified currently mostly used and proper techniques and features of each service virtualization tool for end-to-end testing a microservice architecture.

Regarding group C, the experts had access to the non-functional requirements and functional requirements. They will then be asked to analyse the solution and provide feedback regarding the achievement of the requirements. To do that, they will use the Likert scale to inform their evaluation from “Strongly Disapprove” (grade 1) to “Strongly Approved” (grade 5), for each group of requirements.

5.2.1 Choice of Participants

Since an important aspect of the tool study and implementation is focused on its usefulness in a company environment, the participants will be the highest number of available employees in the company, making up a convenience sample (47).

The job titles of the participants must also require extensive technical knowledge and software architecture experience and they should execute end-to-end tests as part of their recurring responsibilities, therefore the range of the participating positions which be withing the ones below:

- Principal Engineer.
- Engineering Lead.
- Senior Engineer.
- Quality Assurance Engineer

- Software Engineer.

The mean of experience of all the participants should be a value that assures that the participants are highly experienced and are able to provide value by evaluating the results, but there will not be a minimum of years of experience required to enter the survey as the pool of participants may not be very large.

5.2.2 Hypothesis

The mean answer of group B and C will be calculated and mapped to its specific interval. The description of the defined intervals gives some insights regarding the results of the group.

To also have an overall evaluation, the mean of the question groups mentioned will also be calculated. This final grade will be used to test the value of this work.

Using the Likert scale, if a value is bigger than 3, then it is on the positive side of the range. Consequently, it is possible to consider that the work is valuable if the final mean is higher than 3. Therefore, we must understand if the final mean value is on the positive side of the scale to test the following hypotheses.

- Group B hypotheses:

$$H_0 \colon Experts\ of\ the\ field\ consider\ the\ choice\ in\ evaluated\ charecteristics$$

$$not\ valuable\ to\ the\ field$$
$$H_0 : \mu \leq 3$$

$$H_1 \colon Experts\ of\ the\ field\ consider\ the\ choice\ in\ evaluated\ charecteristics$$

$$valueable\ to\ the\ field$$
$$H_1 : \mu > 3$$

- Group C hypotheses:

$$H_2 \colon Experts\ of\ the\ field\ consider\ this\ work\ results\ not\ valuable\ to\ the\ field$$
$$H_2 : \mu \leq 3$$

$$H_3 \colon Experts\ of\ the\ field\ consider\ this\ work\ results\ valueable\ to\ the\ field$$
$$H_3 : \mu > 3$$

If the mean is greater than 3, H_0 and H_2 are refuted and therefore it is valid to say that the work results and evaluated characteristics are valuable to the field.

5.3 Execution and Results

The questionnaire will be directly provided to specific professionals with recognized extensive knowledge in the microservices field mention in the section above. The main objective of this will be to obtain feedback directly from experts and not from a wide group of professionals. It was possible to get answers from 30 participants.

5.3.1 Expert's background

Figure 41 illustrates the job titles of the participants. All of them require extensive technical knowledge and software architecture experience.

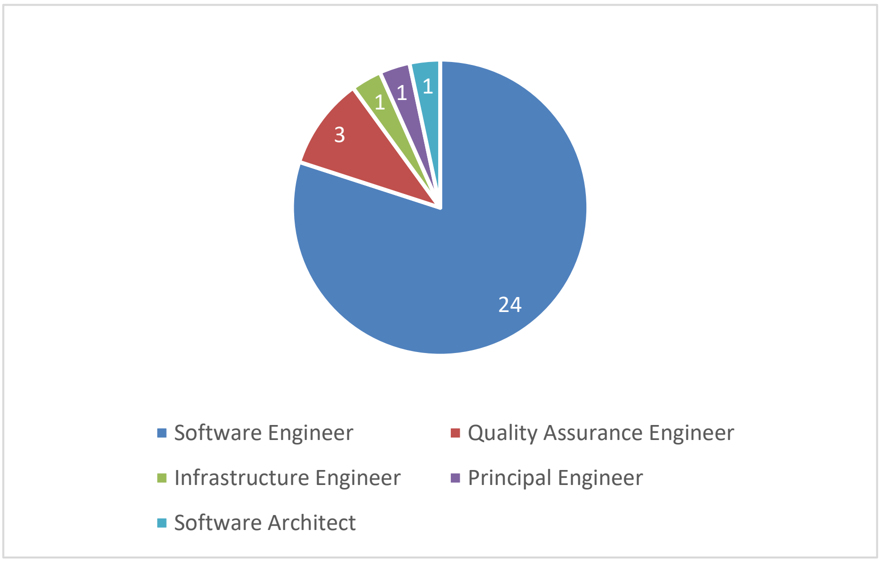


Figure 41 - Participant's job titles.

Also, the participants answered how many years of experience they have, which can be analysed in Figure 42.

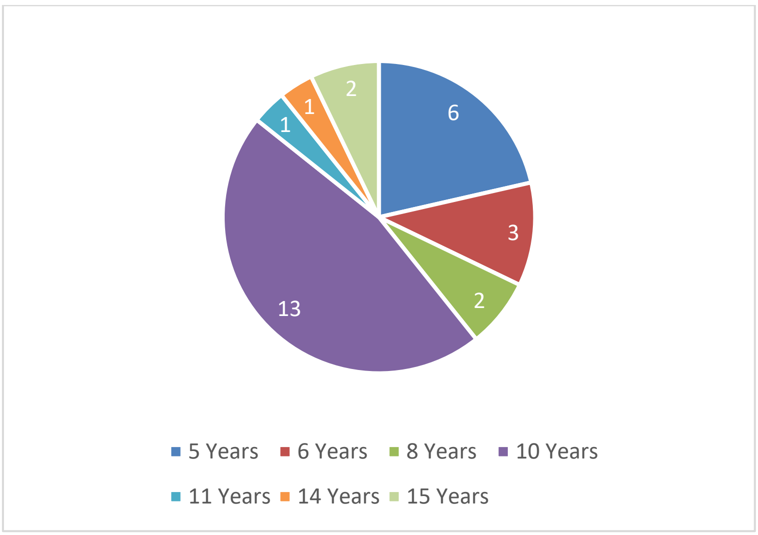


Figure 42 - Participant’s years of experience.

All the participants have at least 5 years of experience. The most experienced participant has 15 years of experience. Also, the average experience of the 30 participants is 8,8. Therefore, the participants are highly experienced and can provide value by evaluating this work results.

The next question about the expert’s background looked to clarify the composition of the system the participants most recent worked with. In this case the results will not be presented in a graph because 100% of participants answered with “My system consists of more than 5 services, but some of them are Monolithic”, meaning the systems the expert’s use mostly have a microservice architecture meaning most of the system are quite updated.

Figure 43 complements the previous paragraph information with the quantity of systems the participants currently work with and 19 of the 30 participants work with 12 services, meaning the company environment has a considerable size and the minimum number of services is 5, meaning all the systems the participants work with have a considerable complexity going by their size.

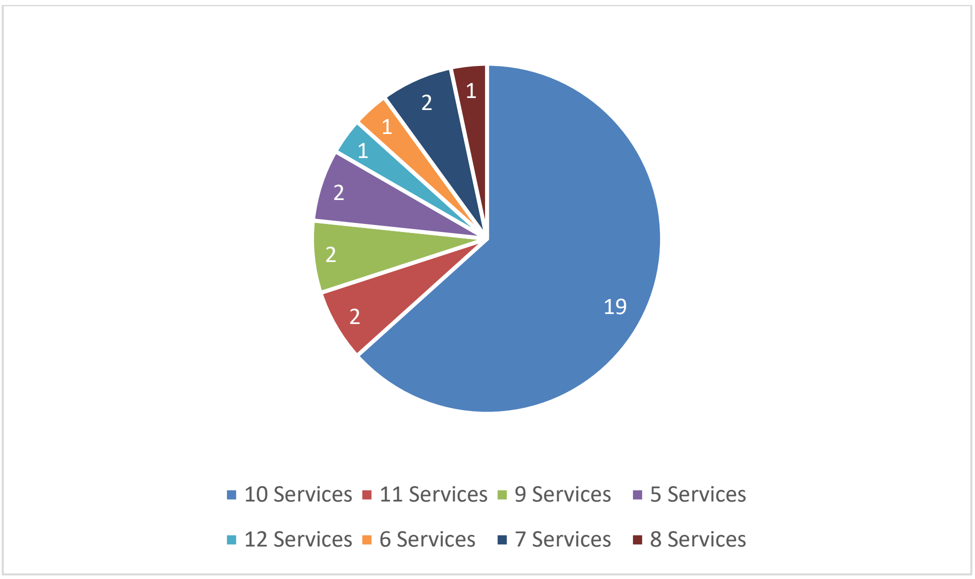


Figure 43 – How many systems the participants currently work with.

5.3.2 Choice of characteristics to evaluate the tools

After gathering some information regarding the participant's profiles, they were asked to provide feedback regarding the main aspects of the tools that were evaluated and their importance when choosing one (questions group 2).

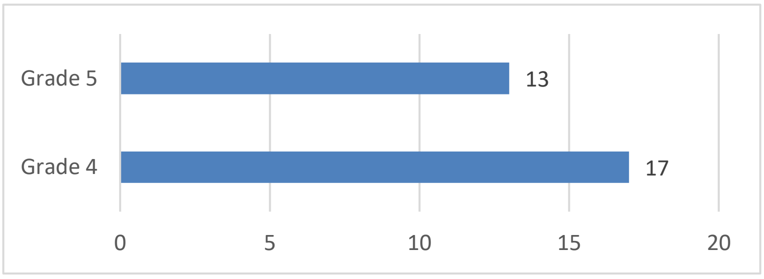


Figure 44 – Results for the question "How important is the test execution time when choosing a service virtualization tool?".

Figure 44 describes the grades provided by experts regarding the importance of test execution time of a tool. Thirteen of the experts provided the maximum grade of 5, while the lowest grade was 4. Therefore, all the experts provided a positive evaluation (more than 3 in the Likert scale).

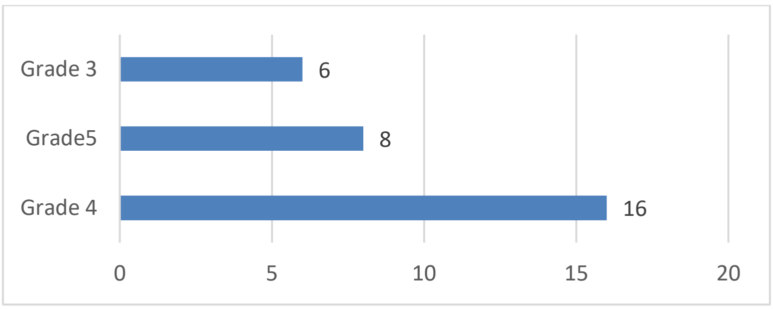


Figure 45 – Results for the question "How important is the fact that the tool is available as a nugget when choosing a service virtualization tool?".

Figure 45 describes the grades provided by experts regarding the importance of the availability of the tool as a NuGet. Sixteen of the experts provided the maximum grade of 5, while the lowest grade was 3. The evaluation was positive, because all grades were 3 or higher, but not as overwhelming as in the previous one.

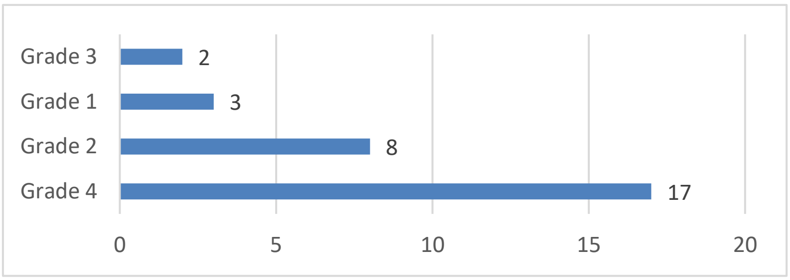


Figure 46 – Results for the question "How important is the fact that the tool is available as an external, standalone service, that must execute outside of the IDE environment, when choosing a service virtualization tool?".

Figure 46 describes the grades provided by experts regarding the importance of the tool being able to run as an external service. Seventeen of the experts provided the grade of 4 and none gave the maximum grade, while the lowest grade was 1, meaning that this time the evaluation was not all positive which indicates that this property is less desirable to some experts but not entirely to the vast majority.

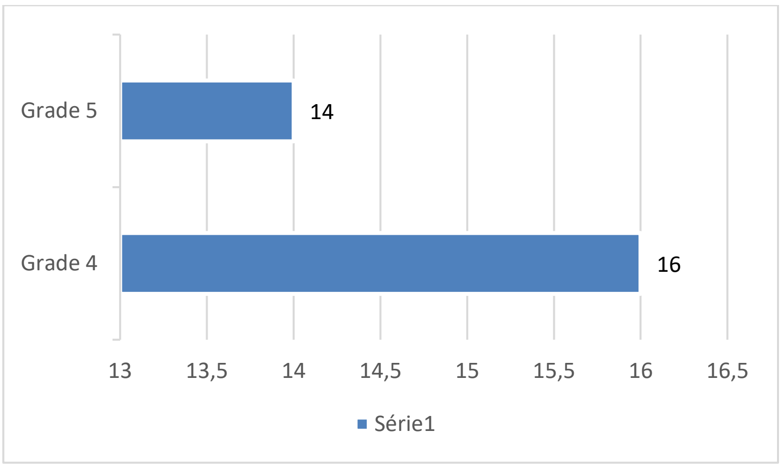


Figure 47 – Results for the question "How important is the fact that the tool is Open Source, when choosing a service virtualization tool?".

Figure 47 describes the grades provided by experts regarding the importance of the tool being Open Source. This aspect was evaluated only by two grades, 4 and 5, and the majority, 16 participants voted 5, meaning that this factor is highly important and highly desirable by the experts.

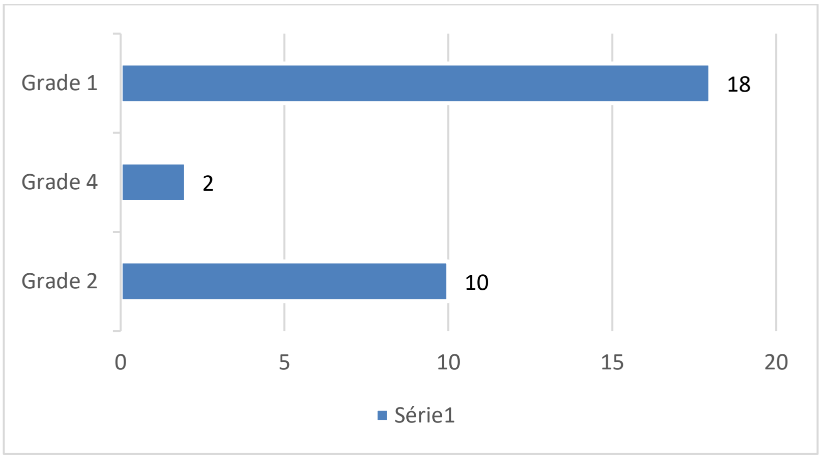


Figure 48 – Results for the question "How important is the fact that the tool is free but with a limited amount of uses per month, when choosing a service virtualization tool?".

Figure 48 describes the grades provided by experts regarding a tool being free but with limited uses. Most of the participants, 18 of the 30 participants, gave this aspect the lowest grade possible, meaning that this evaluation was negative, and it is an aspect of a tool that is not desirable, but even so, there were 2 experts that gave the grade 4, meaning that there is some room for compromise within the sample.

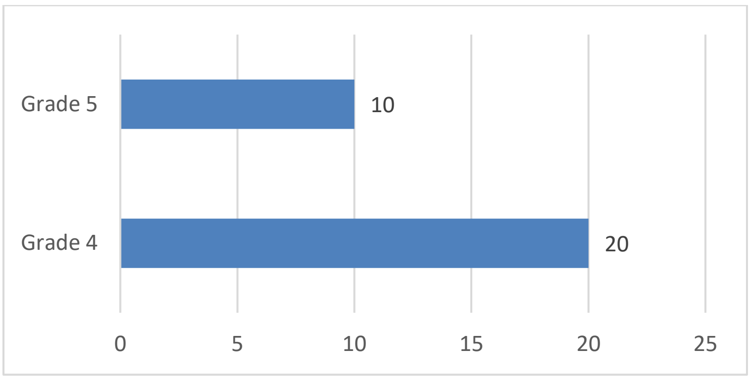


Figure 49 – Results for the question “How much does the fact that a tool has a paid subscription in order to be use, contribute to the rejection of the tool?”.

Figure 49 describes the grades provided by experts regarding the possibility of rejecting the tool if it was only available by paying a subscription. Overwhelming, this aspect was evaluated positively by the participants, in which 10 gave the highest grade possible, meaning that if a tool has this characteristic, it is most likely to not be used.

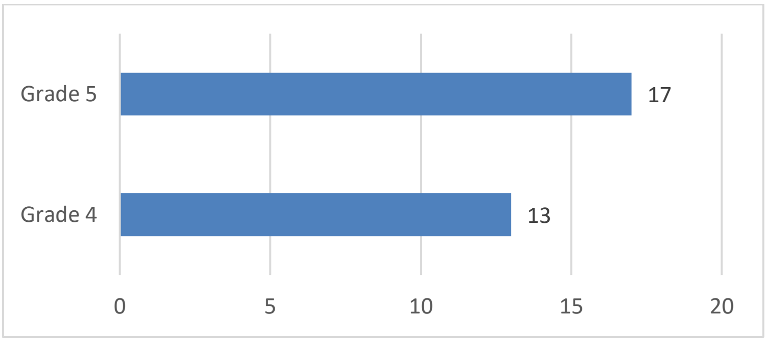


Figure 50 – Results for the question “How important is the fact that the tool has updated documentation, when choosing a service virtualization tool?”.

Figure 50 describes the grades provided by experts regarding the documentation of a tool and how updated it is. Seventeen participants gave the maximum grade to this property making it have a highly positive evaluation, meaning that updated documentation is highly important to the participants when choosing a tool.

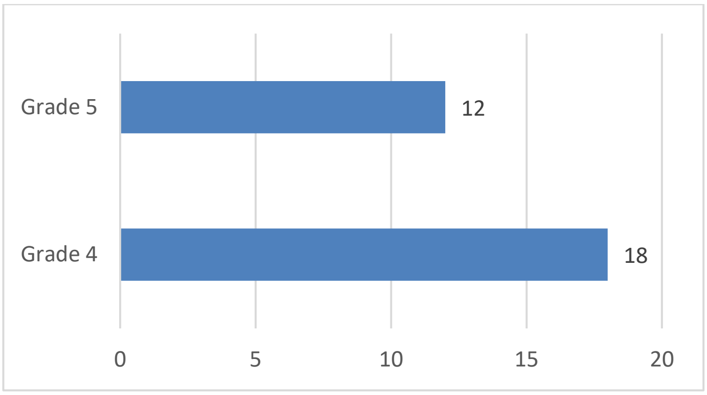


Figure 51 – Results for the question "How important is the fact that the tool has updated examples, when choosing a service virtualization tool?".

Figure 51 describes the grades provided by experts regarding the fact that the tool has updated examples of its implementation as part of its documentation. Much like the previous results this property is also highly regarded by the participant sample, has eighteen of them have given the highest grade possible, continuing the tendency that updated documentation is important when choosing a tool.

5.3.3 Hypotheses test

In Table 13 it is possible to visualize the means of all the answers regarding the importance of the choice of tool properties to evaluate (group B).

Table 13 – Mean of every answer regarding importance of tool properties.

Question	**Mean**
How important is the test execution time when choosing a service virtualization tool?	4,433
How important is the fact that the tool is available as a nugget when choosing a service virtualization tool?	4,067
How important is the fact that the tool is available as an external, standalone service, that must execute outside of the IDE environment, when choosing a service virtualization tool?	3,1
How important is the fact that the tool is Open Source, when choosing a service virtualization tool?	4,467
How important is the fact that the tool is free but with a limited amount of uses per month, when choosing a service virtualization tool?	1,533
How much does the fact that a tool has a paid subscription to be use, contribute to the rejection of the tool?	4,333
How important is the fact that the tool has updated documentation, when choosing a service virtualization tool?	4,567
How important is the fact that the tool has updated examples, when choosing a service virtualization tool?	4,4
Total Mean	3,863

By using the means of the results, it possible to corroborate the affirmations made in the previous paragraphs. The property with the lowest score was “How important is the fact that the tool is free but with a limited amount of uses per month, when choosing a service virtualization tool?” and the one with the highest was “How important is the fact that the tool has updated documentation, when choosing a service virtualization tool?”.

The total mean of this group was 3,863 which means it had an overall positive evaluation.

$$\mu = 3{,}863$$
$$3{,}863 \geq 3$$

With this value, H_0 is refuted confirming that the properties selected to evaluate the tools were correct and in tune with what the experts perceive as valuable when selecting a tool for E2E testing.

The results of the two questions in Group C with the focus on the quality of the tool analysis results and the conclusion of which of them is the most suited to be used are presented in the following figures.

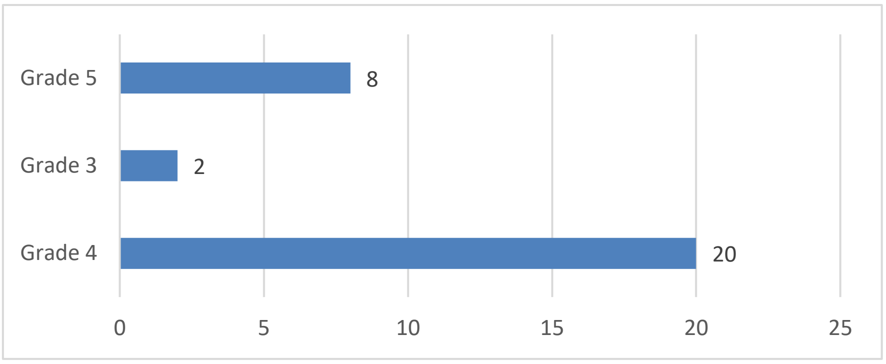


Figure 52 – Results for the question “Please evaluate from (strongly disapprove) to 5 (strongly approve) the analysis of the tools.”

Figure 52 describes the grades provided by experts regarding the general way in which the analysis of all the tools was conducted. Most of the participants gave the second highest grade (Grade 4), meaning the analysis was overall accepted and approve but with some room for improvement as it is also possible to conclude by the two grades of 3.

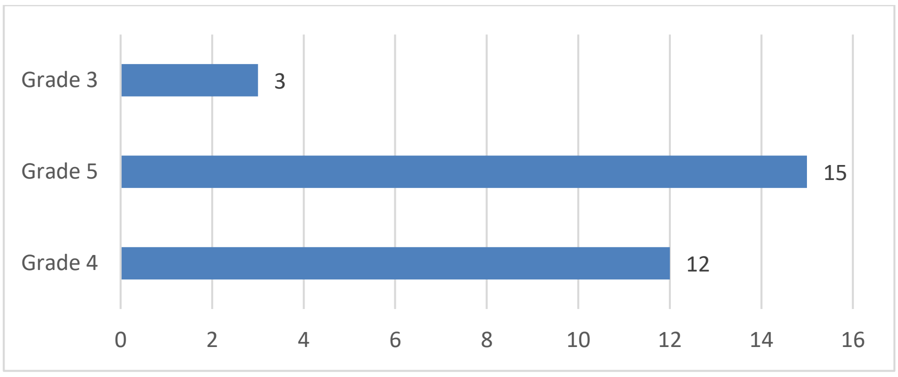


Figure 53 – Results for the question "Please evaluate from (strongly disagree) to 5 (strongly agree) the best tool identified is WireMock."

Figure 53 describes the grades provided by experts regarding their agreement with the statement that WireMock is the most suited tool. Half of the participants gave this statement the maximum score (Grade 5) making it very positively evaluated although there were still three participants who remained neutral.

To conclude the evaluation, the hypotheses stated in section 5.2.2 must be tested. As previously mentioned, this will be obtained by calculating the total mean of the answers and positioning it on the Likert scale. If the total mean of this evaluation is more significant than 3 it will be valid to say that the work results are valuable to the field. The calculations are present in Table 14.

Table 14 – Work Evaluation total means.

Question	Mean
Please evaluate from (strongly disapprove) to 5 (strongly approve) the analysis of the tools.	4,2
Please evaluate from (strongly disagree) to 5 (strongly agree) the best tool identified is WireMock.	4,4
Total mean	4,3

The total mean of this evaluation is 4.3, which is more significant than 3, positioning the evaluation in the positive side of the Likert scale.

$$\mu = 4.3$$
$$4.3 \geq 3$$

With this value, H_2 is refuted and therefore it is valid to say that the work results are valuable to the field.

5.4 Result Limitations

The present study also comes with its share of limitations regarding its results. One of the most visible ones is the fact that the tools were only used and tested in a .Net, or C# developed solutions, meaning that the results of their execution run time may not be directly extrapolated to other languages or frameworks, even though the tools chosen to be evaluated can be used by any language, such as Java for example. The same can be said about the tools ease of implementation because there is a chance that the difficulty might increase in another environment, since for example NuGet is exclusive to .Net.

Another limitation to point out is the fact that the professionals that answered the questionnaire belonged to the same company, which may lead to some developing habits that are common between most participants and it is likely that the primarily work with a focus on single developing language as well, bordering the perspectives of the evaluation.

6 Conclusions

This chapter concludes this document by analyzing and comparing the initially defined objectives with the work outputs and outcomes. The difficulties identified during this work are described here, along with possible future work.

6.1 Achieved objectives

In Section 4.1 the main objective of this work was defined. In this section, the achievement of this objective is evaluated and justified with corresponding evidence.

In Chapter 4, the tests cases and the systems they were implemented on were described. All the tools, except one, were successfully implemented on the systems, including the one used in a real company environment and the path and difficulties faced were described.

In Chapter 5, 30 experts in microservices with an average of 8,8 years of experience in the field provided a positive grade (4.3) in the Likert Scale (1 to 5) and feedback regarding both the tools research study and the tool selected and also the characteristics by which these tools were evaluated, considering the work valuable to the field, and providing further evidence of the objective achievement.

6.2 Difficulties along the way

During the development of this work, different difficulties were faced influencing the results:

- **Tool implementation challenges** – all the tools analyzed worked in different ways and each had their own rules to follow which was quite a tolling task for one developer to try to come up with ways in which these could fit with the system that need to be tested and still be comparable amongst themselves.

- **Tool inaccessible by pay wall** – the fact that one of tools, Traffic Parrot, could not be implemented meant a somewhat incomplete study regarding the implementation process experimentation and the test execution time measurement but it also provided an extra aspect of consideration when selecting or in this case rejecting the use of a tool.

- **Confidentiality issues** – due to confidentiality issues it was not possible to provide all the information regarding the company, its system that was used and the participant sample.

- **Experts' availability** – another difficulty found during this work was to find availability from industry experts to validate the results of this work as it required them to read the results and the fact that they had to answer two different questionnaires, albeit in different timeframes, which demands some time.

6.3 Future work

Even though this work achieved its objectives, there are always improvement points. Also, this work's contributions identify essential challenges for further research in the microservices E2E testing field.

One aspect to point out is the number of tools analyzed. Considering the number of tools available and the lack of tool study reports that exist, the number of tools can be increased in the future and perhaps a more suited tool can be found.

Speaking of increasing the number of tools studied, the number of systems in which these tools are implemented can also increase to achieve a better reach with the results and a stronger statement with the analysis.

Finally, this work was written in English so that it can reach a higher number of readers. Also, it was structured with the possibility of publishing an article on a recognized platform or conference, further increasing the work reach. Even though it was not possible to achieve this in the timeframe available for this work, this task may be completed in the future, so that a different perspective of reviews can be gathered, and future research influenced positively.

References

1. *Architecting Microservices: Practical Opportunities and Challenges.* **Baškarada, S., Nguyen, V. and Koronios, A.** s.l. : Journal of Computer Information Systems, 2018.

2. **Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L.** *Microservices: yesterday, today, and tomorrow.* Springer, Cham. : In Present and ulterior software engineering, 2017.

3. *From monolith to microservices: a classification of refactoring approaches.* **Fritzsch, J., Bogner, J., Zimmermann, A. and Wagner, S.** Springer, Cham. : In International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment, 2018.

4. *Migrating towards microservices: migration and architecture smells.* **Carrasco, A., Bladel, B.V. and Demeyer, S.** s.l. : In Proceedings of the 2nd International Workshop on Refactoring, 2018.

5. **Fowler, Martin and Lewis, Jerry.** Microservices. *martinfowler.com.* [Online] [Cited: 02 6, 2021.] https://martinfowler.com/articles/microservices.html.

6. *Verification in the age of microservices.* **PANDA, Aurojit, SAGIV, Mooly and SHENKER, Scott.** s.l. : Proceedings of the 16th Workshop on Hot Topics in Operating Systems, 2017.

7. **Sundar, A.** An insight into microservices testing strategies. *Infosys.com.* [Online] [Cited: 2 6, 2021.] https://www.infosys.com/it-services/validation-solutions/white-papers/.

8. *Network virtualization in multi-tenant datacenters.* **Koponen, T. and Amidon, K.** s.l. : NSDI, 2014.

9. **Mendes, David Garcia .** *Automated Testing for Provisioning Systems of Complex Cloud Products.* Lisboa : Faculade de Ciências e Tecnologia, 2019.

10. *Architectural Technical Debt in Microservices.* **De Toledo, Soares.** 2019.

11. **Instituto Superior de Engenharia Informática.** Simpósio de Engenharia Informática. *Simpósio de Engenharia Informática.* [Online] ISEP, 2021. [Citação: 11 de 10 de 2021.] http://sei.dei.isep.ipp.pt/index.html.

12. **Koen, Peter, Ajamian, G M and Boyce, S.** *Fuzzy front end: effective methods, tools, and techniques.* s.l. : The PDMA toolbook 1 for new product development, 2002.

13. **Rich, N. and Holweg, M.** *Value analysis. Value engineering: Innoregio: dissemination of innovation and knowledge management techniques.* United Kingdom : Lean Enterprise Research, 2000.

14. *The brainstorming myth.* **Furnham, Adrian.** s.l. : Business strategy review, 2000.

15. *Combining the AHP and TOPSIS to evaluate car selection.* **Ulkhaq, M, et al.** s.l. : In Proceedings of the 2nd International Conference on High Performance Compilation Computing and Communications, 2018.

16. *Software architectures - Present and visions.* **Strîmbei, C., Dospinescu, O., Strainu, R.M. and Nistor, A.** s.l. : Informatica Economica, 2015.

17. **Fowler, Martin.** Monolith First. *martinfowler.com.* [Online] [Cited: 2 6, 2021.] https://martinfowler.com/bliki/MonolithFirst.html.

18. **Newman, Sam.** *Monolith to Microservices.* United States of America. : O'Reilly, 2020. 978-1-492-07554-7.

19. **Apache Software Foundation.** INTRODUCTION. *kafka.apache.* [Online] 2017. https://kafka.apache.org/intro.

20. *Learning to Reliably Deliver Streaming Data with Apache Kafka.* **wu, Han, Shang, Zhihao and Wolter, Katinka.** Berlin : 0th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2020. DSN48063.2020.00068.

21. *Apache Kafka: Real Time Implementation with Kafka Architecture Review.* **Shaheen, Javed Ahmed .** Faisalabad, Pakistan : International Journal of Advanced Science and Technology, 2017. 10.14257.

22. *The Future of Software Quality Assurance.* **Goericke, Stephan.** Cham : Springer, 2020.

23. *Application performance management: State of the art and challenges for the future.* **Heger, C, et al.** s.l. : In Proc. 8th ACM/SPEC Int. Conf. on Performance Engineering, 2017.

24. *Analysis of Server Virtualization Service Performance Using Citrix Xenserver.* **Tenggono, Surahmat and Tenggono, Alfred.** s.l. : Journal of Physics: Conference Series, 2020. 1500 012098.

25. **Hossain, Arafat.** *Discovering Context Dependent Service Models for Stateful Service Virtualization.* Melbourne, Australia : Swinburne University of Technology, 2020.

26. **Shrimann , Upadhyay, Hrishikesh , Mukherjee and Arup Abhinna , Acharya Abhinna .** *Continuous Testing of Service-Oriented Applications Using Service Virtualization.* s.l. : ResearchGate, 2019. 10.9790/0661-17258892.

27. *Comparison of runtime testing tools for microservices.* **Sotomayor, Juan.** s.l. : IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), 2019.

28. **Heorhiadi, Victor.** *Gremlin: Systematic Resilience Testing of Microservices.* s.l. : 36th International Conference on Distributed Computing Systems (ICDCS), 2016.

29. **Gremlin Inc.** Docs. *gremlin.com.* [Online] [Citação: 16 de 2 de 2021.] https://www.gremlin.com/docs/.

30. **SpectoLabs.** Docs. *hoverfly.io.* [Online] [Citação: 17 de 2 de 2021.] https://docs.hoverfly.io/en/latest/.

31. **D'Amore, Antonio.** *Implementation of a serverless application.* Torino : Politecnico Di Torino, 2020.

32. **Akehurst, Tom.** Docs. *wiremock.org.* [Online] [Citação: 18 de 2 de 2021.] http://wiremock.org/docs/.

33. **MockLab.** Docs. *mocklab.io.* [Online] [Cited: 2 20, 2021.] https://www.mocklab.io/docs/getting-started/.

34. **Traffic Parrot.** Documentation. *Traffic Parrot.* [Online] [Cited: 2 20, 2021.] https://trafficparrot.com/.

35. **Gabrielova, Eugenia.** *End-to-End Regression Testing for Distributed Systems.* Las Vegas : Association for Computing Machinery, 2017. 978-1-4503-5199-7.

36. *Efficient test execution in End to End testing.* **Augusto, Crstian.** New York, : International Conference on Software Engineering, 2020. 978-1-4503-7122-3.

37. *Web Test Automation: Insights from the Grey Literature.* **Ricca, Filippo and Stocco, Andrea .** Italy : ResearchGate, 2020.

38. **Cazzulino, Daniel.** moq. *Github.* [Online] Moq. [Cited: 5 18, 2021.] https://github.com/moq/moq.

39. **Microsoft.** An introduction to NuGet. *Microsoft Docs.* [Online] 5 24, 2019. [Cited: 7 3, 2021.] https://docs.microsoft.com/en-us/nuget/what-is-nuget.

40. **Bose, Shreya.** End To End Testing: A Detailed Guide. *BrowserStack.* [Online] BrowserStack, 2 5, 2020. [Cited: 4 30, 2021.] https://www.browserstack.com/guide/end-to-end-testing.

41. **Cohen, Julie, Plakosh, Dan and Keeler, Kristi.** *Robustness Testing of Software-Intensive Systems: Explanation and Guide.* Carnegie Mellon University, Software Engineering Institute : Carnegie Mellon University, 2005. 10.1184/R1/6583508.v1.

42. **Heyenrath, Stef.** WireMock.Net. *Github.* [Online] 2021. [Cited: 6 9, 2021.] https://github.com/WireMock-Net/WireMock.Net/wiki/Stubbing.

43. **Acohen, Daan.** How WireMock.NET Can Help in Doing Integration Testing of a .NET Core Application. *Code Project.* [Online] [Cited: 6 9, 2021.] https://www.codeproject.com/Articles/5267354/How-WireMock-NET-Can-Help-in-Doing-Integration-Tes.

44. **Neumann, Andy, Laranjeiro, Nuno and Bernardino, Jorge.** *An Analysis of Public REST Web Service APIs.* s.l. : IEEE Computer Society, 2018.

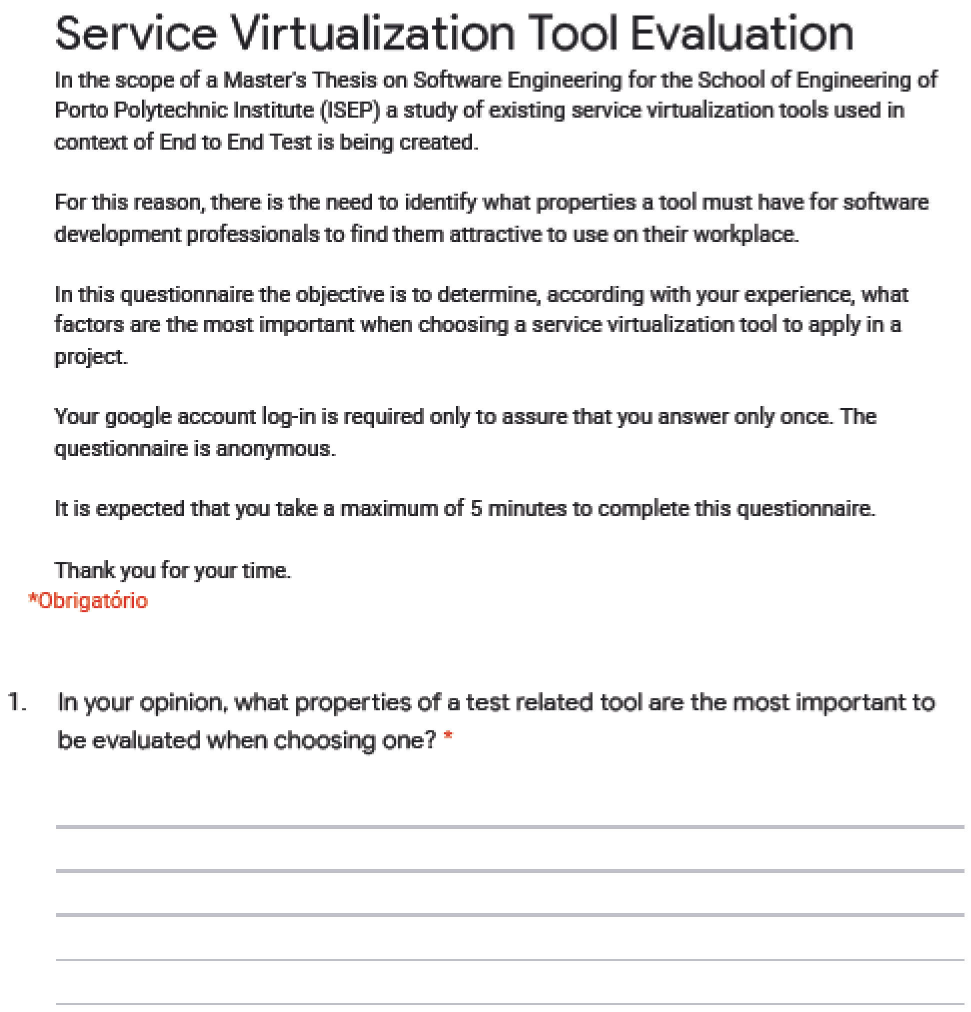
45. **Normén, Fredrik.** Hoverfly C# Nugget. *Github.* [Online] [Cited: 6 8, 2021.] https://github.com/fredrikn/hoverfly-dotnet.

46. *A technique for the measurement of attitudes.* **Likert, R.** s.l. : Archives of psychology, 1932.

47. **Bornstein, Marc H., Jager, Justin and Putnick, Diane L.** *Sampling in Developmental Science: Situations, Shortcomings, Solutions, and Standards.* 2017. PMID 25580049.

48. **Neuman, S.** *Building Microservices: Designing Fine-Grained Systems.* s.l. : O'Reilly Media, 2015.

Appendix A



Service Virtualization Tool Evaluation

In the scope of a Master's Thesis on Software Engineering for the School of Engineering of Porto Polytechnic Institute (ISEP) a study of existing service virtualization tools used in context of End to End Test is being created.

For this reason, there is the need to identify what properties a tool must have for software development professionals to find them attractive to use on their workplace.

In this questionnaire the objective is to determine, according with your experience, what factors are the most important when choosing a service virtualization tool to apply in a project.

Your google account log-in is required only to assure that you answer only once. The questionnaire is anonymous.

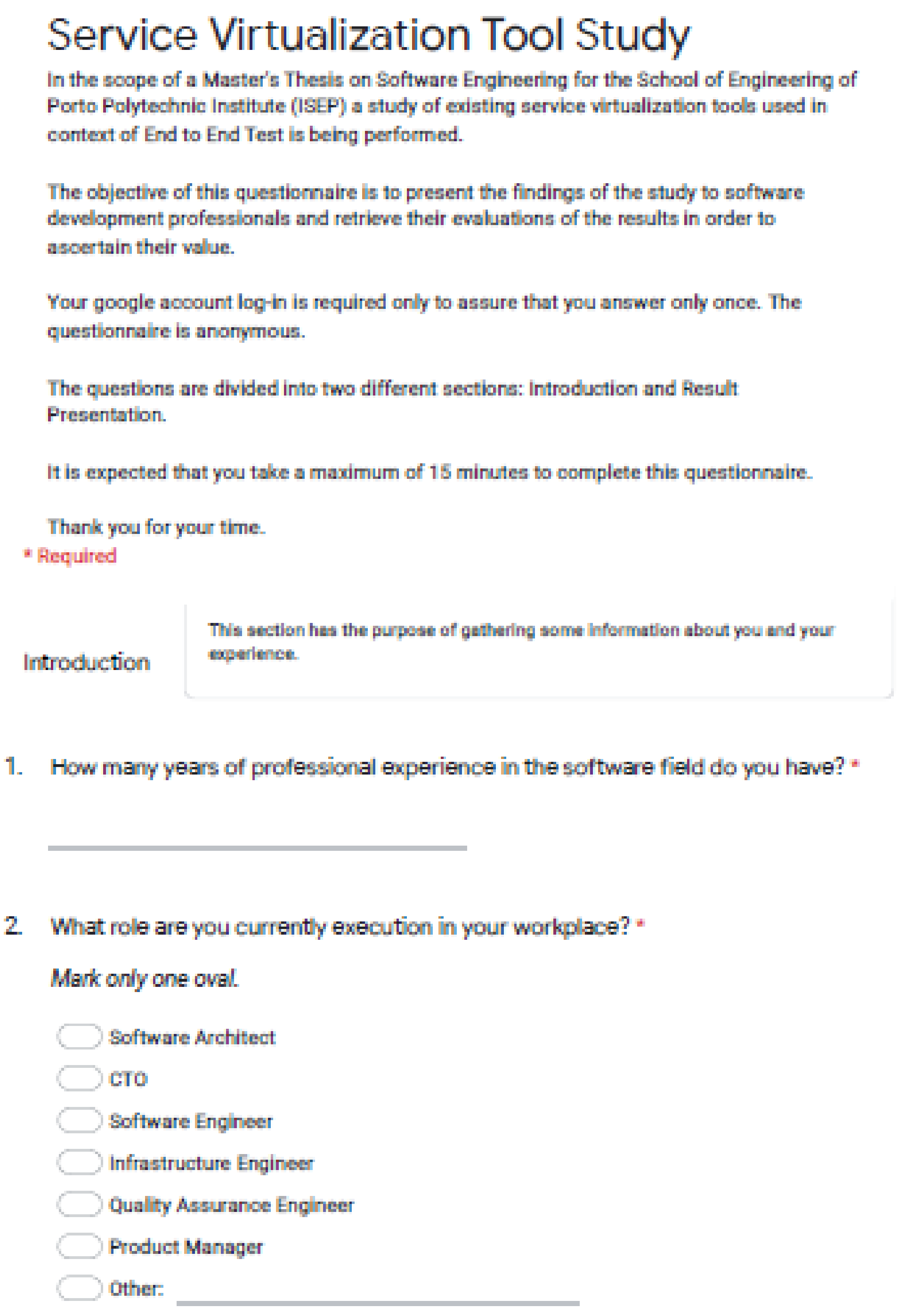
It is expected that you take a maximum of 5 minutes to complete this questionnaire.

Thank you for your time.

*Obrigatório

1. In your opinion, what properties of a test related tool are the most important to be evaluated when choosing one? *

Appendix B



Service Virtualization Tool Study

In the scope of a Master's Thesis on Software Engineering for the School of Engineering of Porto Polytechnic Institute (ISEP) a study of existing service virtualization tools used in context of End to End Test is being performed.

The objective of this questionnaire is to present the findings of the study to software development professionals and retrieve their evaluations of the results in order to ascertain their value.

Your google account log-in is required only to assure that you answer only once. The questionnaire is anonymous.

The questions are divided into two different sections: Introduction and Result Presentation.

It is expected that you take a maximum of 15 minutes to complete this questionnaire.

Thank you for your time.

* Required

Introduction

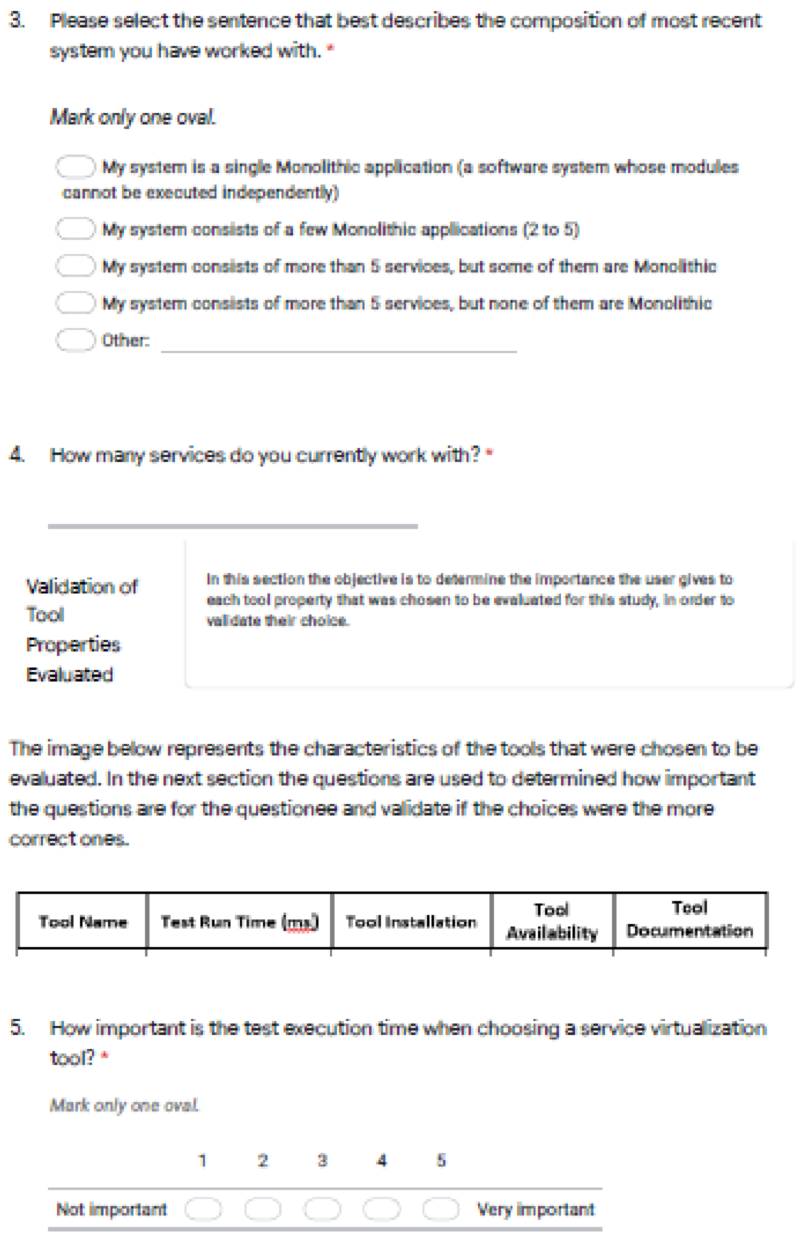
This section has the purpose of gathering some information about you and your experience.

1. How many years of professional experience in the software field do you have? *

2. What role are you currently execution in your workplace? *

Mark only one oval.

- ◯ Software Architect
- ◯ CTO
- ◯ Software Engineer
- ◯ Infrastructure Engineer
- ◯ Quality Assurance Engineer
- ◯ Product Manager
- ◯ Other: ____________



3. Please select the sentence that best describes the composition of most recent system you have worked with. *

Mark only one oval.

- ◯ My system is a single Monolithic application (a software system whose modules cannot be executed independently)
- ◯ My system consists of a few Monolithic applications (2 to 5)
- ◯ My system consists of more than 5 services, but some of them are Monolithic
- ◯ My system consists of more than 5 services, but none of them are Monolithic
- ◯ Other: ____________________

4. How many services do you currently work with? *

____

Validation of Tool Properties Evaluated

In this section the objective is to determine the importance the user gives to each tool property that was chosen to be evaluated for this study, in order to validate their choice.

The image below represents the characteristics of the tools that were chosen to be evaluated. In the next section the questions are used to determined how important the questions are for the questionee and validate if the choices were the more correct ones.

Tool Name	Test Run Time (ms)	Tool Installation	Tool Availability	Tool Documentation

5. How important is the test execution time when choosing a service virtualization tool? *

Mark only one oval.

	1	2	3	4	5	
Not important	◯	◯	◯	◯	◯	Very important

6. How important is the fact that the tool is available as a nugget when choosing a service virtualization tool? *

Mark only one oval.

	1	2	3	4	5	
Not important	◯	◯	◯	◯	◯	Very important

7. How important is the fact that the tool is available as an external, standalone service, that must execute outside of the IDE environment, when choosing a service virtualization tool? *

Mark only one oval.

	1	2	3	4	5	
Not important	◯	◯	◯	◯	◯	Very important

8. How important is the fact that the tool is Open Source, when choosing a service virtualization tool? *

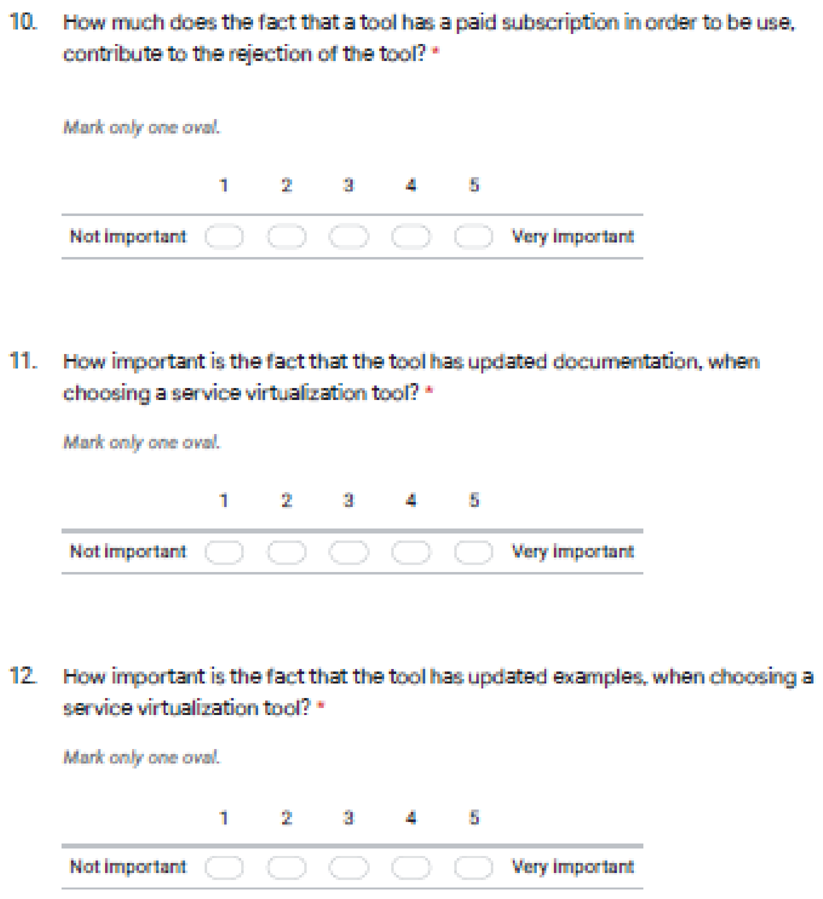
Mark only one oval.

	1	2	3	4	5	
Not important	◯	◯	◯	◯	◯	Very important

9. How important is the fact that the tool is free but with a limited amount of uses per month, when choosing a service virtualization tool? *

Mark only one oval.

	1	2	3	4	5	
Not important	◯	◯	◯	◯	◯	Very important



10. How much does the fact that a tool has a paid subscription in order to be use, contribute to the rejection of the tool? *

Mark only one oval.

	1	2	3	4	5	
Not important	◯	◯	◯	◯	◯	Very important

11. How important is the fact that the tool has updated documentation, when choosing a service virtualization tool? *

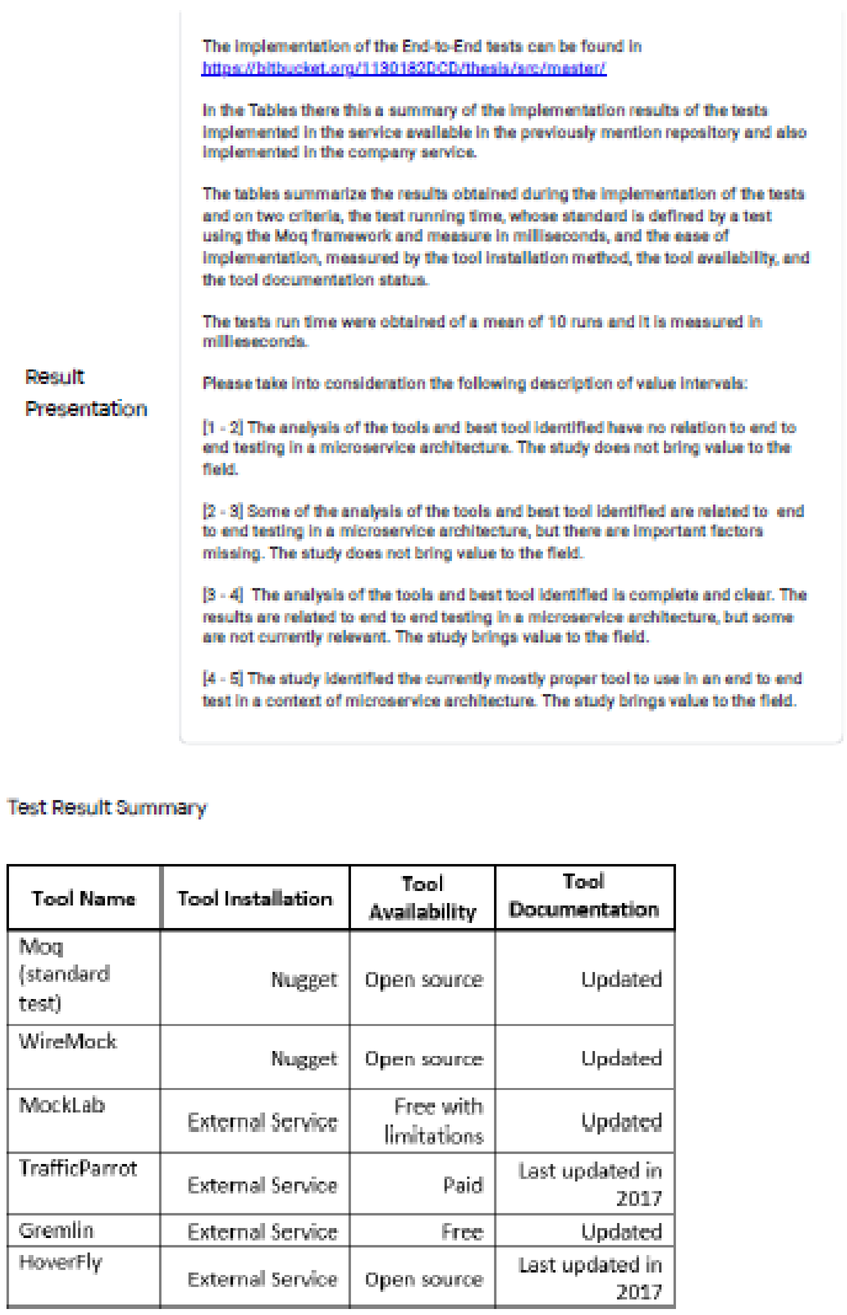
Mark only one oval.

	1	2	3	4	5	
Not important	◯	◯	◯	◯	◯	Very important

12. How important is the fact that the tool has updated examples, when choosing a service virtualization tool? *

Mark only one oval.

	1	2	3	4	5	
Not important	◯	◯	◯	◯	◯	Very important



Result Presentation

The Implementation of the End-to-End tests can be found In
https://bitbucket.org/1130182DCD/thesis/src/master/

In the Tables there this a summary of the implementation results of the tests implemented in the service available in the previously mention repository and also implemented in the company service.

The tables summarize the results obtained during the implementation of the tests and on two criteria, the test running time, whose standard is defined by a test using the Moq framework and measure in milliseconds, and the ease of implementation, measured by the tool installation method, the tool availability, and the tool documentation status.

The tests run time were obtained of a mean of 10 runs and it is measured in millieseconds.

Please take into consideration the following description of value intervals:

[1 - 2] The analysis of the tools and best tool identified have no relation to end to end testing in a microservice architecture. The study does not bring value to the field.

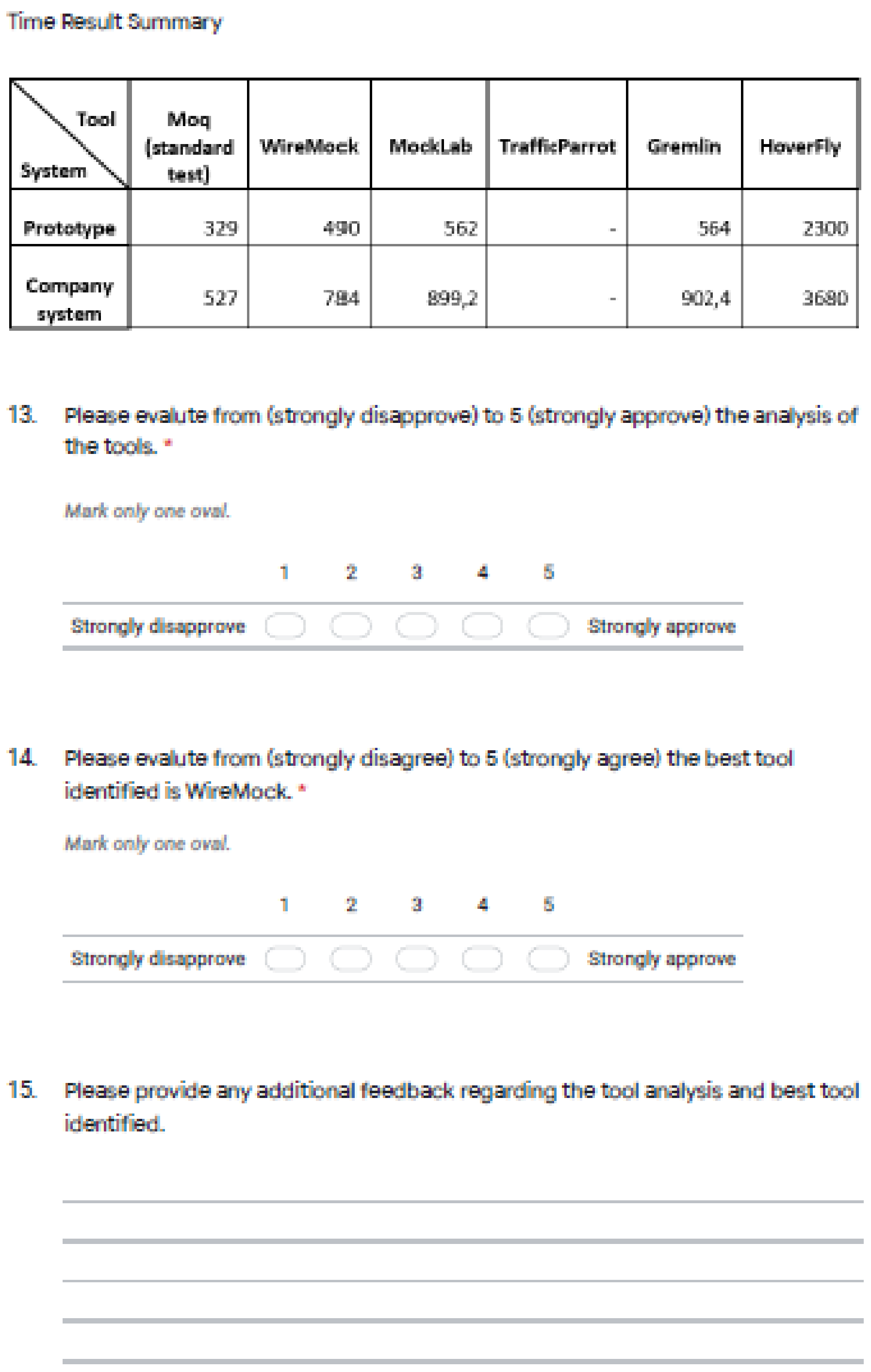
[2 - 3] Some of the analysis of the tools and best tool identified are related to end to end testing in a microservice architecture, but there are important factors missing. The study does not bring value to the field.

[3 - 4] The analysis of the tools and best tool identified is complete and clear. The results are related to end to end testing in a microservice architecture, but some are not currently relevant. The study brings value to the field.

[4 - 5] The study identified the currently mostly proper tool to use in an end to end test in a context of microservice architecture. The study brings value to the field.

Test Result Summary

Tool Name	Tool Installation	Tool Availability	Tool Documentation
Moq (standard test)	Nugget	Open source	Updated
WireMock	Nugget	Open source	Updated
MockLab	External Service	Free with limitations	Updated
TrafficParrot	External Service	Paid	Last updated in 2017
Gremlin	External Service	Free	Updated
HoverFly	External Service	Open source	Last updated in 2017


Time Result Summary

Tool / System	Moq (standard test)	WireMock	MockLab	TrafficParrot	Gremlin	HoverFly
Prototype	329	490	562	-	564	2300
Company system	527	784	899,2	-	902,4	3680

13. Please evalute from (strongly disapprove) to 5 (strongly approve) the analysis of the tools. *

Mark only one oval.

	1	2	3	4	5	
Strongly disapprove	◯	◯	◯	◯	◯	Strongly approve

14. Please evalute from (strongly disagree) to 5 (strongly agree) the best tool identified is WireMock. *

Mark only one oval.

	1	2	3	4	5	
Strongly disapprove	◯	◯	◯	◯	◯	Strongly approve

15. Please provide any additional feedback regarding the tool analysis and best tool identified.