# Evolving Legacy Enterprise Systems with Microservices-Based Architecture in Cloud Environments

**Safa Habibullah**

School of Computing

Edinburgh Napier University

This dissertation is submitted for the degree of

*Doctor of Philosophy*

December 2020

# Abstract

Many legacy enterprise systems suffer from a number of common and critical problems. Such systems have often been implemented in the past with hardware and software technologies which are now out of date. Furthermore, they have often been modified in piecemeal so as to allow them to cope with changed requirements, and the need for new functionalities, which have come to light since their initial implementation. Thus, they are often 'messy' in their implementations and difficult to use: the modules are no longer well-structured, and many dependencies exist across module boundaries; also some new functionalities may prove impossible to incorporate within them. The old-fashioned technologies on which they are based frequently are unable to deliver the speed and throughput required by the business environment, and such technologies usually do not offer the ease of modification provided by modern service oriented and networked systems. Further, new technologies are often available on a much wider range of platforms and are generally more scalable and flexible – leading to much greater ease of use.

Therefore, there is a need to migrate legacy systems to the newer technologies and in this process to construct the more well-structured systems. One path by which this can be done is to migrate the system to the Cloud-based technology, and in the course of this migration, to re-structure it into a microservices-based architecture. By doing so, the hope is that the resultant system will be easier to modify, offer higher performance, and offer other benefits as well, such as better security. To attain these hoped-for benefits, it is vital that the migration is performed in an appropriate approach.

The contributions of this thesis are to propose and validate an approach to the migration of a legacy system to a microservice-oriented architecture and Cloud based system. This approach is predicated on the creation and use of two sets of rules: a set of feature-driven microservice transformation rules and a set of feature-driven cloud migration rules. It is hypothesized that the correct interpretation of, and the appropriate adherence to, these rules will lead to the implementation of a new microservices-oriented and Cloud based system, which will replace the functionality of the legacy system, improve the QoS offered by this, in terms of non-functional requirements, and be far easier to modify in the future in order to cope with further functional and other requirements which may emerge.

To verify that the proposed approach and its associated rules are fit for the above purpose, two case studies are embarked upon. One involves the comprehensive conversion of a legacy system, via the rules, into a Cloud and microservices architecture based system implemented within a container technology environment – an environment which, according to the literature, is the one best suited to this purpose. The testing and implementation involved with that case study is focused on the microservice-oriented system's compliance with non-functional requirements such as throughput. The second case study is analysis-intensive, focusing on how a much more complex and much larger legacy system could be migrated in the same way. This latter case study is focused on interoperability, testability, maintainability, availability, and scalability. The results from these case studies verify the validity and efficacy of the approach and the rules which are developed for it, and lead to some insightful suggestions for future research.

# Acknowledgements

Just a few days and nights ago, all the hours I had were filled with work. Some moments from that time will never be forgotten; they will remain forever engraved in my heart. The hard work that this project entailed represents a link between tomorrow and a bright new future.

This project has been manifested and documented, just as my hopes and wishes have been manifested by the people who, with their involvement, have made such a difference to my life. Words cannot express my gratitude to these people: the pen stands respectfully still in their presence, the ink starts to go dry before we can begin mentioning their favours.

All praise is due to Allah, the giver of all knowledge. Thank you for blessing me with the strength and knowledge to complete this research work.

I am most indebted to my beloved husband, Abdulkader, who has been beside me at all times and he has always been a shoulder to rest on; he has supported me throughout this journey. Indeed, I would not be where I am today without him. I am grateful to my precious boys, Yousef and Omar. They are the light of my life and being with them encourages me to cope and to go through all the challenges necessary for providing them with a bright future.

My deepest thanks go to my parents, Hussain and Badreeyah for their endless love and support. Although they live far away in Saudi Arabia, they have always encouraged me and given me consistent spiritual support. Special thanks go to my brothers and sisters. Specially to my sister Afnan who has spent some time with me in Edinburgh looking after my kids.

I would like to express my huge appreciation of my director of study Prof .Xiaodong Liu and my supervisor Dr.Zhiyuan Tan for their untiring support , assistance, valuable advice and feedback in the course of my PhD study. My thankfulness also goes to my panel chair, Prof.Ahmed Aldubai who has shown an understanding of my situation and the difficulties I have faced during my PhD - and for all the support and help he has provided.

I would like to thank university of Jeddah which has given me the opportunity to continue my studies in the United Kingdom and has helped us to overcome all the difficulties which, otherwise, might have overcome us during our stay abroad.

Finally, a special thanks goes to my parents in law, my family in Saudi Arabia and also my friends in the UK who have become like a family.

# Publications from the PhD work

## Conference papers

1. Habibullah, S., Liu, X., Tan, Z., Zhang, Y., & Liu, Q. (2019). Reviving legacy enterprise systems with microservice-based architecture within cloud environments. In *Proceedings of the 5th International Conference on Software Engineering (SOFT 2019)*, Copenhagen, Denmark, June 2019. https://doi.org/10.5121/csit.2019.90713

2. Habibullah, S., Liu, X., & Tan, Z. (2018). An Approach to Evolving Legacy Enterprise System to Microservice-Based Architecture through Feature-Driven Evolution Rules. *International Journal of Computer Theory and Engineering*, 10(5).

## Journal paper

1. Habibullah, S., Liu, X., Tan, Z., Jaroucheh, Z., Zhang, Y. and Liu, Q., n.d. An Approach to Evolving Legacy Enterprise Systems to Cloud-Based Lean Microservice-Oriented Architecture of the Wiley-Software: evolution and process (under review).

# Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and acknowledgements.

Safa Habibullah

December 2020

# Contents

**List of Figures** **xi**

**List of Tables** **xiv**

**1 Introduction** **1**

1.1 Motivation and Problem Statement . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Aim and Objectives of Proposed Research . . . . . . . . . . . . . . . . . . 2

1.3 Research question and Contribution . . . . . . . . . . . . . . . . . . . . . 4

1.4 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.5 Thesis Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**2 Literature Review** **7**

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.2 Software Evolution: Maintenance Development and Migration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.2.1 Importance of Evolution, Development and Migration . . . . . . . 9

2.3 Legacy System . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.3.1 Software evolution vs legacy system migration . . . . . . . . . . . 13

2.4 Service-Oriented Architecture . . . . . . . . . . . . . . . . . . . . . . . . 16

2.5 Microservices . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.6 Microservices architecture vs Monolithic architecture . . . . . . . . . . . . 17

2.7 Bounded Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

2.8 Feature-Driven Evolution . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

2.9 Cloud Computing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

2.10 Container . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.11 Design Pattern . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.12 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

**3 Related Work** **29**
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.2 A Systematic Study on Microservices . . . . . . . . . . . . . . . . . . 29
3.3 Microservice-Oriented Architecture and Design . . . . . . . . . . . . . 31
3.4 Evolution into Microservice-Oriented Architecture . . . . . . . . . . . . 34
3.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

**4 Research Framework** **37**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.2 The Conceptual Framework of the Approach . . . . . . . . . . . . . . . 38
4.2.1 The Overview of the Framework System . . . . . . . . . . . . . 38
4.2.2 The Legacy System . . . . . . . . . . . . . . . . . . . . . . . 40
4.2.3 Middle Layer – Microservice Transformation . . . . . . . . . . . 41
4.2.4 Target System – Cloud Migration . . . . . . . . . . . . . . . . . 44
4.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**5 The Feature-Driven Migration Approach** **47**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.2 Feature-driven evolution . . . . . . . . . . . . . . . . . . . . . . . 48
5.2.1 Features in enterprise systems . . . . . . . . . . . . . . . . . . 48
5.2.2 Method to derive the microservice Evolution rules . . . . . . . . . 50
5.2.3 Feature relationships in enterprise systems . . . . . . . . . . . . . 53
5.3 Preconditions for feature-driven rules . . . . . . . . . . . . . . . . . 55
5.4 Feature driven microservice transformation rules . . . . . . . . . . . . 57
5.5 Feature Driven Cloud Migration Rules . . . . . . . . . . . . . . . . . 77
5.6 Microservices Transformation Roadmap . . . . . . . . . . . . . . . . . 87
5.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

**6 Case Study and Evaluation** **91**

6.1 Introduction . . . 91

6.2 Case Study 1 . . . 92

6.3 The Current, Monolithic, Architecture of RosarioSIS . . . 92

6.4 New Microservices Architecture of RosarioSIS . . . 96

6.5 Deployment in a Cloud Environment . . . 107

6.6 Test and Analysis . . . 109

6.6.1 Performance Test . . . 110

6.7 Case Study 2 . . . 116

6.7.1 The Legacy System Transformation to Microservice Architecture . 116

6.7.2 Deployment . . . 137

6.7.3 Evaluation . . . 140

6.8 Summary . . . 144

**7 Conclusions** **146**

7.1 Introduction . . . 146

7.2 Critical Analysis . . . 147

7.2.1 Objective I: Understand the theoretical background relating to the migrating of legacy systems, by proposing a holistic framework approach for such migration. . . . 147

7.2.2 Objective II: Apply UML diagram in order to specify the migration rules . . . 148

7.2.3 Objective IV: Evaluation with Case Studies . . . 149

7.3 Contributions . . . 149

7.4 Conclusion . . . 151

7.5 Limitation and Future Work . . . 152

**References** **153**

**Appendix A Performance Test samples** **160**

# List of Figures

1.1 The research methodology phases. . . . . . . . . . . . . . . . . . . . . . . . 5

2.1 Monolithic architecture vs microservice architecture . . . . . . . . . . . . 18

4.1 Overview of the framework . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.2 The conceptual framework of the approach . . . . . . . . . . . . . . . . . 40
4.3 Microservices transformation process . . . . . . . . . . . . . . . . . . . . 42
4.4 Cloud Migration Guidance . . . . . . . . . . . . . . . . . . . . . . . . . . 45

5.1 Typical Features Model in Enterprise Systems . . . . . . . . . . . . . . . . 52
5.2 Update Master DB based on Request priority. . . . . . . . . . . . . . . . . 82
5.3 The Process to transform RosarioSIS legacy system to a microservice-based architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

6.1 Monolithic architecture of RosarioSIS . . . . . . . . . . . . . . . . . . . . 93
6.2 RosarioSIS Monolithic ERD . . . . . . . . . . . . . . . . . . . . . . . . . 95
6.3 Grades' Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
6.4 Students' Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
6.5 School Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
6.6 Users' Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
6.7 Microservice Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 101
6.8 Microservice Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
6.9 User API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
6.10 School API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
6.11 Grade API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

6.12 Student API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
6.13 SAPI GET operation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
6.14 API input Fields . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
6.15 Example of API Message Response . . . . . . . . . . . . . . . . . . . . . 106
6.16 The RosarioSIS Legacy System Deployment . . . . . . . . . . . . . . . . . 108
6.17 The RosarioSIS Microservice-Based Architecture Deployment . . . . . . . 109
6.18 Average response times for the RosarioSIS legacy system and the RosarioSIS microservices architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 114
6.19 Average throughput for the RosarioSIS legacy system and the RosarioSIS microservices architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 115
6.20 ERP2 Entity Relationship Diagram. . . . . . . . . . . . . . . . . . . . . . 118
6.21 Core microservices from the ERP2 legacy system . . . . . . . . . . . . . . 121
6.22 Supply-chain domain . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
6.23 Human resource management domain . . . . . . . . . . . . . . . . . . . . 122
6.24 Financial management domain . . . . . . . . . . . . . . . . . . . . . . . 123
6.25 Planning and scheduling domain . . . . . . . . . . . . . . . . . . . . . . 123
6.26 Quality domain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
6.27 Product domain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
6.28 Customer relationship domain . . . . . . . . . . . . . . . . . . . . . . . 125
6.29 Service domain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
6.30 Inventory subdomain . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
6.31 Forecasting and Demanding subdomain . . . . . . . . . . . . . . . . . . . 128
6.32 Supplier and Customer subdomain . . . . . . . . . . . . . . . . . . . . . 128
6.33 Purchase subdomain . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
6.34 Project subdomain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
6.35 Training and recruitment subdomain . . . . . . . . . . . . . . . . . . . . 130
6.36 Warehouse subdomain . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
6.37 Manufacturing subdomain . . . . . . . . . . . . . . . . . . . . . . . . . 132
6.38 Purchase subdomain . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
6.39 Payroll subdomain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
6.40 Enterprise microservices . . . . . . . . . . . . . . . . . . . . . . . . . . 138

6.41 Monolithic Enterprise application. . . . . . . . . . . . . . . . . . . . . . . . 141

A.1 User Module Summary Report for 40 requests in 0.5sec . . . . . . . . . . . 160
A.2 User Module Summary Report for 20 requests in 0.5sec . . . . . . . . . . . 161
A.3 User Module Summary Report for 60 requests in 0.5sec . . . . . . . . . . . 161
A.4 View User Module Result in Table for 60 requests . . . . . . . . . . . . . . 162
A.5 View User Module Result in Table for 60 requests . . . . . . . . . . . . . . 162
A.6 View User Module Result in Table for 80 requests in 4.48 sec (1) . . . . . . 163
A.7 View User Module Result in Table for 80 requests in 4.48 sec (2) . . . . . . 163
A.8 View User Module Result in Table for 80 requests in 4.48 sec (3) . . . . . . 164
A.9 Student Module Summary Report for 45 requests in 0.5 sec . . . . . . . . . 164
A.10 Student Module Summary Report for 150 requests in 3.5 sec . . . . . . . . 165

# List of Tables

5.1 Main concerns for the feature driven rules . . . . . . . . . . . . . . . . . . . 56

6.1 Monolithic Instance Details . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
6.2 Microservice Instance Details . . . . . . . . . . . . . . . . . . . . . . . . . . 109
6.3 Requirements for Data Modelling . . . . . . . . . . . . . . . . . . . . . . . . 135
6.4 Feature-Driven Migration Rules selection. . . . . . . . . . . . . . . . . . . . 140

# Chapter 1

# Introduction

Microservice-oriented architecture represents a modern, alternative way to develop an application. This style of development is designed to solve several challenges associated with monolithic enterprise applications. The idea of microservice-oriented architecture is Relatively new, and it is rapidly becoming popular in the world of software development due to its support to the key characteristics of service-oriented architecture and its suitability for deployment within the cloud. The microservices themselves are small and purpose-built and have no dependencies outside of themselves; they do not need to know anything about other microservices' implementation or structure. Owing to the benefits of these technical attributes, this thesis will examine them in greater detail, with regard to migrating a legacy system to the microservice-oriented architecture.

## 1.1 Motivation and Problem Statement

Over the last decade, the cost and availability of hardware (physical or virtual) has had a significant impact on system evolution. In the past, systems were developed and designed for specific hardware at a fixed scale. More scalable technologies, such as virtualisation, were subsequently introduced, providing greater flexibility in terms of enterprise system development. Recently, cloud computing has resulted in another wave of innovation. Cloud computing is becoming ubiquitous, spreading across many different sectors, and has become a key element that creates valuable benefits for many enterprises. One of the main purposes of adopting cloud technology is its flexibility in terms of cost per use, as well as of its other features, such as scalability, and efficiency, and so on.

Furthermore, the continuous development of cloud computing, both in academia and industry, means that new concepts and techniques are frequently being introduced. This evolution has been achieved via the combination of two basic concepts: the microservice and the container. This mechanism presents innovative features, such as fast start-up, the separation of units, etc. The new techniques allow a monolithic application to be broken down into smaller independent services with respect to their functionality. Microservices can communicate with other services via an Application Programming Interface (API) gateway which allows developers to update each service independently without affecting others.

Owing to the aforementioned technical advantages, microservices provide an innovative solution to migrate legacy systems toward modern day practices. Implementing a microservice in the cloud represents a new method to modernise software applications and has been adopted by enterprises for next-stage system deployment. However, microservices are not without their issues and challenges. It is important to understand this new paradigm, as it is changing how enterprises deal with information, and to resolve its weaknesses to build the best possible foundation for enterprises.

The present research develops a novel approach to legacy system evolution towards microservice-based architecture and a Cloud hosted environment. The method used comprises a set of feature driven rules and a comprehensive illustration of the working process. Also, the rules are applied in two case studies and in an evaluation of a set of non-functional attributes.

## 1.2 Aim and Objectives of Proposed Research

Driven by the above motivations, the vision of this research is to provide a comprehensive process for a combination of microservice and cloud computing. To a wider extent, this would consequently enhance system functionality and evaluation. Therefore, the aim of this thesis is to convey the legacy system features, attributes and modules to the microservice-oriented architecture through a conceptual framework. In this thesis, the framework is defined as a layered structure which involves a combination of microservice-oriented architecture, cloud computing technology and relevant techniques. Within the framework, the relevant components are interrelated and work together to serve a certain part of the aim. More specifically, the aim is delivered through the following objectives:

1. **propose a holistic approach framework based on the theoretical background relating to the migration of a legacy system**

In order to mitigate the migration process, a conceptual framework have been proposed. The framework is composed of three layers:

**The first layer** concentrates on understanding the legacy system, through reading the source code manually, defining the modules and documenting these modules and relationships among them.

**The second layer** focuses on how to build a (micro)service by identifying the candidate service to be transformed into microservices. During this step, based on the feature-driven transformation rules, a module of the legacy system is selected. From this module, a new microservice can be implemented to enable the existing microservice to be more conducive to the continuous change.

**The third layer** defines the host environment and new architecture requirements to support the complete delivery of microservice-oriented architecture based on the feature-driven migration rules.

2. **To propose a set of feature-driven rules for transforming a legacy enterprise system into a lean architecture populated with microservices.**

A set of rules is designed to transform legacy architecture into microservice-oriented architecture, to assist in the process of extracting and implementing microservices. Each rule shows a specific scenario to choose from and then apply.

3. **To propose a set of feature-driven cloud-oriented migration rules that define the best principles and practice to migrate an enterprise application to the Cloud, addressing the various technical challenges, including security, functionality and performance.**

A set of rules is specifically designed for the process of cloud migration. Companies will benefit from this rule if they have the option to deploy microservices in cloud environments.

4. **To carry out concrete case studies in order to evaluate the effectiveness of the proposed approach.**

The final objective is the validation and evaluation of the proof of concept through case studies and two sizes of enterprise system are applied: small and medium. The case studies ought to provide performance comparison between the legacy system and the new architecture. Also, how the new architecture enhances functional and non-functional requirements.

## 1.3 Research question and Contribution

As a result of the findings from the literature review, the following research question has been formulated, relating to the current state of the art in cloud-based enterprise computing: ***how to engage the latest Cloud computing technologies, service-oriented architectures and software evolution to transform legacy enterprise systems into the best computing solution possible?*** The proposed research aims to develop a novel approach to the migration of a legacy enterprise system into a lean system architecture which is supported with microservices and hosted by cloud platforms. The new system will enjoy the benefits of cloud computing and the microservice-oriented architecture and the much-improved performance, maintainability, reusability and security that this can offer.

The present research attempts to overcome the challenges faced by monolithic architectures by offering, as an alternative, a comprehensive microservice architecture deployed on a cloud service, as informed by the comprehensive literature review. Furthermore, it closes the research gaps found by providing an approach to the migration of legacy systems towards microservice-based architectures within cloud environments. Through this research, several contributions to knowledge are made:

- The definition and implementation of a complete process for the migration of legacy systems toward microservices architectures hosted by Cloud platforms. This process considers the three evaluation issues of performance, security, and functionality.
- A set of feature-driven microservices-specific transformation rules. These rules define the process for modernising an existing legacy system, with a special emphasis on analysing the implications regarding runtime performance, functionality, security,

scalability, maintainability and testability, aiming to provide guidance for the migration of legacy enterprise systems.

- A set of feature-driven Cloud migration rules, which constitute a plan for migrating a system toward cloud computing, are defined by proposing rules for specific situations and activities.
- The case studies and related evaluations for proof-of-concept validation and evaluation, where two case studies are used to examine the proposed approach. These case studies involve different sized enterprise systems: small and medium size. The experimental results intend to provide a comprehensive detailed analysis of how the rules are followed to move from a legacy to a microservices system which is then deployed in the Cloud.

## 1.4 Methodology

This research concerns legacy systems, cloud computing, and microservices. It aims to develop a highly effective scheme, driven by migration frameworks and patterns, whereby legacy systems may be modernised. A novel approach is developed to the migration of legacy enterprise systems toward lean system architectures which are supported by microservices and cloud computing. This research will be carried out via three phases with respect to its objectives, as shown in Figure 1.1.

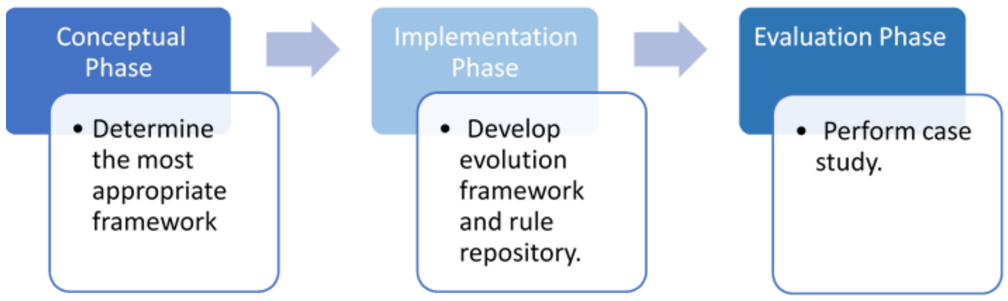

Conceptual Phase
• Determine the most appropriate framework
Implementation Phase
• Develop evolution framework and rule repository.
Evaluation Phase
• Perform case study.

**Figure 1.1** The research methodology phases.

The methodology includes a systematic literature review, to survey the current state of the art of cloud computing, service-oriented architectures, enterprise systems, design patterns and microservices.

Once the framework and the rules were ready for testing, two case studies were then conducted to justify and evaluate the proposed approach and the functionality and efficiency of the subsequently developed system.

At the same time, a number of papers were published based on milestones in the present research. These inform the research community of the exact contribution to the field that this research makes and, moreover, they allow other researchers to assess the work in terms of advances in the field.

## 1.5 Thesis Structure

This thesis is organised as follows:

Chapter 1 Introduces the research along with the problem statement, the aim and the objectives of the research, the contributions to knowledge and the research methodology.

Chapter 2 Provides an overview of the background and environment of this research including discussions surrounding software migration, legacy systems, cloud computing, microservices and major challenges.

Chapter 3 Discusses previous studies related to this research in detail, focusing on: a systematic study of microservices, microservice-oriented architectures and design, and migration toward microservice-oriented architectures.

Chapter 4 Defines the proposed framework which enables the migration of legacy system toward microservice-oriented architectures in cloud environments.

Chapter 5 Presents the feature-driven rules which are split into two groups: feature-driven transformation to microservices and feature-driven migration rules to the cloud.

Chapter 6Presents the two case studies aimed at validating the feature-driven rules and evaluating the resultant performance, functionality, and security.

Chapter 7 Summarises the research by presenting the conclusions and suggestions for future work.

# Chapter 2

# Literature Review

## 2.1 Introduction

This chapter discusses software migration and why migration is often essential. A definition and description of the challenges legacy systems face and how microservices can tackle these issues are presented. Also, this chapter introduces the concept of service-oriented architecture, comparing it to the concept of microservices architecture. Cloud computing is presented as a means of deployment and the bounded context idea is shown as a technique for defining microservices boundaries. In addition, feature driven migration is presented as a method for supporting the definition of rules driven by features. Finally, the chapter is summarised with respect to the main challenges of the research.

## 2.2 Software Evolution: Maintenance Development and Migration

With the passage of time, software will evolve; it will incorporate enhancements which may make it more functional. When it comes to technicalities, software issues can be fixed, or the application's features can be expanded. In addition, the software may be made to perform better or become more available, maintainable, or secure.

Without being able to distinguish between the different types of software evolution, from software maintenance and software migration, understanding these types of changes would be difficult. Swanson in [1], considers three foundations upon which modifications can be made: (1) migration to, for instance, a new architecture; (2) the refinement of existing services through, for instance, the addition of a new feature, i.e. development/evolution; and (3) the 'correction' of a current system, i.e. maintenance.

Constructing a new but related design out of the current system would be considered software development rather than software maintenance. Software development is often associated with migration (to a new architecture and/or new kind of platform) rather than with software evolution as such. However, migration may also deal with the enhancement of the interoperability of the system, making the system perform better, and providing support for new features [2] [3].

The modification of proprietary software is associated with a number of behaviours - which have been identified by Lehman and his colleagues [4].Many people also refer to their specification as Lehman's Laws. These eight laws constitute the Laws of Software Evolution and discuss the ways in which the process depends on feedback. According to these laws, it is not possible to escape change and this is not the result of poor programming [4].When it comes to the safe implementation of new functionalities and updates, changes are often necessary because the system comes up against limitations, as detailed in Lehma's Laws:

1. *Continuing change:* The system may begin to lose its usability if continual modifications are not made to it in order to satisfy client requirements.

2. *Increasing complexity:* The complexities of the system will start to increase because of maintenance-based changes, if no remedial work is undertaken to reduce such complexities.

3. *Self-regulation:* The process of evolution creates measures of processes and products that maintain nearly normal distributions; this shows self-regulation in the evolutionary process.

4. *Conservation of organizational stability:* Over the entire system lifespan, an evolving system has the same average effective global activity rate . Put differently, there is little difference when it comes to how much effort is required, on average, to produce a new release.

5. *Conservation of familiarity:* Satisfactory evolution can only be realised if the system's behaviour and content has been clearly understood by every person involved, including users and developers, in relation to system evolution. Poor understanding may result if a particular release has too great a change from precious iterations. Considering this, there should be consistency in the average incremental growth experienced by an evolving system.

6. *Continuing growth:* With the passage of time, further customer requirements are fulfilled by continually increasing a system's functional content.

7. *Declining quality:* If new operational environments have not been considered and the system's design has not been diligent fine-tuned, it should be considered that the system features will decline with the passage of time.

8. *Feedback system:* The various activities that are part of the evolution process relating to the system include feedback on multiple levels, agents, and loops. For a current system to continuously undergo evolution by providing better quality attributes and more functionalities, the recognition of such complicated interactions becomes necessary for developers.

Large organisations often develop specially tailored and large systems that follow these laws. However, all processes of system modification are not equally, nor clearly, addressed by them. Moreover, the type of the organisation (medium, small, or large) has also been excluded from the specifications above [5]. The laws consider modification of the present components and addition of new components to be the only actions necessary for implementing changes, i.e. software evolution.

### 2.2.1 Importance of Evolution, Development and Migration

Organisations invest a great deal of money into their key business resources, and among the most important of these are their software systems. It is important to frequently alter and modernise an enterprise's software system so that its value can be preserved and maintained. Rather than the development of new software, maintenance and improvement of the current software is what consumes most of the company's software budget.

The software lifecycle involves making key modifications to a software system, sometimes known as software evolution phases. Software evolution also relies heavily on incremental change, which is the piecemeal addition of new features to the software. Incremental change can be guided with the use of the software evolution concept, which also provides a framework for an application case study. Moreover, there is great significance in the domain concept [6].
Software evolution is a term for a software alteration process which responds to design and requirement changes. Software evolution can be considered an important concept in various cases that involve carrying out changes to a system in production. Current information technology is heavily dependent, at every level, on software development. All sectors of economic activity rely on software, whether private, manufacturing, commerce, transportation, industry, or the government sector. In general, the software development process must be based on change which is focused on mitigating the adverse impact of software aging [2] [5].

Because of the adverse impact of software aging, industries experience major social and fiscal changes across every sector. Considering this, developments and improvements in the methods and tools used are necessary to avoid or counteract the main issues which present themselves in relation to software aging [2] [7]. As a result, the creation of more methods and tools for the purposes of developing or protecting a software's positive features, regardless of its complexity and size, is the research problem which presents itself when it comes to software evolution. According to the Lehman's Laws of software evolution, which take into account the notable fact that this is a software-driven era, changes to a system can bring about degradation in the quality of a piece of software if no dynamic counter actions and procedures are used. This is a consequence of the software systems involved becoming less presentable, accessible and consistent, as well as the loss of other such characteristics [2]

There are several different factors which dictate the course of software evolution, including organisational issues, such as an organisation's software update and maintenance procedures, and the skillsets of the people available to make the required changes. Evolution is something which may take place across the entire lifetime of a software system; the need for change may be identified at any time while the software is in use. Thus, change identification is the first and most significant step in a software evolutionary process. Once a necessary or desirable change is identified, proposals may be made with regard to it. At least one of these proposals will then receive the go-ahead resulting in the next update to the [**?** ].

Once the software has been amended, various handover issues may become apparent; how these manifest will depend on the approach taken in the course of the development. Where agile development has been the paradigm, and the team involved with bringing the system on-line is not particularly cognisant of agile techniques, then documentation at a detailed level must be requested and provided. This is not something which is within the purview of agile techniques per-se. Where a pre-planned approach is employed, as opposed to an agile approach, the development team may be more likely to start from scratch with modules, and design/plan from first principles [8]. A wholesale planned process of this kind may dictate the employment of automated tests and so on which would otherwise be unnecessary. Around 65% of the total costs of a piece of software, over its entire lifespan, may be attributable to software evolution, and this may reach 75% when a system enjoys a particularly long lifespan [8]. One of the challenges which leads to large costs being incurred is having to cope with the migration of large quantities of data (to different formats, etc.). Once a necessary change has been identified, the subsequent modification, reformatting and updating of the system's data can be an enormous task. In addition, high levels of complexity are often encountered when trying to achieve this task, and the problems caused by coding errors in the originally implemented system. Thus, it is often the case that the developers must have a full understanding of the system before they are attempting to change and make the desired amendments. Further, very often, amendments lead to additional (and perhaps previously unsuspected) ramifications and the developers must deal with these as well as make the changes which were their original intent.

In conclusion, this discussion stipulates that the challenges faced by an evolution team generate a need for better strategies to be adopted by business entities when developing software, in order to reduce the challenges in the evolutionary stages.

## 2.3 Legacy System

Because technology progresses, enterprises face the dilemma of legacy systems. Various definitions of 'legacy systems' can be found in the literature, including:

1. Bennett [9] defined a legacy system as a large system that was originally developed over 40 years ago, but, nevertheless, is still used and plays a critical role in an enterprise.

2. Sommerville [5], stated that a legacy system is any older system that is still Runing, and is important to a business' operation. Such systems are often built using now obsolete technologies and languages. Moreover, such systems also include legacy processes and procedures.

Here, legacy system is defined simply as an outdated system that is in use by an organisation and has been so for more than 10 years, despite much more recent technology related to the application being adopted by many other organisations. A further attribute of such systems is that money must be spent on them to maintain their usefulness.

In the literature, there is some confusion between the terms “legacy system” and “lean system”, in that if a legacy system is still an important element within an organisation, it could be considered a lean system to its users. However, lean and legacy systems are two separate concepts. A lean system means that the system has no redundant components while a legacy system means an old, out-of-date system. By and large, a legacy system is not lean but there could be some legacy systems who are monolithic and lean. In this work, the two concepts are considered distinct and only legacy systems are discussed.

Normally, a legacy system is not same as the original system. This is because the legacy system has evolved but the evolutionary process has not gone well due to a number of factors; the system was changed, and many different developers were engaged in these changes, making it difficult for any single developer to understand the entire system [5]. [7]. Enterprises and organisations are always seeking more modern systems that meet their needs and requirements but replacing the legacy system with a modern system can incur many challenges and risks. For example, there may be no complete document that contains all the specifications of the legacy system. Even if such a document exists, it might not point to all the detailed changes required. In addition, unexpected problems may appear during a transformation process [5].

### 2.3.1 Software evolution vs legacy system migration

System evolutionary processes can be classified into three distinct categories: maintenance, modernisation and replacement. Different evolutionary activities are applied at different stages of the software system life cycle. Repeated maintenance can improve a software system and allow it to meet the growing business requirements, at least temporarily. However, as the system becomes obsolete and outdated, the effectiveness of mere maintenance falls behind the business needs. At this point, modernisation becomes crucial, although this represents a greater effort in terms of time and functionality than maintenance activity. Lastly, when the system can no longer be upgraded or evolved, it must be replaced [10] [11]. Maintenance, modernisation and replacement are briefly discussed below to give a better understanding of the various kinds of development that a legacy system can undergo.

1. **Maintenance:**

   Maintenance is a piecewise amendment process which is applied to a software system. A change to a system's architecture does not come under the heading of 'maintenance', since that focuses on bug corrections and/or small enhancements. Maintenance is an on-going procedure and supports the evolution of any system, but it does have many limitations. One of these limitations arises from the fact that new technologies will become available. Second, the cost of legacy system maintenance grows over time. Third, legacy systems must be modified in order to satisfy new business requirements and this becomes an increasingly difficult process as time goes on [10] [11].

2. **Modernisation:**

   Modernisation, as compared to maintenance, results in more extensive changes to a software system. These changes will often involve the restructuring of a legacy system, improving its reliability and enhancing its functionality. 'Modernisation' implies the gradual and partial enhancement of a legacy system, using new technologies, while retaining a significant portion of the existing system. Such changes often include system restructuring, significant functional enhancements and/or improved quality attributes such as maintainability. Modernisation is applied when a legacy system requires more ubiquitous changes than those possible in the course of maintenance. However, where the system still has some value, and where this must be preserved, is considered. Software modernisation attempts to evolve a legacy system, or components of such a system, when piecemeal evolutionary practices, such as maintenance, can no longer achieve the desired result [10] [11].

3. **Replacement:**

   Replacement means the building of a new system from scratch. Replacement is at least one option for a system that has been found to be unable to keep pace with business requirements. Replacement may have several risks associated with it; these should be considered before replacement is carried out. First, the replacement process involves the development of a new system and this is likely to be extremely resource intensive. Moreover, the IT personnel may not all be familiar with the new technology proposed for use in the new system. As well as this, the replacement process requires more testing for validation processes than other approaches. Finally, there is no absolute guarantee that the new system will be as robust and functional as the old one [10] [11].

   On the other hand, legacy system issues, such as the fact that they may run on very old processors which are much slower than those currently available, affect system performance and make expansion difficult, if not impossible. To overcome these issues several solutions have been suggested: wrapping, re-development, and migration.

1. **Wrapping**

   Wrapping is a method whereby a legacy component may be modified and/or improved. A wrapper does not alter the source code explicitly but nevertheless results in the modification of the functionality of the legacy features. Wrapping means that the legacy component is encapsulated within a new software layer that provides new functionality and hides the complexities of the old component [12].

2. **Re-development**

   Re-development means, essentially, the rewriting of an existing application; building a new system from scratch using a modern architecture, probably running on a different, more up-to-date, environment, and applying different tools and database functionalities [12].

3. **Migration**

   In circumstances where wrapping cannot produce an acceptable result, and re-development is not reasonable due to significant risks (cost and time), the migration of a legacy system to a new modern architecture is an alternative solution. If such a migration process is successful, it will offer long-term advantages: better system awareness, faster maintenance, lower costs and greater flexibility in relation to meeting future needs. When purely migrating software, developers will seek to keep as much of the system the same (as the legacy system) as possible; this can be in terms of features, overall design, and so on. Thus, migration is distinct from re-development. The changes required by the effort to migrate often include programme redesign and attribute modification. Nevertheless, the main features of the current system will be maintained [12].

   In terms of the development of legacy systems, system migration differs from system upgrades, such as in those in the evolutionary process. System migration is the process of replicating one system and integrating it into another system, while system upgrade is the process of replacing your existing system with a newer version of the same system. The upgrade process can be straightforward if the new version meets the requirements and has a similar database structure, files, etc., to the current version of the legacy system. Upgrades are cheaper than migration to a new environment and come in different types, such as hardware upgrades, due to the need for more memory to run the software, or software upgrades, which are desired because of the promise of new or improved features.

   Legacy system migration is a powerful means of developing new systems and/or enhancing old ones to keep pace with the latest technologies: activities mentioned in Lehman's Laws. Based on these goals, to have a successful journey towards them, the most appropriate migration plan should be provided. The challenges involved in evolving a legacy system in any way are to understand the functionality, performance, security and operation of the legacy system to determine what type of change should be applied [13].

## 2.4 Service-Oriented Architecture

Service-oriented architecture (SOA) is a method of designing and managing the deployment of applications and software into business services that are executable and accessible based on public standards for interoperability [14]. Service orientation is an architectural approach that promotes the integration of business functions as linked to repeatable services and tasks. Some of the key technical concepts of SOA include services, interoperability and loose coupling. In SOA, the application is divided into components that provide the various necessary services through protocols over networks. The co-operation of these services creates the functionality of the whole system. The fundamental concept is to have as few dependencies between the services as possible. The core components of a SOA architecture are:

- SOA services: A service is an item of self-contained business functionality: that is, an SOA service is designed to encapsulate some functionality that is meaningful to the business. The functionality may be simple (as in retrieving a customer address) or complex (e.g., encapsulating the business process for fulfilling a customer order).
- The Enterprise Service Bus (ESB): The infrastructure that enables high interoperability. This mediates between service providers and consumers, provides value added services, and offers a platform of high throughput, reliability, and scalability (dependent on the infrastructure) catering for both providers and consumers.
- Service monitoring and management: : Service management via a service registry/ repository provides service discovery and lifecycle management essential to efficiently manage significant numbers of services.

Loose coupling is a crucial concept in SOA; the above components and functionalities are underpinned by this concept. It is a fundamental principle of SOA by design, minimising dependencies between components as much as possible. This is achieved via removing the close connections that exist between service providers and consumers, and this is supported via the use of service proxies and an ESB. All these uncoupling activities involve abstracting the service descriptions, and designing the SOA services appropriately [15] [16].

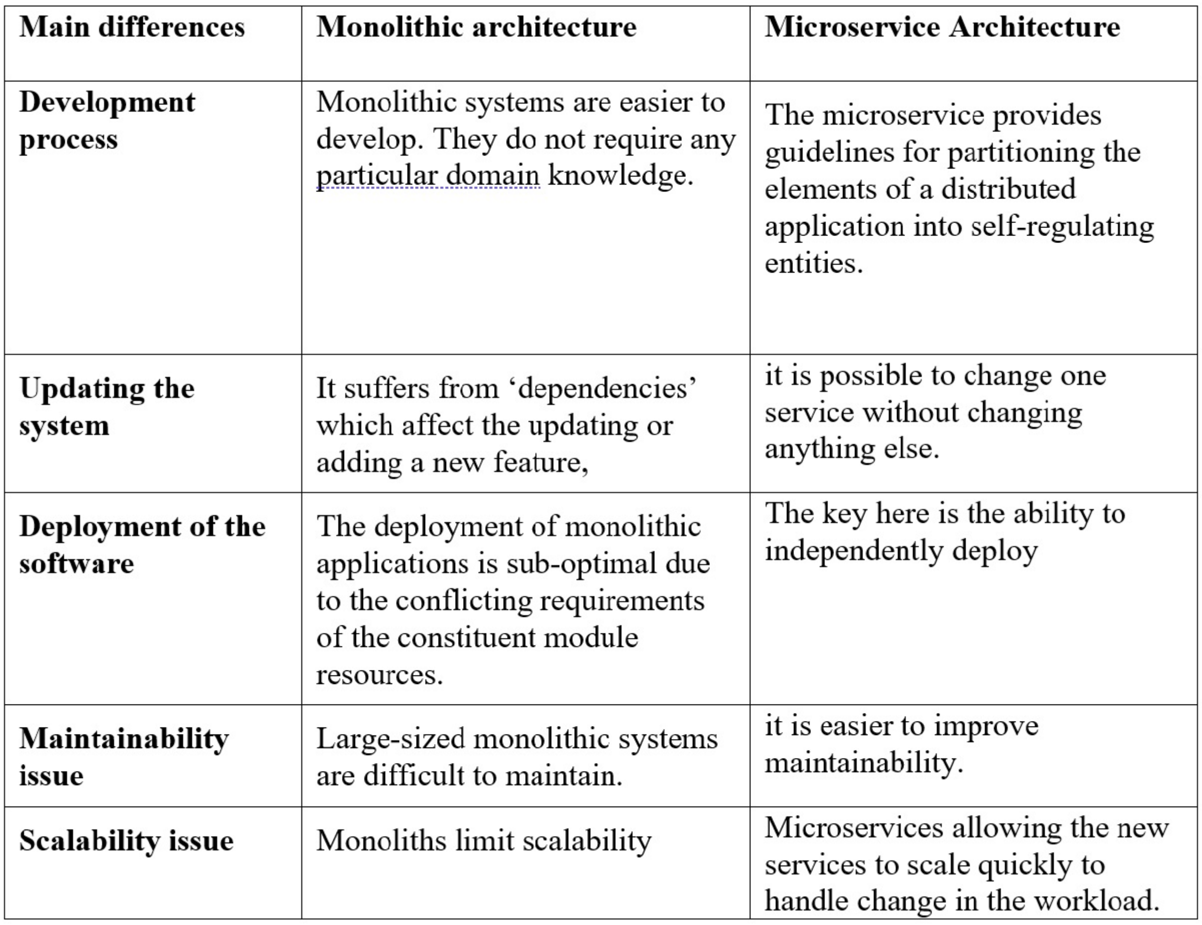
## 2.5 Microservices

An architecture based on microservices represents a lightweight version of an SOA [17] [18]. Although many consider microservices as distinct architectures [19]. a microservices architecture implies the use of a collection of small services with independent responsibilities. There are also a number of other distinguishing characteristics, including the fact that each microservice can choose its own architecture, technology, programming languages and platform, and crucially can be managed, deployed and scaled independently [20] [21] [22]. [87]. To make this independence aspect more effective, each microservice should have their own data within their boundaries [23].The modular and loosely coupled approach that is based on the microservices architecture eliminates dependencies and promotes quick testing and deployment of code changes [24]. There is a relationship between microservices and state-of-the-art technologies which simplifies automated deployment [25]. Self-contained microservice deployment units can be produced thanks to the concept of containers. In addition, the elements are heterogenised as a number of teams can experience design autonomy while all working on the same project, due to the paradigm of microservices. These advantages, which are enhanced by loose coupling, greater modularity and reduced dependencies, all hold the promise of simplifying the integration task. Above all, microservices architectures may provide more flexibility and open up many more choices regarding how to solve migration and other problems in the future [17] [26].

A microservice is usually created with three layers: an interface layer, a business logic layer, and a data storage layer, all contained within a much narrower bounded context than that of the equivalent legacy feature [27]. TThe business logic of each microservice deals with its (the microservice's) specific responsibilities or business purpose, these are independently planned, built and deployed, microservice by microservice [28].

## 2.6 Microservices architecture vs Monolithic architecture

The detailed examination in Figure 2.1Table 2.1 looks at how microservices were proposed to cope with the monolithic architecture problems, how it compares with these, and elements that must be considered before making the switch.



| Main differences | Monolithic architecture | Microservice Architecture |
|---|---|---|
| **Development process** | Monolithic systems are easier to develop. They do not require any particular domain knowledge. | The microservice provides guidelines for partitioning the elements of a distributed application into self-regulating entities. |
| **Updating the system** | It suffers from 'dependencies' which affect the updating or adding a new feature, | it is possible to change one service without changing anything else. |
| **Deployment of the software** | The deployment of monolithic applications is sub-optimal due to the conflicting requirements of the constituent module resources. | The key here is the ability to independently deploy |
| **Maintainability issue** | Large-sized monolithic systems are difficult to maintain. | it is easier to improve maintainability. |
| **Scalability issue** | Monoliths limit scalability | Microservices allowing the new services to scale quickly to handle change in the workload. |

**Figure 2.1** Monolithic architecture vs microservice architecture

Monolithic architectures tend to rely on the sharing of the resources which are resident on the same machine, including memory, files and databases. Monoliths are often single executable artefacts, whose modules cannot be executed independently because they rely on shared resources. This makes it difficult for monoliths to distribute naturally without using specific frameworks or ad hoc solutions [26] [29]. In the context of migration to cloud-based distributed systems, this represents a critical limitation, as it leaves the synchronisation responsibilities to the developer. Some of the significant obstacles include:

1. The difficulty of maintaining large-sized monolithic systems.
2. The fact that these suffer from 'dependencies' which affect the updating of, and the adding to, libraries because such libraries can easily become inconsistent - making it difficult to run and/or compile systems.

3. A change in any of the modules of the monoliths often requires the rebooting of the entire application.

4. The deployment of monolithic applications is suboptimal due to the conflicting requirements of the constituent module resources; some are computational-intensive, others memory-intensive, and others need specific components such as SQL-based databases. When it comes to choosing a deployment environment, a developer may have to pick a one-size-fits-all configuration that is either sub-optimal or expensive with respect to individual modules.

Monoliths limit scalability, and the usual strategy used to handle the increasing of inbound requests is to create a new copy of the application and split the load among the resultant instances. However, it could be the case that increased traffic stresses a subset of the modules, which makes the allocation of new resources for other components inconvenient [30].

Thus, microservices were proposed to cope with these problems. But the key here is the ability to independently deploy [31]. Each microservice is equipped with dedicated memory persistence tools such as databases, and since all the components of a microservice architecture are, by definition, small services, it derives its distinguishing behaviour from the coordination and composition of its parts through messages. The microservice style does not favour or forbid any particular programming paradigm. However, it provides guidelines for partitioning the elements of a distributed application into self-regulating entities, each addressing one of the concerns [32].
Because a microservices architecture consists of small loosely coupled services, it is possible to change one service without changing anything else. Each service performs its own process and a suitable programming language can be chosen for each of these [31].
The principles of a microservices architecture help project managers and developers and provides guidelines for the design and implementation of distributed applications [33]. Following such principles, developers can focus on the execution and testing of several similar functionalities.

## 2.7 Bounded Context

The bounded context method is a type of domain-driven design (DDD) that is applicable to microservices [34]. One of the strictures of DDD s that all the languages to be used in the development must have context strictness. Bounded contexts is a paradigm which is focused on collaboration barriers. Using bounded contexts, it is necessary to create a set of variables in order to optimize, strategically, by using such variables with respect to properties of their objectives. A bounded context is merely a boundary definition; and so can be used with any number of widely applicable language. Outside of the boundaries which are specified, the language in which they are embedded may have multiple and various meanings thus, results may also vary [35]. Clearly, multiple bounded contexts may be delimited, and each one may give rise to a separate model. Any developmental objective may be addressed by such definitions - no limit to their applicability exists, although the features of these instruments remain the same. In DDD, the selected language is taken to be the structural model. Further all of the bounded contexts defined via this, taken together, in turn defines the models' applicability. This means, of course, that the models' validity is restricted to the areas within the defined bounded contexts. Although the features of the concept are limited in restricted as regards their functions, nevertheless, they can prove very efficacious for "general developmental objectives" [36] [37].

In relation to microservices, bounded contexts are a kind of opposite . Here, bounded contexts define to the developer the nature of the largest services that they can possibly create, or to the users the definitions of services that do not give rise to models that can in incur any inhibiting conflicts. This is important for developers to ensure correct operation, or in some cases the continued development of what already happens. Boundaries which are crossed involve conflicts, and these could have a highly detrimental impact on operations. Both large and small monoliths, in terms of DDD, are valid bounded contexts so long as there are no conflicting model components within them. Considering all of this, while all microservices can be defined as bounded contexts, not all bounded contexts are defined as microservices [37] [38].

## 2.8 Feature-Driven Evolution

Feature-Driven Development (FDD) is a gradual, iterative approach to system development, including both the software design and the coding phases. This methodology began to be used around the year 2000 and was mostly applies then to Java modelling. FDD defines five main activities: the development of a comprehensive model, the construction of a feature list, planning in accordance with the features, designing in accordance with the features, and building the software in accordance with the features. Constructing the model and features list involves Boundaries and approaches are established in the course of the effort to construct the model and the features list. Further. planning in accordance with the features results in facilities that can then be handed over to the owners of the system. In fact, the final, fourth and fifth activities, are those which generally consume the majority of the developmental resources (including time) -n these steps generally take around ¾ of such resources [39] [40]. This is not surprising since a great many activities such as program modelling and design; quality assessment and QA tests, and various labelling/packaging processes are included in these steps [36] [38].

As far as its use with FDD is concerned, the paradigm is thought of as one which can help with both architectural concerns, and the design of clients and features. Features often involve the whole system: for instance, the calculation of the sum of sales, the validation of passwords, the processing of sales transactions, and other common business procedures. The requirements and planning inputs are derived from such features, and so they are of paramount importance.

## 2.9 Cloud Computing

Cloud computing is a paradigm which offers to the user universal, on-demand, shared, and convenient access to configurable computing resources, servers, networks, applications, services and storage. These resources can be provided quickly and delivered with little effort or interaction by the service provider.The Cloud model offers five essential attributes, four models for deployment and three models relating to service [41].The main attributes of the Cloud are as follows: rapidity, elasticity, resource pooling, on demand working, network access and regularity of service. The following models are used for deployment: private

Cloud, public Cloud, community Cloud and hybrid Cloud. The three- types of service currently offered by the Cloud are: Software as a Service (SaaS), Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) [42]. There are a number of service providers who have taken the lead in terms of offering cloud computing and, as a result, are motivated to push forward the most important affordances of the Cloud as they relate to their own vendor-specific technologies.

- The most basic type of cloud computing services is known as Infrastructure as a Service (IaaS). Here, a vendor will simply offer storage and some elementary processing on the Cloud. [43].
- Next, in terms of sophistication, is PaaS, i.e., Platform as a Service, whereby the vendor will allow access to the application layer of their platform so that customers may build their own applications to run on the Cloud. The customers are able to make use of the operating system, databases and development environment of the platform offered by the vendor [26].
- The third level of service is described as Software as a Service (SaaS). This involves vendors who have supported specific kinds of service requests for some time. When the servers hosting the application become virtualised [43].

The Cloud is a scalable concept. Various kinds of services dealing with various volume levels, in terms of requests, can be offered. A Cloud platform will generally offer a number of such services rather than just one, and the services it offers will be generic. The implementation of Cloud infrastructures means that the services which may be obtained via grid computing and virtualisation are widened. Other empowering technologies are used in conjunction with the Cloud. A prime example of such would be service-oriented architectures; these are used specifically to support enterprise systems. Companies can make use of Cloud computing in their enterprise systems by applying enabling technologies such as service-oriented and microservices architectures (SOAs). By these means they may choose to emply the Cloud to support both their new and existing business systems.

The Cloud represents a technical advancement which is closely related to current software development paradigms. As opposed to the provision of on-site‘mainframe’computers, which is very expensive, Cloud technologies, e.g., Software as a Service (SaaS) or Infrastructure as a Service (IaaS) are user friendly [44]. The siting of the enterprise system on the Cloud is cost-effective and frees the companies' computing staff from the tedious work of mere system management, which means they can concentrate instead on production and innovation [45].

On the other hand, moving from reliance on an in-house data centre to the use of the Cloud is not without difficulties. Often, as far as the end-users are concerned, the system does not seem to operate as well as it used – immediately after migration to the Cloud, Also, from the IT staff's point of view, the process of migration may appear overly complex. In order to mitigate these problems, the most optimal approach to migration must be determined beforehand. The new system must answer the concerns which may be raised about non-functional requirements such as security, integrity, and reliability, as especially related to the use of an external network. So when it becomes necessary to determine whether the use of the Cloud is appropriate to the next evolution of an enterprise system, the enterprise's management must examine all of the potential issues. A fair and thorough judgement of whether the use of the Cloud is appropriate must be made.

Here, we look at the challenges which emerge when an attempt is made to apply Cloud computing to an enterprise system, and the best approach, we believe, is to apply a rule-based framework to the potential migration.

## 2.10 Container

There are two, to some extent competing, software paradigms often used for providing Cloud services: Virtual Machines (VMs), and containers. The former is based on emulating the affordances of an actual, physical computer, and so are a convenient platform for implementing systems which have been migrated from other environments [46],However, containers are, in general, more efficient, particularly when used to support data hosting applications: for instance, online databanks, where tight control is necessary. Microservices can be implemented on a container platform. This is a felicitous combination because containers, as opposed to VMs, may have many concurrent run-time instances, so enabling users to run a number of applications (or instances of the same application) at the same time. Thus, for instance, upload and download operations can be undertaken by the same user concurrently [47] In contrast to Virtual Machines, containers combined with microservices do not suffer from online-library access issues. In addition, most container-based systems include anti-malware software, which clearly will improve the reliability and security of the services offered.

Containers can also manage challenges related to upload and download. Thus, information loss associated with these issues can be avoided – in particular, losses due to negligence or hacking. An example of this is that containers are more able to operate in a way which is abstracted from the operating system used to support them, so that, for example, no data corruption occurs when files are downloaded onto a client system which uses a different operating system. Finally, microservices systems supported by containers facilitate the storage and retrieval of data at more reasonable levels of cost. [48].

The Cloud is often used by both individuals and organisations to store (and retrieve) data, and so to negate the need to use local servers for this purpose; In fact, microservices, based on containers, support this kind of service, though they may not be visible at the user level.

Docker [49] provides an example of a data container which has been in use for many years. According to that author, data management applications have often been offered as open source platforms on which to build data storage and retrieval services. Also, the system described by that author is often looked on as seminal by sysadmins: people who are charged specifically with the responsibility of maintaining and correctly configuring online data-related systems – to keep them function-rich and reliable.

## 2.11 Design Pattern

The initial debugging and testing are probably the most time-consuming phase of software development: especially where a system has been created from scratch. One methodology which has often been useful in speeding up this process is that of the use of "design patterns" [50]. To be useful, such must be patterns which have been successful within previous software designs. "Reusing design patterns helps to prevent issues that can cause major problems and improves code readability for coders and architects familiar with the patterns" [51]. A system which has been designed using such patterns must still go through the full testing cycle, including beta-testing; however, the probability that the system will be able to operate initially at least without critical failure will be greatly improved. In addition, the use of design patterns means that the developers of a system need not become bogged down with "re-inventing the wheel" and can concentrate instead on the more unique and central aspects of the specific development in question, "Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem" [51] [52]. This methodology enables the re-use of work already created by other skilled professionals.

Design patterns have been classified in a number of different ways. An example of a classification which is commonly used is that of "creational" design patterns, "While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done" [51]. There are many other examples of categories of design pattern: e.g., structural design patterns "use inheritance to compose interfaces" and "define ways to compose objects to obtain new functionality" [51] [53]. Another example is behavioural design patterns, these "are all about class's objects communication. Behavioural patterns are among those that are most specifically concerned with communication between objects". There are design patterns which can be used at every stage of development, from initial brainstorming and design to the determination of the structure of the implementation and the protocols via which its various components will communicate [51].

On the other hand, there are criticisms which can be levelled at the use of design patterns. The most important of which, perhaps, is that it can restrict innovation: "The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code". Therefore, perhaps, the idea of "best practices" can also be criticised [51].

However, it has always been the case that even the most innovative designs can include elements of previous designs. Without being able to use such elements developers have to fall back on a "rip-n-replace approach to legacy modernization" [54]. In contrast, it is possible to make use of design patterns as components of a gradual process of legacy system replacement, and as [54]argues, "the gist of the above patterns is to slowly build a new system around the edges of the old system... This is done incrementally until we can kill the old system". Predictability is an advantage of the use of design patterns; however they do allow adequate flexibility in terms of creating systems which will cope with future demands - via replacement and rebuilding, rather than starting from scratch each time a new requirement emerges: "It's impossible to predict the future; we can only be prepared for the future by designing our systems to be modular and highly flexible to change. Build an architecture that can evolve with time and be future-ready and not try to be future-proof" [54].

Design patterns, like any other paradigm, bring with them their own issues. However, they do assist in terms of making the task of software development much more straightforward. Open source patterns exist; these can be useful for free and are accompanied by online tutorials. Creative sharing and dialogue of this kind will generate innovation by enhancing the communal aspects of creating designs. Changes to existing, accepted designs remain, of course, possible, even though these have been used as patterns or include patterns. The use of design patterns is expected to carry on into the foreseeable future.

## 2.12 Conclusion

Switching to a microservice-oriented architecture and transposing legacy system into a new architecture may lead to new challenges in an organisation, especially if the process of the transformation is not planned very well. Such concerns may be technical, in this case the developer should differentiate between the strategies of simply substituting each service in the existing monolithic with a microservice and building a new microservice-oriented architecture from scratch. In the first case, the biggest challenges are how to extract the services from the old system, and how this will affect the functional and non-functional requirements.

Splitting up a monolithic system into microservices can lead to performance issues as stated in [55]. Particularly, an increase in the communication necessary between services can occur; if the services are too fine grained, then the fact that each interservice requests add an extra network latency must be considered. Also, the use of a microservice-based architecture is expected to require more computational resources, i.e. CPU cycles to communicate used for each microservice to communicate with another [56].

Another concern is that this technique will raise the issue of the presence of security challenges that usually do not exist in legacy applications. The use of a microservice-oriented architecture will break the system up into multiple components. As a result of having multiple small independent services, the a microservice-oriented architecture often expands the attack surface presented, as several microservices may communicate remotely. Now there are hundreds of entry points to worry about instead of there being just one or two entry points. Dragoni et al. [36] argues that there will be difficulty in monitoring, debugging and auditing microservices. An attacker, of course, will benefit from this by being able to launch attacks against each service; Moreover, these different components will often need to communicate with each other across widespread locations. Later on, this work will describe the various different techniques used to secure service to service communication and the application of authentication and authorisation in a microservice-oriented architecture. Later on , this work considers role-based access control (RBAC) which is used to secure microservice-oriented architecture authentication and authorisation.RBAC is important in cyber systems because users and resources change periodically and data and privacy should always be ensured. The first step to anlyse the cyber securiy is to clarify the exact meaning of confidentiality, integrity and availability in cybersecurity. Availability, in cybersecurity, does not mean that the system is always available. It means the system is available to users when they need to access the system based on their privileges.Availability guarantees that systems, applications and data are available to authorised users when they need them. Confidentiality means that unauthorized users do not have access to the data. Integrity is the ability to ensure that a system and its data have not suffered unauthorized modification. In RBAC, permission is always associated with roles and access privileges are divided into different levels in distributed systems with a large number of users. In RBAC, authorisation information is linked to roles not to individuals by correctly identifying the roles and assigning those privileges to each role, to allow the user to accomplish their task [57].

The next concern to be examined is the fact that many companies might feel anxious about the system's functionality. Enterprises need to ensure that the migration process is clear in order to overcome any incompatibility that can arise in the course of the migration. It is important to consider how decoupling the functionality of the legacy system will help to deal with changes in requirements, and how microservices can take full advantages of this situation. The transformation process consists of pulling out of all the functions of the legacy system and separating them into microservices until the microservices set-up, on its own, can perform all the required functionality. The key to a successful migration is thorough, careful, documented planning and execution, along with the realization that a complete migration of a large monolith can take several long years.

A deep investigation has been carried out for existing open source microservices applications that would be suitable candidates for a microservices benchmark based on the work requirements. To the authors knowledge, every application is different in their platform and environments and such benchmark systems are not yet agreed by the community. Therefore, the effectiveness of these benchmarks is not discussed here.

The purpose of this project is to develop a novel approach to legacy system migration towards the microservice-based architecture, which can guide an organisation to their best enterprise-system solution. This chapter has explored the concepts of software evolution and has highlighted the differences between software maintenance, software evolution, and software migration. In addition, Lehman's laws are presented in order show how the limitations of the various forms of software evolution can be handled. Moreover, a definition of the legacy concept is introduced, and the differentiation between the development of software, generally, and migration from a legacy to a more modern system is made.

The goal of this chapter has been to give the reader a big-picture model so that they may understand microservices systems via the discussion of service-oriented architecture versus microservice-oriented architectures that was introduced; the key differences between these architectures are discussed. Also, the main issues relating to legacy systems specifically and are presented along with how the use of microservices can tackle these obstacles. Furthermore, we look at bounded contexts as a way to explain how such define service boundaries. Another approach was also explored – which was feature-driven development and its relation to the application of microservices.

Within this chapter, also, cloud computing is discussed. A comparison between virtual machine and containers is made, and the Docker type is given as an example of a container type. Lastly, we introduced the design and deployment challenges which we will cover in more detail throughout the rest of this project.

# Chapter 3

# Related Work

## 3.1 Introduction

The concept of 'microservice' is informed by the latest service-oriented computing paradigm, as illustrated in the previous chapter, and these ideas have evolved from the creation of business solutions. The use of microservices has gained popularity in recent years, and a clear understanding of this concept is critical in order to sensibly transpose an application to a microservices-based system. Therefore, this chapter presents an in-depth investigation to identify the characteristics of state-of-the-art microservice architectures, and what strategies are implemented in practice.; Also, an evaluation of the various approaches used and the benefits and drawbacks of their implementation is carried out. A study of recent literature reveals that such research has focused on the scalability, availability and performance of these architectures. In the following section, these studies are reviewed in relation to three different categories of research, to provide a fair and comprehensive evaluation of the strengths and weaknesses of microservice-oriented architectures.

## 3.2 A Systematic Study on Microservices

Francesco et al. conducted a systematic mapping study in [58] which clearly summarises the nature of microservices and this work serves as a solid foundational reference for both academia and industry professionals. The study also identified a shortfall in terms of the system quality attributes, such as security, portability, and testability.

Another systematic mapping study by [59] shows that some of these quality attributes have not been thoroughly investigated, in particular, security features as related to microservice architecture. This study also provides a broad overview of the challenges facing microservices architecture and related technologies. Such an overview is very useful as a guide for the research community in relation to widening this field of study.

In a systematic mapping study focused on microservices, Hamzehloui et al. [60], conducts an analysis to identify the areas of research that have been undertaken with respect to microservices architectures. The researchers concluded that the infrastructure is at the forefront of current research, although more work is needed in the area of monitoring and automation. In terms of deployment and management, this study essentially neglects three key aspects: security, maintenance, and costing.

Taibi et al. conducted a systematic mapping study in [61] on several different architectural styles and patterns to define their significant benefits and drawbacks. They classified the patterns they found into three categories: the orchestration and coordination pattern, the deployment strategies pattern, and the data storage pattern. Also, the authors provided a summary of the advantages and disadvantages of microservices architectures which must be taken into account. This paper highlighted an important point; the deployment of microservices is not yet clearly based on a particular type of implementation.

A study by Soldani et al. [62], which was a systematic literature review on the pain and gain of using microservices, addressed two concerns regarding microservices architectures, based on industrial practices. The main concern was the technical and operational benefits and drawbacks of the microservices architecture regarding the design, development, and operational phases. For instance, an approach to the simplification of the transformation process (to microservices) and how to handle the data storage issued was discussed. Another concern was defining the gap and limitations with regard to consensus which exists between researchers in the academic and industrial fields. More specifically, the security attribute was considered to be of greater significance by researchers and practitioners working within industrial environments. This confirms the relevance of the current work which aims to develop a novel approach to migrating legacy enterprise systems into the structures of microservices architecture.

The systematic review in [63], which defines what the authors label 'bad smells'. These 'bad smells' were identified to assist software developers in recognising when to modify their designs in order to avoid or evade such identified pitfalls which are labeled as 'refactoring.'. The paper covers seven of these architectural 'bad smells' and 16 architectural refactorings. One of the solutions mentioned in this review is to apply a circuit breaker to prevent violation of the isolation of failure principle caused by a wobbly service interaction 'bad smell'. A number of different 'bad smells' mentioned in [63]are cited as violating the decentralisation concept, one of which is shared persistence. This 'bad smell' occurs when two or more microservices access the same database. The review presents three solutions to this problem: merge the services, split the database, or add a data manger. Related considerations are covered in this research, i.e. the separate-out-database rule and the master data access microservices. These issues are discussed in relation to designing microservice-oriented architectures based on feature-driven transformation rules. However, the present research has quite a different focus on assisting researchers and developers to understand the main issues and to engage the most applicable rules associated with non-functional requirements. A case study is presented to allow the evaluation of the rules provided.

## 3.3 Microservice-Oriented Architecture and Design

The 'microservices' approach deals with the issues inherent in updating a large, monolithic enterprise system by dividing its facilities into small, manageable independent services. Indeed, the term 'microservice' refers mostly to small, well-defined services created for just such an 'agility' centred context [64]. A set of microservices do not all have to be written in the same language. A variety of languages and data storage techniques may be used even within the same enterprise system.

Balalaie et al. [65] addressed a number of problems relating to the understanding of this kind of architecture; ; more specifically, these problems most specifically were those associated with the process of migrating to a microservices architecture. Regarding these challenges, Balalaie et al. [65] presented a set of patterns describing the types of introductory repositories that can be employed for such a process. Several different patterns were indicated as being significant in relation to these issues, and their identification led to considerations focused on recommended resolutions.

Brown and Woolf [66] attempted to show how microservices ought to be designed, the ways they can fit into a larger architectural picture, and the ways in which they can be built so that they operate efficiently. Their study deals particularly with matters relating to microservices efficiency, systems design, and microservices design. According to Taibi et al. [61], the microservices architectural pattern can be determined via a catalogue. In accordance with their systematic mapping study, they showed the drawbacks and benefits of each pattern, so that developers can select the most appropriate for their purposes. However, for practitioners, this process of identifying patterns or, indeed, of not being able to identify a pattern, is unclear since the process has not actually been implemented.

Knoche et al. [67] showed that, according to a survey carried out among German industry professionals, there were a number of specific major motivations driving companies and developers to adopt a microservices architecture. The desire for high elasticity and scalability was identified as one of the more important of these motivations. In contrast, it was emphasised that the lack of developers having the necessary skills was a major obstacle to the application of microservices architectures. The survey from which these results were extracted included only Germany thus it is impossible to generalise these results to rest of the world.

In [68], a different method was discussed in terms of how to portion functionalities into microservices. A Domain Driven Design (DDD) method was used for portioning a microservice into a set of the subdomain. This technique guides the developers in sizing the microservices, and the way in which each microservice is allocated to a specific function and a single-purpose-database; these represent what is to be implemented at a later stage.

The 11 microservices bad-practice scenarios provided by Taibi and Lenarduzzi [69] were determined and examined through interviews with 72 developers who had experience of working with microservices architectures. In general, they stated that the main issue was the separation of the service from the monolithic system and linked data usage; this could lead to possible maintenance challenges where the splitting had not been carried out properly.

On the basis of this, one of the objectives in the present work is to find the right approach to separating out microservices, based on rules.

Other researchers have uncovered various findings concerning the assessment of microservices that are relevant to ongoing research and developments within the affected areas of industry. Pahl and Jamshidi [37] assessed microservice systems and explained that they had been initially developed alongside specific architectural strategies for optimising the building, management, and development processes using self-contained units. This resulted in improvements some areas of development, and the general strategy paralleled developments in cloud technology. Researchers have attempted to contribute to the knowledge base and fill the research gaps regarding the classification of microservices and their applicability in association with cloud technologies, focusing on their potential to complement the use of container technologies in platform as a service (PaaS). Assessing 21 previous research conclusions for relevant patterns, the analysts concluded that the microservices strategy could be more useful than other, more conventional, strategies in these regards, and recommended increasing their more frequent use.

Later, Zimmermann [38] assessed various aspects of microservices,, in particular agile approach strategies as used in service development processes, and focused on deployment. They reported that some microservice and bounded context applications result in new structures and/or implementations relating to service orientation. This creates varying motivations when considering the usefulness of the microservices model for an application. The analyst claimed that microservices are a a major development, and that they have the potential to address the issues raised in prior approaches to service orientation, i.e. their (microservices') use of both technologies and models. It was recommended that, to create and frame a database of architectural information to support microservices implementation, microservices be assumed to constitute a specific implementation approach to service creation and use.

Other researchers have continued to contribute to the knowledge base for other relevant aspects of microservice operation. For instance, Kratzke and Quint [36] assessed the historical trends and developmental directions of microservices use. These researchers reported that citations of this approach to development had become increasingly common in specifications of best practices and strategies. They also considered a range of practicality issues and potential methods that could be effective in overcoming various types of developmental barriers. They further argued that attempts to optimise the service-based processes informed by new approaches can, in themselves, create challenges. This has the potential to be a deterrent, regardless of specific feature applicability [36].

## 3.4 Evolution into Microservice-Oriented Architecture

Dragoni et al. [36], described a real-world case study of a mission critical system - the FX (Foreign eXchange) core system at Danske Bank. This was constructed using a legacy system architecture, to address major concerns involving the scalability of the system. It was revealed that, in addition to using a legacy system, a microservices architecture might be a promising methodology for reducing code complexity. Furthermore, numerous microservices could be decoupled via the use of service discovery. As pointed out in [36], switching to a microservices architecture brings about enhanced scalability through the application of several techniques, e.g. load balancing, horizontal scaling, and cache and clustering methods [13]. The FX system can be improved through the implementation of the aforementioned methods for the purpose of supporting additional qualitative characteristics. A case in point is including additional security with the intention of delivering, to the client, an innovative user-experience.

Villamizar et al. [64] described an enterprise case study where the application under examination was established on the Cloud podium. Two versions were constructed, one using microservices and the other using a monolithic architecture. The authors analysed the performance of both architectures in terms of response times and they recognized that, so far as the microsystems architecture was concerned, additional performance provisions should be considered for the future. [36] highlighted that the performance downgrade which appears to be involved when moving to a microservices architecture is mostly due to significantly increased network use and the existence of the container.

Gouigoux and Tamzalit [70] dealt with some feedback after moving the MGDIS SA (a French software vendor editing application) monolithic software to an independent service. Among the benefits they encountered was an up-surge in performance, and this is somewhat contrary to the results from our case study.

Bogner et al. [71] interviewed experts from 10 different companies based in Germany; some of these companies are active Europe-wide or even globally. The discussion covered three main areas: the technology used to implement the microservices, the popular features that play a significant role in developers' decisions regarding adopting a microservices architecture, and the impact of the microservice on different software quality attributes. The analysis rated maintainability as the most improved attribute when moving to microservices architectures.Participant opinion was divided when it came to performance; one group noticed significant improvements in terms of response time, while the other group believed that the response time was not affected. In addition, there still exists much debate, among the interviewees and elsewhere, about the security issues relating to microservices and how to deal with these challenges.

There appears to be no clear upshot to the performance issues indicated above. Therefore,it is necessary to continue investigating the differences between the performance of applications implementing microservices and those that use a monolithic architecture. tThe performance issues are believed to differ on a number of factors, for instance, the programming languages, the host environment, and the container technology used.

Alwis et al. [72] proposed a technique to be a guide for splitting a monolithic system into microservices based on structural and behavioural properties. The first type is structural properties which focuses on the functional splitting of the system by addressing the key functions and the business objects of the create, read, update- delete (CRUD) operations. The other a type of functional splitting is based on behavioural properties, focusing on the operation execution. The approach in this work differs from Alwis et al. [72] in the way in which the splitting of the legacy enterprise system is approached. The main focus in this work is the non-functional requirements and the way in which the rules are designed around enhancing non-functional attributes such as scalability, availability,etc. Both approaches propose measures for evaluating the quality of the slicing solution that they present.

## 3.5 Conclusion

In summary, a gap in the academic literature was identified relating to the state of practicse regarding microservices. The main motivation of this study is to try to fill this gap by using a literature review to obtain an understanding of the current microservices state-of-the-art and then build an approach to migrate a legacy system to microservices as hosted on the Cloud computing podium. In terms of the issues presented in this chapter, the following aspects were observed:

- A narrower focus on the proposed architectural style and onto the emerging patterns to determine the main research direction and the advantages and disadvantages of these patterns;
- The architectural issues and how microservices sustain over the long term;
- Referring to the literature review, it seems that the performance attribute needs more attention in relation to microservice-oriented architectures;
- From the stance of various different studies, it was noticed that the security attribute is one of the main challenges in industrial environments;
- How to design a service for a single function;
- A plan for building an approach framework to guide developers through the transformation from functions in a monolith to microservices;
- Through a comprehensive analysis of the literature, the idea of architectural rules to derive certain architectural structural properties of microservices has emerged.

# Chapter 4

# Research Framework

## 4.1 Introduction

A main contribution of this thesis is the development of a novel approach whereby a legacy system can be migrated to a lean enterprise system architecture, supported by microservices and hosted by a cloud platform. This research focuses on migrating such legacy systems using the latest microservices concepts and technology while applying the lean concept in order to work effectively with various enterprise systems.

A legacy system migration approach has been developed which incorporates both microservices and container techniques and a framework of its working process and key components has been defined. Referring to this framework enables the determination of how effective any new microservices are in terms of functionality, performance, and security. This framework is supported with a rule repository which must be used bearing in mind how these rules support the functionality, performance and security.

This approach to evolution consists of:

1. A framework for the evolution and key components in the approach.
2. A rule-repository designed so that any resultant codebase can be organised around the set of microservices transformation rules it contains. These rules facilitate the breakup of a legacy system into a set of independent functional services; this must be achieved prior to any migration away from local hardware. Using a microservices

repository allows new services to be created and tested without affecting any other (micro) services.

3. A repository of migration rules designed to enable the preservation of all the functional details of a service (in terms of the processes and activities required as identified by the above microservice repository) as it is migrated to the cloud.

4. A set of criteria for evaluating the above regarding the improvement of functionality, performance and security.

## 4.2 The Conceptual Framework of the Approach

### 4.2.1 The Overview of the Framework System

This section describes the process of evolving a legacy system via a concise conceptual framework and clear guidelines for understanding, implementing, and evaluating the methods used.

The knowledge, and understanding, of such a problem domain and its solution are achieved by:

- Describing the boundaries of the legacy system to be migrated via a conceptual framework which enables the understanding of the transformation steps which must be made towards a resultant microservices artefact.

- Developing a set of rules as a guideline for conducting and evaluating the transformation process.

These represent concrete prescriptions that enable researchers and practitioners to understand and address the problems inherent in migrating to, and successfully implementing, such a microservices-based system.

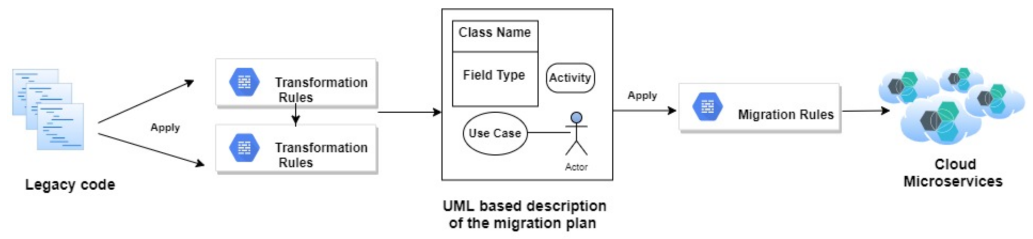


**Figure 4.1** Overview of the framework

Figure 4.1 shows that the original input to this framework is a legacy application (i.e. the legacy code) and the next element is the set of transformation rules that will be applied to the legacy system to make decisions about its evolution. The third element is the environment into which the application can be deployed, based on the migration rules, in an affordable way. Via the framework, the legacy code is employed as the input to the rules. Subsequently, an output will be generated based on the applicable rules. The output consists of several UML-based descriptions relating to the evolution plan which:

- Define the microservices-based architecture and its deployment;
- Define where each service will be located and how these services will interact.

To facilitate the necessary paradigm shifts, a set of features-driven microservice transformation rules and cloud migration rules have been developed.

The proposed evolution process, as shown in Figure 4.2, consists of three major phases:

1. The first phase focuses on understanding and analysing the legacy system in terms of the module dependencies. The legacy system is analysed with respect to its legacy software architecture and an entity relationship model.
2. The second phase forms an intermediate layer which focuses on how to determine the boundaries of each microservice, by applying a set of feature-driven transformation rules.
3. The third phase is the creation of the target system. In this phase, the ways in which the microservices are deployed in the cloud podium is determined along with what the most applicable solutions are to maintain system performance, functionality and security.

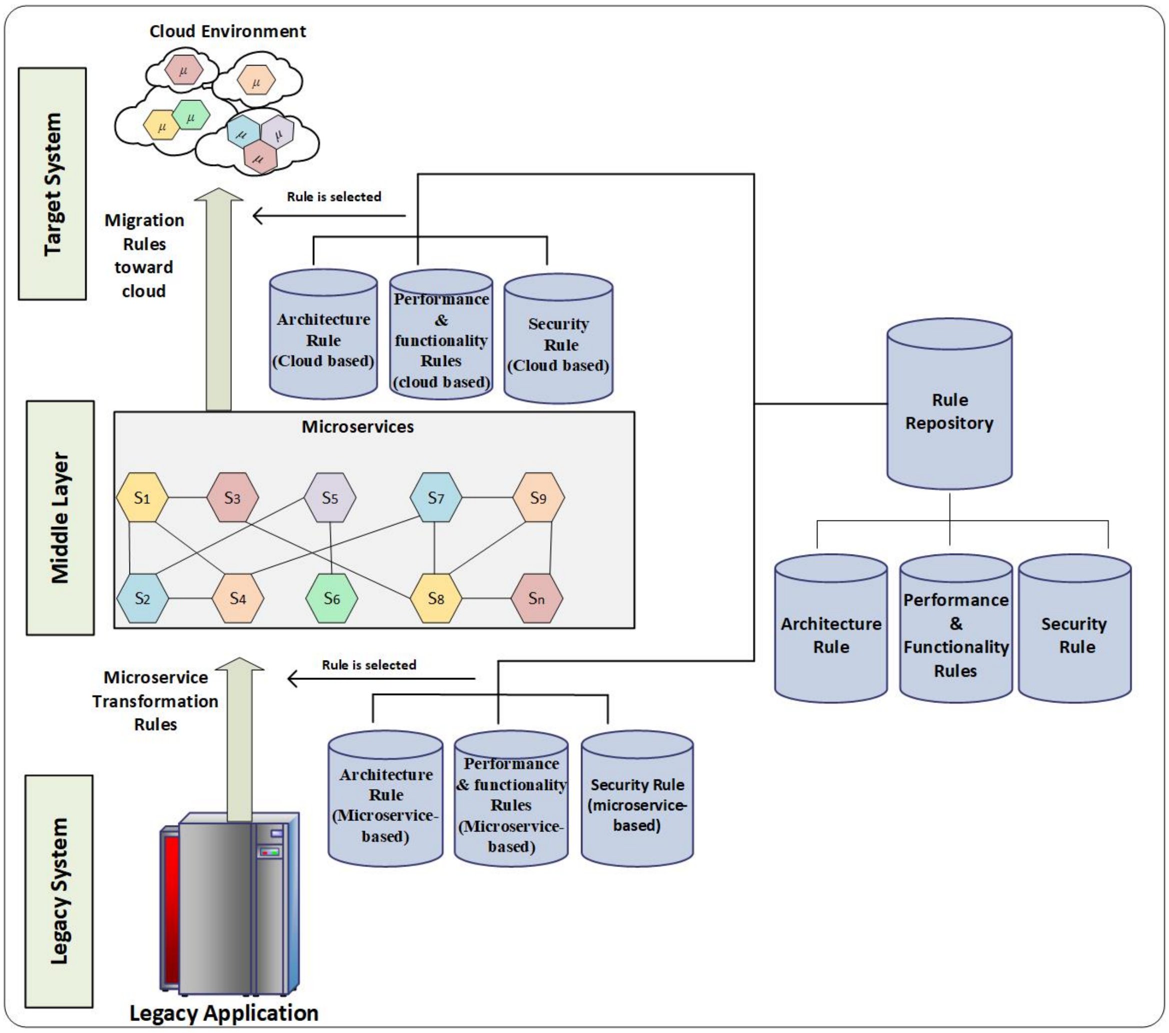

Cloud Environment
Target System
Migration Rules toward cloud
Rule is selected
Architecture Rule (Cloud based)
Performance & functionality Rules (cloud based)
Security Rule (Cloud based)
Rule Repository
Middle Layer
Microservices
S1
S3
S5
S7
S9
S2
S4
S6
S8
Sn
Architecture Rule
Performance & Functionality Rules
Security Rule
Microservice Transformation Rules
Rule is selected
Architecture Rule (Microservice-based)
Performance & functionality Rules (Microservice-based)
Security Rule (microservice-based)
Legacy System
Legacy Application

**Figure 4.2** The conceptual framework of the approach

## 4.2.2 The Legacy System

A legacy system is an application which is outdated and incompatible with the new technologies, and it difficult and costly to replace or modify [9]. It is considered the backbone of an organisation and handles all the crucial operations within and/or outside the organisation. However, this type of system often encounters a number of challenges. For example:

- if part of the system stops working, this may cause the failure of the whole system;
- it may be difficult to repair faulty parts of the system without interfering with other components;

- providing new functionality to the legacy system may require modification to all the dependent modules; and
- the maintenance costs of old hardware platforms can become excessive.

Owing to these obstacles, many organisations decide that they would like to move their legacy systems to new environments that facilitate easy replacement of components and improve system functionality. Thus, to redesign the system with the appropriate function module would make more efficient use of resources and come at a lower cost. However, when moving to such new systems, it is important to consider the details of system migration carefully, particularly with regard to components that are difficult to maintain because they need to change over time.

These challenges are addressed through, first, a thorough understanding of the legacy system. To achieve such an understanding, a reverse engineering method has been adopted here. Reverse engineering is defined as the identification of components of a system and their interconnections and the subsequent creation of a system representation at a high level of abstraction. Technical information about the software's design, architecture, and internal modules have to be examined to attain a comprehensive understanding of the system architecture and the system domain, and to identify functionalities, extract modules and map data flows between them [73].

Secondly, the data model of the legacy system should be analysed. There are various ways to organise data, such as, hierarchical, relational, object oriented, etc. In general, the older the data model is, the harder it is to retrieve a clear picture of the data structure.

Finally, the application's structure, behaviour and business processes, as well as the data structure and dependencies of these, are presented in a unified modelling language, entity relationship diagrams, or related notations.

### 4.2.3 Middle Layer – Microservice Transformation

This subsection defines the transformation from a legacy application to a microservice-based architecture. Advantages of the microservice-based architecture are better performance, fewer system errors, and enhanced functionality, security, and scalability. The documentation generated in phase one enables the division of the legacy application functionality and so enables extraction to candidate microservices.

The transformation to the microservice-based architecture, including the identification of the candidate microservices, is shown in Figure 4.3:

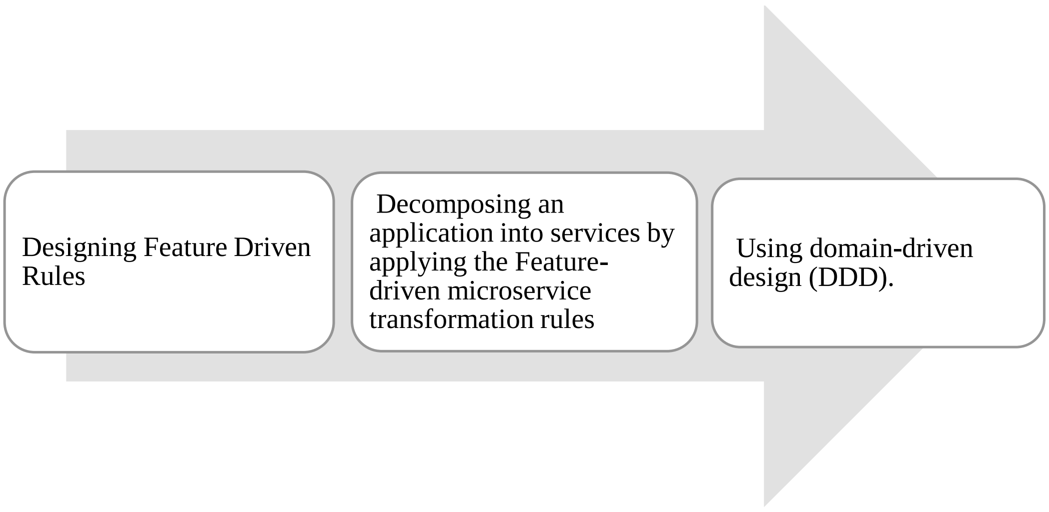

Designing Feature Driven Rules
Decomposing an application into services by applying the Feature-driven microservice transformation rules
Using domain-driven design (DDD).

**Figure 4.3** Microservices transformation process

1. Designing and selecting feature driven rules:

   At this stage, the non-functional features of the new system are defined; among them, performance, scalability, availability, and security are the main elements. Then, a featured model of the enterprise system is built, as presented in Chapter 5, to represent the interrelationships between the non-functional elements. Different non-functional features are represented in different layers.

   To select the rules, the features necessary for this task must first be selected from the enterprise feature model. After that, a strategic transformation plan is provided as the basis for the development of the microservice-oriented architecture. From these elements, transformation rules that the development of the microservice-based architecture must follow are derived. The scenario design will be based on:

   - a clear definition of the problems inherent in the legacy system which affect its non-functional attributes;

- a consequent definition of what needs to be done in terms of correcting these problems; and
- the basis upon which the solution must be determined and planned for.

2. Decomposing an application by applying transformation rules:

   The strategy for migrating to a microservices architecture is to first deconstruct the legacy system by applying transformation rules. Each rule constitutes a frame for constructing and guiding the evolution plan. The rules govern the transformation with regard to different perspectives, architecture, performance and functional objectives. Each rule represents a situation and a response to that situation; these rules help to decide how to extract services from the legacy code. More concretely, a candidate microservice is extracted from the legacy system by analysing the legacy code, the rule to guide the implementation of this microservice is then applied, then the data, logic, and user interface is extracted to form a new service.

   During the transformation, it may become clear that some of the services need to be split further into multiple (micro) services or to combine a number into one service. This could be due to excessive data access volumes for one (combined) microservice, or excessive inter-process communication between a number of microservices which have been extracted from one service; these challenges are covered by specific rules.

3. Using Domain-Driven Design (DDD):

   DDD is an approach to building complex software by portioning functionality into subdomains and bounded contexts. It defines multiple domain models and each has a different scope, defining separate subdomains for each domain. A bounded context represents the scope of a domain; its identification ensures information and features are assigned correctly to solve the design problems related to each domain [35].

   The DDD concepts of subdomain and bounded context can be usefully applied to feature-driven microservice transformation rules to make services deconstruction efficient and easier to do.

### 4.2.4 Target System – Cloud Migration

This layer represents the design of the target cloud-based architecture to be used in migration. Migration to the cloud has become one of the main priorities for many enterprises because of the effective integration facilitated by public cloud capabilities, simplified IT management, elastically scalable resources, flexible costs, and the broad accessibility of cloud resources. Before deciding on the migration process, it is crucial to examine technical requirements, system goals, and system workloads.

To deliver microservices, a set of feature-driven cloud migration rules have been proposed in this project. The main focus of the migration rules is to govern the way in which the microservices can be deployed in the cloud to enhance microservice performance. This is in terms of workload distribution, specific resources requirements, minimising request latency, increased availability, and greater agility.

Figure 4.4 shows how the best candidate rules for migrating microservices to the cloud are selected. This process begins by asking the following questions:

- Will the microservices be deployed in a multi-cloud environment (public, or private)? If the answer is “No”, it is recommended that an alternative type of cloud (e.g. hybrid) is found. If the answer is “Yes” then the following, further, questions must be answered:
- Are the microservices containerised?
- Do the microservices need to run in multiple instances?
- Can microservices run in the same region?

  Answering the above questions will give a clear picture of how to determine which rules will have the biggest impact on the microservices deployment. For instance, when a microservice application is ready to deploy, there is an architectural concern; does the system depend on high availability and thus need dedicated resources? In such a scenario, the most suitable migration rules to adopt are the containerised microservices rule and the deploying and managing extra loads rule. In the cases where the new architecture must achieve high performance, the geolocation rule is the best candidate. . These rules are presented in detail in 5).

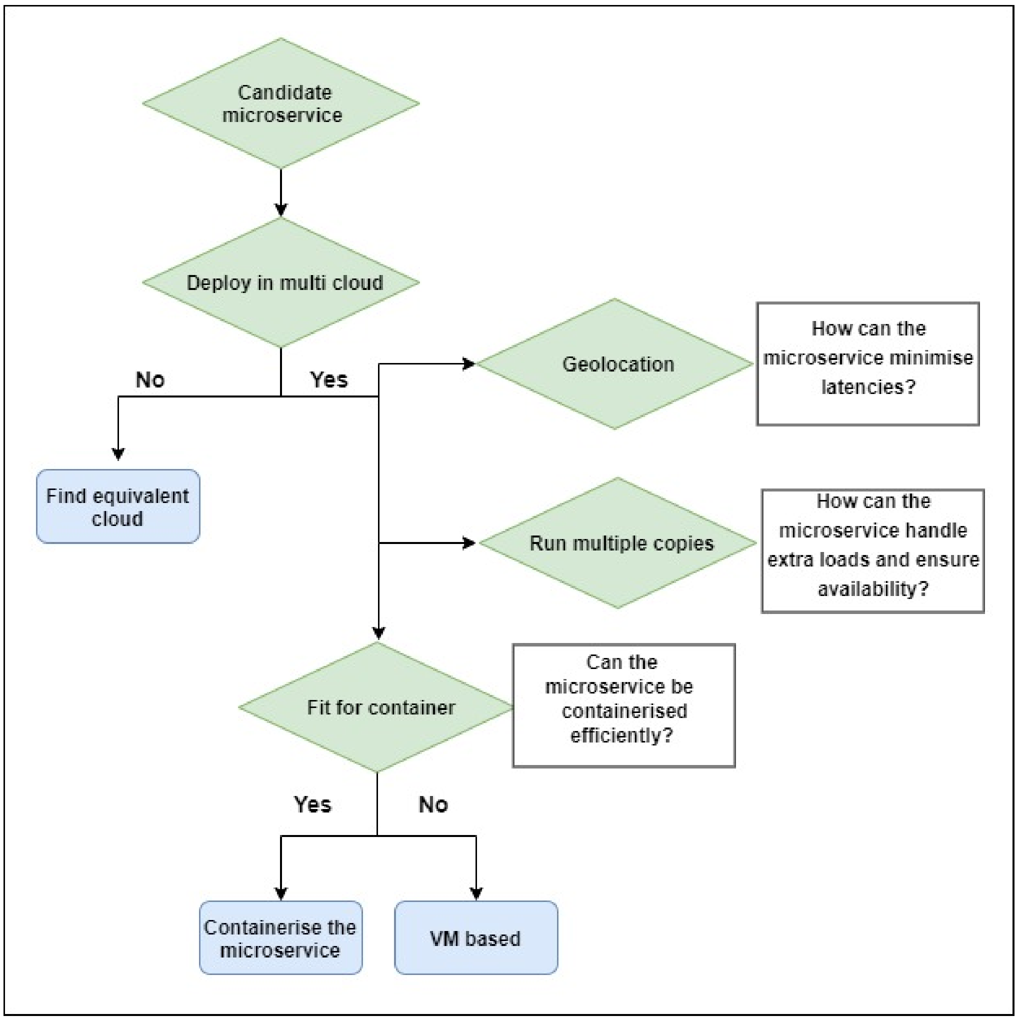

Candidate microservice
Deploy in multi cloud
No
Yes
Find equivalent cloud
Geolocation
How can the microservice minimise latencies?
Run multiple copies
How can the microservice handle extra loads and ensure availability?
Fit for container
Can the microservice be containerised efficiently?
Yes
No
Containerise the microservice
VM based

**Figure 4.4** Cloud Migration Guidance

Moving to the cloud is not the endpoint of software development, but rather represents a starting point in relation to future challenges and opportunities [74].

## 4.3 Summary

In this chapter, a conceptual framework of the approach for evolving legacy systems to microservice-based architectures in a cloud-based environment is presented. This framework provides a solid foundation for extracting microservices.

The framework helps to define an appropriate evolution path across a number of stages. The first stage is to understand the legacy system and how the components/modules are interrelated. The next stage is an intermediate layer which highlights the non-functional considerations and suggests rules based on the system situation. A comprehensive understanding of the functional and non-functional interactions which are inherent in the enterprise system enables the generation of a feature model; this plays an essential role in designing the necessary feature-driven rules.

Then, the transformation into the targeted architectural paradigm is accomplished by applying the feature driven rules to extract microservices from the monolithic code. The microservice technique involves the substitution of each part of the legacy application functionality with a microservice, thus breaking the entire legacy system into services. Such services can deploy and scale independently. The last stage of the migration process is the target layer, which is inherent within microservices running in the cloud. Thus, multiple services will be deployed in order to provide higher availability and functionality.

# Chapter 5

# The Feature-Driven Migration Approach

## 5.1 Introduction

Microservice-oriented architecture is a recent paradigm that focuses on building a system in the form of a set of distributed interacting microservices. Developing a system based on this architecture requires specific consideration because the new system will be distributed. In terms of the microservice architecture, one of the most significant challenges is how to partition the modules into separated services. Also, each microservice must be responsible for a precisely definable functionality. Data management, in particular, requires attention because a poor design can critically affect the performance of the new architecture, becoming a bottleneck of the system.

This chapter focuses on how to migrate to a microservice based architecture, which solves most of the legacy issues which have been discussed, by developing and applying a set of feature-driven rules. The goal of these rules is not just to have a set of small services but also to address the problems and limitations of large monolithic architectures.

The main challenges in this research lie in attempting to improve understanding surrounding the quality attributes inherent in a microservice architecture that can benefit enterprise systems. Additionally, in the attempt to develop the feature-driven rules for transforming legacy monolithic systems into microservices based architecture which exhibit these quality attributes.

## 5.2 Feature-driven evolution

The agile methodology can be applied to several different software development techniques. Some of the existing agile methods include Adaptive Software Development (ADS), eXtreme Programming (XP), Feature-Driven Development (FDD), and Scrum. However, FDD is applied in this work as it focuses primarily on the features that will be implemented. It helps to explore the relationship between the attribute and sub-attributes in the list of features. It distinguishes between the requirement that makes sub-attribute more important than others. Also, it renders the attributes in the feature list should be divided into sub-attributes until the attributes are small enough to customise the rules.

The FDD goal is to discover the system target and how they may be reached - to prevent costly reworking activities. For the best results, the set of requirements must be determined entirely before the system design and implementation begins.

Such system requirements should be concerned with identifying and modeling both the functional and the Non-Functional Requirements (NFRs) of a system. Having a clear explanation of these requirements helps create an appropriately focused set of rules and an understanding of the scope of the target system. In the following subsections, the techniques that were used to develop the features model, and how the rules were derived from it, will be discussed.

### 5.2.1 Features in enterprise systems

FDD method is applied to modernise, or evolve, a system to ensure that new system requirements can be met correctly and consistently. The goal here is to explore further how new plans which have emerged in relation to a system can be met with minimum reworking. For the best results, in terms of evolution, a set of requirements must be precisely identified ahead of any changes' commencement.

These requirements, or features, can be used as a guide for the further development of the system and thus represent a very important aspect of the business process involved. Such requirements are widely used and play an important role in system evolution. They are essential because they limit the list of functional requirements and NFRs that must be considered to those that are proven to be of value for an enterprise and its users. A feature will generally reflect a particular business outcome, the attainment of which must be improved or enhanced: e.g., runtime performance, maintainability, security or functionality , which are the main focus of this research. A features model will be used to handle the selection of the rules which are deemed applicable to the evolution of legacy enterprise systems dealt with here. This model will, for example, describe the additional features, the hierarchy of each feature, and the relationships between them. Each decision regarding evolution should be informed by details of the means which will become available to enhance NFRs.

In the examination of the problem, it is assumed that the formulation and use of the evolution rules are performed per feature. Considering that a complete model which comprehensively accounts for all features would be difficult to define, the focus here is on performance, security, availability, functionality and maintainability. The feature model presented in Figure 5.1 shows, in particular, the relationships between the various different features which play a role in defining the feature-driven rules. This features model was designed on the basis of FDD; i.e. defining a hierarchy of features along with the differing constraints that exist among them. To reduce complexity, all the root features and sub-features remain constant; they are not modified during the rule development process.

FDD is one of the most widely used agile methods and, as the name suggests, features are the main focus. Designing using FDD results in the proposal of an optimal design that exhibits only the required and requested features. FDD focuses on supporting a continuous process, on integration and on small releases, to reduce the occurrence of conflicting changes [75] [76].

FDD uses the following five main steps:

- Develop an overall model by addressing the project goal and understanding the system domain.
- Build a features list based on the knowledge obtained from the initial step, Step 1.
- Plan the development, in accordance with the features.
- Create the design, in accordance with the features.
- Build the design, in accordance with the features..

The final two steps are concerned with actual implementation, in accordance with the design. As with all agile methodologies, the first step in a development driven by FDD is to gain an accurate understanding of the system content. Then a clear understanding of the requirements is needed, as determined by the system's functional and non-functional characteristics. Functional features are not usable without the necessary non-functional attributes being in place. Both functional and non-functional characteristics must be considered in the development of a system. Once all the necessary goals have been recognised, the FDD model can then be developed [39] [75].

In state-of-the-art service computing, engaging microservices, containers and FDD is necessary to transform a legacy system into an architecture populated with containerised microservices. The proposed approach rearchitects a legacy enterprise system based on reengineered legacy features in order to realise the proposed evolution's requirements. Such system requirements should be concerned with identifying and modelling both the functional requirements and NFRs of a system. Having a clear explanation of these requirements helps the engagement of an appropriately focused set of evolution rules and an understanding of the scope of the target enterprise system. In the next subsection, the methodology used to develop the features model and the feature-driven rules is discussed. The derived microservice evolution rules and cloud migration rules are described in the two subsequent subsections.

### 5.2.2 Method to derive the microservice Evolution rules

In this study, two in-depth investigations are carried out into the evolution of legacy monolithic systems to microservices architectures. In both cases, the migration was motivated by the need to address some major non-functional concerns of the system. In particular, the improvement of performance and the enhancement of availability and security. Monolithic structures are not sufficiently agile to respond to significant increases in workload. The solution proposed is to deconstruct the application into a set of smaller services. Separating out components can reduce dependencies, which may lead to several advantages such as scalability and performance, and agility in response to change.

The main steps by which the rules were designed to govern migration are described as follows:

**To identify main non-functional requirements:** the main NFRs of the system, generally, are performance, functionality scalability, availability and security.

**To analyse the quality attributes:** each quality attribute is analysed, and each attribute is categorised into a relevant sub-feature.

**To define the relationships among the attributes:** based on the impact of each attribute its dependencies, trade-offs, circumstances of inclusion, and shared relationships are defined. The result of this analysis is a hierarchy of system NFR based features, as shown in Figure 5.1, This set of relationships among the NFRs is used as a constraint to build the feature-driven rules. The identification of NFRs can be challenging to integrate into the development of a system. This is because it is difficult to manage NFRs during the development process.

**To develop evolution rules:** once the requirements have been identified in a granular enough fashion, designing the rules and implementing according to these can begin. This is done by looking at several different scenarios in relation to handling the transformation process and considering (1) the relations between the non-functional features; and (2) the adjustment to other requirements triggered by the NFRs when certain conditions hold.

The upshot of this process is that a set of feature-driven evolution rules is proposed to drive the transformation of legacy enterprise systems to a microservice-oriented architecture, and a set of cloud migration rules is developed to guide the deployment of the new microservice-oriented system in cloud computing environments. The concept of feature-driven rules is used to guide and further improve microservice-based legacy system evolution. Thus, this work sets out to develop a feature-driven microservice evolution rule repository. Each of these rules define a transformation that can be applied to the software architecture of the legacy systems to maintain and evolve the system's goals, requirements and objectives. The evolution rules are intended to address the issues of the legacy software architecture during both implementation and operation. Testing the rules in relation to a real-world case study is accomplished by verifying that the systems facilities—as they have been redefined using the rules—meet the new NFRs and that the new system performs its functions correctly.

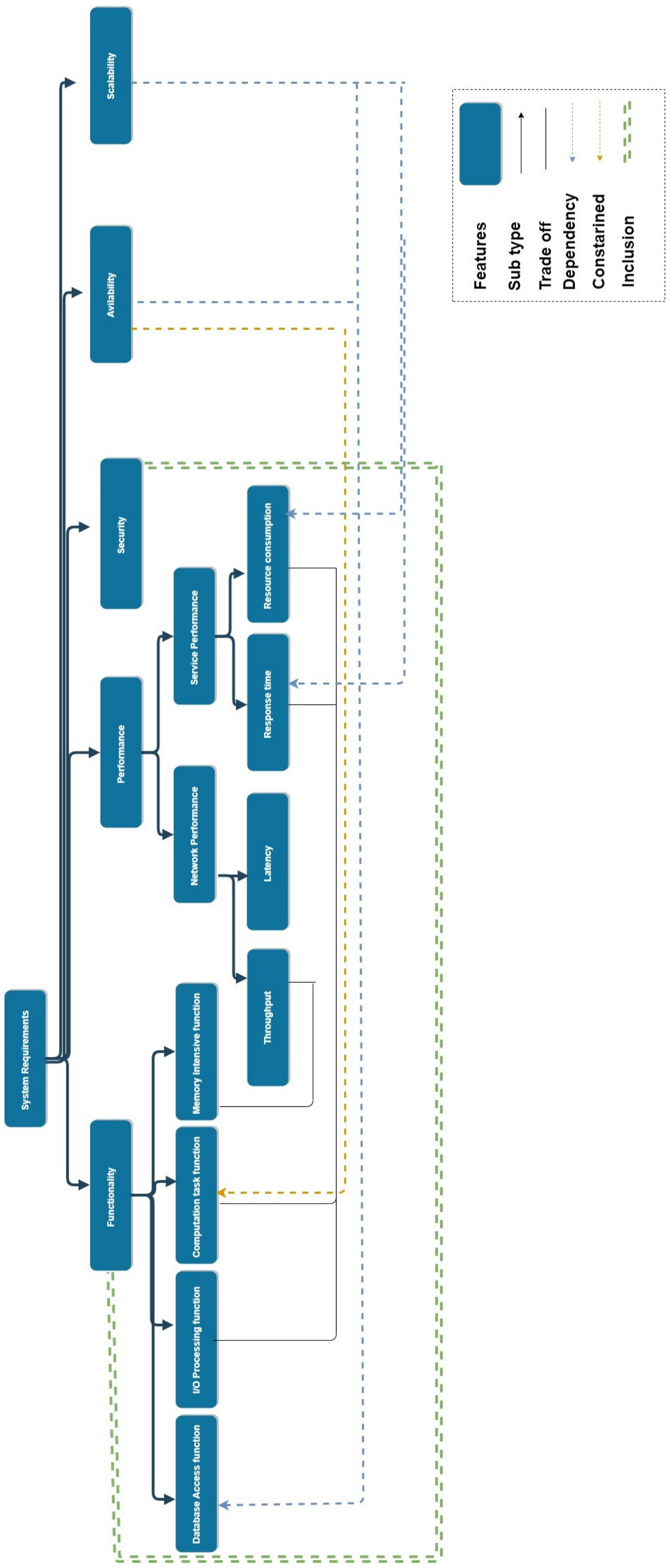


**Figure 5.1** Typical Features Model in Enterprise Systems

### 5.2.3 Feature relationships in enterprise systems

This section defines four types of relationships between features which are based on the system requirements, namely Trade-off, Dependency, Inclusion, and Shared. These play important roles in the design of feature-driven rules.

1. **Trade-off Relationship:**

   - **Assumption:**
     - $\mathscr{A}$ is a feature; $\mathscr{B}$ is another feature.
   - **Syntax of the Relationship:**
     - $\mathscr{A} \xleftrightarrow{T} \mathscr{B}$
   - **Semantics:**
     - The increase of $\mathscr{A}$ will result in the decrease of $\mathscr{B}$
     - The increase of $\mathscr{B}$ will result in the decrease of $\mathscr{A}$
   - **Examples:**

     (a) **If** the application's performance needs to be enhanced, **then** the computational task's response time should be minimised by distributing the workload between services.

     (b) **If** the application's response time needs to be improved, **then** the number of unnecessary input/output (I/O) processing tasks should be reduced.

     (c) **If** the system has insufficient memory and a lack of available space, **then** the throughput will be extremely slow.

     (d) **If** the system has fewer computational tasks, **then** the network latency should be improved, which means the system can perform more processing.

2. **Dependency Relationship:**

   - **Assumption:**
     - $\mathscr{A}$ is a feature; $\mathscr{B}$ is another feature.
   - **Syntax of the Relationship:**
     - $\mathscr{A} \underset{\rightarrow}{D} \mathscr{B}$
   - **Semantics:**
     - To enhance $\mathscr{A}$ implies that $\rightarrow$ $\mathscr{B}$ needs to be enhanced
   - **Examples:**

     (a) **If** the system suffers from resource consumption, **then** consider scaling the system.

     (b) **If** the user cannot access their data, **then** the system has to handle and guarantee the availability of data for create, update, and delete operations.

3. **Inclusion Relationship**

   - **Assumption:**
     - $\mathscr{A}$ is a feature; $\mathscr{B}$ is another feature.
   - **Syntax of the Relationship:**
     - $\mathscr{A} \leftarrow\leftarrow \mathscr{B}$
   - **Semantics:**
     - To preserve $\mathscr{A}$ implies that $\mathscr{B}$ needs to be included.
   - **Example:**

     (a) **If** the user interface has provided a trusted path, **then** the trusted path must be able to ensure the security of transmitted data.

4. **Shared Relationship**

   - **Assumption:**
     - $\mathscr{A}$ is a feature; $\mathscr{B}$ is another feature.

- **Syntax of the Relationship:**
  - $\mathscr{A} \ll \mathscr{B}$
- **Semantics:**
  - If $\mathscr{A}$ and $\mathscr{B}$ refer to the same context, then there exists a shared relationship between $\mathscr{A}$ and $\mathscr{B}$.
- **Example:**

  (a) **If** the system has suffered from repeated failures, **then** a circuit breaker is an efficient way of preventing cascading failures and of ensuring the system is available and returned to the normal working state.

  (b) **If** one of the servers has completely failed in the past, **then** a backup data server will be used to ensure data and service availability (this will bring with it a variety of security issues, as the data is stored in at least two different locations).

## 5.3 Preconditions for feature-driven rules

The first step toward the microservice-oriented architecture is to populate a method or rule repository, which has been designed in the following section as feature-driven rules. It is important to emphasise that each of these rules have been developed to address five primary concerns. This step is presented with a proper definition of the issue, determining why it is needed, what the effects of the issues are, and, for each Precondition, indicating which rules are used. After this, they are transferred to a rule template, as presented in sections section 5.4 and section 5.5.

| ***Problem to be solved*** | ***Why it is needed*** | ***What effect the Precondition can achieve*** | ***Refer Rule*** |
|---|---|---|---|
| Large monolithic systems are complex for any developer to fully understand. In addition, they are not easily maintained and evolved due to their inherent complexity. As a result, adding a new feature becomes a hard and time-consumingtask and it is not easy to debug the system. | To improve system agility, reliability and scalability. | Achieving better performance and reliability through increasing the instances of any services facing a bottleneck, better scalability due to full or partial transformation to the micro services-oriented architecture and reduced maintenance cost. | Rules (1- Decomposition of a legacy application into microservice , 2- Single responsibility principle) |
| Swift upgrading is needed to meet the competitive environment requirements or integrate with systems on different platforms. | To improve system resilience and enable effective use of data. | It speeds up data flow and reduces operational cost. | Rules(1- Decomposition legacy system , 5- performance optimisation, 6- reduction of I/O processing, 12- master data microservice) |
| Some existing modules provided by legacy systems are not the best alternative to satisfy requirements. | To build a reliable system and extend it. Also, taking an advantage of applying a diversity of technology to use during the implementation of microservices to suit the enterprise requirements and needs. | To make it possible to change the implementation of the microservices and gives you the choice of the service location. | Rules(5-performance optimisation ,6- reduction of I/O processing ,7-improvement of throughput ,8-resource optimisation ,10-resource complementary services) |
| The monolithic system takes time to diagnose and fix system failure. | It helps to avoid wasting time on handling failures. | The service will be easier to understand, change and deploy. It also enhances the availability of an application | Rule(4- inclusion of circuit breaker) |
| Rapidly changing data structures or large data streams. | To keep schema separation as it might be in the future, more service the separation will be necessary. | It will reduce system complexity | Rules (3-database separation ,9-prioritisation of data access workload ,11-customised data management) |

**Table 5.1** Main concerns for the feature driven rules

## 5.4 Feature driven microservice transformation rules

A set of feature-driven microservice transformation rules is proposed in this section. These rules are documented in the form of pseudo-code. Each rule consists of a Precondition, a transformation and a statement regarding its impacts on one or more features of the enterprise system.

The Precondition specifies the main system issue that the rule is designed to cope with. The transformation defines the solution to this issue. Lastly, the impact statement describes the concerns which might arise when the rule is applied.

**Rule1 - Decomposition of a legacy application into microservice-based architecture**

***Precondition***

- A monolithic system needs to scale and improve in its modularity.
- Individual parts of a modular application may be independently deployed.

***Transformation***

- Decomposing a legacy system to a microservice-based architecture, through building an application from internal lightweight services, will improve modularity and simplify the scaling of a particular service to meet new demands and requirements.
- Decomposing may happen in order to meet a number of different NFRs and the size of the service as a whole depends on the complexity of the problem domain.

**Algorithm 5.1:** Decomposition of a legacy application into microservice-based architecture

```
1 LS = Legacy System;
2 $\mu$= Microservice based architecture;
3 $\mu$_Mo = Microservice Modularity
4 $\mu$_S = Microservice Scalability;
5 $\mu$_M = Microservice Maintainability;
6 if (LS==true) then
      • Create a new $\mu$ in order to Improve ($\mu$_Mo) $\wedge$ Improve ($\mu$_S ) $\wedge$ Improve ($\mu$_M);
7 end
```

***Impact on features***

- Enhances system scalability and modularity.
- Separate processes add complexity and new problems, including network latency.
- Increases the complexity in managing dependencies and deployment.

**Rule 2 - Single responsibility principle**

***Precondition***

- A monolithic application has suffered from tight coupling and dependencies between modules.

***Transformation***

- Deconstructing a system into small services minimises dependencies and leads to loose coupling, which is assisted by the application of the Single Responsibility Principle (SRP) concept.
- The scalability of each service can be improved by separating the dependent services in the legacy system so that they become independent microservices [17].
- This allows the developer to change the implementation or modify the services and/or replace them without any downstream impact.

**Algorithm 5.2:** Single responsibility principle

```
1 LS_D = legacy system dependency;
2 μ_SRP = microservice architecture based on the single responsibility principle;
3 μ_D = dependency;
4 μ_S = scalability;
5 if (LS_D == true) then
      • Create a new μ_SRP in order to Reduce (μ_D) ∧ Improve(μ_S);
6 end
```

***Impact on features***

- The core complexity of this rule increases memory consumption in the new architecture. As microservices consumed persisted memory directly during requests instead of using CPU.

**Rule 3 - Separating out databases**

***Precondition***

- A monolithic application has been decomposed into a set of services to ensure that the services are loosely coupled (i.e. they can be developed, deployed and scaled independently from each other).

***Transformation***

- Keeping a separate data store for each service, whereby each such service becomes responsible for persisting their data [77].

- Manipulating the data of other microservices is prohibited as only one microservice can access each schema.

**Algorithm 5.3:** Separating out databases

```
1 $\mu$_DB = microservice architecture with separate data store;
2 LS_TC = Legacy System _Tightly Coupled;
3 $\mu$_D = Microservice Dependency;
4 $\mu$_A = Microservice Availability;
5 $\mu$_S = Microservice Scalability;
6 $\mu$_M = Microservice Maintainability;
7 if ( LS_TC == true) then
      • Create a new $\mu$_DB in order to
      • Reduce($\mu$_D) $\wedge$ Improve ($\mu$_A) $\wedge$ Improve ($\mu$_S) $\wedge$ Improve ($\mu$_M);
8 end
```

***Impact on features***

- Breaking the data up may make data management more complicated.

- Implementing queries that join data becomes more challenging.

- Having a separate data store improves service scalability. availability, and maintainability.

**Rule 4 – Inclusion of circuit breaker**

***Precondition***

- If one or more services are unavailable or suffer from high latency from time to time, or have coding issues, this can result in cascading failure.

***Transformation***

- Adding a circuit breaker prevents new requests when detecting a service problem.
- Allowing the microservice to continue to operate without waiting for the fault to be fixed [78].

**Algorithm 5.4:** Inclusion of circuit breaker

```
1 $\mu$_ts = Microservice transient faults;
2 $\mu$_ci = Microservice code issues;
3 $\mu$_b = Microservice very busy;
4 $\mu$_A = Microservice Availability;
5 CB = Circuit Breaker ;
6 D = dependency;
7 if ($\mu$_ts == true) $\vee$ ($\mu$_ci == true) $\vee$ ($\mu$_b == true) then
      • Add (CB) in order to
      • Improve ($\mu$_A)
8 end
```

***Impact on features***

- The circuit breaker could be accessed by a large number of concurrent requests which should not be blocked but could be limited.
- In case the microservice is slow or down, the circuit breaker can cope with this situation by returning cached data or by reducing load, which allows the microservice to recover.
- This rule ensures stability and availability of the microservice by stopping resources from being consumed by requests that have no, or little, chance of being processed.

**Rule 5- Performance optimisation of computational Tasks**

***Precondition***

- Legacy applications have a serious effect on the response times in relation to single requests.

***Transformation***

- Dividing legacy systems into microservices means the computation must divide into smaller tasks which are distributed out to each microservice, to improve the service's computational capabilities.

**Algorithm 5.5:** Performance optimisation of computational Tasks

```
1 Response time $\leftrightarrow$ Computation Task (trade off)
2 $\mu\_t$ = Microservice response time;
3 $\mu\_ct$= Microservice computation task;
4 $\mu\_P$= Microservice Performance;
5 $LS\_t$ = Legacy System response time;
6 while ($\mu\_t < LS\_t$) do
      • Reduce ($\mu\_ct$)
      • Improve ($\mu\_P$)
7 end
```

***Impact on features***

- Decreasing response time in comparison with that of the corresponding monolithic application. Response time of a request is an essential indicator of the user-perceived performance of the system.

- Raising the number of microservices could increase latency which may occur due to more connections.

**Rule 6 – Reduction of I/O processing**

***Precondition***

- The legacy system suffers from longer than average response times due to a bottleneck in one of the components, which should be eliminated to improve the system's response time.

***Transformation***

- Deconstructing the system into a set of microservices.
- Eliminating all unnecessary I/O processing.

**Algorithm 5.6:** Reduction of I/O processing

```
1 Response Time $\leftrightarrow$ I/O Processing (Trade off)
2 $\mu$_t = Microservice response time;
3 $\mu$_P= Microservice Performace;
4 LS_t= Legacy System response time;
5 I/O = I/O processing;
6 while ($\mu$_t< LS_t) do
      • Improve ($\mu$_P)
      • Reduce (I/O)
7 end
```

***Impact on features***

- The response time of a new microservice-based architecture will be influenced by the I/O intensive processes, which this new architecture dictates.
- Poorly defined of a microservices can lead to a chatty I/O that affects performance.

**Rule 7- Improvement of throughput**

***Precondition***

- Legacy systems cannot reduce the latency of the requests they handle.

***Transformation***

- Using microservices.
- Reducing the quantity/frequency of the memory-intensive functions employed [79].

**Algorithm 5.7:** Improvement of throughput

```
1 Throughput ↔ Memory Intensive (trade off)
2 $\mu$_l = Microservice latency;
3 LS_l = Leagcy System latency;
4 $\mu$_th= Microservice network throughput;
5 LS_th = Legacy System throughput;
6 Mi = Memory intensive function;
7 $\mu$_P = Microservice Performance;
8 while ($\mu$_l < LS_l ) do
      • Improve ($\mu$_th)
      • Reduce (LS_th)
      • Improve ($\mu$_P)
      • Reduce (Mi)
9 end
```

***Impact on features***

- Improving throughput (i.e. the number of messages processed), as compared to that of the monolithic application.
- Improved throughput, however, may not lead to decreased latency.

**Rule 8 - Resource optimisation**

***Precondition***

- The various modules of a monolithic application have their unique resource requirements, e.g. they may be resource-intensive or computation-intensive.

***Transformation***

- Deconstructing modules in a monolithic application with unique resource requirements into a set of microservices.
- Prioritising the system resource.
- Allocating each service with its required resources.

**Algorithm 5.8:** Resource optimisation

```
1 LS_m= legacy system module;
2 $\mu$= microservice;
3 $\mu$_S= Microservice Scalability;
4 if (LS_m) require unique resources then
      • Break (LS_m)
      • Create a new ($\mu$) in order to Improve ($\mu$_S)
5 end
```

***Impact on features***

- Ensuring sufficient resources are available to each microservice to function and cope with scalability requirements, with regard to data volumes and throughput.
- Increased costs result from an inefficient placement of resources.

**Rule 9 - Prioritisation of data access workload**

***Precondition***

- A module in a legacy application suffers from a heavy database access load.

***Transformation***

- Monitoring of the legacy system's database access load.
- Identifying modules that must frequently deal with the heaviest workloads.
- Transforming the module in the legacy system into a microservice to relieve the heavy database access load.
- Prioritising all operations performed by a system.
- Encapsulating a high prioritised operation in an individual, resource-rich microservice.

**Algorithm 5.9:** Prioritisation of data access workload

```
1 LS_m= Legacy System module;
2 $\mu$= Microservice;
3 $\mu$_S = Microservice Scalability;
4 $\mu$_A = Microservice Availability;
5 $\mu$_P = Microservice Performance;
6 if (LS_m) experience heavy access load then
      • Break (LS_m)
      • Create a new ($\mu$) in order to
      • Improve ($\mu$_S)$\wedge$
      • Improve ($\mu$_A) $\wedge$
      • Improve ($\mu$_P)
7 end
```

***Impact on features***

- Optimising system performance
- Improving the availability, reliability and scalability of services.

- Implementing systems, where transaction boundaries span multiple microservices, is challenging in general as implementing distributed data management and distributed transactions could be daunting tasks.

**Rule 10 - Resource-complementary services**

***Precondition***

- The various modules of a monolithic application have their unique resource requirements: e.g., they may be resource or computation intensive.

***Transformation***

- Encapsulating modules with unique resource requirements in separate microservices.
- Platforming complementary containerised microservices that have different resource profiles on the same virtual machine. (For example, given Service A needs less CPU but more memory, and Service B needs more CPU but less memory, the two services are recommended to be hosted on the same VM).
- Limiting the number of containerised microservices running on each VM instance with respect to the capacity and cost of VM instance, as well as the sizes of the microservices.

**Algorithm 5.10:** Resource-complementary services

```
1 LS_m= legacy system module;
2 $\mu$= microservice;
3 $\mu$_A= Microservice Availability;
4 C = Cloud cost;
5 while (LS_m) require unique resources do
      • Break (LS_m)
      • Create a new ($\mu$)
      • Containerise ($\mu$a & $\mu$b) in order to
      • Improve ($\mu$_A) $\wedge$ Reduce (C)
6 end
```

***Impact on features***

- Providing efficient sharing of resources of a host operating system.
- Reducing costs.

- Improving availability.
- Monitoring the resource consumption of each microservice is challenging.

**Rule 11 - Customised data management**

***Precondition***

- In a legacy application using a mega-sized database system, some of the modules may demand a specific type of database to manipulate and store their data optimally.

***Transformation***

- Transform each of these modules of the legacy system into a microservice with its own database. One significant advantage of partitioning data management is the ability to take advantage of polyglot persistence. Different types of data have different storage requirements, e.g. for some service, a relational database will be the best choice while other services might require a NoSQL database to handle their complex, unstructured and query graph data.
- developer has to Choose the most appropriate type of database for managing and storing the data used in each microservice.

**Algorithm 5.11:** Customised data management

```
1 LS_m= Legacy System module;
2 $\mu$_DB = microservice with specific type of database;
3 $\mu$_A = Microservice Availability;
4 $\mu$_S = Microservice Scalability;
5 $\mu$_P = Microservice Performance;
6 $\mu$_Se = Microservice Security;
7 if (LS_m) require specific database then
      • Break (LS_m)
      • Create a new ($\mu$_DB) in order to
      • Improve ($\mu$_A) $\wedge$
      • Improve ($\mu$_P) $\wedge$
      • Improve ($\mu$_S) $\wedge$
      • Improve ($\mu$_Se)
8 end
```

***Impact on features***

- Resulting in better system performance, scalability and availability.
- Improving security; data can be separately stored in different databases in accordance with their level of sensitivity.
- It is challenging for the development team to handle several different types of database.
- The operating costs involved with supporting multiple databases might become relatively high.

**Rule 12 - Master data access microservice**

***Precondition***

- Data from the microservice-based system needs to be merged into the existing legacy system due to the difficulties involved in breaking up some components of the legacy system into microservices.

***Transformation***

- Creating a MDAM that manages data access to selected components of the legacy database and the microservice database; microservices do not need to 'know' each other's underlying database structure.
- Replicating only a selective portion of the data in the master data system.
- The idea is that each microservice can follow the SRP.

**Algorithm 5.12:** Master data access microservice

```
1 LS_DS= Legacy System Data Store;
2 $\mu$_DS = Microservice Data Store;
3 $\mu$_MD =Microservice Master data;
4 $\mu$_A =Microservice Availability;
5 $\mu$_I =Microservice Interoperability;
6 if (LS_DS Join $\mu$_DS) then
      • Create ($\mu$_MD) in order to
      • Improve ($\mu$_A) $\wedge$
      • Improve ($\mu$_I);
7 end
```

***Impact on features***

- Maintaiting coordination of the microservices, and between the microservices and the residual legacy system, within the MDAM reduces the complexity of coordination.
- Ensuring the availability of data at the right time.
- Enhancing interoperability between two systems.

- Bringing in an additional latency problem, especially when the microservices are hosted in environments which are geographically distributed.
- Forming a potential performance bottleneck at the MDAM, and even resulting in denial of service in the new architecture and affecting all the depending microservices if the MDAM runs into a fatal failure.

**Rule 13 - Database view**

***Precondition***

- Where there is a need to join or aggregate the data from multi-data sources, the idea of a virtual database is applicable. For example, one of the databases involved could be that of the legacy monolithic application, and so the other one could be one or more of the new microservice. Alternatively, a virtual database could be the means by which two microservices co-operate.

***Transformation***

- Create a database view by creating another representation of the model that are suite the use case . Views represent a subset of the data contained in a table. They can join and simplify multiple tables into one virtual table.

**Algorithm 5.13:** Database view

```
LS_Ds= Legacy System Data store;
$\mu$_Ds = Microservice Data store;
DBV = Database View;
$\mu$_P = Microservice Performance;
if (LS_DS Join $\mu$_DS) $\vee$ ( $\mu$_DS Join $\mu$_DS) then
    • Create DBV in order to
    • Reduce ($\mu$_P)
end
```

***Impact on features***

- The use of this technique might bring performance issues, especially view depends on frequency of view accesses.
- It provides secure access to the underlying table
- It consumes little memory storage as the database contains the view definition not the data.

**Rule 14 - Security access control roles**

***Precondition***

- The legacy system struggles to deal with a particular, identified, threat such as unauthorised access due to the inflexibility of the old system, in terms of it accommodating fixes, amendments, or updates.

***Transformation***

- The legacy system modules are 'broken out' into separate microservices. This transition generally results in a system implemented in fewer lines of code than was necessary to implement the monolith. Such efficiencies may result in there being fewer loopholes for attackers to exploit, especially in terms of authentication and access control.
- This transformation protects the microservices from unauthorised access and reduces the risks derived from privileged user credentials being stolen (both in terms of likelihood and severity). The functionally different user-roles (involving access control) are separated out and the traffic and user-roles across all the services become more segregated.
- One microservice will be responsible for obtaining the permissions pertaining to a specific user and will administer a set of database tables which are used to maintain user-roles, responsibilities and permissions. User-roles will be determined during user login by associating each user-role with a token and storing this in the database.

**Algorithm 5.14:** Security Roles

```
LS_S= Legacy System Security issue;
$\mu$= Microservice Architecture;
$\mu$_S = Microservice Security;
if (LS_S == true) then
    • Create a new ($\mu$) in order to
    • Improve ($\mu$_S)
end
```

***Impact on features***

- This rule leads to a reduction of the latency issue related to the securing of microservices.

## 5.5 Feature Driven Cloud Migration Rules

The migration mechanism to the cloud is an optional part, but many enterprise systems and even microservice enterprise systems need it and choose to deploy to the cloud due to the unique advantages of cloud computing. How new microservices migrate to the cloud is a question that needs to be tackled due to the practical needs of those who use this approach to remain within the microservices , or to move further and deploy in the cloud.

Microservices-based applications can be difficult to deploy. There are hundreds or thousands of different kinds of microservices, each with their own configuration and scaling requirements. For these reasons, a set of feature-driven cloud migration rules is proposed in this section to assist in the process of deploying microservices efficiently in the cloud.

**Rule 1 - Deployment in multi-cloud environments**

***Precondition***

- An enterprise application consists of a set of microservices with diverse workloads or unique platforms and databases.

***Transformation***

- The cloud offers several choices in terms of microservice deployment. However, it is critical to have an in-depth understanding of microservices application environments and the unique prerequisites of each microservice to determine which microservices to deploy into clouds - whether private, public, or hybrid.

**Algorithm 5.15:** Deployment in multi-cloud environments

```
$\mu$= Microservice Architecture;
$\mu$_A = Microservice Availability;
$\mu$_P = Microservice Performnce;
if ($\mu$) Require specific DB $\vee$ unique resources $\vee$ handle intensive workload then
    • Deploy ($\mu$) in Cloud environment in order to
    • Improve ($\mu$_A) $\wedge$
    • Reduce cost $\wedge$
    • Improve $\mu$_P
end
```

***Impact on features***

- Improving service availability.
- Minimizing the cost scaling as needed;
- Appropriate performance;
- providing consistent security protections and policies by keeping sensitive data in a private cloud;
- Furthermore, there are also the generally available benefits of cloud deployment.Such as elasticity and pay-as- you-go.
- Managing multi-clouds-based applications may be challenging.

**Rule 2 - Deploying and managing extra loads**

***Precondition***

- Microservices may need to handle additional loading while still ensuring high availability. Some microservices may have to be run in multiple copies to handle additional loads and offer high availability.

***Transformation***

- Deploy one or more instances of a microservice, depending on the deployment requirements.
- The API gateway manages and controls the distribution of requests between multiple instances.
- The instances can be added or removed, based on the loading.
- All the instances are completely isolated.

**Algorithm 5.16:** Deploying and managing extra loads

```
1 $\mu$= Microservice Architecture;
2 $\mu$_R= Microservice Reliability;
3 $\mu$_A = Microservice Availability;
4 if ($\mu$) Require to handle intensive workload then
      • Deploy ($\mu$) in multiple instance in order to
      • Improve ($\mu$_A) $\wedge$
      • Improve($\mu$_R);
5 end
```

***Impact on features***

- The API gateway ensures that the data associated with the microservice based application will persist, thereby providing high reliability;
- One drawback is that many different versions of the microservice must be handled simultaneously.
- Increased response time due to the network bottleneck at the API gateway.

**Rule 3 - Geolocation microservice - master database synchronization**

***Precondition***

- Microservices may need to be deployed across many regions and it must be ensured that such distributed microservices provide fast, more than adequate, system performance for the globally distributed users.

***Transformation***

- Deploy the microservices across different regions.
- Each region will consist of multiple zones; each zone will be comprised of one or more microservices. Each region is completely independent in terms of power, separate databases, memory and backbone network connectivity.
- Create master databases and synchronise the data by replicating the necessary data items from each region in the master database.
- In case any of the microservices in any of the regions becomes unavailable, the user can access the same data from any other location.

**Algorithm 5.17:** Geolocation microservice - master database synchronization

```
1 $\mu$= Microservice Architecture;
2 $\mu$_DI = Microservice Data Integrity;
3 $\mu$_DA = Microservice Data Availability;
4 MDB = Master Database;
5 if ($\mu$) Require to deploy in different region then
      • Create MDB in order to
      • Improve ($\mu$_DA ) $\wedge$
      • Improve _DR;
6 end
```

***Impact on features***

- This rule ensures data availability and integrity via synchronous replication.
- This replication of data reduces data access latencies to the user.

- There might be temporary delay issues due to network latency or the load on the replication facility.
- Security and privacy may become an issue with distributed microservices since the client's data is stored across multiple regions.
- To increase read throughput, it may be necessary to allow multiple machines to serve read-only requests.

**Rule 4 - Geolocation microservice – prioritised request**

***Precondition***

- Microservices may need to be deployed across many regions and it must be ensured that such distributed microservices provide fast, more than adequate, system performance for the globally distributed users.

***Transformation***

- Deploy separate instances of each microservice in each region, including the data stores and the services that access them.
- Employ a master database whereby, depending on the request type and the request priority, the master database will be updated.
- If a transaction does not need to be processed instantly, asynchronous update will be used; otherwise synchronous updating will be applied, as described in Figure 5.2.

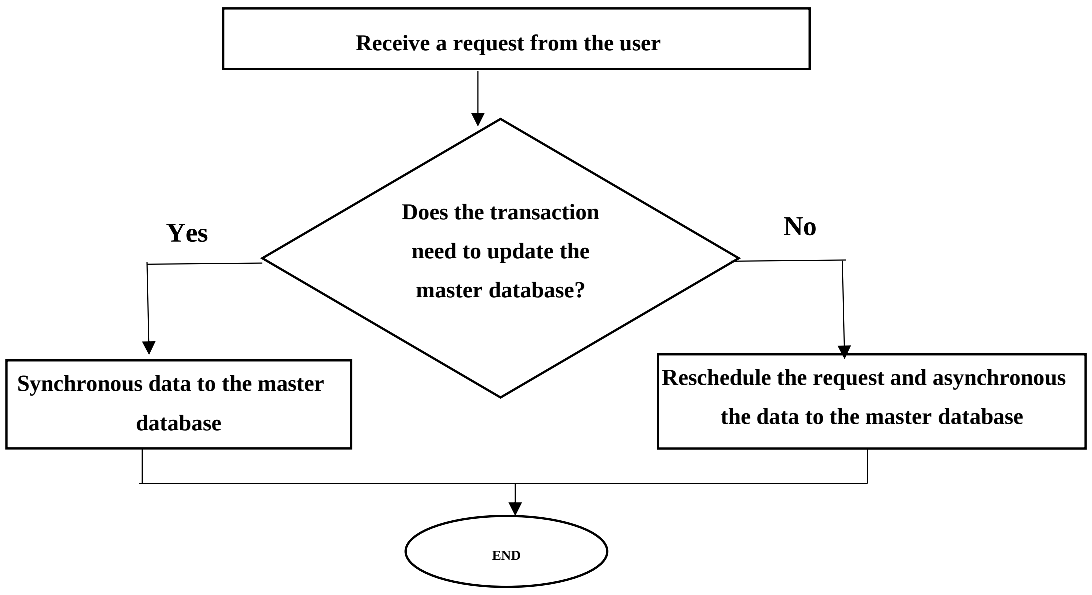

Receive a request from the user
Does the transaction need to update the master database?
Yes
No
Synchronous data to the master database
Reschedule the request and asynchronous the data to the master database
END

**Figure 5.2** Update Master DB based on Request priority.

- In the single queue approach, the number of consumers can be scaled back as necessary. High priority messages will still be processed first (although possibly more slowly), and lower priority messages might be delayed for longer.

- If the multiple message queue approach is implemented, with separate pools of consumers for each queue, the pool of consumers allocated to the lower priority queues can be reduced. Processing can even be suspended for some very low priority queues by stopping all the consumers that listen for messages on those queues.

**Algorithm 5.18:** Geolocation microservice - prioritised request

```
1 $\mu$= Microservice Architecture;
2 $\mu$_P= Microservice Data Performance;
3 MDB = Master Database;
4 if ($\mu$) Require to deploy in different region then
      • Deploy $\mu$ in separate instance $\wedge$
      • Create MDB
      if Transaction require to be updated on the MDB then
          • Synchronous data to the MDB
      else
          • Asynchronous data to the MDB in order to
          • Reduce opertional costs $\wedge$
          • Improve _P;
      end
5 end
```

***Impact on features***

- The multiple message queue approach can help maximise application performance and scalability by partitioning messages based on processing requirements. For example, vital tasks can be prioritised to be handled by receivers that run immediately while less important background tasks can be handled by receivers that are scheduled to run at less busy periods.
- It can help to minimise operational costs.
- Security and privacy may become an issue with distributed microservices since the client's data is stored across multiple regions.

**Rule 5 - Geolocation microservice – local/global master database**

***Precondition***

- Microservices may need to be deployed across many regions and it must be ensured that such distributed microservices provide fast, more than adequate, system performance for the globally distributed users.

***Transformation***

- Deploy microservices across different regions;
- Each region consists of multiple zones. Each zone supports one or more microservices, each with a separate database. For example, services located in the US or European zones will all use the power networks, backbone networks, and data-storage available in the same zone. Create a local master database in each region and a global master database.
    1. Synchronise data to the local master database.
    2. Synchronise data from the local master database to the global master database.
- However, if the transaction type is such that its result needs to be available in other regions instantly, the request will be routed accordingly and synchronised to the global master database [80].

**Algorithm 5.19:** Geolocation microservice - local/global master database

```
1 $\mu$= Microservice Architecture;
2 $\mu$_DA = Microservice Data Availability;
3 $\mu$_P= Microservice Database performnce ;
4 LMDB = Local Master Database;
5 GMDB = Global Master Database;
6 if ($\mu$) Require to deploy in different region then
      Create LMDB$\wedge$ create GMDB $\wedge$
      Synchronise data from LMDB to GMDB in order to
      Improve ($\mu$_DA ) $\wedge$
      Improve ($\mu$_P);
7 end
```

***Impact on features***

- There might be temporary delay issues due to network latency or the loading on the replication process.
- Security and privacy may become an issue with distributed microservices since the client's data is stored across multiple regions.

**Rule 6 - Containerised microservice**

***Precondition***

- Microservices must be deployed and scaled independently; they also need to be isolated from one another. When one service misbehaves, other microservices should not be impacted.

***Transformation***

- Package the service as a container image; this makes its image portable across the different operating systems involved and means that it runs in an isolated environment. It is possible to run several of containers on a single VM without any interference between them.

- Containers are very lightweight and consume only what is needed from the host OS,which reduces the possibility of resource conflicts.

**Algorithm 5.20:** Geolocation microservice - Containerised microservice

```
1 $\mu$= Microservice Architecture;
2 $\mu$_DA = Microservice Data Availability;
3 if ($\mu$) Require to deploy independently $\wedge$ isolated from other ($\mu$) then
      Run $\mu$ in container in order to
      Reduce overhead processing $\wedge$
      Improve $\mu$_DA ;
4 end
```

**Impact on features**

- Adopting containers will lower computational overhead and lead to better isolation.

- Containers are not as secure as VMs and to manage this issue the developer must make the configurations of infrastructure use as secure as possible; the infrastructure should be monitored at all times.

## 5.6 Microservices Transformation Roadmap

In this section, candidate features are examined to see how effectively they can be transferred to microservices. For this, the manual identification of the candidate features was implemented by navigating the code and defining the main reason of each class. The functionality of the system helps to define the individual's entities in the legacy system. The migration process was done by one feature at a time and one function at a time.

The ultimate goal of adopting a microservices architecture is to benefit from the single responsibility and independent deployment of each service. However, how to design and build each microservice must be carefully considered, in isolation, and then all the services must be tested working together in the implementation stage before the final release. It is also critical to think properly about the scope and size of each microservice. To construct a microservices architecture, the system had to be analysed in-depth so that the entire architecture was understood. The process used to transform the system into a microservices-based architecture is shown in Figure 5.3 and is described below.

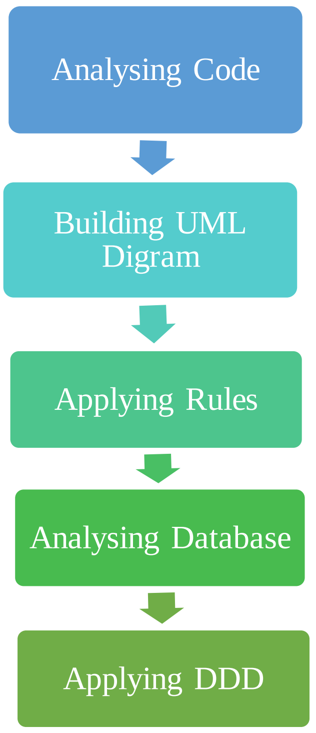


**Figure 5.3** The Process to transform RosarioSIS legacy system to a microservice-based architecture

(a) **Analysing the code and define the system's boundaries**

Transposing the legacy system to a microservices-based architecture began with an analysis of the existing system to identify, from the legacy code, all service elements used within it, including files, modules, classes and procedures. To discover the most data intensive procedure, in other words the one that reads/writes the most data, the procedures were then reviewed to determine where they were defined, what they did, and where procedure calls to them were located. This revealed the most commonly referenced procedures in the code; these also tend to be the most data intensive. Finally, all the connected modules were explicitly defined to identify the structure of the system and the relations between the modules.

(b) **Building a UML diagram**

Next, a class diagram for the existing system was drawn to abstract the functionality of all the modules in the system. This assisted in the understanding of the relationships between the system classes, and in addition plays an essential role in determining what the services should be. Besides all this, such a diagram shows how the existing code can be turned to new uses, via well targeted modification.

(c) **Applying Rules**

Once an abstracted overview of the existing system was formed, transformation rules had to be chosen and applied to redesign the system in consideration of the objectives of involvement. One candidate heuristic is to break up the code so that functions (mostly) remain local to a service. Each service should have a dedicated function. This rule creates clear boundaries between each microservice and indicates which piece(s) of code should be changed for the functionality of each service. The outcome of this step was identifying main functionalities and responsibilities, the key task was the use of the feature-driven rules.

(d) **Analysing the database schema**

This step focuses on building an understanding of the shared database schema. Generally, the database is broken up and data tables are extracted and redefined in such a way that each can be placed in an isolated, independent database which may be used later by only one of the microservices. These activities concern data management, which is the most essential component of any application. The main challenge presented at this stage was to identify which data services are closely related. In a microservices architecture, to achieve the implementation of independent and loosely coupled microservices, one dedicated database per microservice (Rule 3) is applied. This means that all the microservices' databases are independent of each other. A change in one database should not affect any other databases.

(e) **Applying Domain Driven Design**

Applying a Domain-Driven Design (DDD) technique, in which a main module is broken into a set of services, minimises the possibility of data sharing between two or more microservices. It is possible to use DDD and still end up with quite large services. To avoid that, the feature-driven rules (Rule 2 – Single Responsibility Principle) ensure that each service concentrates on one responsibility. Migrating data from a legacy software system requires careful planning, depending on each case, by identifying the tables corresponding to each service and creating a new database schema for each of the corresponding services. Then migrating one service at a time. To perform the transformation, developers can choose whatever technologies and tools are best suited for each service. Each case study has different requirements and priorities.

Given the need to deploy these services in different cloud environments, feature-driven migration rules are applied. There are several options to used based on the microservices requirements and specifications. Enterprises have the option to deploy a microservices architecture on public or private cloud. Also, runing microservices on container improve quality, reliability, and resiliency of the microservice architecture.

## 5.7 Summary

This chapter addressed how the FDD is used to identify and specify NFRs. Also, it discussed how this classification can act like a guideline to the introduction of feature-driven rules. Based on the FDD, a set of system requirements for microservice-based legacy system transformation and migration were defined. The system requirements, in which NFRs are included, play a vital role in the success of a software development process. The interpretation of the NFRs depends on the system that is being implemented. To achieve one NFR may mean strengthening, or alternatively losing, another NFR. To base the development/migration on a solid foundation, a set of key features were identified; performance, functionality, availability and scalability. Then, a list of sub-features were identified along with the relationships that exist between them. These defined requirements expose the relationships between functional and non-functional elements which are necessary to be aware of to build a new system. Each feature is important in terms of defining the scope of the system and in terms of providing a clear picture of the required result.

Once the above definitions were determined, a set of feature-driven transformation rules were derived. These rules are like the map that the software architect must navigate; steer through the rules and analyse the options to determine the suitability of each rule to the legacy monolithic system's goals, to reach the final destination of a successful microservices-based system. Also, these rules represent a realistic approach to the exploration of both the benefits and the drawbacks of transformation. The microservice-based architecture is constructed by first decomposing the monolithic system into small independent components, having specific functions, and isolated databases. This converts a module, possibly using significant resources, to a separate service based on the rules. Furthermore, rules are provided to facilitate the analysis of the deployment of the microservice-based architecture to the cloud and how that affects the quality attributes, such as performance, scalability and availability.

# Chapter 6

# Case Study and Evaluation

## 6.1 Introduction

Chapter 5 proposed a set of feature-driven rules that help determine when to migrate and where migration begins. This chapter introduces two case studies which include the evaluation of these feature-driven rules and presents the results of these evaluations.

In the first case study, migration was implemented using a relevant sub-set of the rules laid out in Chapter 5 and was tested using the Jmeter tool [1]; this measures how long each of the systems (monolithic, legacy system and microservices system) takes to process the user request, in relation to a pre-specified time limit. Additionally, the comparison of the two system implementations, using the two different architectures, reveals the impact on performance of the transformation to a microservices architecture.

In the second case study, the microservice system was designed by selecting the most suitable feature-driven rules with respect to the system's aims and functionality. The rules were combined to define the best route to migrate the legacy systems to a microservices architecture. The usability of the rules was evaluated by analysing the impact of the new microservices-based architecture on the NFRs. The evaluation highlighted the way in which this new, microservices, paradigm is becoming crucial as it is changing how enterprises handle information and resolve legacy system weaknesses such as scaling and maintenance challenges.

## 6.2 Case Study 1

This section presents a case study based on a monolithic system called ‘RosarioSIS’ [2], which is an open-source legacy Student Information System (SIS). This system was analysed and then transposed to a new microservice-based architecture. The new system was fully implemented and deployed on a cloud platform. This section first introduces the original monolithic RosarioSIS system and then provides a description of its migration into a microservices architecture.

RosarioSIS provides services across several schools, helping to manage the schools, their staff, the students, grades, payment, etc. The new microservices architecture was built using a Laravel and Swagger API framework [81] [82].However, to ensure the case study remained relatively simple, the migrated application was designed to support only the four most popular services. The first service, ‘School’, is for managing the school itself. The second service, ‘Student’, is employed for adding, updating, and deleting student records. The third module is the ‘User’ microservice, which presents the user-profiles and the teacher programs. Finally, the ‘Grades’ service manages the GPA and presents the user’s grades to them on request.

## 6.3 The Current, Monolithic, Architecture of RosarioSIS

RosarioSIS [83] runs as a publicly accessible web application. The prime objective of RosarioSIS is to provide schools with a platform where they can manage their staff and students. The system consists of numerous different components via which the school admin can manage teachers, students, attendance, fees, events, courses, and resources.Furthermore, there are several roles recognised by the system, such as teacher, staff member, payroll, and administrator; the latter can only be registered with the system by the current school admin. Each role has a certain number of duties associated with it and these are determined by the school admin. Moreover, each student will have their own web panel from which they can manage their leave, fees, and other important details.

current system has a 3-tiered architecture, as shown in Figure 6.1.

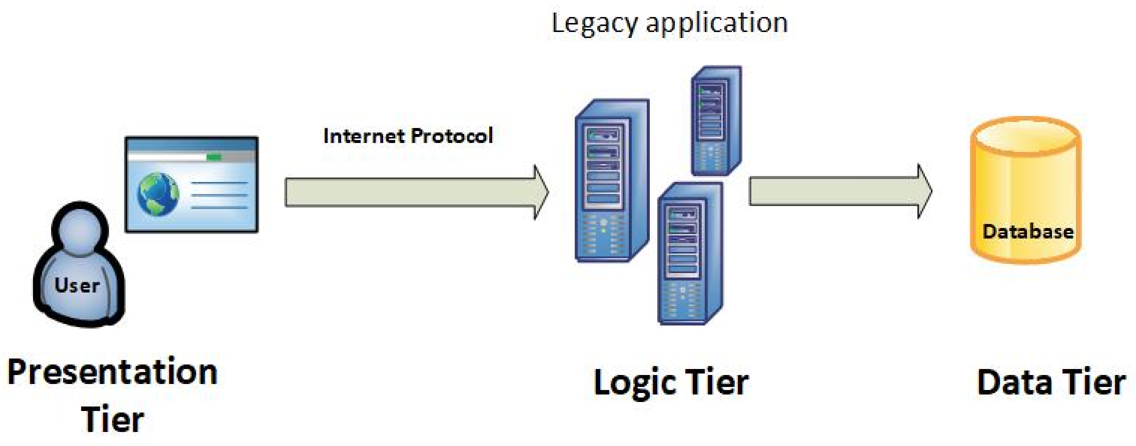


**Figure 6.1** Monolithic architecture of RosarioSIS

**The Presentation Tier** is the user interface, which is responsible for translating the results of the various operations into a readable format for the user and displaying them. Users can access the system via any web browser they may have. The Hypertext Markup Language (HTML) components of the presentation tier assemble the data received from the logic tier, described below, and then display it. The user communicates with the web server via the Hypertext Transfer Protocol (HTTP), to send and receive requests.

**The Logic Tier** works as a coordinator, controlling interactions between the presentation tier and the data tier, described below. User requests are received from the user interface (i.e. the presentation tier) and then converted into actions to be performed via appropriate communications with the data tier. Any results are sent back to the logic tier which will then pass them on to the presentation tier for display. For instance, the logic tier processes a client's request to view students' records, the logic tier receives this request and then passes it to the data tier to process and extract all the information relating to the students from the database so that these can be displayed.

**The Data Tier** Tier holds the application data, such as students, teachers, timetables, assignments and grades data, etc. This information is stored in a relational database management system in order to facilitate data reusability and manageability. All the data needed by the application are retrieved from this database. In addition, all the computation results from the logic tier are stored in the data tier.

The RosarioSIS legacy system was implemented as a set of web applications. Its architecture is very monolithic albeit with a modular structure. It may be accessed through an intranet where there is no Internet access (i.e. offline). The system's components are:

- The web application developed using PHP 5.3. The relational database employed was PostgreSQL.
- The front-end application, developed in HTML-5, CSS 3, and jQuery.

The system can be considered as one package containing all the necessary modules. Such a system structure makes it an excellent candidate for adopting a microservice-based architecture. Its major functionality can be identified as following as an Entity Relationship Diagram (ERD), shown in Figure 6.2 and described below.

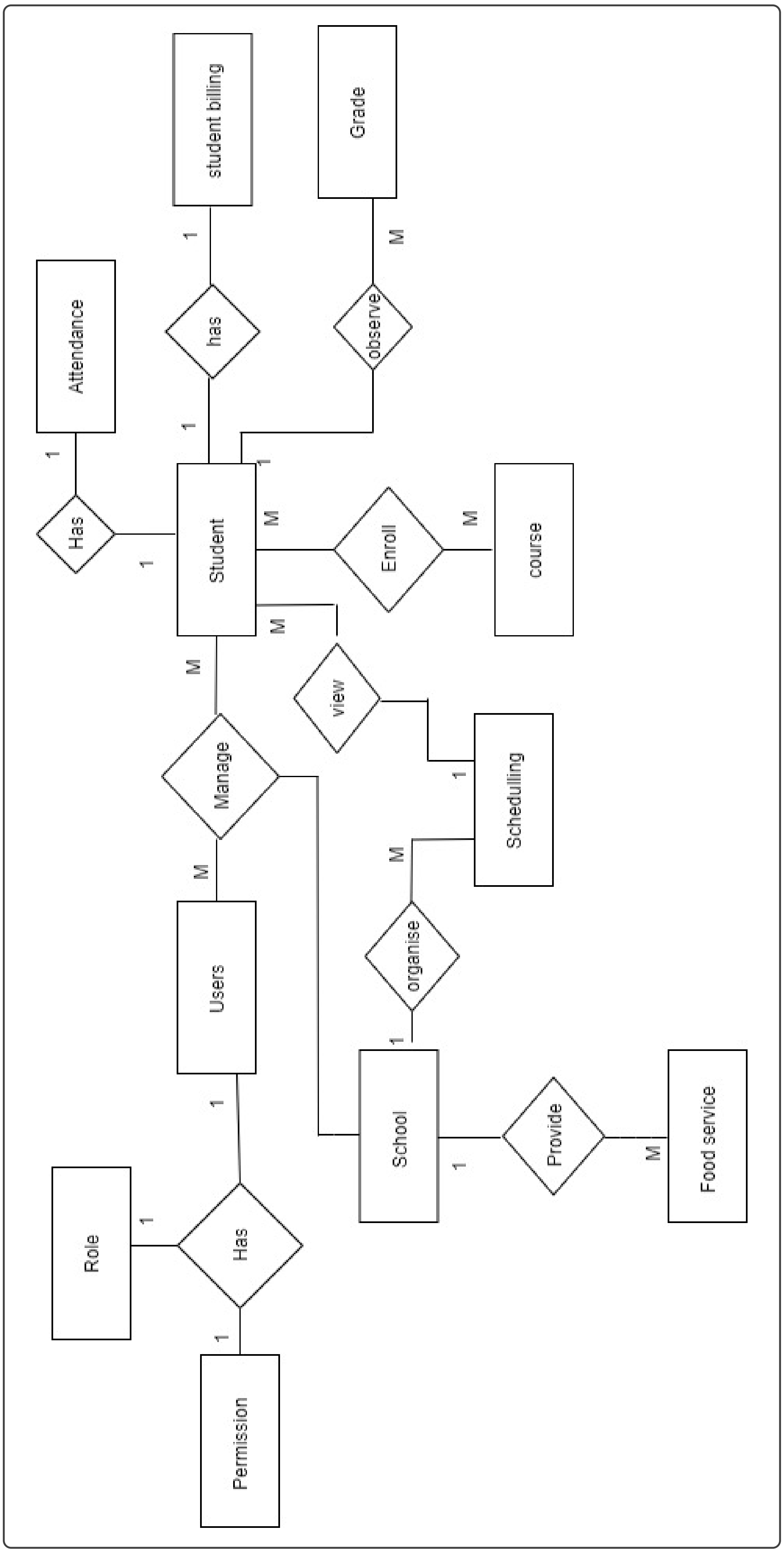


**Figure 6.2** RosarioSIS Monolithic ERD

- The ‘School’ module is used to add a school, set up marking periods, set up grade levels, manage a school’s calendar and schedule school events. In addition, this maintains the publishing notes and allows the system to be configured.
- The ‘Student’ module facilitates student enrolment and allows student information to be edited. It is also responsible for generating and printing advanced reports and formatted letters to communicate with students.
- The ‘User’ module manages user profiles by facilitating the addition of the various user types (administrator, teacher, parent ) and the editing of their information. The permissions under which each user must operate are determined in this module.
- The ‘Scheduling’ module organises school subjects, courses, and student schedules. It also generates and prints schedules and class lists.
- The ‘Grade’ module is responsible for creating assignments, managing the gradebook and recording final grades. As well as this, the students can consult it to access their grades and print transcripts and honour lists.
- The ‘Student billing’ module handles the school’s expenses, staff salaries, and student fees. It also generates and print statements.
- The ‘Food services’ module tracks the meals served and manages student and staff accounts.

## 6.4 New Microservices Architecture of RosarioSIS

To deliver an accessible demonstration and evaluation of the proposed rule set, a legacy, monolithic application was chosen for migration to micro services and the cloud. To simplify matters, the microservices architecture was designed to support only the four most popular services from the legacy application. The first service, ‘School’, is for managing the school itself. The second service,‘Student’, is employed for adding, updating, and deleting student records. The third module is the ‘User’ microservice, which maintains the user-profiles and the teacher programs. Finally, there is the ‘Grades’ service, which manages the GPA and presents the grades to the user on request.

Creating an application based on microservices is not like a monolithic application. For this reason, functionalities in monolithic systems are divided into different services. To do this, it is important to follow an adequate design and structure for each of the microservices according to its requirements.

The implementation stage is responsible of dividing the legacy system into logical parts and groups them according to their existing relationship. Feature-driven transformation rules take care of defining which concrete elements support each of the microservices, for example, where the data is stored, or breaking modules based on specific requirements. A feature-driven rule is a process solution to a recurrent problem in a real-world application development. Feature-driven rules will make software more stable and reliable. During this stage, the main focus is to transform the RosarioSIS legacy system to microservice architecture and the main goal is to have as stable and reliable a system as possible. Some of the feature transformation rules will be used, such as the legacy system decomposition into microservices (Rule 1), SRP rule (Rule 2) and one dedicated database per microservice (Rule 3).

The transformation process starts by following the understanding that the RosarioSIS legacy architecture, dependencies and relationships between modules have to be recognised. There are tools available to help read the source code and generate a visual diagram that has all the modules with their relationships. However, in this case study, this step has been done manually by reading all the code, trying to identify the relationships between the modules, and drawing the ERD by dividing the functionality of the RosarioSIS legacy system. To understand the architecture at this level, dependencies and communication relationships between the microservices must be known. However, analysing communication relationships is difficult. This makes it possible to verify the implemented architecture, adjust it to the planned architecture, and follow the evolution of the architecture over time.

The process of isolation in this case study was done using DDD, which provided an idea of how to determine the boundaries of the problems with respect to the various RosarioSIS legacy sub-systems, such as the user management, educational, and food service sub-systems. Breaking the larger context into smaller chunks provides a clearer idea of how data moves from one component to another. As microservices must be isolated, it is critical that each component remains in its own bounded context and has an obvious responsibility. For instance, there is a student module in RosarioSIS which performs the student management tasks, and thus the information which relates to each student must be kept within this subdomain. Also, each student is associated with one or more parents, so parents' profiles must also be maintained here. The extraction is not easy, all the modules that are connected or being used by this module have to be checked. Following this, a new table in a new database must be created, so the student microservice will use it.

The same kind of process of analysis was applied to the other modules until a clear structure of domains and subdomains, in relation to the RosarioSIS legacy system, was identified. The bounded contexts of students, school, users, and grades are presented in Figures (6.3, 6.4, 6.5, 6.6).

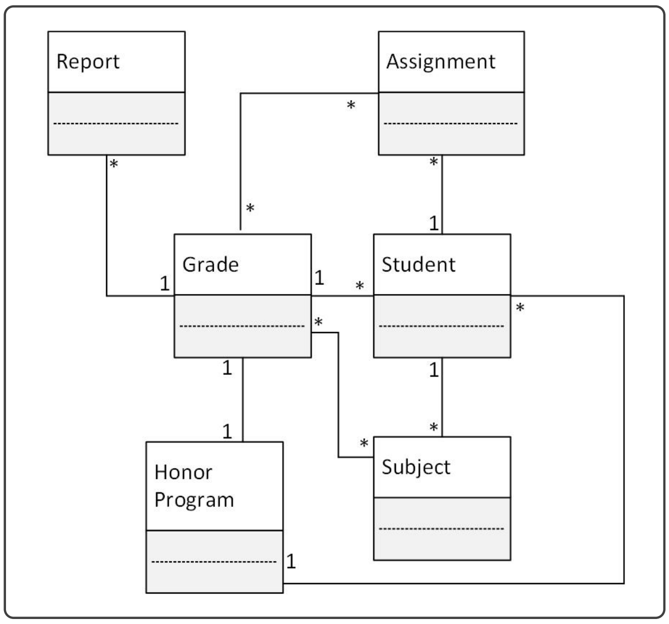


**Figure 6.3** Grades' Context

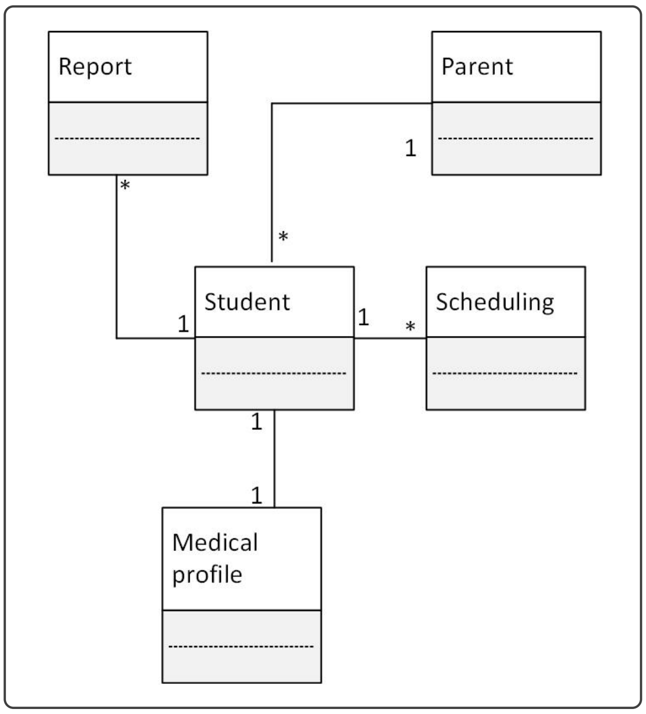


**Figure 6.4** Students' Context

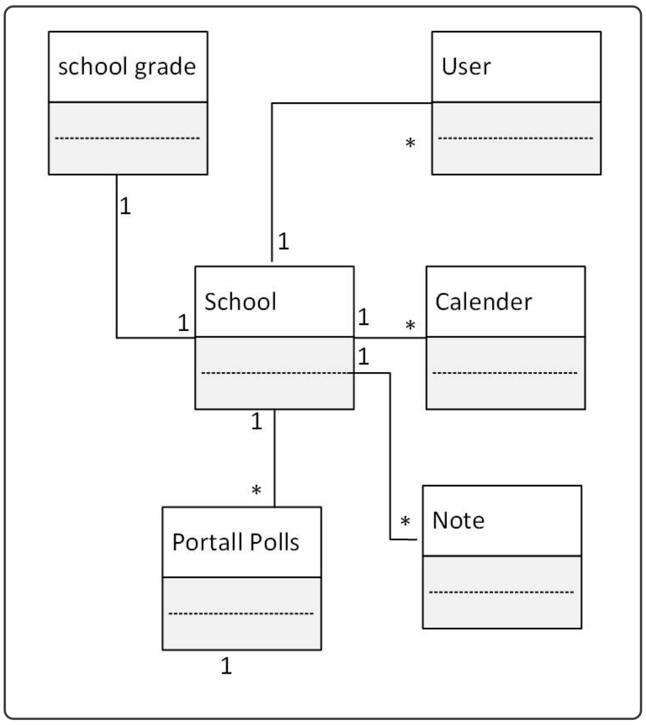


**Figure 6.5** School Context

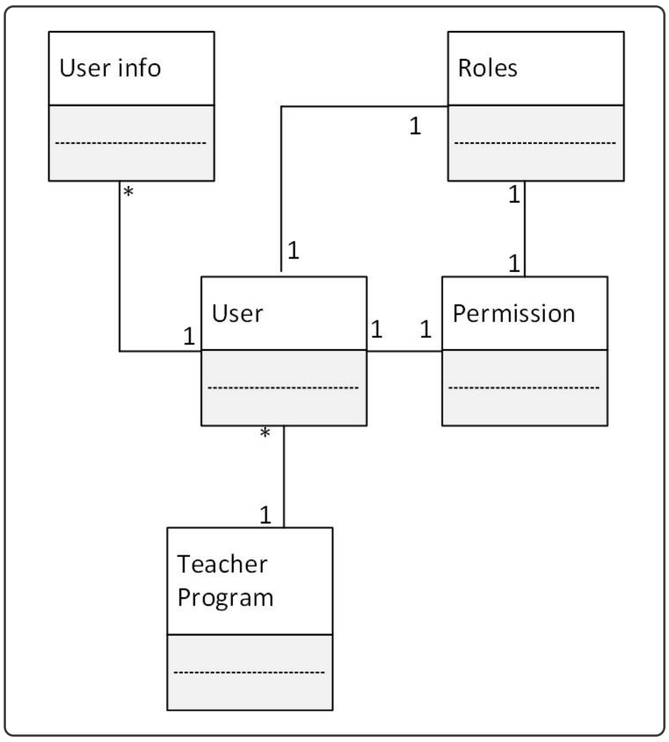


**Figure 6.6** Users' Context

After splitting the functions of the monolithic system into different microservices, the services were prioritised in terms of which needed to be built first to ensure the main services remained available. As a result, four services, were derived from the legacy PHP code. Each service addresses a specific business scope and they are fully decoupled from each other.

When adopting a microservices architecture, it is important, early on, to decide on the number of microservices to be built. For this case study, the above four microservices were selected as the only ones to be implemented due to the time constraints of the current work.

### a ) Implementation

Defining the technology stack that will be used to construct the framework is one of the core aspects of the architecture. Different technologies can be used and defining a standard technology for all the microservices is not mandatory. However, in this case study, due to the time limitation, all the microservices are built with the same technology stack . Also, there are benefits when all the microservices apply the same framework and tools; it will reduce complexity , and it will be easier for the developer to solve the issues, etc. This section will focus on the tools that were used to implement the RosarioSIS microservice architecture. Each microservice was developed as an independent Laravel MVC framework. The gateway was developed as a light web application which receives requests from end-users (browsers) through the Internet and consumes the private services offered by the microservices through Representational State Transfer (REST). The message exchange protocol which connects the browsers to the gateway, and the gateway to each microservice, is JSON. The gateway does not store any information. The microservices architecture was deployed on the Amazon ECS cloud platform, as illustrated in Figure 6.7.

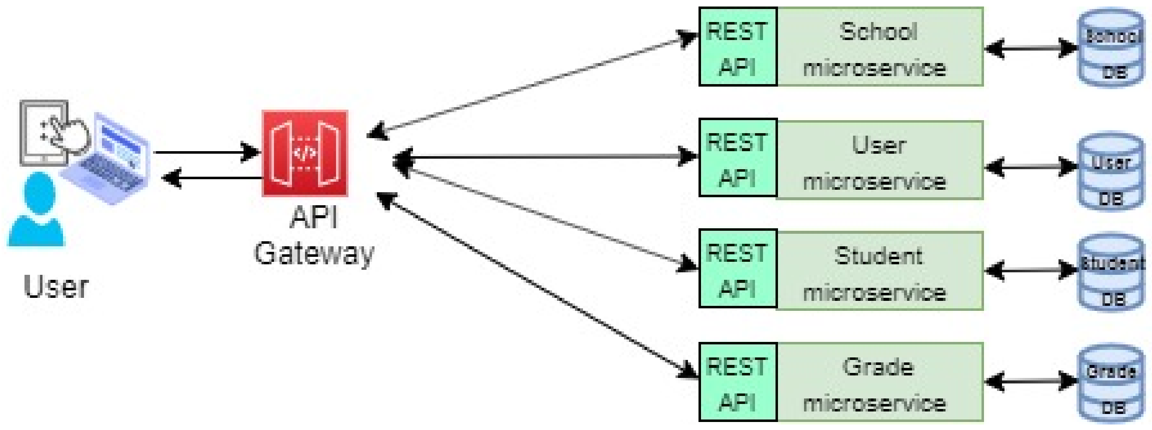


**Figure 6.7** Microservice Architecture

After the transformation of the system into one framed in a microservice-based architecture, this microservices architecture was implemented as four independent applications, supported by the AWS cloud: the 'School' microservice ($\mu$S1); the 'User' microservice ($\mu$S2); the 'Student' microservice ($\mu$S3); and the 'Grade' microservice ($\mu$S4). These were all developed using PHP 7.1 and the MySQL5.0.12 relational database system. The Swagger API gateway was used for developing the four APIs relating to the four microservices.

A microservice is built in a specific way that incorporates three main components: an API, a microservices logic unit, and a data store. These are shown in Figure 6.8 and detailed in the sub-sections below.

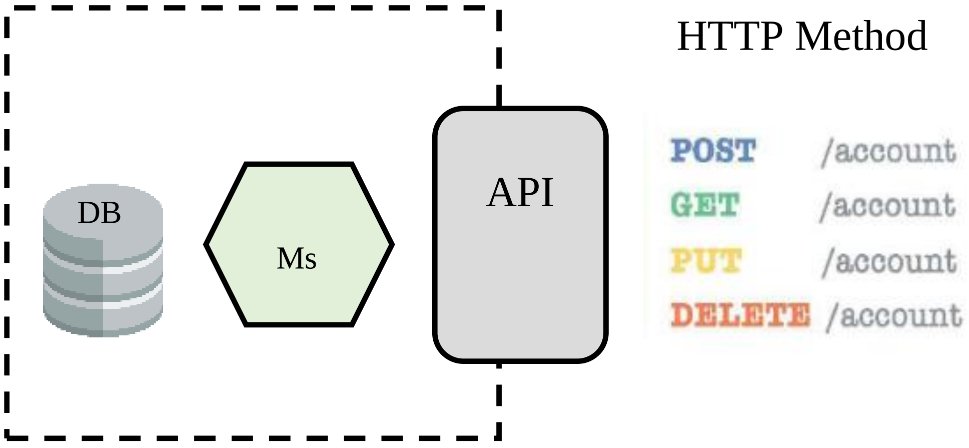


**Figure 6.8** Microservice Design

- **The API gateway and its Design**

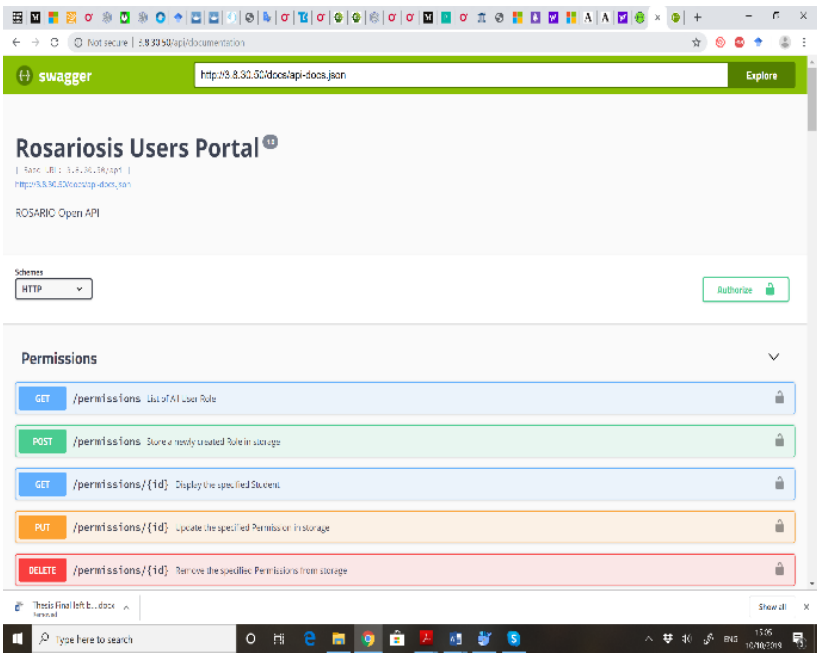
The APIs have been implemented using the Swagger gateway, an open-source API development framework [81],and were coded in the JSON format. The gateway defines the API operations which can be used, including POST, GET, DELETE, and PUT; these are supported at each endpoint. Swagger is used as an open API specification for identifying the functionality of the API gateway associated with a microservice. This specification defines how to use the API, how to enter values, etc. The API gateway forms a layer between the user and the microservices that encapsulates the internal system architecture and provides a tailored API for each client. Requests sent through the API gateway are routed to the applicable backend microservice. Requests to the backend services are independent of each other. The API gateway is a way to bridge the services; it has a certain amount of information which it holds to assist it in understanding the overall microservices system.

1. The API gateway can also have other responsibilities, such as authentication, input validation, monitoring, and response handling. The APIs are responsible for direct request and protocol translation, independently of each other. The API gateway implementation includes the following tools, Swagger and Passport:

1. **Swagger** is a framework for designing an API in different languages. Swagger is a set of specifications or rules for describing REST APIs. It can be used by different frameworks or tools. In terms of generating documentation for use with Laravel

projects, Swagger-PHP has proven to be the most reliable and problem-free system to use [81]. Swagger makes sure all the microservices on the system are available and easy to understand without accessing the code or the service documentation.

The Figure (6.9, 6.10, 6.11, 6.12) below show the format used for the description of the APIs for the four microservices; this format provides detailed information, including data inputs and outputs and the authentication method employed. Each API has different methods (get, put, post, delete ) and is expandable. By clicking on each method, such as in Figure 6.13, a full description of the parameters will be obtained along with an example, as shown in Figure 6.14. All the values can be tested, and an API response message appears in JSON based on the result of the value, as shown in Figure Figure 6.15.



http://3.8.30.50/docs/api-docs.json

Rosariosis Users Portal

[ Base URL: 3.8.30.50/api ]
http://3.8.30.50/docs/api-docs.json

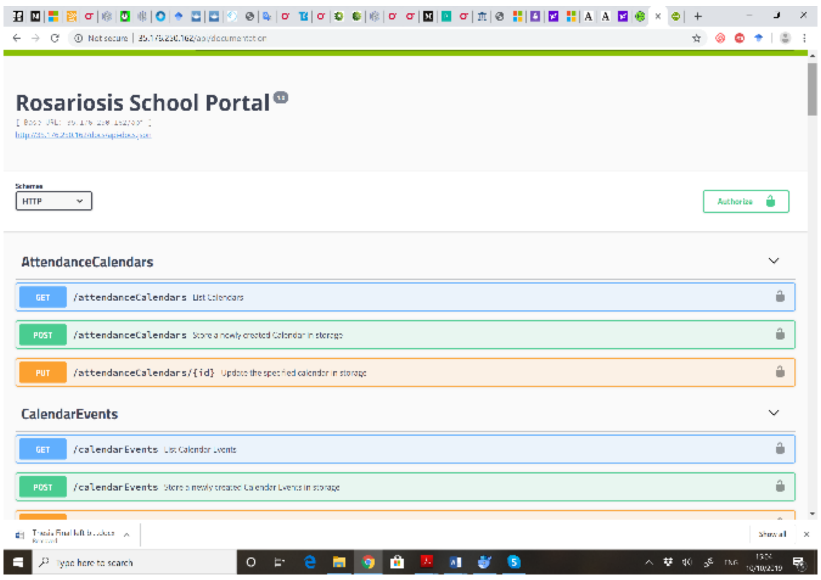
ROSARIO Open API

Schemes
HTTP

Permissions

GET /permissions List of All User Role
POST /permissions Store a newly created Role in storage
GET /permissions/{id} Display the specified Student
PUT /permissions/{id} Update the specified Permission in storage
DELETE /permissions/{id} Remove the specified Permissions from storage

**Figure 6.9** User API


Rosariosis School Portal

Schemes: HTTP

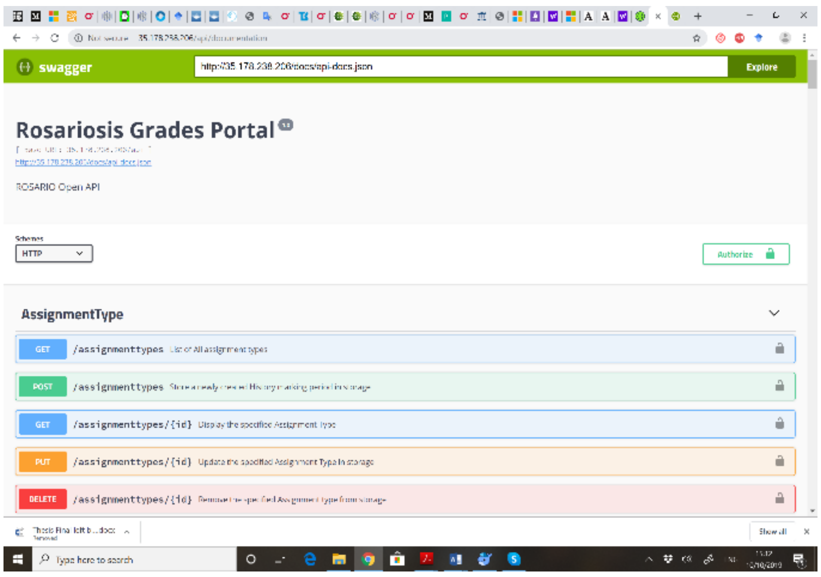
AttendanceCalendars

```
GET    /attendanceCalendars       List Calendars
POST   /attendanceCalendars       Store a newly created Calendar in storage
PUT    /attendanceCalendars/{id}  Update the specified calendar in storage
```

CalendarEvents

```
GET    /calendarEvents  List Calendar Events
POST   /calendarEvents  Store a newly created Calendar Events in storage
```

**Figure 6.10** School API



Rosariosis Grades Portal

http://35.178.238.206/docs/api-docs.json

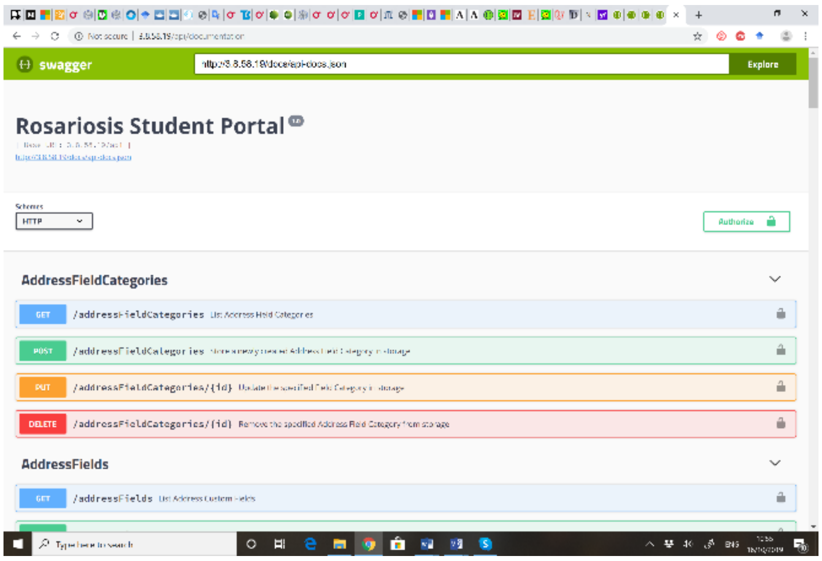
ROSARIO Open API

Schemes: HTTP

AssignmentType

```
GET     /assignmenttypes       List of All assignment types
POST    /assignmenttypes       Store a newly created History marking period in storage
GET     /assignmenttypes/{id}  Display the specified Assignment Type
PUT     /assignmenttypes/{id}  Update the specified Assignment Type in storage
DELETE  /assignmenttypes/{id}  Remove the specified Assignment Type from storage
```

**Figure 6.11** Grade API



Rosariosis Student Portal

AddressFieldCategories

GET /addressFieldCategories List Address Field Categories

POST /addressFieldCategories Store a newly created AddressField Category in storage

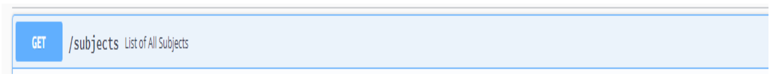
PUT /addressFieldCategories/{id} Update the specified Field Category in storage

DELETE /addressFieldCategories/{id} Remove the specified Address Field Category from storage

AddressFields

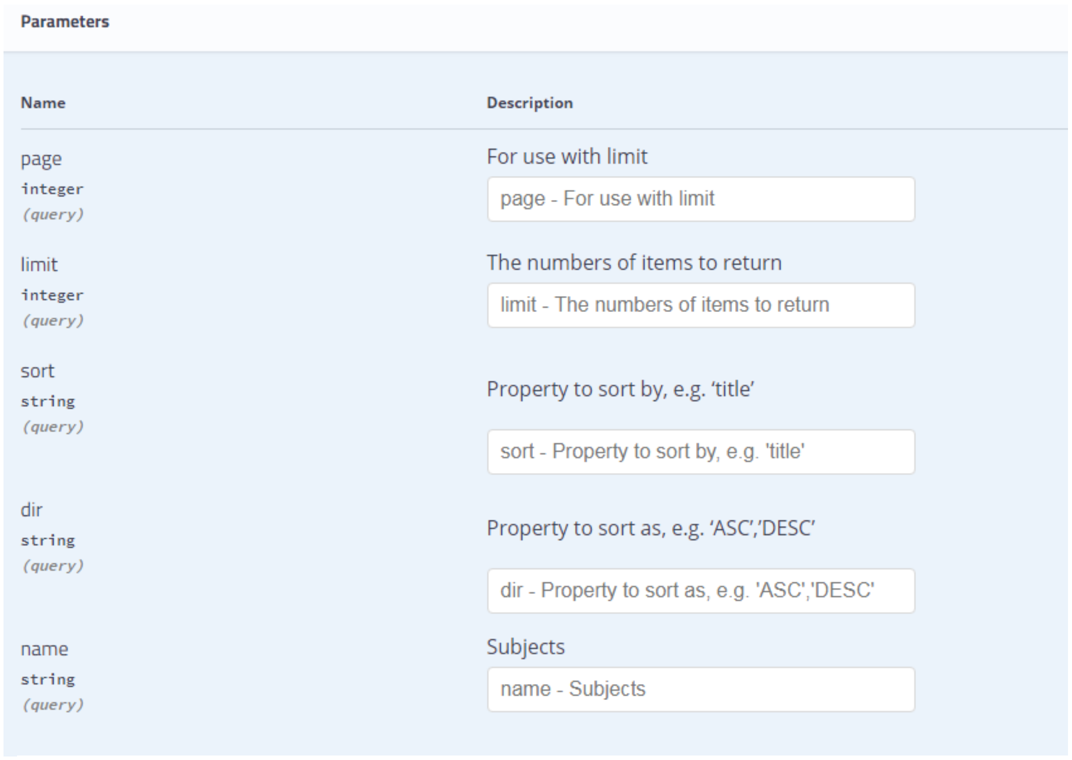
GET /addressFields List Address Custom Fields

**Figure 6.12** Student API



GET /subjects List of All Subjects

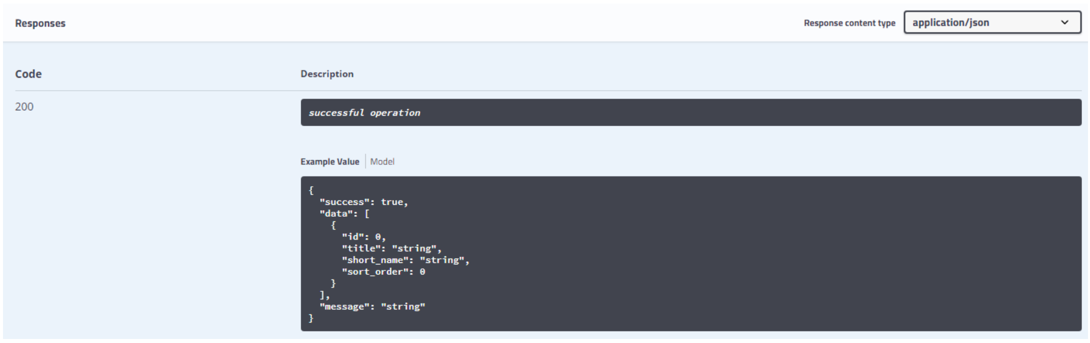
**Figure 6.13** SAPI GET operation



**Parameters**

| Name | Description |
|---|---|
| page<br>`integer`<br>*(query)* | For use with limit<br>page - For use with limit |
| limit<br>`integer`<br>*(query)* | The numbers of items to return<br>limit - The numbers of items to return |
| sort<br>`string`<br>*(query)* | Property to sort by, e.g. ‘title’<br>sort - Property to sort by, e.g. 'title' |
| dir<br>`string`<br>*(query)* | Property to sort as, e.g. ‘ASC’,’DESC’<br>dir - Property to sort as, e.g. 'ASC','DESC' |
| name<br>`string`<br>*(query)* | Subjects<br>name - Subjects |

**Figure 6.14** API input Fields



**Responses**

Response content type: application/json

| Code | Description |
|---|---|
| 200 | *successful operation* |

Example Value | Model

```
{
  "success": true,
  "data": [
    {
      "id": 0,
      "title": "string",
      "short_name": "string",
      "sort_order": 0
    }
  ],
  "message": "string"
}
```

**Figure 6.15** Example of API Message Response

2. **Passport** is a user authentication technology. There are several authentication methods included and these vary between a log in, with a username and password, and the use of an OAuth provider. In an API-based system, a token is used to authenticate users. The Laravel framework uses the Laravel passport; this employs an OAuth2 technology to issue a token to a user [84]. To understand the microservice security regime implemented here, it is important to first look at the way security works in monolithic applications. This will help to see the differences between the authentication and authorisation mechanisms of the RosarioSIS monolithic system and the RosarioSIS microservices system. .

   In a monolithic application, the purpose of authentication is always to verify the identity of a user. In addition, authorisation manages what a user can or cannot access (in other words, permissions). Also, the data which is passed between the client and the server can be encrypted with reference to the user's identity. Usually, the user enters a username and password through a web browser, and the server then verifies these given credentials. A 'session' is created, and this is stored in the database. A cookie and session ID will be kept in the web browser (client side) and the session will be removed from both the web browser and the server side once the user logs out.

   In contrast, here the microservices API authentication has been implemented with the use of Laravel passports. This represents a form of token-based authentication. The user enters a username and password, the server then verifies these user credentials and generates a token. The token is stored on the client side and the server verifies the token (a JSON web token) and returns the required data. Once the user logs out the token is destroyed. By applying this technique, it is ensured that the service user is allowed to access each specific service that they require.

## 6.5 Deployment in a Cloud Environment

To compare the infrastructures which support each architecture, both the monolithic architecture and the microservices architecture were first deployed using the Amazon Elastic Container Service (Amazon ECS) [85]. The deployment process is conducted two stages.

1. Server installation on Amazon Web service.
   In this step, an Amazon instance was initiated by setting up the software configuration first, such as the operating system, application server, storage system and the server (the location which was decided upon was EU West (London) regional server).

2. Creating an ECS Container on the AWS and configuring the cluster
   Both the RosarioSIS legacy system and the RosarioSIS microservices-based architecture were deployed on an Amazon cloud testbed. Five EC2 instances (virtual machines) running the Amazon Linux operating system were allocated to this testbed. The type of instance employed reflects the level and type of resources which it is expected to use while running the application. Each instance type has its own configuration of resources, resources such as CPU, memory, storage and networking.

   The RosarioSIS legacy system was run at the t2.micro level with only one virtual CPU, and a 'budget' of six CPU credits per hour. Therefore, it should be noted that, when a t2 instance runs out of CPU credits at peak usage, it is restricted to its baseline performance and this results in the application becoming slow. The deployment is illustrated Figure 6.16.

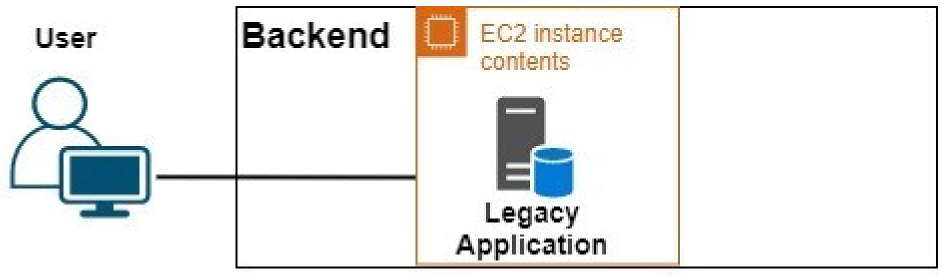


**Figure 6.16** The RosarioSIS Legacy System Deployment

On the other hand, the RosarioSIS microservices were run at t3 (small instance type). At t3.small, two virtual CPUs and a budget of 24 CPU credits per hour are allocated. Also, at this level, up to five times more bandwidth can be used when sending or receiving network traffic between instances within the same or different regions. This deployment. This deployment is illustrated in Figure 6.17.

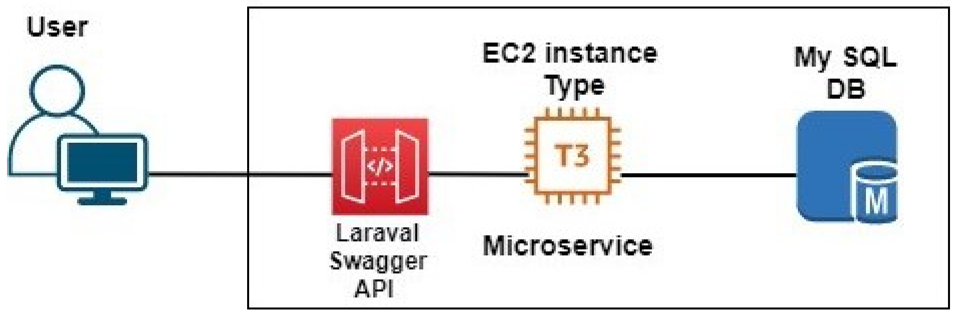


**Figure 6.17** The RosarioSIS Microservice-Based Architecture Deployment

Different deployment instance types were used for each type of architecture; this was intended to ensure that the same level of performance was attained by both. Both the legacy RosarioSIS monolithic architecture and the new RosarioSIS microservices-based architecture were deployed on their VM instance, as shown in Table 6.2, Table 6.1.

**Table 6.1** Monolithic Instance Details

| **Instance type** | **vCPU** | **CPU Credits /hour** | **Mem (GiB)** | **Network Performance** |
|---|---|---|---|---|
| T2.micro | 1 | 6 | 1 | Low to Moderate |

**Table 6.2** Microservice Instance Details

| **Instance type** | **vCPU** | **CPU Credits /hour** | **Mem (GiB)** | **Network Performance** |
|---|---|---|---|---|
| T3.small | 2 | 24 | 2 | Up to 5 |

## 6.6 Test and Analysis

Tests were conducted using the JMeter tool, because it is free and open source [81], against three performance metrics; error rate, throughput, and average response time. These metrics help to measure how the systems behave alongside a change in the number of users in real time. The comparison began by performing a targeted test for each microservice individually, to identify the maximum number of users that each service could handle simultaneously. Then, a performance test was carried out for the monolithic architecture to calculate the number of requests it could handle at the same time, in a specific ramp-up period.

The development/modification of the application undertaken for this case study allowed the performance of each architecture to be compared by performing stress tests on each. The stress tests evaluated the systems' behaviours under heavy load, varying the expected number of requests. The corresponding results from the experiments are described below.

### 6.6.1 Performance Test

To test and compare the performance and infrastructures of the four microservices, the same scenario was defined for all the microservices and the response time required for each was configured.

In this test, the response time and throughput were examined via the JMeter tool [86] by increasing the number of simulated users or requests loading the monolithic system's login page, user-detail page, school page, grade page, and student-detail page. The results of this test were then monitored. The response times yielded by the four microservices were all evaluated using the JMeter tool. The stress test saved records in the MYSQL databases of the RosarioSIS microservices applications and in the PostgreSQL database of the RosarioSIS monolithic application; the number of records was almost 500 for both architectures.

For the RosarioSIS legacy system and RosarioSIS microservices stress tests, the same number of requests were run on each system; this was to measure their average response times and average throughputs appropriately, which are provided below.

(a) **Response time**

To perform the stress test on the microservices architecture, the scenario was initiated by simulating a small number of requests (n=10) and then increasing this in increments of 10 until the application began to generate errors or the time limit defined for responses from S1, S2, S3, or S4 could no longer be met. The timeframe within which the maximum number of requests the system could handle before failing was examined and varied between 0.5 and 5 seconds.

The stress tests performed on the RosarioSIS monolithic architecture were based on the number of requests per second that were found to be supported by the microservices architecture. The performance tests were executed with the goal of identifying the maximum number of requests that could be supported by the monolithic architecture as implemented via Amazon AWS. This number was found by increasing the number of requests each time until the application began to generate errors, or the time limits defined for responses from the monolithic architecture were not being met. The timeframe within which requests were expected to be dealt with varied between 0.5 to 5 seconds; as before, given that this timeframe was determined by the results from the stress tests on the microservices architecture. Both the monolithic and the microservices architectures were platformed on the Amazon ECS2 cloud.

There is a noticeable difference between the length of the response times for the two architectures: all the microservices, S1, S2, S3 and S4, as Figure 6.18 indicates, yielded relatively slow response times from the beginning and these worsened rapidly with the increase in the number of requests per second. The highest number of requests attempted in one test was 80, which the microservices architecture completed in a total of 4.5 secs. Attempts to test the system with more requests than this resulted in errors being generated. On the other hand, the monolithic application , as shown in Figure 6.18, yielded relatively quick response times from the outset, but these slowed gradually as the number of requests increased. When the number of requests attempted reached 80, the average response time increased to 158 ms (milliseconds). The testing was discontinued when the number of requests reached 80, in order to easily construct a meaningful comparison. See Appendix A for further information regarding the tests, including the JMeter tests for both architectures.

(b) **Throughput**

The stress test was carried out to compare average throughputs between the legacy and the microservice architecture. The bar chart in Figure 6.19, shows that the average throughput attained by the microservices S1, S2, S3, and S4 remained relatively steady while the average throughput of the RosarioSIS legacy system rose steadily each time the number of requests increased.

These results show that the performance of both architectures will decline as the number of requests increases. An increase in the number of requests correspondingly increases the response time and this leads to a lower throughput. However, the monolithic architecture provides better performance overall, in relation to 40, 60, and 80 threads/requests. Based on this test's scenario and with respect to the defined time-frame, the monolithic application performs much better than the microservices architecture. This is due to the move from the monolithic system to the microservices architecture, which leads to an increase in the amount of computing resources that must be expended across several processes, e.g. network waits and the number of I/O operations. In contrast, in the monolithic system the whole computation (for each request) takes place in a single process. Referring to transformation rules (5&7)focused on computation tasks and throughput respectively, to improve throughput and decrease response time for the microservices architecture it is essential to reduce memory intensive functions by allocating an appropriate amount of resources to each microservice. Furthermore, if there is only one instance type for all the microservices then that instance type becomes over utilized. This means that the resources available for that EC2 instance, such as memory and CPU time, are usually at peak consumption all the time. Thus, if the resources available were to increase by having a unique instance type for each service, better performance would be achieved. Also, the more types of instance, and therefore instances, there are, the less failure will be observed.

To summarise, there is no way to know in advance precisely how a particular configuration will impact a specific application and set of requirements; the proposed set-up must be tested against relevant performance metrics.

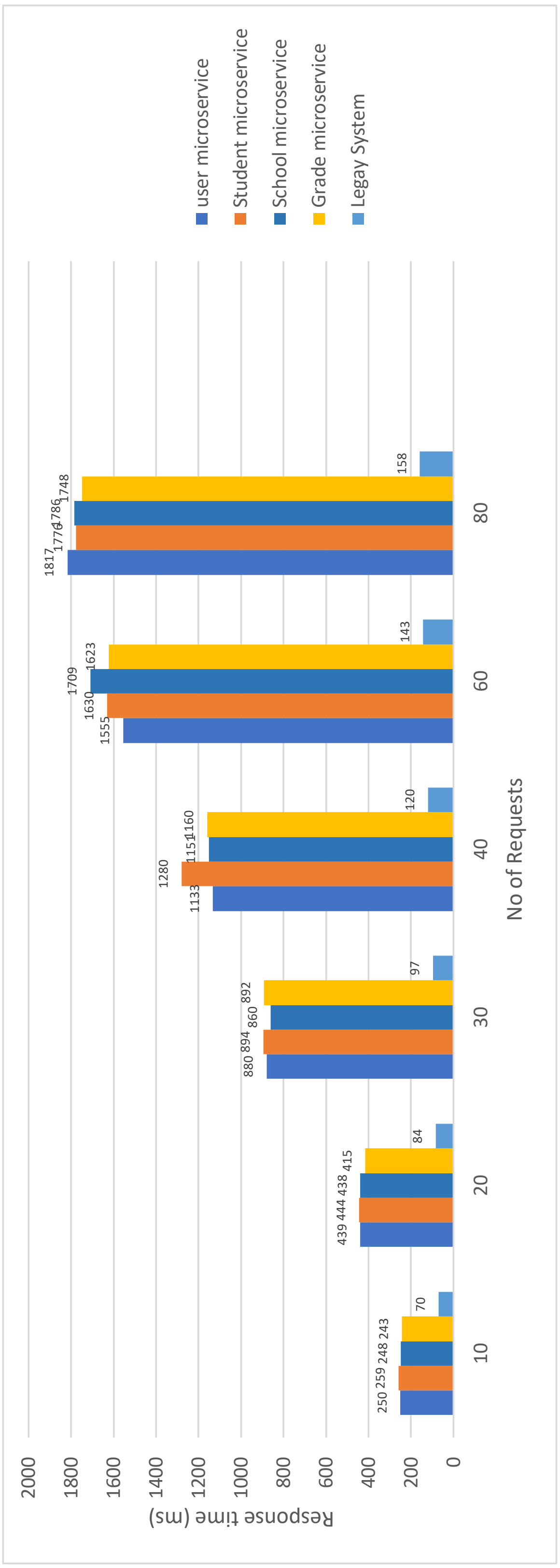


**Figure 6.18** Average response times for the RosarioSIS legacy system and the RosarioSIS microservices architecture

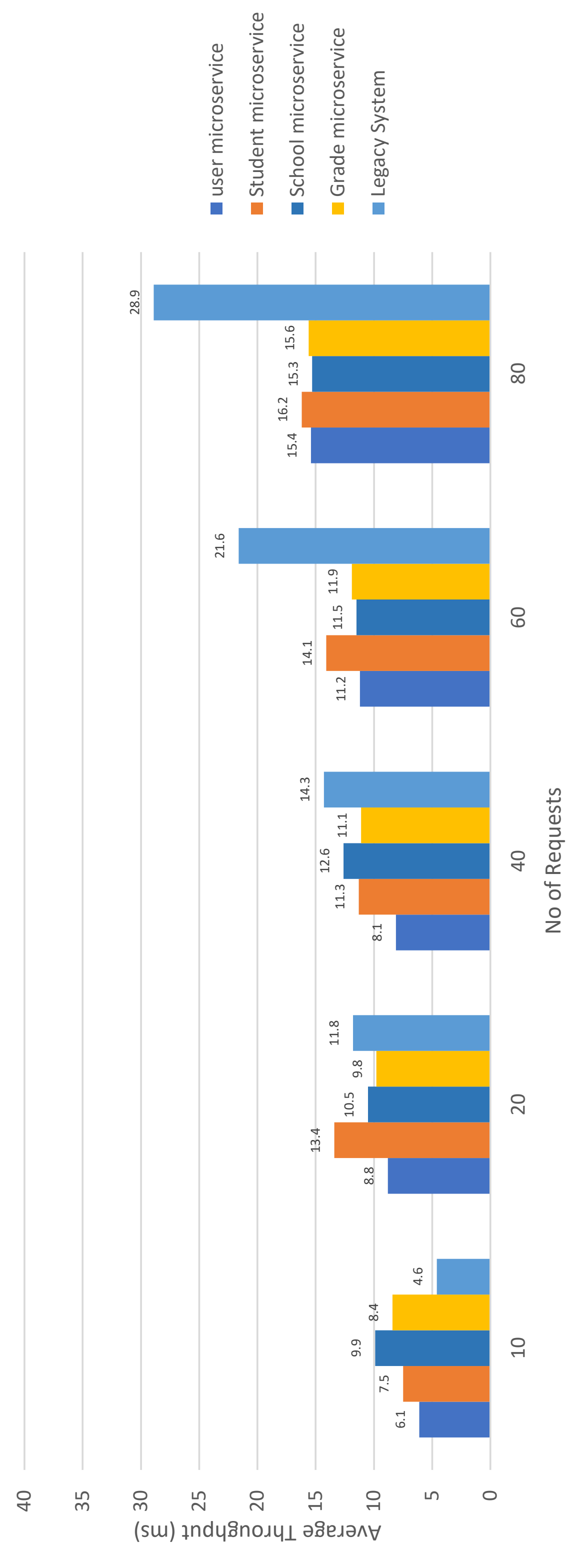


**Figure 6.19** Average throughput for the RosarioSIS legacy system and the RosarioSIS microservices architecture

## 6.7 Case Study 2

This section presents the second case study relating to the use of feature-driven rules. Here, a monolithic enterprise system 'ERP2' [87] is scrutinised and reframed into a new microservice-oriented architecture. The aim of this case study is to apply the evolution rules to a more complex and larger legacy enterprise system than that examined in Case Study 1. As such, the efficiency and scalability of the evolution framework and feature-driven rules are fairly evaluated. This case study also provides a comprehensive explanation of the use of feature-driven rules with respect to an enterprise system. The rules are framed in such a way as to enable developers and architects to choose the most appropriate ones for planning their transformations and migrations, combined with enhanced satisfaction of NFRs. The monolithic system and its core business components, such as supply chain management, human resource management, finance, and customer relationships, are reviewed and studied in in Sections section 6.7. An evaluation of the re-architected 'ERP2' is undertaken in Section subsection 6.7.3.

### 6.7.1 The Legacy System Transformation to Microservice Architecture

The ERP2 is a complex system and it consists of a set of interconnected modules with many-to-many relationships between most tables in its database. An overview of its data infrastructure is illustrated in Figure 6.20. . The ERP2 legacy system is implemented using out-of-date software and technology. This makes it difficult to upgrade the system, so that it can address new requirements, or to modify it, generally, in response to business demands. Besides, the ERP2 inherits a great range of technical issues as a result of it being a monolith; this challenges system efficiency and code modifiability and re-usability.

To address these issues, this legacy enterprise system is required to move towards a microservice-based architecture, which is technically challenging. Firstly, the migration requires a comprehensive understanding of the ERP2 system's legacy architecture, boundaries and code, as well as the business and technical requirements that the new architecture must meet. Then, it is necessary to identify the target components that must to be restructured in order to transcribe ERP2 into the new architecture smoothly.

The following paragraphs detail how the required understanding has been gained and how this helped to determine which microservice rules should be applied on the ERP2 System in order to transform it.

Understanding how data are stored and how entities relate to each other within the legacy system is the first step. This is important so that the optimal solution, in terms of transformation, could be attained. As the ERP2 is an enterprise system running with a single database supporting all its modules, the system components and its underlying database needed to be conceptualised

The feature-driven rules are applied across two stages, as follows, to extract the services from the legacy system

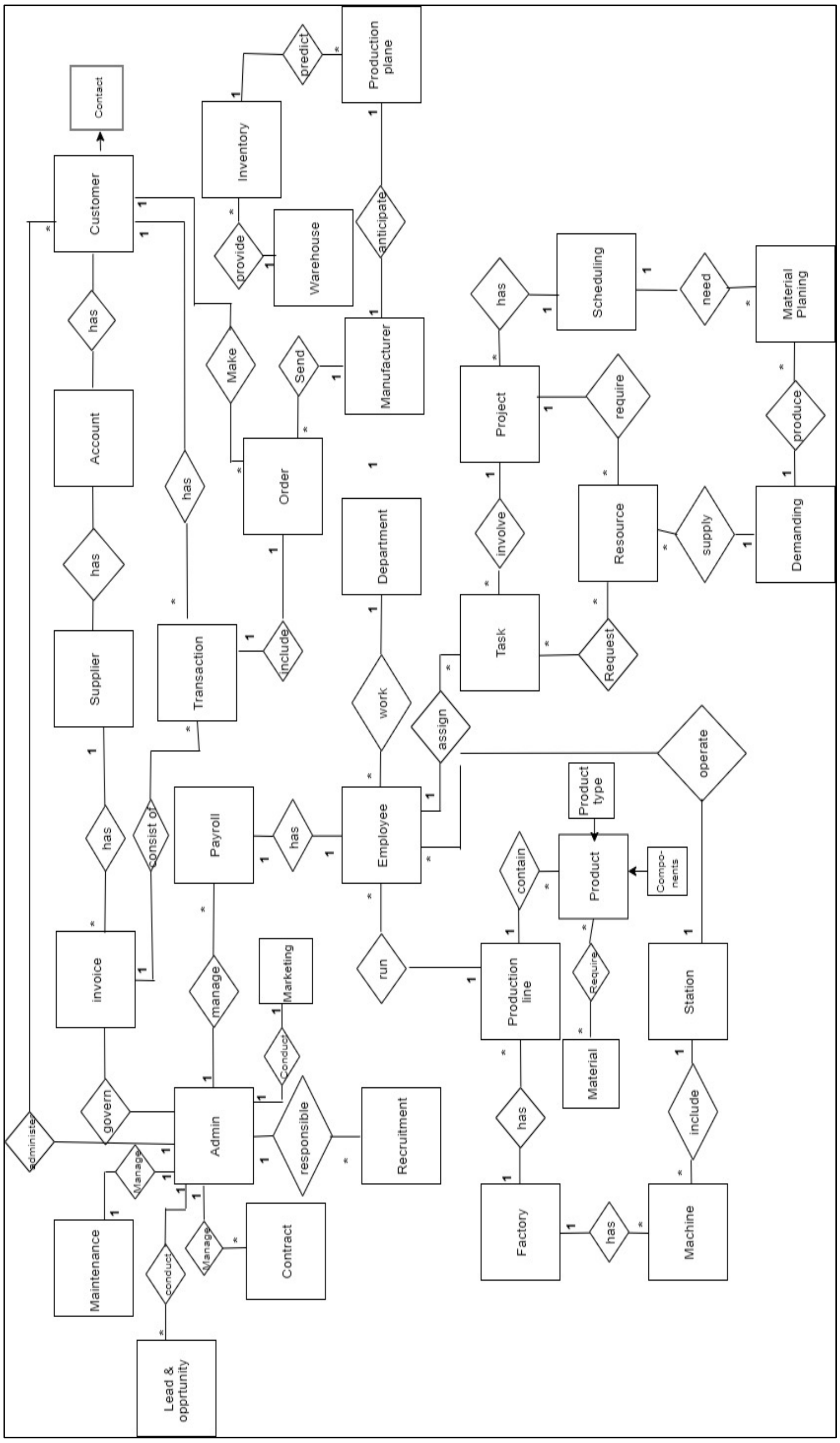


**Figure 6.20** ERP2 Entity Relationship Diagram.

***Step 1: understand the original system***

The aim here is to gain an understanding of the original system, focusing on its system domain to determine the functional dependencies and understand the functional requirements of each module [88]. Guided by this information, the scope of each module is defined. Then it is important to explicitly understand the code, the database schema, and how data are exchanged between entities. Some difficulties emerged in relation to this and appeared to be due to the poor allocation of resources in the monolithic system architecture; the system experienced enormous challenges in terms of the relationships among resources and system services and its performance. In addition, since the monolithic ERP2 system has components of different sizes (which share libraries, executables and resources), the precise identification of the resource requirements and bottlenecks of each module is critical to successful transformation.

***Step 2: Extract Services:***

The objective of this stage is to find an ideal feature-driven rule, which provides a direction in terms of the migration process – towards a microservices architecture. The purpose of such a rule is to facilitate the conversion of modules (or combinations thereof) to microsystems.

- *Decomposition of a legacy application into microservice-based architecture (Rule 1)*

  ERP2 is a large, complex monolithic application consisting of tens of modules, all of which are candidates for extraction and subsequent conversion. Deciding which modules to convert first is often a challenging process. A good approach is to start with the modules that are easy to extract. This provides the developer with a level of familiarity with the general use of microservices, and the extraction process in particular. The thinking behind this rule is that when a monolithic system become unmanageable, it may be beneficial to split the system functionality into standalone microservices.

- *Single responsibly principle (Rule 2)*

  When decomposing a system into small services, the Single Responsibility Principle (SRP) is applied. Things that change for the same reasons are grouped together. A precise service domain is drawn for each microservice following the bounded context concept of DDD. Every derived service accounts for only a single responsibility in order to retain its independence.

  A bounded context is a boundary delimiting an area within a particular domain whereby it is subdivided into independent subsystems that host the differing functions sharing a single domain. Each subdomain is like a linguistic boundary: within the boundary, everyone speaks the same (local) language.

  This (i.e., SRP with DDD) renders an insightful understanding of the system components and their relationships to each other, within a visual context. Monolithic applications, in general, can be deconstructed piece-by-piece and context-by context. They gradually move into a microservices architecture, where microservice domains and the microservices are open to any future refinements and changes [89].

  At this point, the current methods and modules used in the monolithic code are identified. After this, the methods that get called in the code within each module are determined as well as all the methods which depend on each module. For instance, the supply chain module in the ERP2 legacy system is responsible for: item information, order initiation, and checking the item's delivery to the customer. After selecting this module for examination, the methods called by this module and the other modules which depend on it (and in what ways) are reviewed: modules such as check invoice, check inventory, etc. This approach assists in defining a clear perspective on the module. Such information will help to extract this module from the legacy system and transform it into something which can fit into a microservices-based architecture. To proceed to the building of a microservices-based architecture, the rules are applied, using a now more complete understanding of the domain, applying DDD to form the domain modell.This domain model represents the functional perception of the system [90], ], and it provides hints for structuring the bounded context by representing the business process and the interactions between the different components. Following the two rules described above, nine microservices were derived, as shown in Figure 6.21; the domain of each microservice is illustrated from Figure 6.22 to Figure 6.29.

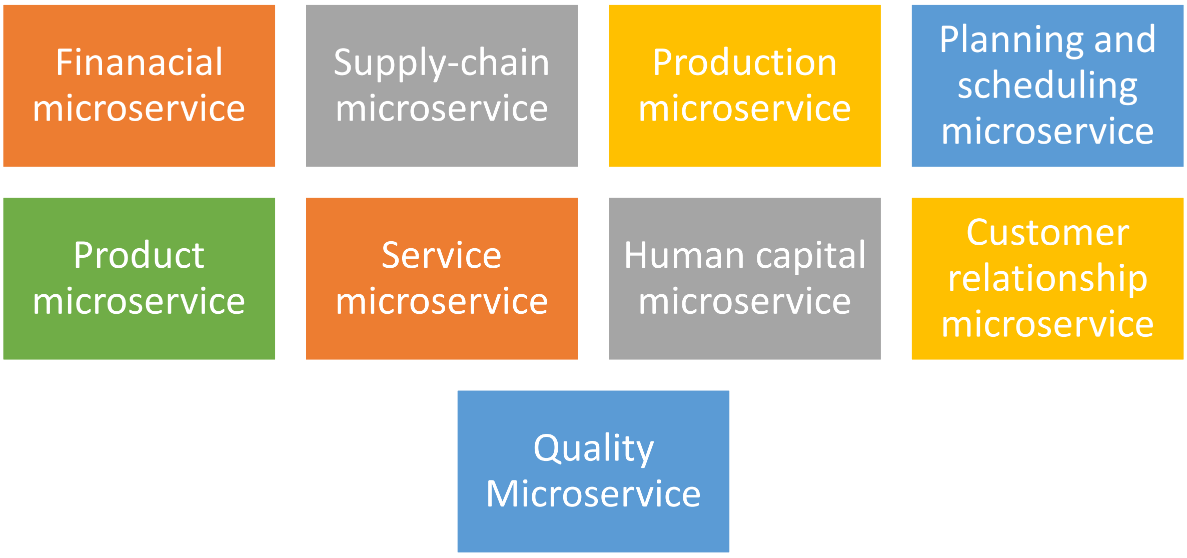

Finanacial microservice
Supply-chain microservice
Production microservice
Planning and scheduling microservice
Product microservice
Service microservice
Human capital microservice
Customer relationship microservice
Quality Microservice

**Figure 6.21** Core microservices from the ERP2 legacy system

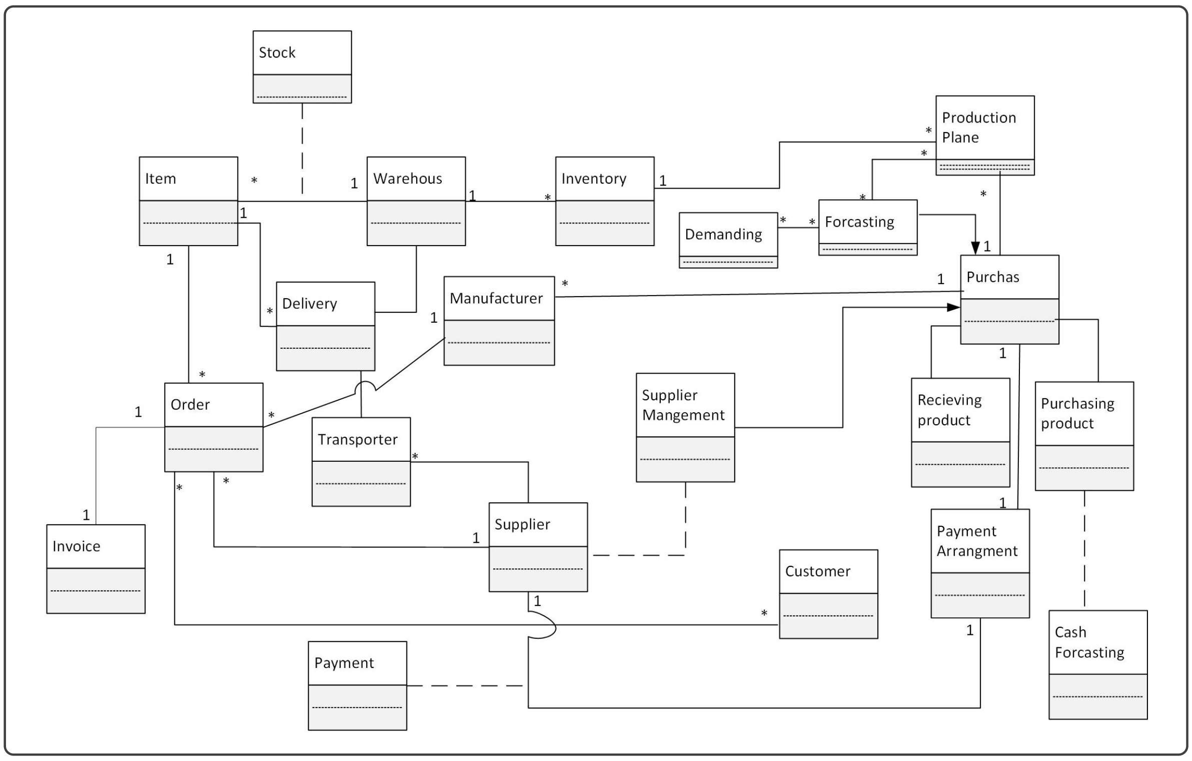


**Figure 6.22** Supply-chain domain

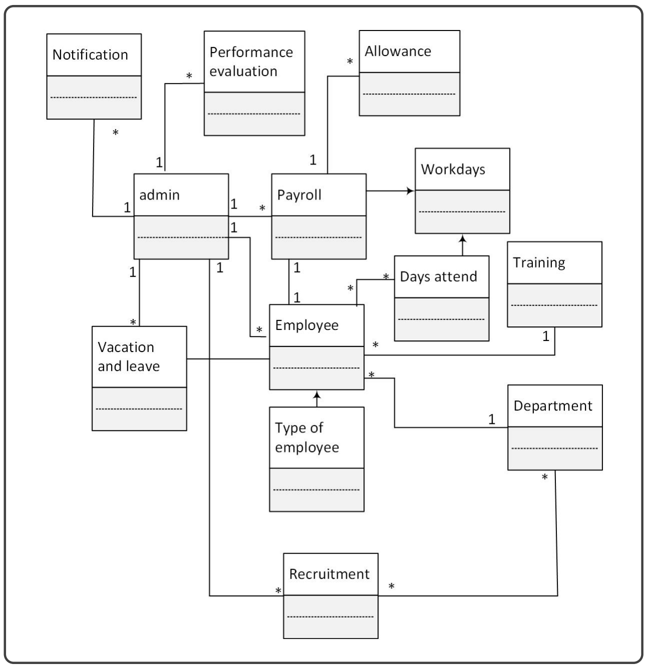


**Figure 6.23** Human resource management domain

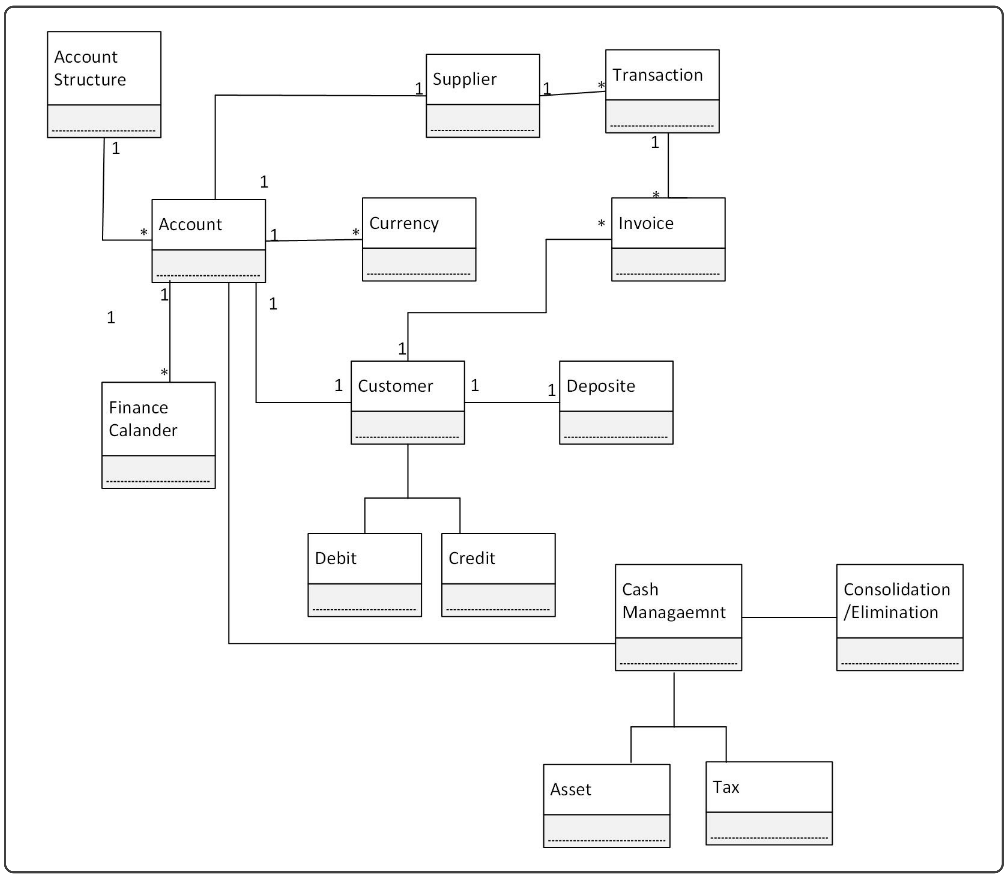


**Figure 6.24** Financial management domain

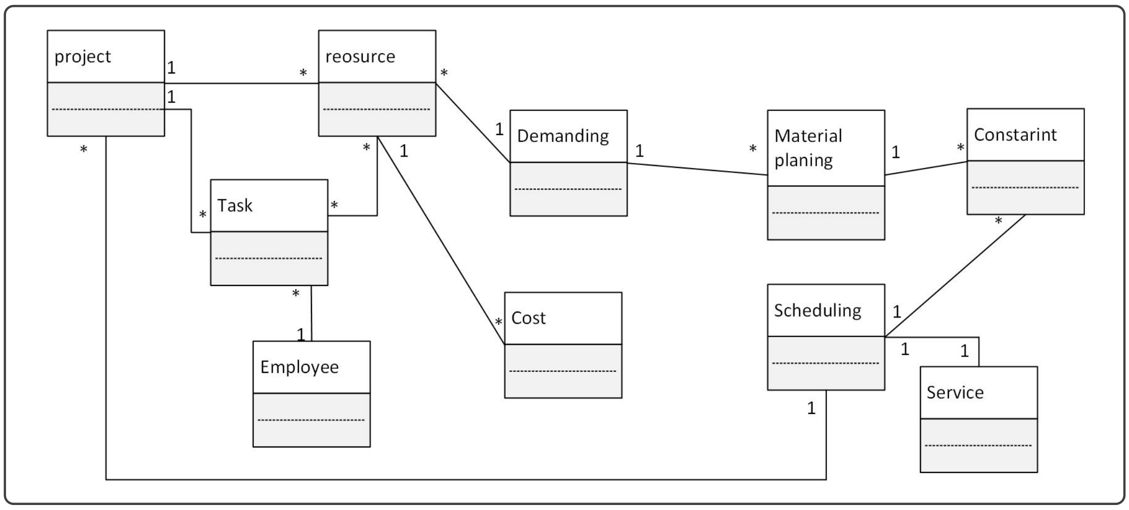


**Figure 6.25** Planning and scheduling domain

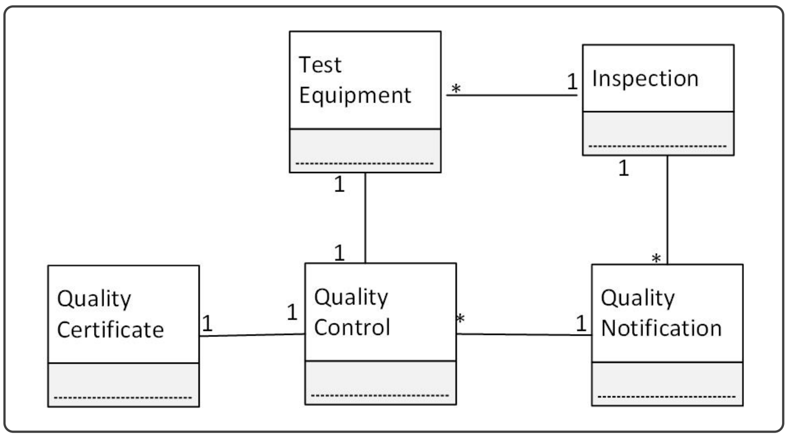


**Figure 6.26** Quality domain

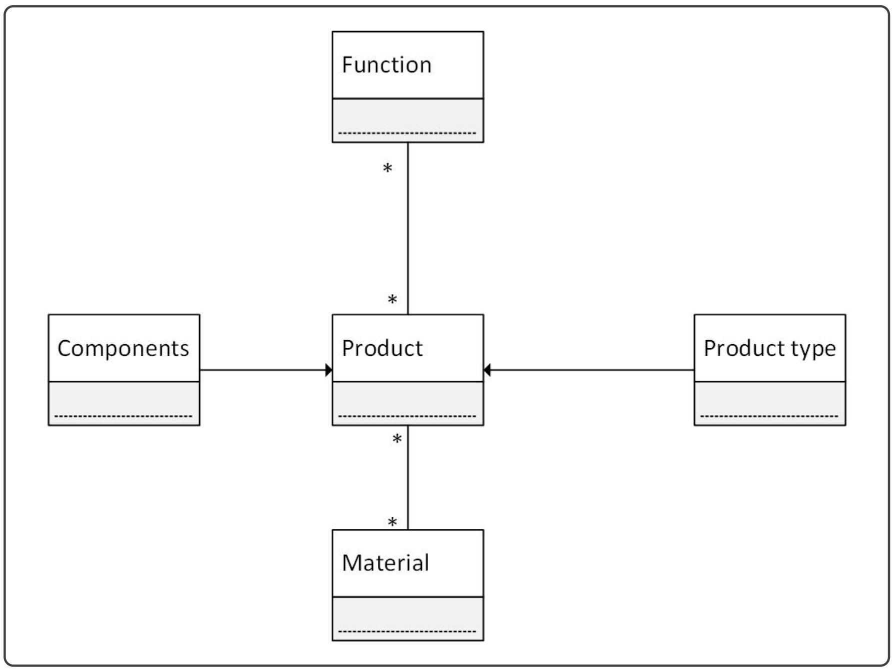


**Figure 6.27** Product domain

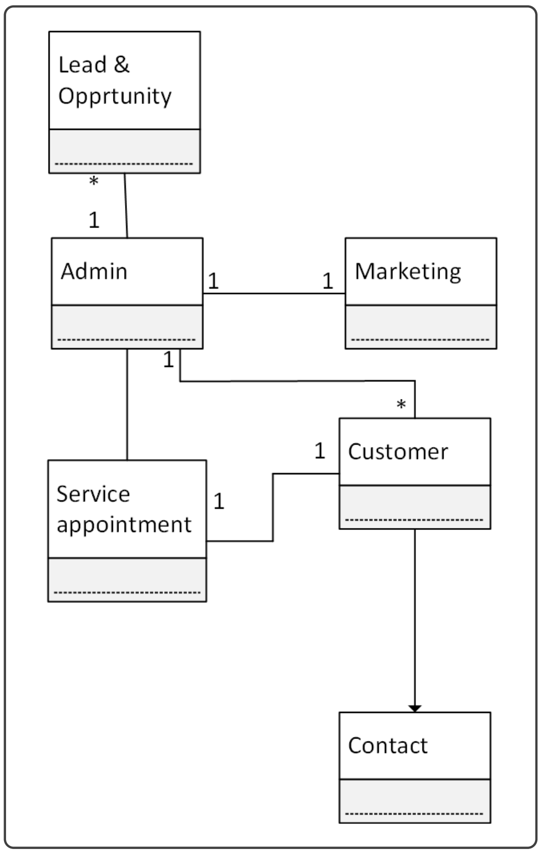


**Figure 6.28** Customer relationship domain

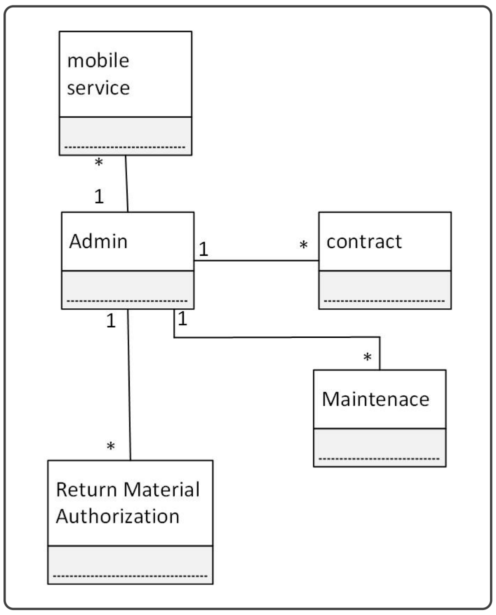


**Figure 6.29** Service domain

- *Resource optimisation (Rule 8)* In the context of this case study, definition of the resource requirements specifies the hardware needed to guarantee the system will function correctly and efficiently. As some modules may require unique resources to accomplish their intensive computational tasks, the resource requirements should be analysed with respect to the modules individually, so that the resources required for each service can be determined. Moreover, the following assumption is introduced to facilitate the analysis:

  "The monolithic system can run on a machine with 4GB memory and a 4-core CPU, the microservices may need 2-4 times those resources - which means about 8GB memory and an 8-core/10-core CPU."

  TThe first case study demonstrated that the monolithic RosarioSIS system needed 1GB memory and 1 vCPU to run, while the new microservice architecture needed 2GB memory and 2 vCPUs. This is because, even though the core executable files were identical in both architectures, the classes files and controller files were different for each microservice - which led to this increase in resources related to the running of these microservices. With thorough consideration of the above, Rule 8 is the best suited to segregate services from the modules. One way to deal with the separation of services that need unique resources is to find resource bottlenecks. Combining this information on resource bottlenecks and their parameters with knowledge concerning the flow of information through the system assists in establishing a baseline which can help developers to decide where to split the services from. It would be illustrative to take the example of the customer-supplier relationship; a vital part of the supply chain model. The customer can use the system to perform CRUD operations (Create, Read, Update, and Delete) and to search. The customer can request one or more items and allocate them for themselves, in accordance with item availability, at a selected date and time. A backend system then informs the supplier about the order which has been thus placed. The suppliers keep the system updated regarding the items purchased. Next, the order is delivered to the customer.

Considering the above scenario, there are two subdomain models:

1) The online ordering system
2) The supplier management system

It is predicted that the online ordering system, which is part of the supply chain module Figure 6.22, will have to deal with around 15,000 new customers, each with orders to be fulfilled, in the next quarter. Every one of these new customers will be responsible for a certain number of requests per second being sent to this module. Based on this, the hardware resource capacity for the online ordering system can be predicted. The supplier management system needs to manage large numbers of purchase requests based on the orders issued, which gives a clear picture of the resources needed by this component as well. Because of the correlation between the two subdomains, it was decided to locate the online ordering system and the supplier management system in a single subdomain. This prevents database dependence occurring between the two subdomains (which could occur via Rule 3, the separating out of databases). Figure 6.32 illustrates the supplier and customer subdomains.

So, in relation to Rule 8, we extracted the following subdomains from the ERP2 legacy system shown in Figure 6.21 in order to form the microservices.

- Inventory Figure 6.30
- Forecasting & demanding Figure 6.31
- Purchase Figure 6.33
- Project management Figure 6.34
- Recruitment and training services Figure 6.35

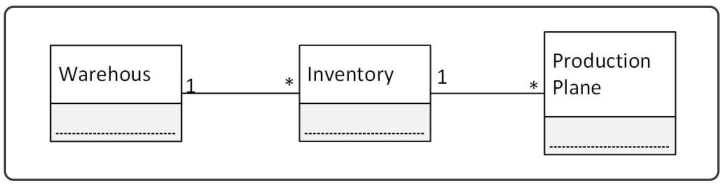


**Figure 6.30** Inventory subdomain

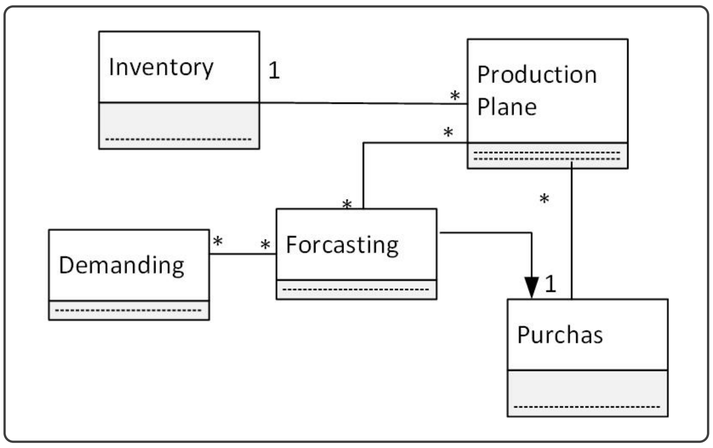


**Figure 6.31** Forecasting and Demanding subdomain

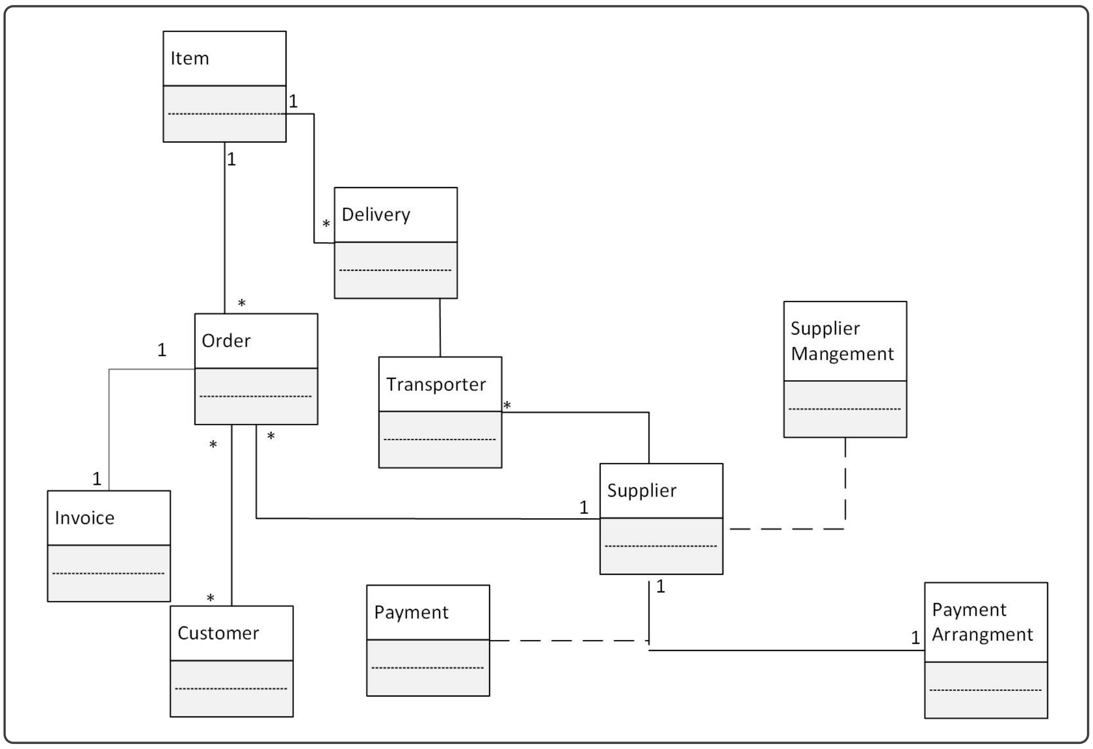


**Figure 6.32** Supplier and Customer subdomain

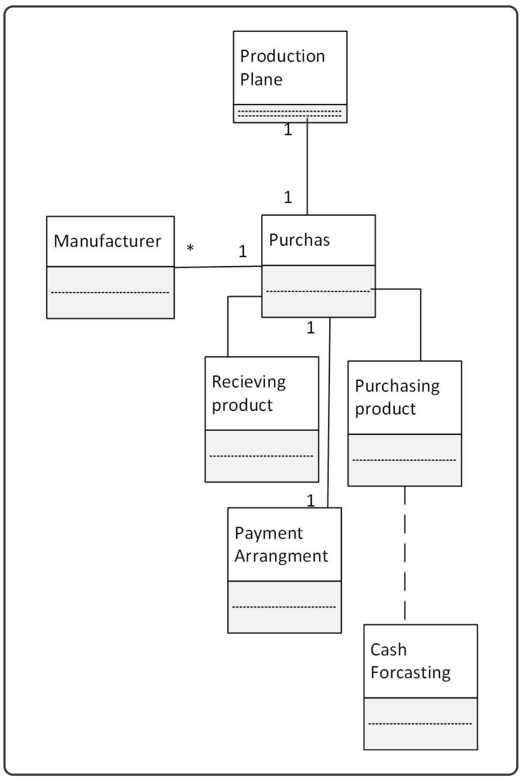


**Figure 6.33** Purchase subdomain

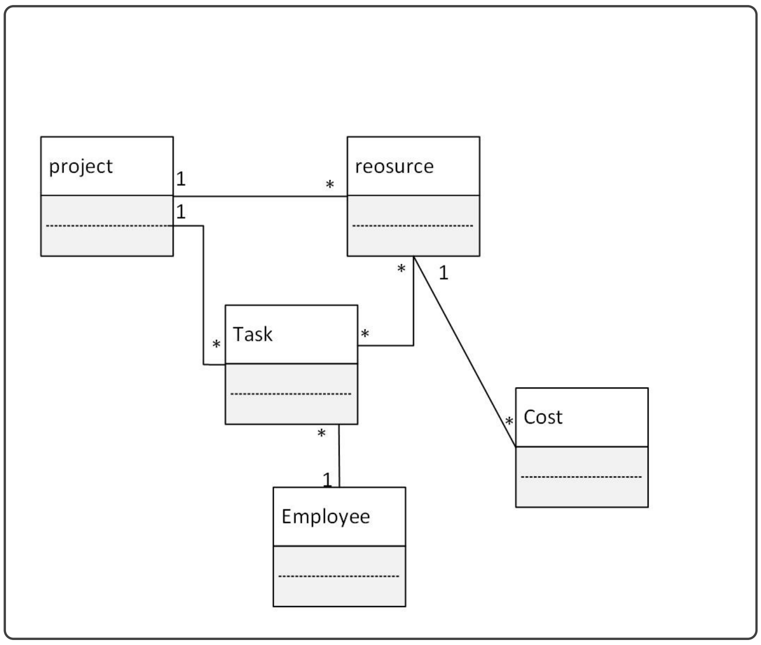


**Figure 6.34** Project subdomain

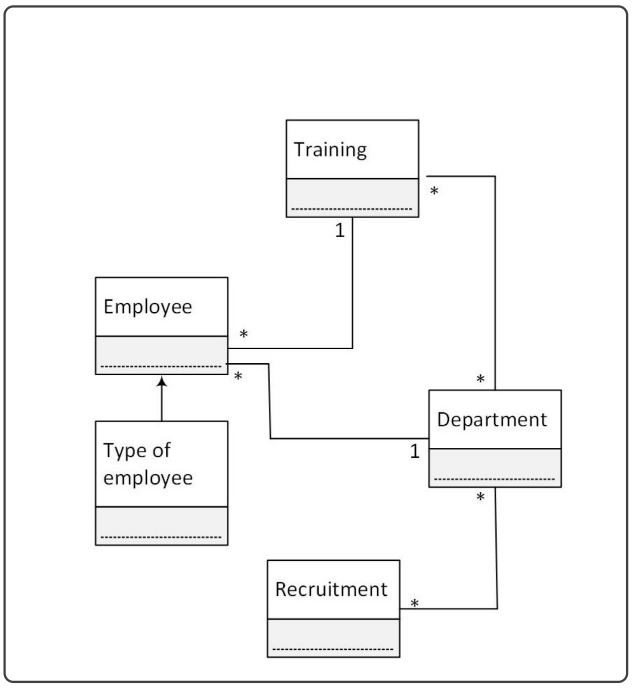


**Figure 6.35** Training and recruitment subdomain

3. *Resource-complementary service (Rule10)*

   After identifying the services which have unique resource requirements with reference to Rule 8, it was easier to determine which services can usefully be within the same instance. It has been found that combining microservices which have different resource requirements within the same instance leads to the best usage of resources and so brings the deployment costs down [56]. ]. For example, if the account payable and account receivable subsystems take 1 GB running independently, but 500 MB each when together, these two services should be run in the same instance to save resources; the option to deploy them separately will always exist.

4. *Prioritisation of data- access workloads (Rule 9).*

   The ERP2 system has many database connections and this could result in a system bottleneck when the number of connections exceed the capacity of the database. Thus, services must be prioritised based on their importance to the key business. For example, timely processing of orders from the point-of-sale to the point-of-delivery is one of the main concerns of the supply chain module. When the number of orders increases, traffic load on the supply chain module grows quickly. So, understanding the traffic patterns, in terms of the number of requests per second, provides a more complete picture about which services should be extracted as separate microservices. The candidate rules ‘prioritisation of data-access workloads’ (rule 9) and ‘customised data management’ (rule 11), are applicable in this context. However, Rule 9 is preferred to Rule 11 as the principle by which to guide the extraction of the microservices because the operating costs involved with supporting multiple databases may become relatively high. The main work relating to this rule is to track the workload of the legacy system at runtime and classify the module, of the ones under investigation, which is repeatedly subject to the heaviest workload. To address a heavy workload identified in this way, the following objectives are identified:

- The response time: how long the system takes to perform a specific request;
- The average resource utilisation: the amount of time for which devices such as the CPU and disk are occupied;
- The average throughput: the rate at which successful responses to requests are delivered.

In the supply chain module of ERP2, database access is triggered by an initial order check and the subsequent scan of the associated bar code. Thus, the average number of scans occurring in the warehouse per second must be calculated. The average time in which an order can be transmitted from the point-of-sale to the warehouse is affected by the length of the software queue in ERP2. Combining the workload information with information about the flow of data through the supply chain model creates a clear picture of the warehouse subdomain which must be extracted from the supply chain domain, as shown in Figure 6-37. Other subdomains extracted based on Rule 9 are purchase, manufacture, and payroll subdomains. The results are shown from Figure 6.36 to  Figure 6.39.

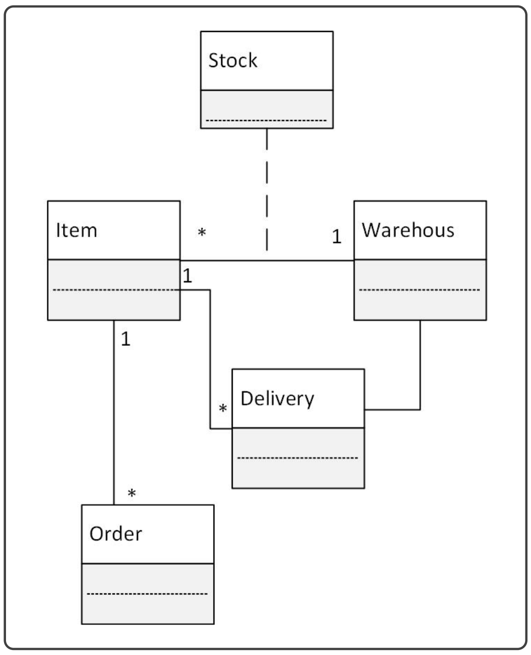


**Figure 6.36** Warehouse subdomain

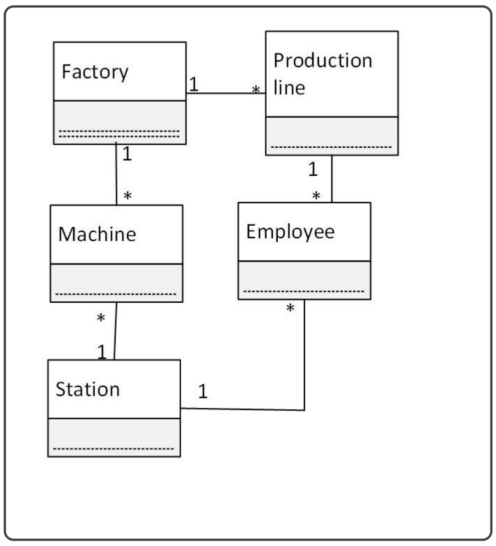


**Figure 6.37** Manufacturing subdomain

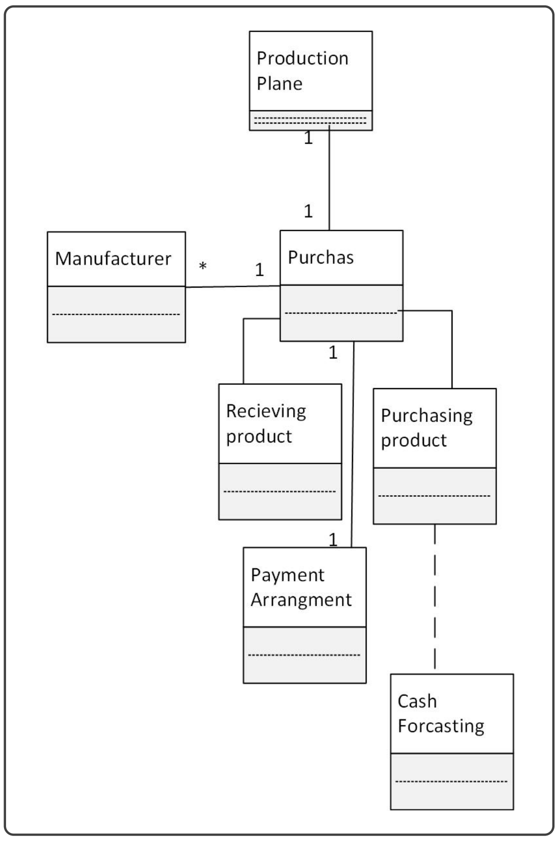


**Figure 6.38** Purchase subdomain

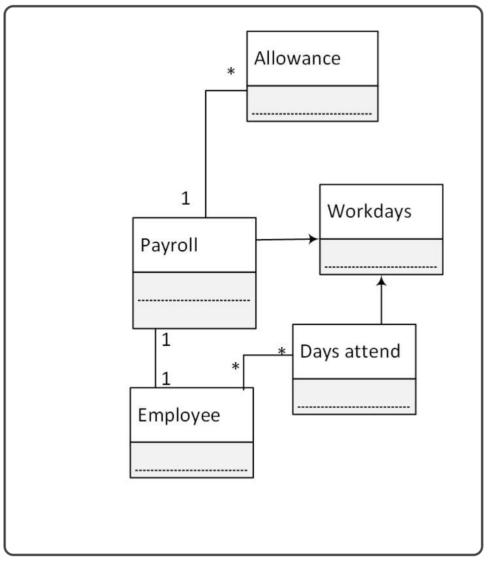


**Figure 6.39** Payroll subdomain

5. *Separation out of databases (Rule 3)*

   To support the derived microservices, the legacy database is divided into separate data stores which are managed and accessed separately based on the service that they correlate with. Having separate databases for each service helps to:

   - Improve scalability: a single database may be scaled-up based on requirements;
   - Improve availability: having separate data stores avoids a single point of failure affecting the whole system. For example, if a server fails, only the data in that server becomes unavailable;
   - Improve security: based on the nature of the data, the data store can be distributed across servers such that sensitive data will reside, for instance, in a private cloud server.
   - Optimise performance: data access operations on each partition take place over a smaller volume of data, and this leads to the minimising of latency. This is what is addressed by Rule 9, prioritisation of data access workload.

6. *Customised data management (Rule 11)*

   To choose the most suitable database structure for the necessary data management, it is important to understand the type of data to be stored and the functional and non-functional requirements via answering the questions below.

   - Will the database schema be changed frequently?
   - Does the database need to scale in the future?
   - What types of data are to be stored?
   - How many requests must be processed per second/minute?
   - How would the database handle an increasing volume of data?
   - What level of security does the database have to meet?

   In order to answer the above questions, data modelling can be used to identify the structure of the data; a crucial step. This process promotes project comprehension through the identification of key features, which must considered to avoid programming and operating errors s [91]. The data model is used to structure the data and help the developer visualise a useful picture of the nature of the data and what must therefore be included in the software. A well-constructed data model can be used to convert the domain model into a database system. Rules 2 and 3 have an impact on the decisions involved in the database modelling, each microservice must be built around a single function and have a separate database. To achieve this, each microservice's data model should be independent of all the other microservices' data models. A change to one data model should not affect another microservice's data model. The data modelling needs identified in this case study are listed in Table 6.3 below.

**Table 6.3** Requirements for Data Modelling

| ***Microservice*** | ***Read performance*** | ***Write performance*** | ***Latency*** |
|---|---|---|---|
| **Inventory** | High | High | Low |
| **Warehouse** | Moderate to high | Moderate to high | Low |
| **Purchase** | High | High | Low |
| **Forecasting** | Moderate | Moderate | Moderate low |
| **Customer & supplier** | Moderate | Moderate | Moderate to low |
| **Manufacturing** | Moderate | Moderate | Moderate to low |
| **Production** | High | High | Low |
| **Project** | High | High | Low |
| **Planning** | Moderate | Moderate | Moderate to Low |
| **Scheduling** | Moderate | Moderate | Moderate to Low |
| **HR** | Moderate | Moderate | Moderate to Low |
| **Training** | High | High | Low |
| **Payroll** | High | High | Low |
| **Product** | Moderate | Moderate | Moderate to Low |
| **Quality** | Moderate | Moderate | Moderate to Low |
| **Customer relationship** | Moderate to high | Moderate to high | Low |
| **Service** | Moderate | Moderate | Moderate to Low |
| **General lodger** | Moderate | Moderate | Moderate to Low |
| **Account receivable** | Moderate | Moderate | Moderate to Low |
| **Account payable** | Moderate | Moderate | Moderate to Low |
| **Finance** | Moderate | Moderate | Moderate to Low |

It is challenging to find a database system which offers all the features that microservices require. Therefore, a combined relational and non-relational database management system are adopted. The microservices, including inventory, purchases, and shipping and receiving, were implemented using a NoSQL database that guarantees good data accessibility and a better than average accessing speed. However, the other microservices used a SQL database.

To gain more improvements in terms of scalability and reduce the complexity implied by having each service possess its own database, the following two rules can be used to combine relevant data from different resources: 'master data access microservice' (rule 12) and 'database view' (rule 13).
Master data access microservice (Rule 12) and database view (Rule 13)

On the one hand, the master data rule (i.e., Rule 12) that governs data from several different datastores must be available from the master data source; this allows various database systems to work autonomously and then merge-update one single master database. Rule 13 allows the creation of multiple virtual tables all with different data. Both rules are, effectively, defined to help deal with data originating from different resources. In some frequently encountered circumstances, data needs to be integrated between at least two separate systems. One may be the old legacy system and the other will then be the microservices-based architecture; alternatively, both systems involved may be within the microservices-based architecture. Database view (DBV) is generated from a query . Data are stored in the physical databases persistently (in relation to the changes to the system). In the context of the new Human Resource Microservice of ERP2, DBV renders detailed views of data in response to the unique requirements/needs of users. The use of a DBV reduces data replication thus strengthening data integrity, and such a layer can manage very complicated scenarios with JOINS operations and by using multiple database technologies.

7. *Security role (Rule14)*

Securing microservices is an objective which must be achieved during implementation. Some of the information stored in a database may be highly sensitive, such as proprietary company data and employees' personal information. This rule imposes a default policy that denies any request to access any microservices. For this restriction to be lifted in relation to a user, a request must be sent to a microservice that manages permissions for such a user by maintaining a collection of database tables which represent user obligations and permits. Each process which must handle the user's authorisations will receive a unique token that expresses the corresponding permissions. Depending on the microservice access request involved, and the token associated with the user, access will be either denied or permitted. Assigning different roles to different users in terms of access control will help to protect access to the data stored on each microservice.

In this case study, the focus is on security authentication and authorisation. During the implementation stage, the security role (Rule 14) acted as a guiding principle for the building of secure microservices. Rule 14 is applied to all the design/implementation levels, from code to architecture. Also, the use of another technology, i.e. Secure Sockets Layer (SSL), was found to be essential to secure data transmissions with the Public Key Infrastructure (PKI). Ultimately, because of the separation and distribution of roles, the new architecture can achieve an acceptable level of microservice security. Figure 6.40 illustrates the microservices that were extracted from the ERP2 monolithic enterprise system based on the rules.

### 6.7.2 Deployment

This section describes the conceptual deployment of the re-architected 'ERP2' system.Approximately twenty microservices are defined in this case study, as stated in Figure 6.40 . Each one is a tiny-application with its own concerns, demands, resources and profile of varying handling loads. There are several different rules which can be used to pave deployment steps, as outlined below:

- (Rule 1) deployment in a multi-cloud environment

.Several options are available regarding the cloud facility used, and the choice of which one should be based on the nature and unique specification of the microservices; the enterprise can choose between public, private or hybrid clouds.

- (Rule 2) Deploying and managing extra loads

This rule is concerned with providing redundancy by running duplicated copies of the microservices, as the availability of the microservices is crucial and no microservices in

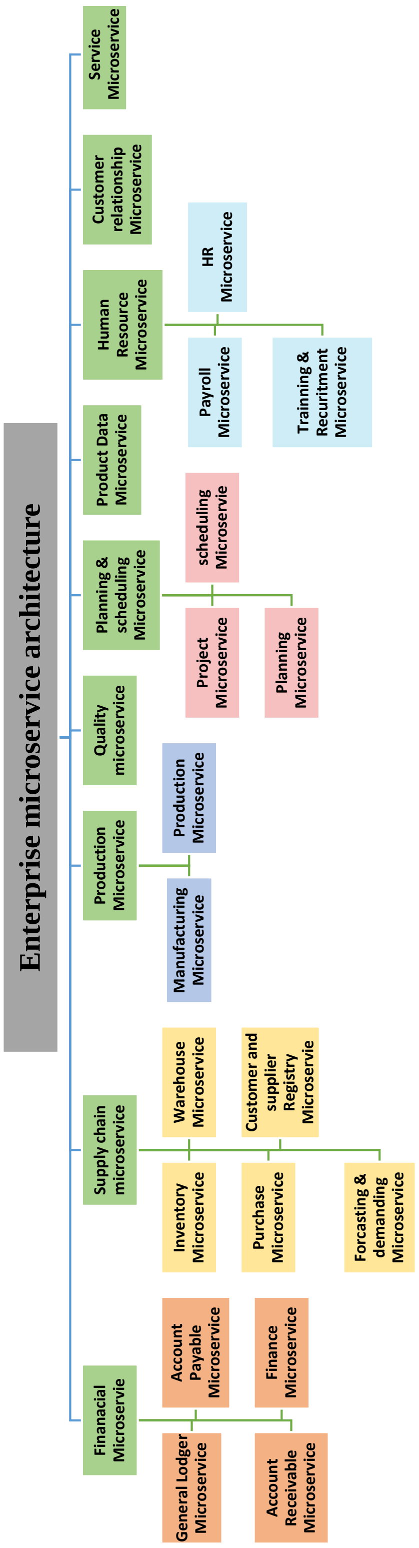

Enterprise microservice architecture
Service Microservice
Customer relationship Microservice
Human Resource Microservice
HR Microservice
Payroll Microservice
Trainning & Recuritment Microservice
Product Data Microservice
Planning & scheduling Microservice
scheduling Microservie
Project Microservice
Planning Microservice
Quality microservice
Production Microservice
Production Microservice
Manufacturing Microservice
Supply chain microservice
Warehouse Microservice
Customer and supplier Registry Microservie
Inventory Microservice
Purchase Microservice
Forcasting & demanding Microservice
Finanacial Microservie
Account Payable Microservice
Finance Microservice
General Lodger Microservice
Account Receivable Microservice

**Figure 6.40** Enterprise microservices

this case study should be critically affected by heavy loads or hardware failures. The API gateway will schedule requests between microservice copies in order to balance the load.

- (Rule 3) geolocation microservice-master database synchronisation, (Rule 4) geolocation microservice-prioritised requests, and (Rule 5) geolocation microservice-local/global master database selection. These rules are concerned with the ability to cope with growing traffic demands from geographically distributed users. This issue can be solved through deploying several different copies of microservices across several different regions to mitigate loads.

  The first option (related to rules 3, 4 and 5) is to create a master database and synchronise the data from each region using that master database. From this, the microservices database can then be updated. The second option is to create a master database and prioritise specific requests, these requests must be processed more swiftly than lower priority requests because they synchronise data with the master database. Asynchronies will be used for lower priority requests which leads to the reduction of operational costs. The third option here is to have a local master database for each geographical region and synchronise the microservices data to the local master databases, then synchronise these with a global master database. However, if a transaction needs to be processed instantly and must then be available in other regions, synchronising to the global master database is a more widely applicable option. The use of these rules facilitates the meeting of business requirements by enhancing performance and microservice availability. Moreover, their use ensures that data can remain close to the users, where required, and allows them (the users) to access their own data from any part of the world.

  – (rule 6) containerised microservices

  The microservice architecture was designed as several small, independent services. Each service must be isolated from all others and modifying one service should be a straightforward matter which does not disturb other services. In short, Rule 6 represents an appropriate solution here. The container offers an isolated environment for each service. The idea is to encapsulate each service and its assets in one package. Containers allow each service to manage its one storage and it is possible to support multiple running containers within one operating system instance [92]; which reduces overhead costs.

Table 6.4 shows a list of objectives and which feature-driven microservices rules are the best fit with each. These rules have been used to construct a pathway whereby a legacy system can be migrated to a microservice architecture. This mapping (Table 6.4) is used to narrow down the rules for selection.

**Table 6.4** Feature-Driven Migration Rules selection.

| **Objective** | **Rule** | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **Availability** | √ | | √ | √ | | | | √ | √ | √ | √ | √ | | √ | √ | √ | √ | √ | √ | |
| **Scalability** | √ | √ | √ | | √ | √ | | √ | √ | | √ | | | | | | | | √ | |
| **Reduce operational cost** | | | | | | | | | | √ | | | | √ | | | √ | | √ | |
| **Maintainability** | √ | | √ | | | | | √ | | | | | | | | | | | | |
| **Interoperability** | | | | | | | | | | | | √ | √ | | | | | | | |
| **Improve Performance** | x | x | | | √ | √ | √ | √ | √ | | √ | x | x | √ | x | x | √ | √ | | |
| **Security** | | | √ | | | | | | | | √ | | | √ | | x | x | x | | √ |

The selected rules can be used to deal with a number of separate issues. For example, understanding how a legacy system is structured, what the components are, whether data security is managed well, etc., which are important issues in the development of the system generally. This approach is not restricted to the use of only the set of rules specified above; it has greater flexibility. For instance, in both the case studies presented, the rules are combined with the DDD concept. Hence, it is possible to incorporate the use of these rules with the latest best solutions to new problems.

### 6.7.3 Evaluation

The following is a summary of the essential capabilities of a microservice infrastructure and how they may benefit the future of an enterprise.

***Scalability:***

This is one of the most significant features in relation to switching to a microservice. The ability of microservices to run in a range of computational environments such as virtual machines and containers, increases system efficiency by utilising resources more effectively. Each service is designed as a separate component. This enables each service to scale quickly in the case of increased demand.

Consider a monolithic enterprise application that has the following modules, shown in Figure 6.41, as an example:

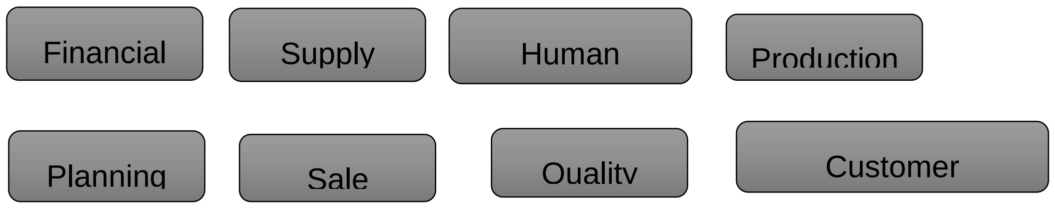


**Figure 6.41** Monolithic Enterprise application.

Each of these modules has a different function, so scaling the entire application to meet increased system needs will lead to inefficient consumption of resources. For example, a user logs in to the system, requests to search for training workshops available for their specialist area and then updates their record and logs off. In this case, the user will only need to access the human resources database, searching and updating a few records. By breaking up the monolithic system into microservices, just the services which have high computational demand will scale according to their loading, thus saving costs.

To ensure that microservices are scalable, a number of different angles should be considered. First, it is important to identify resource requirements which is addressed in Rule 8, giving a full picture of the resource requirements of the service. Second, it is crucial to identify the potential resource bottlenecks of each microservice by applying load testing. Resource bottlenecks occur as a result of high usage of the services at peak times. The main aspects of a system which must be examined in this regard include requests per second, latency, and request duration. Third, a precise picture of the nature of heavy traffic and large data-volume situations should be analysed. In summary, the developer can scale each microservice independently from the other microservices of the system and can react swiftly to changes in the workload, enabling efficient use of resources [93] [94].

***Maintainability:***

In the first case study, the RosarioSIS monolithic system emerged as complex and tightly coupled. Indeed, this represents one of the main reasons for breaking up this particular system in the present work into a number of smaller units, i.e. microservices, each of which focuses on only one business function and aims to deliver it well, so as to achieve a perfect loose coupling. Such loosely coupled components are the key to improving the maintainability of the RosarioSIS services. These smaller ('micro') systems are much easier to understand, change and test. The RosarioSIS microservices are clearly separated from each other as 'independent units', and there is no necessary sharing of information between one service and another. Loose coupling means that the RosarioSIS microservices design is more flexible and will more easily facilitate future changes, error fixes and/or new functionality [95] [96].

In addition, the second case study supports the idea that this new kind of architecture leads to systems which have been designed in such a way that they can be extended and can grow in the future (by the addition of new functionality). Also, the second case study evidenced that microservices systems are easy to change because of their reduced complexity, both in terms of architecture and components. As the degree of complexity increases, testing a system becomes more difficult. To reduces complexity, in relation to both case studies, and increase maintainability, several different methods have been applied. First, both systems have been divided up into smaller components to reduce system complexity. A sizeable monolithic system tends to be complex and challenging to modify. Second, each microservice was built to be totally independent, i.e. so that they are loosely coupled from each other by applying (1&3). Changing one service will not affect another. Loose coupling is correlated with high cohesion, representing consistency between components.

***Availability:***

It is challenging to design and deploy a microservice system in a way that means it will keep running and operating correctly as required by the user. The microservices need to be resilient and able to handle failures adequately. Also, each microservice must maintain data persistence, data consistency and resistance to data loss. As described in the previous chapter rules (3, 4&5), together offer a method for deploying microservices across multiple zones, servers and/or data centres to ensure the continued availability of the system in cases of unplanned downtime. However, in the case of a microservice failure, a circuit breaker (Rule 4) should be used. The traffic heads to the healthy microservices instead and this will give the unhealthy microservice a chance to recover, and restart to the state it was in before the failure occurred.

***Interoperability:***

Interoperability is an underlying concept, used in many enterprises; it is applied to gain the benefits of a new architecture without entirely abandoning the existing one. The term interoperability refers to the ways in which two or more systems can communicate with each other. For applications to interoperate in a convenient manner, they must be able to exchange information appropriately. The developers should be aware of any issues which could interfere with this process. For example, in the case studies, there are two types of system involved, legacy systems and the system in the process of migration, i.e. the 'microservices system'. If interoperability is a requirement, then the two systems of different types need to communicate with each other. Each system operates in a different environment to that of the other, and is built using different techniques, tools, languages, frameworks, etc. To achieve interoperability and ensure efficient working between two systems forming such a pair, creating an orchestration layer between the legacy system and microservices is essential. Adding such a layer increases the overall ability to manage the different data formats used by the two architectures. To facilitate interoperability between two systems, a common technology should be used by both architectures. For example, as here, JavaScript Object Notation (JSON) can be used for the data exchange format and Representational State Transfer (REST) over the Hyper Transfer Protocol Secure (HTTPS) for communications. Moreover, a data layer that acts as a means of communication with the legacy data should be designed. The core concepts are to separate out direct access to the legacy system and to prevent unexpected behaviour (Rules 12 &13). A master data access microservice and virtual databases can be used to deal with any data source that is not synchronised with the new architecture. These rules can be applied to perform the task of exchanging information between different sources efficiently.

***Testability:***

All the components of the RosarioSIS and ERP2 microservices architectures are separate and isolated thus testing can be scoped and also isolated. Since the RosarioSIS and ERP2 microservices are autonomous and loosely coupled, testability is much improved in relation to the legacy RosarioSIS and ERP2 systems. Moreover, regression testing for a specific microservice is much easier than it is compared to a particular functionality of the RosarioSIS or ERP2 monolithic systems. With microservices, it is possible to change part of the system and then isolate it to test it independently from the rest of the system; this represents increased isolability [95].Enhanced software testability leads to the greater efficiency of microservices. Also, it is easier to observe the state of the microservices that are being tested. Testability allows the developer to analyse the input and output operations, and this, in turn, makes it easier to detect if a microservice is working well. Furthermore, performing the required amount of testing in a focused way and using the most suitable tools ensures that the microservice system will be able to deal with the challenges it might encounter. However, increasing the number of microservices will lead to the need for greater collaboration among them and so to increased testing complexity.

## 6.8 Summary

In this chapter, a microservices-based development framework has been applied and evaluated by using the feature-driven microservices migration rules in relation to the RosarioSIS and ERP2 legacy systems. These case studies aimed to guide the migration of the legacy systems with an emphasis on analysing the implications regarding runtime performance, scalability, maintainability, availability, interoperability and testability.

The RosarioSIS case study analysed the differences in the performance test of the monolithic architecture compared to the microservices-based systems, in relation to pre-specified time periods, and observed how both of these systems behaved under heavy loads. The performance decreased as the number of requests increased, which ultimately meant that user requests were not processed within the given time constraint and using the fixed amount of vCPU and memory available.

In the ERP2 case study, a number of different feature-driven migration rules were applied, focused on different needs, to evaluate the usability of the rules and determine whether the approach is scalable and so usable for the transformation of industrial scaled monolithic systems to microservices-based equivalents while maintaining acceptable functionality, maintainability, testability, and security. Furthermore, this case study yielded more details on how each rule fits into a larger architectural picture to enrich the transformational process within a precise context. The study presented the database migration rules and where they applied, as well as exposing details of the data communications issues, including dealing with synchronised and asynchronised data. It also explored various deployment options, such as deploying in one geolocation or across several, and the impact of this kind of issue in terms of managing microservices and their performance.

As a result of these experiments, it is concluded that the microservices architecture has significant value in terms of solving the problems that may arise in relation to legacy enterprise applications. The information gained from these experiments will help in implementing and deploying microservice architectures more efficiently.

# Chapter 7

# Conclusions

## 7.1 Introduction

The research undertaken for this thesis has enabled the development of a conceptual framework which integrates a number of different technologies and methods, including a microservices architecture, a feature-driven method, and cloud computing. Bringing these together has aimed to provide developers with multi-faceted legacy system development recommendations and guidance. The research outcomes involve both conventional techniques and support from the latest theories and are backed by the latest microservices techniques.

This chapter first addresses the above research results in terms of their achievements relating to the previously defined research objectives and their compliance with the various requirements. Next, the conclusions are presented, demonstrating the contributions made. Finally, the future directions for study are briefly examined.

## 7.2 Critical Analysis

### 7.2.1 Objective I: Understand the theoretical background relating to the migrating of legacy systems, by proposing a holistic framework approach for such migration.

The first objective was to develop an approach which could effectively assist in the developing/migration of a legacy system. The objective has been accomplished by successfully delivering the following:

1. **The proposing of a set of feature-driven microservice transformation rules:** A literature review was undertaken to identify and analyse what strategies for migration to microservices have been implemented in practice, what the advantages and disadvantages of the microservice architecture are, and which non-functional requirements have been studied. From this, a research questions was derived:
   - RQ1: How is it possible to extract microservices from a legacy system? To answer this question, a review was carried out on the shifting of existing monolithic structures to microservices architectures. This was motivated by the need to resolve some of the main non-functional problems associated with such systems. In general, better performance, enhanced functionality and system protection are the main concerns guiding the migration of monolithic structures. Monolithic systems are not flexible enough to adapt to major workload increases. The solution proposed here is to deconstruct the system into a number of smaller services. Separating components out can minimise dependencies and migration can bring several advantages to the forefront such as scalability, efficiency and agility. To extract microservices from a legacy system, a set of rules was proposed through:

    * Defining the main non-functional features involved. Generally, these are efficiency, scalability of functionality, availability and security. Every attribute is then analysed, and each attribute is grouped into a specific subfunction.
    * Describing the dependencies, trade-offs, inclusiveness and specific relationships, which depend on the influence of each NFR. This resulted in a hierarchy of NFRs. A collection of relationships between the NFRs were used to construct the feature-driven rules. The rules were constructed by analysing a number of situations relating to the process of transformation and taking into account the interactions between non-functional features. The goal of the framework is to use the feature-driven concept to steer the transformation process (from monolith to microservices) via the proposal of a microservices-based transformation rule repository.

  – **proposing a set of feature-driven cloud-oriented migration rules:**
    To gain more benefit from loosely coupled microservice architectures, Cloud migration rules are proposed. These migration rules are mainly focused on how deployment in the cloud can be employed to increase microservices performance in terms of handling workload, individualised use of resources, minimisation of the latency of requests, increased availability, and increased agility.
    The rules consist of three parts: a precondition, a transformation and the expected impact on features. The precondition element defines the cases where the rule can be applied to the microservice; the transformation indicates the procedure to be followed; and the impact on features reveals the issues that may emerge once the rule is applied.

### 7.2.2 Objective II: Apply UML diagram in order to specify the migration rules

During the migration process, there needs to be an explicit representation of the system components and the flow of data. Using a UML diagram allows the structure of the system to be analysed after application of the rules, with respect to the bounded context concept, and thus define the exact domain of each microservice.

### 7.2.3 Objective IV: Evaluation with Case Studies

For the purposes of critical evaluation, the present research uses two case studies and some associated experiments using a cloud service. First, comprehensive research, searching for an open-source enterprise system appropriate to this study, was undertaken. The target size was small to medium because of the time constraints involved with the implementation stage. Secondly, for the cloud service and evaluation elements of the study, Amazon Web Services (Amazon ECS) was selected; this was specifically chosen to examine the microservices architecture performance versus the legacy system performance(refer to section 6.6). Finally, the ways in which microservices capabilities affect non-functional attributes were examined (refer to subsection 6.7.3).

## 7.3 Contributions

Migrating to a microservices architecture includes many processes that need to be managed carefully. Microservices represent a relatively new architectural style and, as a result, there is no general migration guide for microservices. This work provides a method for shifting a monolithic architecture to microservices. A set of rules is employed to define and implement the migration process.
The contribution consists, in part, of constructing a basis for the effective refactoring of monolithic applications towards microservices style applications hosted in cloud environments. The following work has been undertaken throughout the course of this study.

– **The thesis presents a novel approach to the migration of a legacy system towards cloud computing through the construction of a microservices system:**
  This novel approach determines how effective the application of the microservices architecture is in terms of three requirements: performance, functionality

and security. The approach used a conceptual framework that consists of three layers:

  * ∗ Layer one consists of understanding and analysing the legacy system and its context. To obtain a deeper understanding the notation diagram was used to visualise the system components and their relationships with each other.
  * ∗ Layer two focuses on how to decouple the functionalities of the legacy system to be more flexible in the face of changing requirements. Based on the transformation rules, the services are extracted from the legacy system and built as self-contained services.
  * ∗ Layer three is concerned with defining the desired architecture, describing the operations that should be implemented and ensuring the quality of the systems. In general, the architectures of the systems are built based on a microservices architecture, and the rules guide what can be done to accommodate the demands of this new architecture.

– **The construction of a set of feature-driven microservice transformation rules:**
  A set of rules are presented as principles of the microservices-based architectural style. These rules were determined by identifying the most important quality attributes of such an architecture. Then, the relationships between the attributes were defined and classified. After this, the brainstorming of various scenarios and the ways in which the quality attributes can be managed in relation to these scenarios took place. Finally, each rule was designed in such a way that it would be the most beneficial in extracting microservices from legacy systems (refer to section 5.4).

– **The construction of a set of feature driven cloud migration rules:**
  A set of guidelines which enables the exploration of various scenarios was developed, followed by the determination of which one was the most applicable based on the requirements (refer to section 5.5).
  Each rule consists of three parts: a precondition, which is the main component, a transformation, which specifies the major changes to the legacy system

dependent on the precondition, and the impact on features, which delineates the consequence of these changes.

– **The conducting of the case studies and evaluations:**
  Two case studies were undertaken to test the suggested framework in terms of proof-of-concept, validation and evaluation. These case studies involved differently scaled enterprise systems: small and medium. The experimental findings provide a detailed description of how to follow the rules in order to switch from a legacy to a microservices system effectively, and then deploy the latter in the Cloud. Finally, an analysis of the success of the new architecture is presented via the evaluation of several different quality attributes.

## 7.4 Conclusion

This project aimed to develop an approach to evolve legacy systems, through designing a framework that incorporates a microservice-oriented architecture. This framework architecture determines how effective is the new architecture in terms of functionality, performance and security. The framework focuses on the repository of the microservice and cloud rules and how these rules support the three features. The framework was evaluated by applying feature-driven evolution rules, which include transformation and migration rules, to two different case studies; the 'RosarioSIS' and 'ERP2' legacy systems. The first case study analysed the differences in the legacy system performance compared to the microservice-based architecture over pre-specified time periods and observed how these systems operate under heavy load. The second case study analysed other qualitative attributes such as scalability, maintainability, interoperability, testability and availability in relation to the feature driven of the transformation and migration rules . The experiments suggested that the microservices design has a major benefit in solving problems that can occur in legacy enterprise applications.

## 7.5 Limitation and Future Work

The challenges encountered in this project included finding a suitable open source enterprise system. This was difficult as some such systems are incomplete, do not have source code for some of their modules, and/or do not have adequate documentation explaining how the system works and runs. Also, during the migration process, defining the responsibility for each microservice is sometimes a challenging task due to the dependencies between components and how the definition of the responsibilities reflects the communication between microservices. In addition, sometimes it is necessary to model services based on the business services that are provided by the enterprise. Moreover, some types of legacy system are not supported by this rule because the legacy system is a shared business, meaning the microservices will depend on each other in complex ways which will make them more difficult to manage.

Regarding research focused on the study of legacy system development, future work could concentrate on extending the proposed rules to include what happens when the number of microservices increases (e.g. perhaps to thousands or millions); will that affect performance and in what ways? Also, the addition of more non-functional features to indirectly support a more complete application functionality. Applying an automation tools to the detaching of the legacy system and to other elements of the migration process may speed this up, and so this is worthy of investigation. Also, monitoring microservices would be a means to provide knowledge sources for services searches and recommendation tasks. Furthermore, only small and medium sized enterprise systems were studied in this work. A large enterprise system could be considered in future work, to further evaluate the framework.

# References

[1] E. B. Swanson, "The dimensions of maintenance," in *InProceedings of the 2nd international conference on Software engineering*, pp. 492–497, 1976.

[2] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," pp. 13–22, 2005.

[3] D. Stéphane, T. Gîrba, M. Lanza, and S. Demeyer Moose, "A collaborative and extensible reegineering environment," *Tools for Software Maintenance and Reengineering (RCOST)*, 2005.

[4] M. W. Godfrey and D. M. German, "On the evolution of lehman's laws," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 613–619, 2014.

[5] I. Sommerville and S. Engineering, *New York.* NY: Pearson Education, 2015.

[6] K. H. Bennett, V. T. Rajlich, and N. Wilde, "Software evolution and the staged model of the software lifecycle," in *Advances in Computers (Vol*, pp. 1–54, Elsevier: 56, 2002.

[7] T. Girba, S. Ducasse, and M. Lanza, "Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 40–49, IEEE, 2004.

[8] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," in *2009 IEEE International Conference on Software Maintenance*, pp. 51–60, IEEE, 2009.

[9] K. Bennett, "Legacy systems: Coping with success," *IEEE software*, vol. 12, no. 1, pp. 19–23, 1995.

[10] N. H. Weiderman, J. K. Bergey, D. B. Smith, and S. R. Tilley, "Approaches to legacy system evolution.," tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1997.

[11] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing legacy systems: software technologies, engineering processes, and business practices.* Addison-Wesley Professional, 2003.

[12] J. Bisbal, D. Lawless, B. Wu, and J. Grimson, "Legacy information systems: Issues and directions," *IEEE software*, vol. 16, no. 5, pp. 103–111, 1999.

[13] V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch, “How to adapt applications for the cloud environment,” *Computing*, vol. 95, no. 6, pp. 493–535, 2013.

[14] J. Footen *’Service Oriented Architecture  Cloud Computing in Media Industry’*, vol. 2011, pp. 1–23, 2011.

[15] Z. Xiao, I. Wijegunaratne, and X. Qiang, “Reflections on soa and microservices,” in *2016 4th International Conference on Enterprise Systems (ES)*, pp. 60–67, IEEE, 2016.

[16] R. Welke, R. Hirschheim, and A. Schwarz, “Service-oriented architecture maturity,” *Computer*, vol. 44, no. 2, pp. 61–67, 2011.

[17] S. Newman, *Monolith To Mircoservices*. Inc: O’Reilly Media, 2019.

[18] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, and R. Metz, “Reference model for service oriented architecture 1.0–oasis standard,” *Saatavilla oasis-open*, vol. 1, 2006.

[19] T. Cerny, M. J. Donahoo, and J. Pechanec, “Disambiguation and comparison of soa, microservices and self-contained systems,” in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, pp. 228–235, IEEE, 2017.

[20] D. Namiot and M. Sneps-Sneppe, “On micro-services architecture,” *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[21] J. Thönes, “Microservices,” *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[22] O. Al-Debagy and P. Martinek, “A comparative review of microservices and monolithic architectures,” in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI) (pp*, pp. 149–00015, 0004). IEEE, 2018.

[23] C. Richardson and F. Smith, “Microservices: from design to deployment,” *Nginx Inc, pp*, pp. 24–31, 2016.

[24] A. A. Al-Rashedi, “E-government based on cloud computing and service-oriented architecture,” *International Journal of Computer and Electrical Engineering*, vol. 6, no. 3, p. 201, 2014.

[25] N. Kratzke and P. C. Quint, “Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study,” *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[26] D. Gonzalez, *Developing Microservices with Node. js*. Packt Publishing Ltd, 2016.

[27] Y. Yu, H. Silveira, and M. Sundaram, “A microservice based reference architecture model in the context of enterprise architecture,” *In*, vol. 2016, pp. 1856–1860, 2016.

[28] K. Indrasiri and P. Siriwardena, "Microservices for the enterprise," *Apress, Berkeley*, 2018.

[29] H. H. S. da Silva, G. D. F. Carneiro, and M. P. Monteiro, "An experience report from the migration of legacy software systems to microservice based architecture," in *16th International Conference on Information Technology-New Generations (ITNG 2019) . , Cham*, pp. 183–189, 2019.

[30] Q. Hu and Y. Y. Du, "Service architecture and service discovery oriented to service clusters," *Journal of Computer Applications*, vol. 33, no. 8, pp. 2163–2166, 2013.

[31] M. Fowler and J. Lewis, "Microservices a definition of this new architectural term," *URL: http://martinfowler.com/articles/microservices.html*, vol. 22, 2014. http://martinfowler.

[32] M. Bruce and P. Pereira, *Microservices In Action*. Shelter Island, NY: Manning Publications Co, 2018.

[33] R. Jardim-Goncalves, A. Grilo, and A. Steiger-Garcao, "Challenging the interoperability between computers in industry with mda and soa," *Computers in industry*, vol. 57, no. 8-9, pp. 679–689, 2006.

[34] V. Vernon, *Implementing domain-driven design*. Addison-Wesley, 2013.

[35] M. Fowler, "Boundedcontext," vol. 1, p. 2017, 2014. https://www.martinfowler.com/bliki/BoundedContext.html.

[36] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and ulterior software engineering . , Cham*, pp. 195–216, 2017.

[37] C. Pahl and P. Jamshidi *Microservices: A Systematic Mapping Study*, vol. 1, pp. 137–146, 2016.

[38] O. Zimmermann, *Mircroservices tenets: Agile approach to service development and deployment*. In Proceedings of the Symposium/Summer School on Service-Oriented Computing, 2016.

[39] J. Hunt, "Feature-driven development," *Agile Software Construction, pp*, pp. 161–182, 2006.

[40] S. Thakur and H. Singh, "Fdrd: Feature driven reuse development process model," in *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*, pp. 1593–1598, IEEE, 2014.

[41] B. Hayes, *Cloud computing*. NY, USA: ACM New York, 2008.

[42] J. Lenhard, *Portability of Process-Aware and Service-Oriented Software: Evidence and Metrics (Vol. 23)*. University of Bamberg Press, 2016.

[43] M. B. Mollah, K. R. Islam, and S. S. Islam, "Next generation of computing through cloud computing technology," in *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 1–6, IEEE, 2012.

[44] B. Furht and A. Escalante, *Handbook of cloud computing (Vol. 3).* New York: Springe, 2010.

[45] M. Petrenko, D. Poshyvanyk, V. Rajlich, and J. Buchta, "Teaching software evolution in open source," *Computer*, vol. 40, no. 11, pp. 25–31, 2007.

[46] K. Boeckman, *Docker containers vs. virtual machines: What's the difference?* NetApp Blog, 2017.

[47] B. Golden, "3 reasons why you should always run microservices apps in containers," *TechBeacon*, vol. 3, February 2018. https://techbeacon.com/app-dev-testing/3-reasons-why-you-should-always-run-microservices-apps-containers.

[48] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, pp. 27–34, IEEE, 2015.

[49] Docker, "About docker engine," Accessed 2020 January. https://docs.docker.com/engine/.

[50] M. Kircher and P. Jain, *Pattern-oriented software architecture, patterns for resource management*, vol. 3. John Wiley & Sons, 2013.

[51] TutorialPoints, "Design pattern overview," November. https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm.

[52] N. Noda and T. Kishi, "Design pattern concerns for software evolution," in *Proceedings of the 4th international workshop on principles of software evolution*, pp. 158–161, 2001.

[53] C. Kramer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," in *Proceedings of WCRE'96: 4rd Working Conference on Reverse Engineering*, pp. 208–215, IEEE, 1996.

[54] N. Nadiu *Design Patterns For Legacy Migration And Digital Modernization*, vol. 3, p. 2020, March 2016.

[55] M. Kalske, N. M"akitalo, and T. Mikkonen, "Challenges when moving from monolith to microservice architecture," in *International Conference on Web Engineering . , Cham*, pp. 32–47, 2017.

[56] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *2016 IEEE international symposium on workload characterization (IISWC)*, pp. 1–10, IEEE, 2016.

[57] V. Muppavarapu and S. M. Chung, "Role-based access control for cyber-physical systems using shibboleth," in *Proceedings of DHS Workshop on Future Directions in Cyber-Physical Systems Security*, pp. 57–60, Citeseer, 2009.

[58] P. Di Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 21–30, IEEE, 2017.

[59] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 44–51, IEEE, 2016.

[60] M. S. Hamzehloui, S. Sahibuddin, and K. Salah, "A systematic mapping study on microservices," in *International Conference of Reliable Information and Communication Technology . , Cham*, pp. 1079–1090, 2018.

[61] D. Taibi, V. Lenarduzzi, and C. Pahl, *Architectural patterns for microservices: a systematic mapping study*. SCITEPRESS, 2018.

[62] J. Soldani, D. A. Tamburri, and W. J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.

[63] D. Neri, J. Soldani, O. Zimmermann, and A. Brogi, "Design principles, architectural smells and refactorings for microservices: a multivocal review," *SICS Software-Intensive Cyber-Physical Systems, pp*, pp. 1–13, 2019.

[64] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," *In*, vol. 2015, no. 10, pp. 583–590, 2015.

[65] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns," *Software: Practice and Experience*, vol. 48, no. 11, pp. 2019–2042, 2018.

[66] K. Brown and B. Woolf, "Implementation patterns for microservices architectures," in *Proceedings of the 23rd Conference on Pattern Languages of Programs*, pp. 1–35, 2016.

[67] H. Knoche and W. Hasselbring, "Drivers and barriers for microservice adoption-a survey among professionals in germany," *Enterprise Modelling and Information Systems Architectures (EMISAJ)-International Journal of Conceptual Modeling*, vol. 14, no. 1, pp. 1–35, 2019.

[68] I. J. Munezero, D. T. Mukasa, B. Kanagwa, and J. Balikuddembe, "Partitioning microservices: a domain engineering approach," in *2018 IEEE/ACM Symposium on Software Engineering in Africa (SEiA)*, pp. 43–49, IEEE, 2018.

[69] D. Taibi and V. Lenarduzzi, “On the definition of microservice bad smells,” *IEEE software*, vol. 35, no. 3, pp. 56–62, 2018.

[70] J.-P. Gouigoux and D. Tamzalit, “From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture,” in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 62–65, IEEE, 2017.

[71] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, “Microservices in industry: insights into technologies, characteristics, and software quality,” in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pp. 187–195, IEEE, 2019.

[72] A. A. C. De Alwis, A. Barros, A. Polyvyanyy, and C. Fidge in *Function-splitting heuristics for discovery of microservices in enterprise systems. In International Conference on Service-Oriented Computing . , Cham*, pp. 37–53, 2018.

[73] H. A. M"uller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, “A reverse-engineering approach to subsystem structure identification,” *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993.

[74] C. Esposito, A. Castiglione, and K. K. R. Choo, “Challenges in delivering software in the cloud as microservices,” *IEEE Cloud Computing*, vol. 3, no. 5, pp. 10–14, 2016.

[75] S. R. Palmer and M. Felsing, *A practical guide to feature-driven development*. Pearson Education, 2001.

[76] F. Anwer, S. Aftab, U. Waheed, and S. S. Muhammad, “Agile software development models tdd, fdd, dsdm, and crystal methods: A survey,” *International journal of multidisciplinary sciences and engineering*, vol. 8, no. 2, pp. 1–10, 2017.

[77] C. Richardson, *Microservice architecture*. 2016. http://microservices.io.

[78] P. R. Chelliah, S. Naithani, and S. Singh, *Practical Site Reliability Engineering: Automate the process of designing, developing, and delivering highly reliable apps and services with SRE*. Packt Publishing Ltd, 2018.

[79] J. H. Saltzer and M. F. Kaashoek, *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.

[80] G. Gousios, “Distributed databases. [online] gousios.org,” *Available at: https://gousios.org/courses/bigdata/dist-databases.html*, Accessed 2019 December.

[81] “Swagger,” *API Design For Swagger And OpenAPI*, vol. 10, p. 2018, March. https://swagger.io/solutions/api-design/.

[82] ThePhp *Framework For Web Artisans*, vol. 10, p. 2018, April. https://laravel.com/docs/8.x.

[83] "Francoisjacquet," *Francoisjacquet/Rosariosis.*, vol. 25, p. 2019, April. https://github.com/francoisjacquet/rosariosis/blob/mobile/INSTALL.md.

[84] "Laravel passport," vol. 10, p. 2018, June. https://laravel.com/docs/7.x/passportissuing-access-tokens.

[85] *Amazon Ecs- Run Containerized Applications In Production*, vol. 11, p. 2019, February. https://aws.amazon.com/ecs/.

[86] ApacheJmeter *(Application Designed To Test Functional Behavior And Measure Performance)*, vol. 1, p. 2018, November. http://jmeter.

[87] "Codecanyon," *Full ERP*, vol. 1, p. 2019, November. https://codecanyon.net/item/erp-business-solution-c-project-with-source-code/22449253.

[88] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, and R. Casallas, "Towards the understanding and evolution of monolithic applications as microservices," *In*, vol. 2016, pp. 1–11, 2016.

[89] U. R. Sharma, *Practical Microservices.* Packt Publishing Ltd, 2017.

[90] E. Eric and F. Martin, "Domain-driven design: Tackling complexity in the heart of software," 2013.

[91] R. Elmasri and S. Navathe, *Fundamentals of database systems*, vol. 7. Pearson, 2017.

[92] P. Raj, A. Raman, and H. Subramanian, *Architectural Patterns.* Packt Publishing, 2017.

[93] S. J. Fowler, *Production-ready Microservices: Building standardized systems across an engineering organization. " O'Reilly Media.* Inc.", 2016.

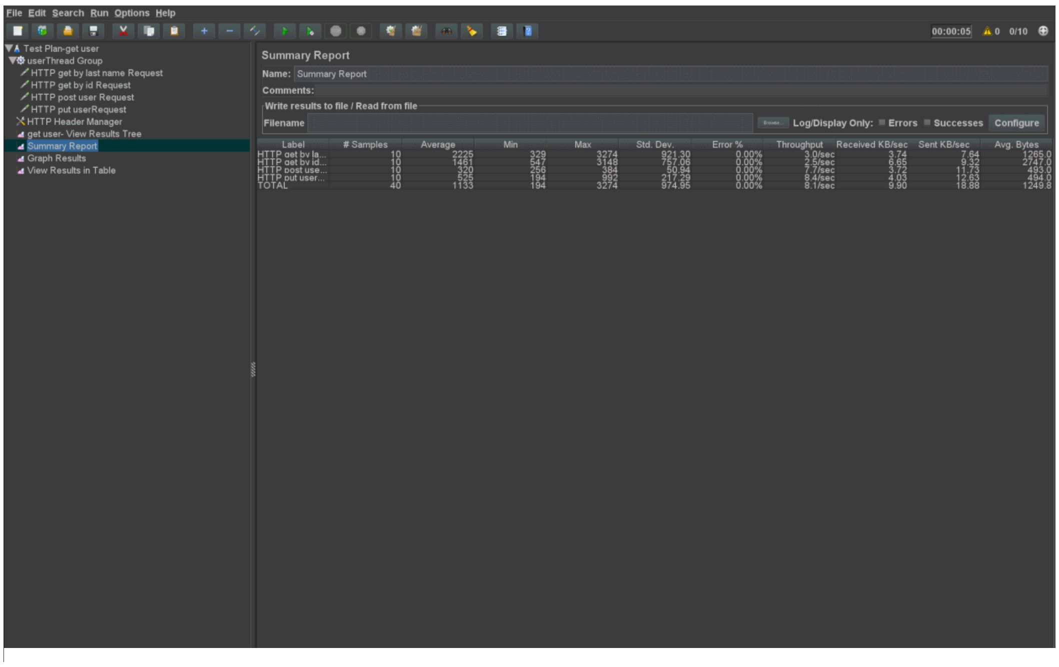
[94] M. L. Abbott and M. T. Fisher, *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise.* Pearson Education, 2009.

[95] N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "Microservices: Migration of a mission critical system," *arXiv preprint arXiv:1704.04173*, 2017.

[96] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice.* Addison-Wesley Professional, 2003.

# Appendix A

# Performance Test samples



Summary Report

Name: Summary Report

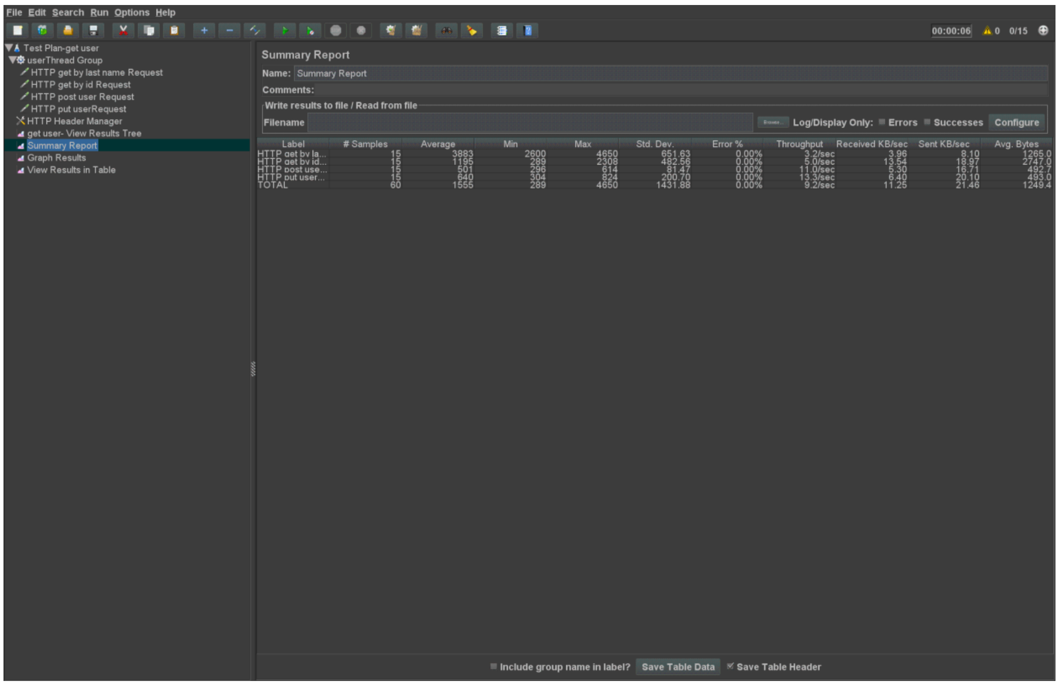
Comments:

Write results to file / Read from file

Filename

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/sec | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| HTTP get by la... | 10 | 2225 | 329 | 3274 | 921.30 | 0.00% | 3.0/sec | 3.74 | 7.64 | 1265.0 |
| HTTP get by id... | 10 | 1461 | 547 | 3148 | 757.06 | 0.00% | 2.5/sec | 6.65 | 9.32 | 2747.0 |
| HTTP post use... | 10 | 320 | 256 | 384 | 50.94 | 0.00% | 7.7/sec | 3.72 | 11.73 | 493.0 |
| HTTP put user... | 10 | 525 | 194 | 992 | 217.29 | 0.00% | 8.4/sec | 4.03 | 12.63 | 494.0 |
| TOTAL | 40 | 1133 | 194 | 3274 | 974.95 | 0.00% | 8.1/sec | 9.90 | 18.88 | 1249.8 |

**Figure A.1** User Module Summary Report for 40 requests in 0.5sec



Summary Report

Name: Summary Report

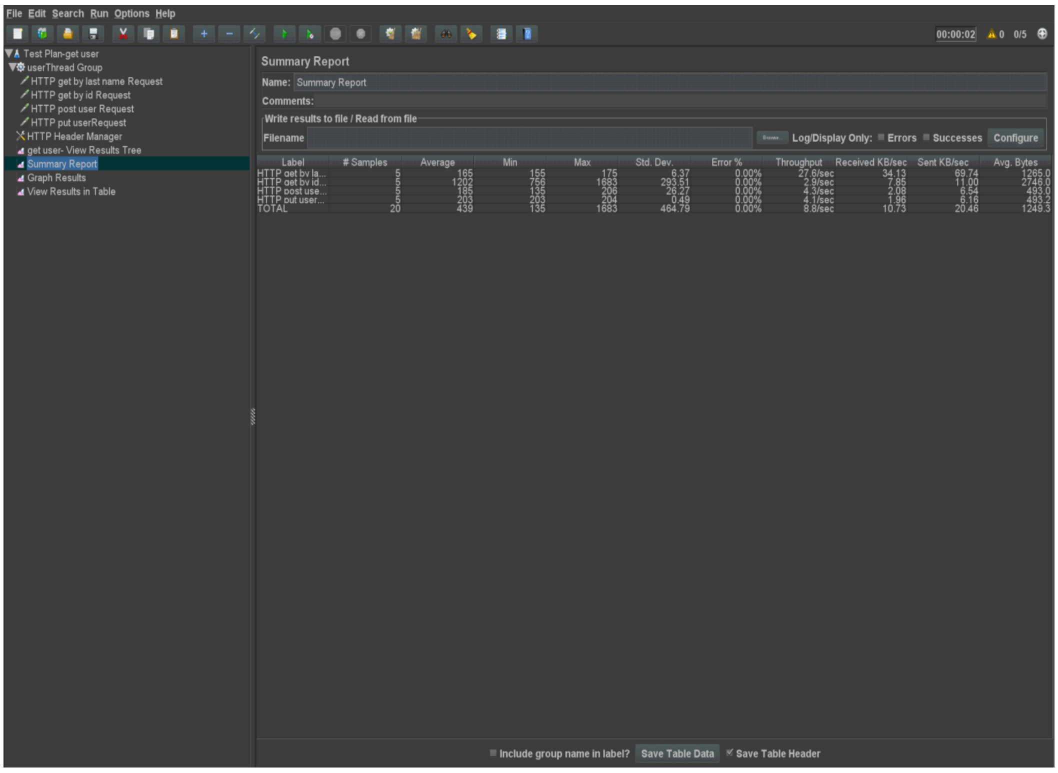
Comments:

Write results to file / Read from file

Filename | Log/Display Only: Errors Successes | Configure

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/sec | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| HTTP get by la... | 15 | 3883 | 2600 | 4650 | 651.63 | 0.00% | 3.2/sec | 3.96 | 8.10 | 1265.0 |
| HTTP get by id... | 15 | 1195 | 289 | 2308 | 482.56 | 0.00% | 5.0/sec | 13.54 | 18.97 | 2747.0 |
| HTTP post use... | 15 | 501 | 296 | 614 | 81.47 | 0.00% | 11.0/sec | 5.30 | 16.71 | 492.7 |
| HTTP put user... | 15 | 640 | 304 | 824 | 200.70 | 0.00% | 13.3/sec | 6.40 | 20.10 | 493.0 |
| TOTAL | 60 | 1555 | 289 | 4650 | 1431.88 | 0.00% | 9.2/sec | 11.25 | 21.46 | 1249.4 |

**Figure A.2** User Module Summary Report for 20 requests in 0.5sec



Summary Report

Name: Summary Report

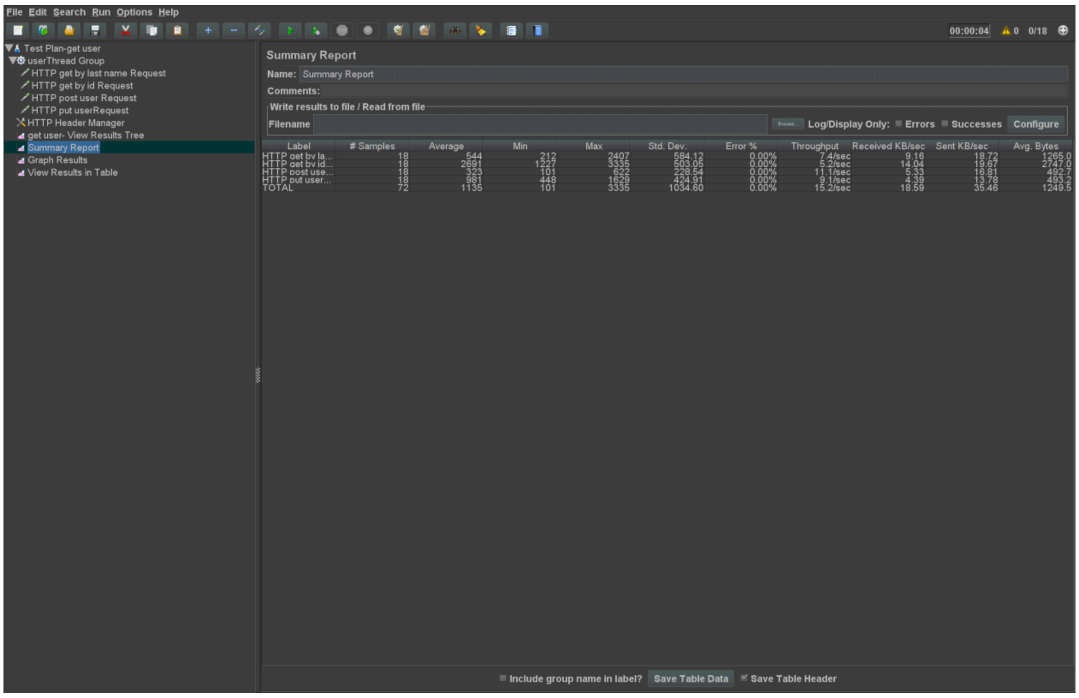
Comments:

Write results to file / Read from file

Filename | Log/Display Only: Errors Successes | Configure

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/sec | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| HTTP get by la... | 5 | 165 | 155 | 175 | 6.37 | 0.00% | 27.6/sec | 34.13 | 69.74 | 1265.0 |
| HTTP get by id... | 5 | 1202 | 756 | 1683 | 293.51 | 0.00% | 2.9/sec | 7.85 | 11.00 | 2746.0 |
| HTTP post use... | 5 | 185 | 135 | 206 | 26.27 | 0.00% | 4.3/sec | 2.08 | 6.54 | 493.0 |
| HTTP put user... | 5 | 203 | 203 | 204 | 0.49 | 0.00% | 4.1/sec | 1.96 | 6.16 | 493.2 |
| TOTAL | 20 | 439 | 135 | 1683 | 464.79 | 0.00% | 8.8/sec | 10.73 | 20.46 | 1249.3 |

**Figure A.3** User Module Summary Report for 60 requests in 0.5sec



Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

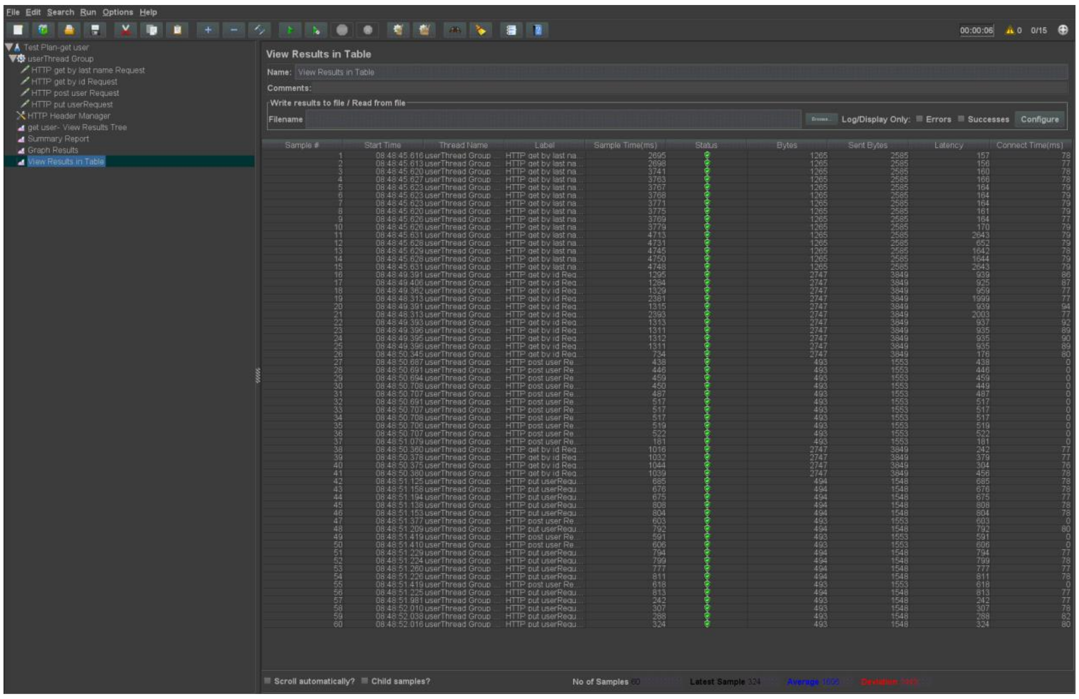
Filename

Log/Display Only: Errors Successes

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/sec | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| HTTP get by la... | 18 | 544 | 212 | 2407 | 584.12 | 0.00% | 7.4/sec | 9.16 | 18.72 | 1265.0 |
| HTTP get by id... | 18 | 2691 | 1227 | 3335 | 503.05 | 0.00% | 5.2/sec | 14.04 | 19.67 | 2747.0 |
| HTTP post use... | 18 | 323 | 101 | 622 | 228.54 | 0.00% | 11.1/sec | 5.33 | 16.81 | 492.7 |
| HTTP put user... | 18 | 981 | 448 | 1629 | 424.91 | 0.00% | 9.1/sec | 4.39 | 13.78 | 493.2 |
| TOTAL | 72 | 1135 | 101 | 3335 | 1034.60 | 0.00% | 15.2/sec | 18.59 | 35.46 | 1249.5 |

Include group name in label? Save Table Header

**Figure A.4** View User Module Result in Table for 60 requests



View Results in Table

Name: View Results in Table

Comments:

Write results to file / Read from file

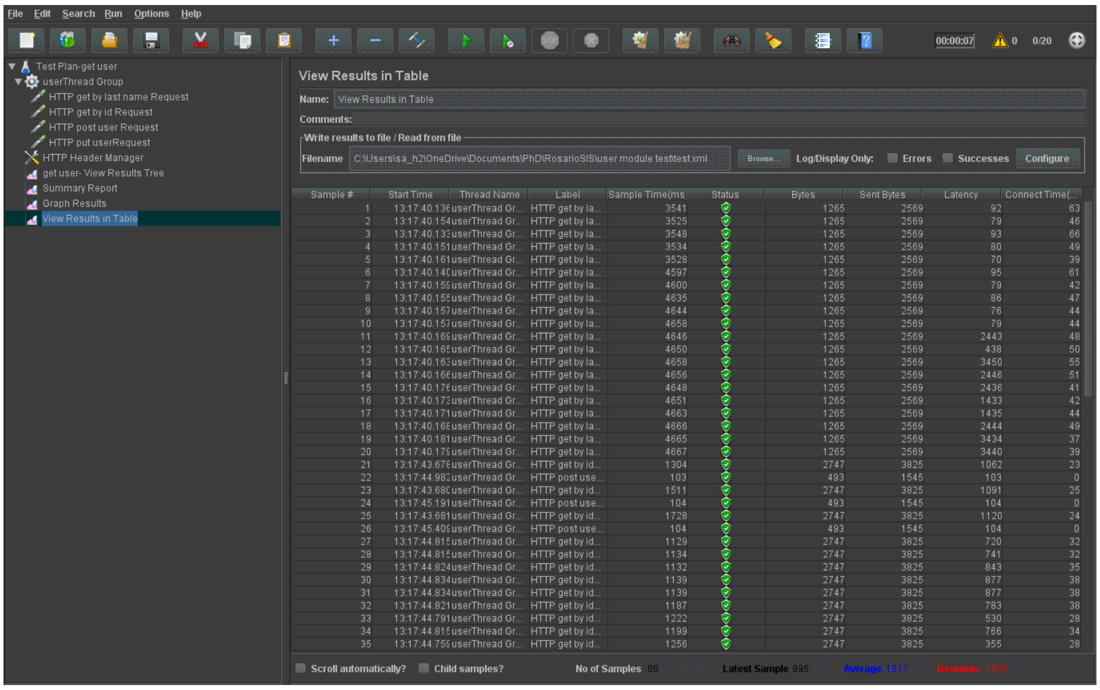
Filename

Log/Display Only: Errors Successes

| Sample # | Start Time | Thread Name | Label | Sample Time(ms) | Status | Bytes | Sent Bytes | Latency | Connect Time(ms) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 08:48:45.616 | userThread Group ... | HTTP get by last na... | 2695 | ✓ | 1265 | 2585 | 157 | 78 |
| 2 | 08:48:45.613 | userThread Group ... | HTTP get by last na... | 2698 | ✓ | 1265 | 2585 | 156 | 77 |
| 3 | 08:48:45.620 | userThread Group ... | HTTP get by last na... | 3741 | ✓ | 1265 | 2585 | 160 | 78 |
| 4 | 08:48:45.627 | userThread Group ... | HTTP get by last na... | 3763 | ✓ | 1265 | 2585 | 166 | 78 |
| 5 | 08:48:45.623 | userThread Group ... | HTTP get by last na... | 3767 | ✓ | 1265 | 2585 | 164 | 79 |
| 6 | 08:48:45.623 | userThread Group ... | HTTP get by last na... | 3768 | ✓ | 1265 | 2585 | 164 | 79 |
| 7 | 08:48:45.623 | userThread Group ... | HTTP get by last na... | 3771 | ✓ | 1265 | 2585 | 164 | 79 |
| 8 | 08:48:45.620 | userThread Group ... | HTTP get by last na... | 3775 | ✓ | 1265 | 2585 | 161 | 79 |
| 9 | 08:48:45.626 | userThread Group ... | HTTP get by last na... | 3769 | ✓ | 1265 | 2585 | 164 | 77 |
| 10 | 08:48:45.626 | userThread Group ... | HTTP get by last na... | 3779 | ✓ | 1265 | 2585 | 170 | 79 |
| 11 | 08:48:45.631 | userThread Group ... | HTTP get by last na... | 4713 | ✓ | 1265 | 2585 | 2643 | 79 |
| 12 | 08:48:45.628 | userThread Group ... | HTTP get by last na... | 4731 | ✓ | 1265 | 2585 | 652 | 79 |
| 13 | 08:48:45.629 | userThread Group ... | HTTP get by last na... | 4745 | ✓ | 1265 | 2585 | 1642 | 78 |
| 14 | 08:48:45.628 | userThread Group ... | HTTP get by last na... | 4750 | ✓ | 1265 | 2585 | 1644 | 79 |
| 15 | 08:48:45.631 | userThread Group ... | HTTP get by last na... | 4748 | ✓ | 1265 | 2585 | 2643 | 79 |
| 16 | 08:48:49.391 | userThread Group ... | HTTP get by id Req... | 1295 | ✓ | 2747 | 3849 | 939 | 86 |
| 17 | 08:48:49.406 | userThread Group ... | HTTP get by id Req... | 1284 | ✓ | 2747 | 3849 | 925 | 87 |
| 18 | 08:48:49.362 | userThread Group ... | HTTP get by id Req... | 1329 | ✓ | 2747 | 3849 | 959 | 77 |
| 19 | 08:48:48.313 | userThread Group ... | HTTP get by id Req... | 2381 | ✓ | 2747 | 3849 | 1999 | 77 |
| 20 | 08:48:49.391 | userThread Group ... | HTTP get by id Req... | 1315 | ✓ | 2747 | 3849 | 939 | 94 |
| 21 | 08:48:48.313 | userThread Group ... | HTTP get by id Req... | 2393 | ✓ | 2747 | 3849 | 2003 | 77 |
| 22 | 08:48:49.393 | userThread Group ... | HTTP get by id Req... | 1313 | ✓ | 2747 | 3849 | 937 | 92 |
| 23 | 08:48:49.396 | userThread Group ... | HTTP get by id Req... | 1311 | ✓ | 2747 | 3849 | 935 | 89 |
| 24 | 08:48:49.395 | userThread Group ... | HTTP get by id Req... | 1312 | ✓ | 2747 | 3849 | 935 | 90 |
| 25 | 08:48:49.396 | userThread Group ... | HTTP get by id Req... | 1311 | ✓ | 2747 | 3849 | 935 | 89 |
| 26 | 08:48:50.345 | userThread Group ... | HTTP get by id Req... | 734 | ✓ | 2747 | 3849 | 176 | 80 |
| 27 | 08:48:50.687 | userThread Group ... | HTTP post user Re... | 438 | ✓ | 493 | 1553 | 438 | 0 |
| 28 | 08:48:50.691 | userThread Group ... | HTTP post user Re... | 446 | ✓ | 493 | 1553 | 446 | 0 |
| 29 | 08:48:50.694 | userThread Group ... | HTTP post user Re... | 459 | ✓ | 493 | 1553 | 459 | 0 |
| 30 | 08:48:50.708 | userThread Group ... | HTTP post user Re... | 450 | ✓ | 493 | 1553 | 449 | 0 |
| 31 | 08:48:50.707 | userThread Group ... | HTTP post user Re... | 487 | ✓ | 493 | 1553 | 487 | 0 |
| 32 | 08:48:50.691 | userThread Group ... | HTTP post user Re... | 517 | ✓ | 493 | 1553 | 517 | 0 |
| 33 | 08:48:50.707 | userThread Group ... | HTTP post user Re... | 517 | ✓ | 493 | 1553 | 517 | 0 |
| 34 | 08:48:50.708 | userThread Group ... | HTTP post user Re... | 517 | ✓ | 493 | 1553 | 517 | 0 |
| 35 | 08:48:50.706 | userThread Group ... | HTTP post user Re... | 519 | ✓ | 493 | 1553 | 519 | 0 |
| 36 | 08:48:50.707 | userThread Group ... | HTTP post user Re... | 522 | ✓ | 493 | 1553 | 522 | 0 |
| 37 | 08:48:51.079 | userThread Group ... | HTTP post user Re... | 181 | ✓ | 493 | 1553 | 181 | 0 |
| 38 | 08:48:50.360 | userThread Group ... | HTTP get by id Req... | 1016 | ✓ | 2747 | 3849 | 242 | 77 |
| 39 | 08:48:50.378 | userThread Group ... | HTTP get by id Req... | 1032 | ✓ | 2747 | 3849 | 379 | 77 |
| 40 | 08:48:50.375 | userThread Group ... | HTTP get by id Req... | 1044 | ✓ | 2747 | 3849 | 304 | 76 |
| 41 | 08:48:50.380 | userThread Group ... | HTTP get by id Req... | 1039 | ✓ | 2747 | 3849 | 456 | 78 |
| 42 | 08:48:51.125 | userThread Group ... | HTTP put userRequ... | 685 | ✓ | 494 | 1548 | 685 | 78 |
| 43 | 08:48:51.158 | userThread Group ... | HTTP put userRequ... | 676 | ✓ | 494 | 1548 | 676 | 78 |
| 44 | 08:48:51.194 | userThread Group ... | HTTP put userRequ... | 675 | ✓ | 494 | 1548 | 675 | 77 |
| 45 | 08:48:51.138 | userThread Group ... | HTTP put userRequ... | 808 | ✓ | 494 | 1548 | 808 | 78 |
| 46 | 08:48:51.153 | userThread Group ... | HTTP put userRequ... | 804 | ✓ | 494 | 1548 | 804 | 78 |
| 47 | 08:48:51.377 | userThread Group ... | HTTP post user Re... | 603 | ✓ | 493 | 1553 | 603 | 0 |
| 48 | 08:48:51.209 | userThread Group ... | HTTP put userRequ... | 792 | ✓ | 494 | 1548 | 792 | 80 |
| 49 | 08:48:51.419 | userThread Group ... | HTTP post user Re... | 591 | ✓ | 493 | 1553 | 591 | 0 |
| 50 | 08:48:51.410 | userThread Group ... | HTTP post user Re... | 606 | ✓ | 493 | 1553 | 606 | 0 |
| 51 | 08:48:51.229 | userThread Group ... | HTTP put userRequ... | 794 | ✓ | 494 | 1548 | 794 | 77 |
| 52 | 08:48:51.224 | userThread Group ... | HTTP put userRequ... | 799 | ✓ | 494 | 1548 | 799 | 78 |
| 53 | 08:48:51.260 | userThread Group ... | HTTP put userRequ... | 777 | ✓ | 494 | 1548 | 777 | 77 |
| 54 | 08:48:51.226 | userThread Group ... | HTTP put userRequ... | 811 | ✓ | 494 | 1548 | 811 | 78 |
| 55 | 08:48:51.419 | userThread Group ... | HTTP post user Re... | 618 | ✓ | 493 | 1553 | 618 | 0 |
| 56 | 08:48:51.225 | userThread Group ... | HTTP put userRequ... | 813 | ✓ | 494 | 1548 | 813 | 77 |
| 57 | 08:48:51.981 | userThread Group ... | HTTP put userRequ... | 242 | ✓ | 493 | 1548 | 242 | 77 |
| 58 | 08:48:52.010 | userThread Group ... | HTTP put userRequ... | 307 | ✓ | 493 | 1548 | 307 | 78 |
| 59 | 08:48:52.038 | userThread Group ... | HTTP put userRequ... | 288 | ✓ | 493 | 1548 | 288 | 82 |
| 60 | 08:48:52.016 | userThread Group ... | HTTP put userRequ... | 324 | ✓ | 493 | 1548 | 324 | 80 |

Scroll automatically? Child samples? No of Samples 60 Latest Sample 324 Average 1606 Deviation [illegible]

**Figure A.5** View User Module Result in Table for 60 requests



View Results in Table

Name: View Results in Table

Comments:

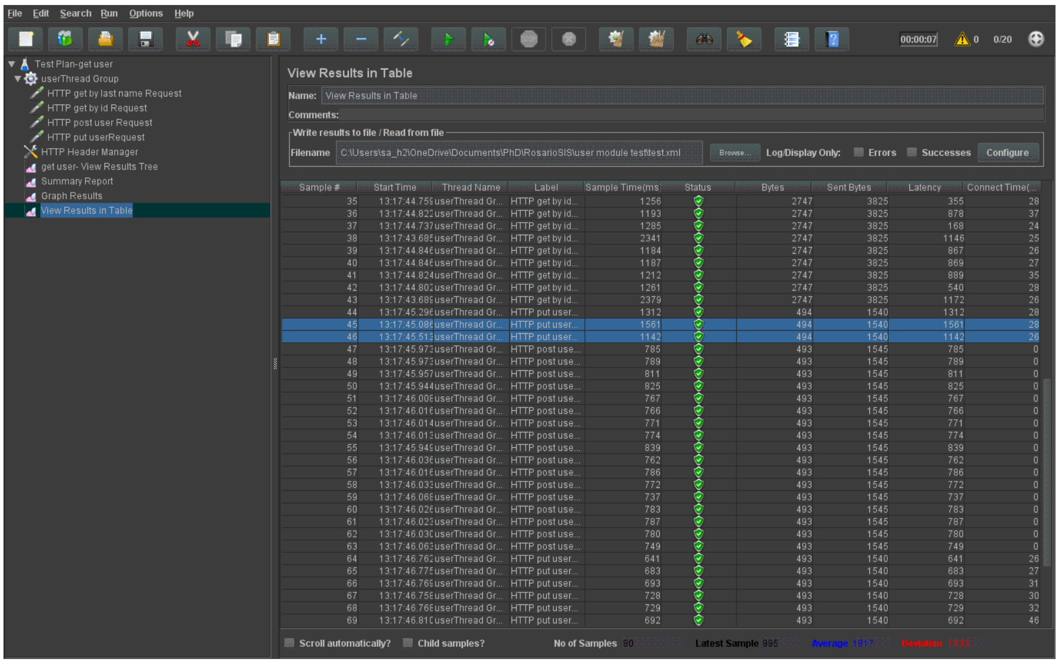
Write results to file / Read from file

Filename C:\Users\sa_h2\OneDrive\Documents\PhD\RosarioSIS\user module test\test.xml Browse... Log/Display Only: Errors Successes Configure

| Sample # | Start Time | Thread Name | Label | Sample Time(ms) | Status | Bytes | Sent Bytes | Latency | Connect Time(... |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 13:17:40.136 | userThread Gr... | HTTP get by la... | 3541 | ✓ | 1265 | 2569 | 92 | 63 |
| 2 | 13:17:40.154 | userThread Gr... | HTTP get by la... | 3525 | ✓ | 1265 | 2569 | 79 | 46 |
| 3 | 13:17:40.133 | userThread Gr... | HTTP get by la... | 3548 | ✓ | 1265 | 2569 | 93 | 66 |
| 4 | 13:17:40.151 | userThread Gr... | HTTP get by la... | 3534 | ✓ | 1265 | 2569 | 80 | 49 |
| 5 | 13:17:40.161 | userThread Gr... | HTTP get by la... | 3528 | ✓ | 1265 | 2569 | 70 | 39 |
| 6 | 13:17:40.140 | userThread Gr... | HTTP get by la... | 4597 | ✓ | 1265 | 2569 | 95 | 61 |
| 7 | 13:17:40.159 | userThread Gr... | HTTP get by la... | 4600 | ✓ | 1265 | 2569 | 79 | 42 |
| 8 | 13:17:40.155 | userThread Gr... | HTTP get by la... | 4635 | ✓ | 1265 | 2569 | 86 | 47 |
| 9 | 13:17:40.157 | userThread Gr... | HTTP get by la... | 4644 | ✓ | 1265 | 2569 | 76 | 44 |
| 10 | 13:17:40.157 | userThread Gr... | HTTP get by la... | 4658 | ✓ | 1265 | 2569 | 79 | 44 |
| 11 | 13:17:40.169 | userThread Gr... | HTTP get by la... | 4646 | ✓ | 1265 | 2569 | 2443 | 48 |
| 12 | 13:17:40.165 | userThread Gr... | HTTP get by la... | 4650 | ✓ | 1265 | 2569 | 438 | 50 |
| 13 | 13:17:40.163 | userThread Gr... | HTTP get by la... | 4658 | ✓ | 1265 | 2569 | 3450 | 55 |
| 14 | 13:17:40.166 | userThread Gr... | HTTP get by la... | 4656 | ✓ | 1265 | 2569 | 2446 | 51 |
| 15 | 13:17:40.176 | userThread Gr... | HTTP get by la... | 4648 | ✓ | 1265 | 2569 | 2436 | 41 |
| 16 | 13:17:40.173 | userThread Gr... | HTTP get by la... | 4651 | ✓ | 1265 | 2569 | 1433 | 42 |
| 17 | 13:17:40.171 | userThread Gr... | HTTP get by la... | 4663 | ✓ | 1265 | 2569 | 1435 | 44 |
| 18 | 13:17:40.168 | userThread Gr... | HTTP get by la... | 4666 | ✓ | 1265 | 2569 | 2444 | 49 |
| 19 | 13:17:40.181 | userThread Gr... | HTTP get by la... | 4665 | ✓ | 1265 | 2569 | 3434 | 37 |
| 20 | 13:17:40.179 | userThread Gr... | HTTP get by la... | 4667 | ✓ | 1265 | 2569 | 3440 | 39 |
| 21 | 13:17:43.678 | userThread Gr... | HTTP get by id... | 1304 | ✓ | 2747 | 3825 | 1062 | 23 |
| 22 | 13:17:44.982 | userThread Gr... | HTTP post use... | 103 | ✓ | 493 | 1545 | 103 | 0 |
| 23 | 13:17:43.680 | userThread Gr... | HTTP get by id... | 1511 | ✓ | 2747 | 3825 | 1091 | 25 |
| 24 | 13:17:45.191 | userThread Gr... | HTTP post use... | 104 | ✓ | 493 | 1545 | 104 | 0 |
| 25 | 13:17:43.681 | userThread Gr... | HTTP get by id... | 1728 | ✓ | 2747 | 3825 | 1120 | 24 |
| 26 | 13:17:45.409 | userThread Gr... | HTTP post use... | 104 | ✓ | 493 | 1545 | 104 | 0 |
| 27 | 13:17:44.815 | userThread Gr... | HTTP get by id... | 1129 | ✓ | 2747 | 3825 | 720 | 32 |
| 28 | 13:17:44.815 | userThread Gr... | HTTP get by id... | 1134 | ✓ | 2747 | 3825 | 741 | 32 |
| 29 | 13:17:44.824 | userThread Gr... | HTTP get by id... | 1132 | ✓ | 2747 | 3825 | 843 | 35 |
| 30 | 13:17:44.834 | userThread Gr... | HTTP get by id... | 1139 | ✓ | 2747 | 3825 | 877 | 38 |
| 31 | 13:17:44.834 | userThread Gr... | HTTP get by id... | 1139 | ✓ | 2747 | 3825 | 877 | 38 |
| 32 | 13:17:44.821 | userThread Gr... | HTTP get by id... | 1187 | ✓ | 2747 | 3825 | 783 | 38 |
| 33 | 13:17:44.791 | userThread Gr... | HTTP get by id... | 1222 | ✓ | 2747 | 3825 | 530 | 28 |
| 34 | 13:17:44.815 | userThread Gr... | HTTP get by id... | 1199 | ✓ | 2747 | 3825 | 766 | 34 |
| 35 | 13:17:44.759 | userThread Gr... | HTTP get by id... | 1256 | ✓ | 2747 | 3825 | 355 | 28 |

Scroll automatically? Child samples? No of Samples 80 Latest Sample 995 Average 1817 Deviation 1533

**Figure A.6** View User Module Result in Table for 80 requests in 4.48 sec (1)



View Results in Table

Name: View Results in Table

Comments:

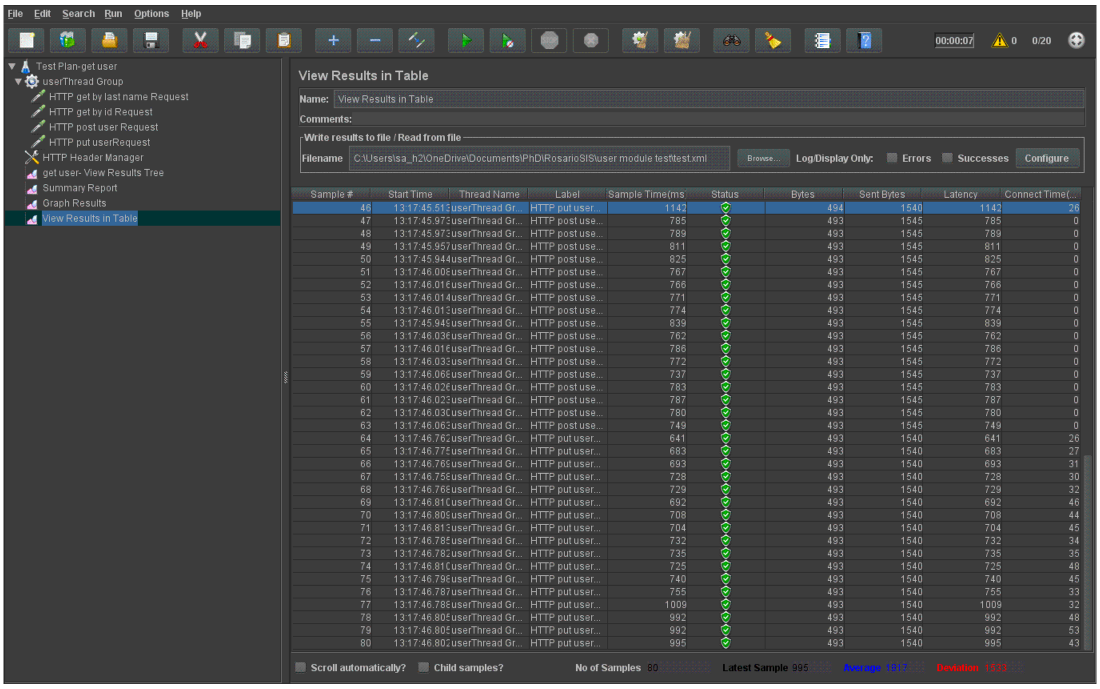
Write results to file / Read from file

Filename C:\Users\sa_h2\OneDrive\Documents\PhD\RosarioSIS\user module test\test.xml Browse... Log/Display Only: Errors Successes Configure

| Sample # | Start Time | Thread Name | Label | Sample Time(ms) | Status | Bytes | Sent Bytes | Latency | Connect Time(... |
|---|---|---|---|---|---|---|---|---|---|
| 35 | 13:17:44.759 | userThread Gr... | HTTP get by id... | 1256 | ✓ | 2747 | 3825 | 355 | 28 |
| 36 | 13:17:44.822 | userThread Gr... | HTTP get by id... | 1193 | ✓ | 2747 | 3825 | 878 | 37 |
| 37 | 13:17:44.737 | userThread Gr... | HTTP get by id... | 1285 | ✓ | 2747 | 3825 | 168 | 24 |
| 38 | 13:17:43.685 | userThread Gr... | HTTP get by id... | 2341 | ✓ | 2747 | 3825 | 1146 | 25 |
| 39 | 13:17:44.846 | userThread Gr... | HTTP get by id... | 1184 | ✓ | 2747 | 3825 | 867 | 26 |
| 40 | 13:17:44.846 | userThread Gr... | HTTP get by id... | 1187 | ✓ | 2747 | 3825 | 869 | 27 |
| 41 | 13:17:44.824 | userThread Gr... | HTTP get by id... | 1212 | ✓ | 2747 | 3825 | 889 | 35 |
| 42 | 13:17:44.802 | userThread Gr... | HTTP get by id... | 1261 | ✓ | 2747 | 3825 | 540 | 28 |
| 43 | 13:17:43.689 | userThread Gr... | HTTP get by id... | 2379 | ✓ | 2747 | 3825 | 1172 | 26 |
| 44 | 13:17:45.296 | userThread Gr... | HTTP put user... | 1312 | ✓ | 494 | 1540 | 1312 | 28 |
| 45 | 13:17:45.086 | userThread Gr... | HTTP put user... | 1561 | ✓ | 494 | 1540 | 1561 | 28 |
| 46 | 13:17:45.513 | userThread Gr... | HTTP put user... | 1142 | ✓ | 494 | 1540 | 1142 | 26 |
| 47 | 13:17:45.973 | userThread Gr... | HTTP post use... | 785 | ✓ | 493 | 1545 | 785 | 0 |
| 48 | 13:17:45.973 | userThread Gr... | HTTP post use... | 789 | ✓ | 493 | 1545 | 789 | 0 |
| 49 | 13:17:45.957 | userThread Gr... | HTTP post use... | 811 | ✓ | 493 | 1545 | 811 | 0 |
| 50 | 13:17:45.944 | userThread Gr... | HTTP post use... | 825 | ✓ | 493 | 1545 | 825 | 0 |
| 51 | 13:17:46.008 | userThread Gr... | HTTP post use... | 767 | ✓ | 493 | 1545 | 767 | 0 |
| 52 | 13:17:46.016 | userThread Gr... | HTTP post use... | 766 | ✓ | 493 | 1545 | 766 | 0 |
| 53 | 13:17:46.014 | userThread Gr... | HTTP post use... | 771 | ✓ | 493 | 1545 | 771 | 0 |
| 54 | 13:17:46.013 | userThread Gr... | HTTP post use... | 774 | ✓ | 493 | 1545 | 774 | 0 |
| 55 | 13:17:45.949 | userThread Gr... | HTTP post use... | 839 | ✓ | 493 | 1545 | 839 | 0 |
| 56 | 13:17:46.036 | userThread Gr... | HTTP post use... | 762 | ✓ | 493 | 1545 | 762 | 0 |
| 57 | 13:17:46.016 | userThread Gr... | HTTP post use... | 786 | ✓ | 493 | 1545 | 786 | 0 |
| 58 | 13:17:46.033 | userThread Gr... | HTTP post use... | 772 | ✓ | 493 | 1545 | 772 | 0 |
| 59 | 13:17:46.068 | userThread Gr... | HTTP post use... | 737 | ✓ | 493 | 1545 | 737 | 0 |
| 60 | 13:17:46.026 | userThread Gr... | HTTP post use... | 783 | ✓ | 493 | 1545 | 783 | 0 |
| 61 | 13:17:46.023 | userThread Gr... | HTTP post use... | 787 | ✓ | 493 | 1545 | 787 | 0 |
| 62 | 13:17:46.030 | userThread Gr... | HTTP post use... | 780 | ✓ | 493 | 1545 | 780 | 0 |
| 63 | 13:17:46.063 | userThread Gr... | HTTP post use... | 749 | ✓ | 493 | 1545 | 749 | 0 |
| 64 | 13:17:46.762 | userThread Gr... | HTTP put user... | 641 | ✓ | 493 | 1540 | 641 | 26 |
| 65 | 13:17:46.775 | userThread Gr... | HTTP put user... | 683 | ✓ | 493 | 1540 | 683 | 27 |
| 66 | 13:17:46.769 | userThread Gr... | HTTP put user... | 693 | ✓ | 493 | 1540 | 693 | 31 |
| 67 | 13:17:46.758 | userThread Gr... | HTTP put user... | 728 | ✓ | 493 | 1540 | 728 | 30 |
| 68 | 13:17:46.768 | userThread Gr... | HTTP put user... | 729 | ✓ | 493 | 1540 | 729 | 32 |
| 69 | 13:17:46.810 | userThread Gr... | HTTP put user... | 692 | ✓ | 493 | 1540 | 692 | 46 |

Scroll automatically? Child samples? No of Samples 80 Latest Sample 995 Average 1817 Deviation 1533

**Figure A.7** View User Module Result in Table for 80 requests in 4.48 sec (2)



View Results in Table

Name: View Results in Table

Comments:

Write results to file / Read from file

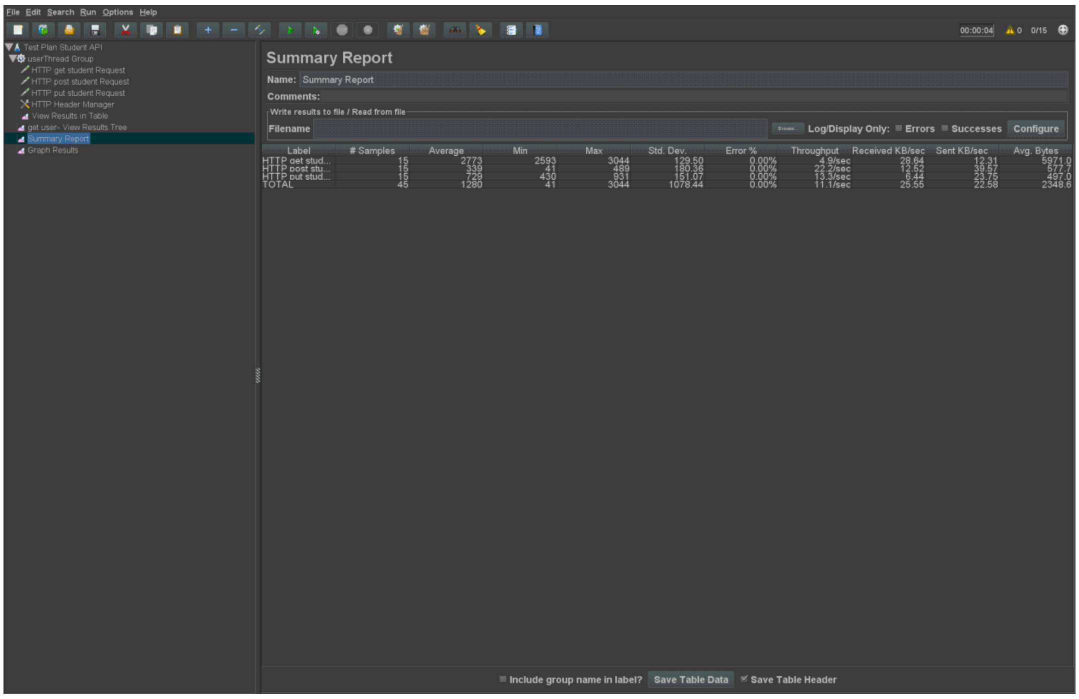
Filename C:\Users\sa_h2\OneDrive\Documents\PhD\RosarioSIS\user module test\test.xml

Log/Display Only: Errors Successes

| Sample # | Start Time | Thread Name | Label | Sample Time(ms | Status | Bytes | Sent Bytes | Latency | Connect Time(... |
|---|---|---|---|---|---|---|---|---|---|
| 46 | 13:17:45.513 | userThread Gr... | HTTP put user... | 1142 | ✓ | 494 | 1540 | 1142 | 26 |
| 47 | 13:17:45.973 | userThread Gr... | HTTP post use... | 785 | ✓ | 493 | 1545 | 785 | 0 |
| 48 | 13:17:45.973 | userThread Gr... | HTTP post use... | 789 | ✓ | 493 | 1545 | 789 | 0 |
| 49 | 13:17:45.957 | userThread Gr... | HTTP post use... | 811 | ✓ | 493 | 1545 | 811 | 0 |
| 50 | 13:17:45.944 | userThread Gr... | HTTP post use... | 825 | ✓ | 493 | 1545 | 825 | 0 |
| 51 | 13:17:46.008 | userThread Gr... | HTTP post use... | 767 | ✓ | 493 | 1545 | 767 | 0 |
| 52 | 13:17:46.016 | userThread Gr... | HTTP post use... | 766 | ✓ | 493 | 1545 | 766 | 0 |
| 53 | 13:17:46.014 | userThread Gr... | HTTP post use... | 771 | ✓ | 493 | 1545 | 771 | 0 |
| 54 | 13:17:46.013 | userThread Gr... | HTTP post use... | 774 | ✓ | 493 | 1545 | 774 | 0 |
| 55 | 13:17:45.949 | userThread Gr... | HTTP post use... | 839 | ✓ | 493 | 1545 | 839 | 0 |
| 56 | 13:17:46.036 | userThread Gr... | HTTP post use... | 762 | ✓ | 493 | 1545 | 762 | 0 |
| 57 | 13:17:46.016 | userThread Gr... | HTTP post use... | 786 | ✓ | 493 | 1545 | 786 | 0 |
| 58 | 13:17:46.033 | userThread Gr... | HTTP post use... | 772 | ✓ | 493 | 1545 | 772 | 0 |
| 59 | 13:17:46.068 | userThread Gr... | HTTP post use... | 737 | ✓ | 493 | 1545 | 737 | 0 |
| 60 | 13:17:46.026 | userThread Gr... | HTTP post use... | 783 | ✓ | 493 | 1545 | 783 | 0 |
| 61 | 13:17:46.023 | userThread Gr... | HTTP post use... | 787 | ✓ | 493 | 1545 | 787 | 0 |
| 62 | 13:17:46.030 | userThread Gr... | HTTP post use... | 780 | ✓ | 493 | 1545 | 780 | 0 |
| 63 | 13:17:46.063 | userThread Gr... | HTTP post use... | 749 | ✓ | 493 | 1545 | 749 | 0 |
| 64 | 13:17:46.762 | userThread Gr... | HTTP put user... | 641 | ✓ | 493 | 1540 | 641 | 26 |
| 65 | 13:17:46.775 | userThread Gr... | HTTP put user... | 683 | ✓ | 493 | 1540 | 683 | 27 |
| 66 | 13:17:46.769 | userThread Gr... | HTTP put user... | 693 | ✓ | 493 | 1540 | 693 | 31 |
| 67 | 13:17:46.758 | userThread Gr... | HTTP put user... | 728 | ✓ | 493 | 1540 | 728 | 30 |
| 68 | 13:17:46.768 | userThread Gr... | HTTP put user... | 729 | ✓ | 493 | 1540 | 729 | 32 |
| 69 | 13:17:46.810 | userThread Gr... | HTTP put user... | 692 | ✓ | 493 | 1540 | 692 | 46 |
| 70 | 13:17:46.809 | userThread Gr... | HTTP put user... | 708 | ✓ | 493 | 1540 | 708 | 44 |
| 71 | 13:17:46.813 | userThread Gr... | HTTP put user... | 704 | ✓ | 493 | 1540 | 704 | 45 |
| 72 | 13:17:46.785 | userThread Gr... | HTTP put user... | 732 | ✓ | 493 | 1540 | 732 | 34 |
| 73 | 13:17:46.782 | userThread Gr... | HTTP put user... | 735 | ✓ | 493 | 1540 | 735 | 35 |
| 74 | 13:17:46.810 | userThread Gr... | HTTP put user... | 725 | ✓ | 493 | 1540 | 725 | 48 |
| 75 | 13:17:46.798 | userThread Gr... | HTTP put user... | 740 | ✓ | 493 | 1540 | 740 | 45 |
| 76 | 13:17:46.787 | userThread Gr... | HTTP put user... | 755 | ✓ | 493 | 1540 | 755 | 33 |
| 77 | 13:17:46.788 | userThread Gr... | HTTP put user... | 1009 | ✓ | 493 | 1540 | 1009 | 32 |
| 78 | 13:17:46.805 | userThread Gr... | HTTP put user... | 992 | ✓ | 493 | 1540 | 992 | 48 |
| 79 | 13:17:46.805 | userThread Gr... | HTTP put user... | 992 | ✓ | 493 | 1540 | 992 | 53 |
| 80 | 13:17:46.802 | userThread Gr... | HTTP put user... | 995 | ✓ | 493 | 1540 | 995 | 43 |

Scroll automatically? Child samples? No of Samples 80 Latest Sample 995 Average 1817 Deviation 1533

**Figure A.8** View User Module Result in Table for 80 requests in 4.48 sec (3)



Summary Report

Name: Summary Report

Comments:

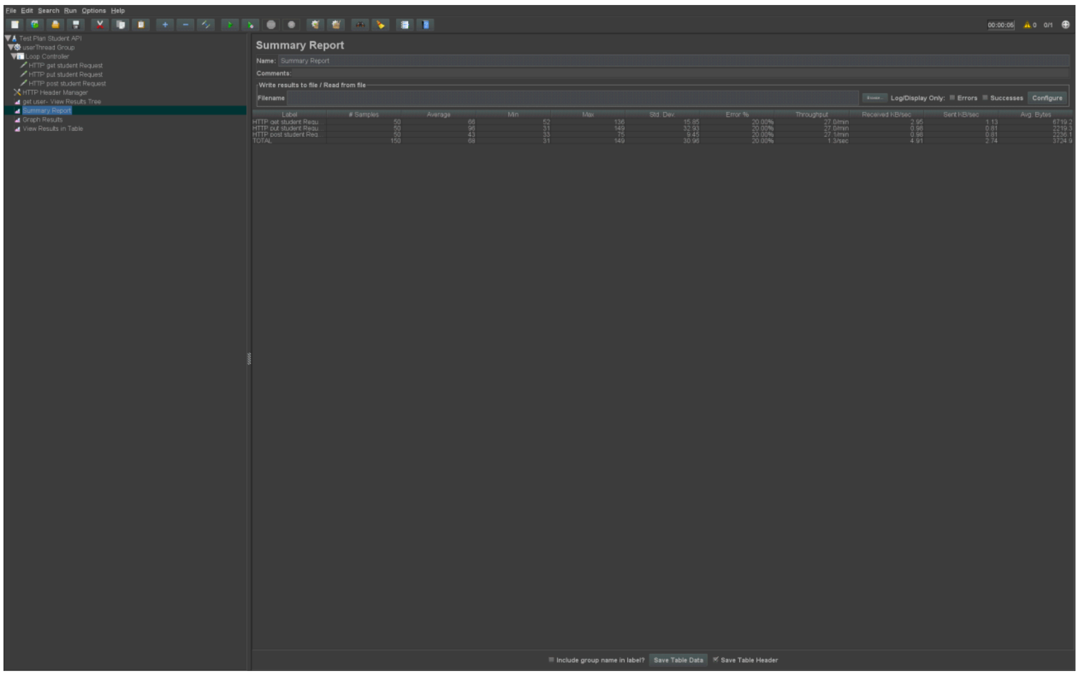
Write results to file / Read from file

Filename

Log/Display Only: Errors Successes

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/sec | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| HTTP get stud... | 15 | 2773 | 2593 | 3044 | 129.50 | 0.00% | 4.9/sec | 28.64 | 12.31 | 5971.0 |
| HTTP post stu... | 15 | 339 | 41 | 489 | 180.36 | 0.00% | 22.2/sec | 12.52 | 39.57 | 577.7 |
| HTTP put stud... | 15 | 729 | 430 | 931 | 151.07 | 0.00% | 13.3/sec | 6.44 | 23.75 | 497.0 |
| TOTAL | 45 | 1280 | 41 | 3044 | 1078.44 | 0.00% | 11.1/sec | 25.55 | 22.58 | 2348.6 |

**Figure A.9** Student Module Summary Report for 45 requests in 0.5 sec



Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/sec | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| HTTP get student Requ... | 50 | 66 | 52 | 136 | 15.85 | 20.00% | 27.0/min | 2.95 | 1.13 | 6719.2 |
| HTTP put student Requ... | 50 | 96 | 31 | 149 | 32.93 | 20.00% | 27.0/min | 0.96 | 0.81 | 2219.3 |
| HTTP post student Req... | 50 | 43 | 33 | 75 | 9.45 | 20.00% | 27.1/min | 0.96 | 0.81 | 2236.1 |
| TOTAL | 150 | 68 | 31 | 149 | 30.96 | 20.00% | 1.3/sec | 4.91 | 2.74 | 3724.9 |

**Figure A.10** Student Module Summary Report for 150 requests in 3.5 sec