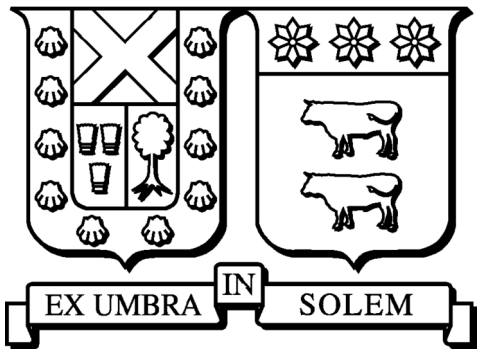
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO – CHILE



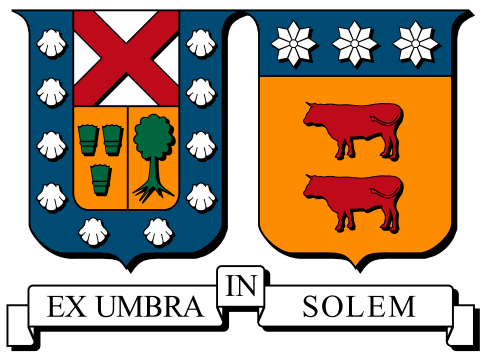
"SEMI-AUTOMATIC GENERATION OF MICROSERVICES ARCHITECTURES TO SATISFY NON-FUNCTIONAL REQUIREMENTS THROUGH ARCHITECTURAL PATTERNS AND TACTICS"

GASTÓN PATRICIO MÁRQUEZ ORTEGA

DISSERTATION THESIS FOR THE DEGREE OF DOCTOR IN INFORMATICS ENGINEERING

Valparaíso, Chile
September, 2020

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO – CHILE



"SEMI-AUTOMATIC GENERATION OF MICROSERVICES ARCHITECTURES TO SATISFY NON-FUNCTIONAL REQUIREMENTS THROUGH ARCHITECTURAL PATTERNS AND TACTICS"

GASTÓN PATRICIO MÁRQUEZ ORTEGA

DISSERTATION THESIS FOR THE DEGREE OF DOCTOR IN INFORMATICS ENGINEERING

Valparaíso, Chile
September, 2020

TÍTULO DE LA TESIS:

Semi-automatic Generation of Microservices Architectures to Satisfy Non-Functional Requirements through Architectural Patterns and Tactics

AUTOR:

Gastón Patricio Márquez Ortega

TRABAJO DE GRADO, presentado en cumplimiento parcial de los requisitos para el Grado de Doctor en Ingeniería Informática de la Universidad Técnica Federico Santa María.

Profesor guía	Dr. Hernán Astudillo
Examinador externo internacional (Universidad Rey Juan Carlos, España)	Dr. Rafael Capilla
Examinador externo internacional (Universidad Federal de Rio de Janeiro, Brasil)	Dr. Guilherme Travassos
Examinadora externa nacional (Universidad de Valparaíso, Chile)	Dra. Carla Taramasco
Examinadora interna (Universidad Técnica Federico Santa María, Chile)	Dra. Erika Rosas
Presidente de la comisión	Dr. Carlos Castro

Valparaíso, Chile

Septiembre 2020

To Ana, Rosa, Eliana, Arturo and Juvenal. When life takes a loved one from you, the memory of their smile is the best way to move forward.

Acknowledgments

I thank my advisor, Professor Dr. Hernán Astudillo Rojas, for teaching me the passion and vigor of scientific research. He has given me the freedom to explore the different challenges that emerge in Software Architecture and has continuously supported me in achieving all my goals. His inspiration has given me a new perspective to face the challenges of computer science and informatics. Additionally, his advice has helped me to form my profile as a researcher. I will always be grateful to him for teaching me to respect, value and enjoy scientific research.

I would also like to thank all those colleagues who helped me finish my thesis. I make special mention of Professor Dr. Marcello Visconti, who thanks to him, I was able to start my Ph.D. program successfully. Additionally, I also thank Professor Dr. Mehdi Mirakhorli for accepting me as an intern student at the Rochester Institute of Technology (RIT). Thanks to him, I was able to advance my thesis.

I also thank the tools and support provided by Universidad Técnica Federico Santa María, the Department of Informatics, and the Toeska research team to conduct my Ph.D program.

I would also like to thank the *Agencia Nacional de Investigación y Desarrollo*, ANID (formerly the *Comisión Nacional de Investigación Científica y Tecnológica*, CONICYT) for the doctoral scholarship (CONICYT-PCHA Doctorado Nacional/2016-21161005) that I received.

I thank my mother, father and brother for their constant support. Their talks and encouragement made it possible for me to finish this adventure. You guys have always been and will be my endless source of support and joy.

Last but not least, I am deeply grateful to Michelle, my wife. You were by my side throughout this process, having good and bad times. You followed me on every adventure that came our way. You were with me when I wanted to give up and you managed to show me that *everything is possible*. You are an incredible woman.

Resumen

Microservicios es un estilo de arquitectura que promueve la facilidad de construir y mantener sistemas desglosando sus capacidades de negocio en servicios más pequeños y distribuidos. A menudo, los profesionales suelen utilizar frameworks y plataformas para proporcionar funcionalidades genéricas que abordan problemas recurrentes relacionados a atributos de calidad de sistemas basados en microservicios. No obstante, en la práctica, la información de los frameworks y plataformas es imprecisa y cambiante, así como los requisitos. Los enfoques de despliegue más realista combinan la exploración de las arquitecturas candidatas con su evaluación en cuanto a la satisfacción de los requisitos y la información confusa e incompleta de los frameworks y plataformas disponibles. Esta tesis doctoral describe una técnica novedosa, llamada μAzimut, cuyo propósito es generar, evaluar y comparar los ensamblajes de frameworks y plataformas usando descripciones potencialmente imprecisas y cambiantes de requisitos no funcionales, frameworks y plataformas. La evaluación del ensamblaje de los frameworks y plataformas se basa en una puntuación de apoyo que permite modelar un conocimiento arquitectónico imperfecto. La técnica es evaluada en tres estudios empíricos. Los resultados señalan que μAzimut genera soluciones cercanas a las que los arquitectos seleccionan para diseñar e implementar arquitecturas de microservicios.

Abstract

Microservices is an architectural style that promotes the facility to build and maintain systems by breaking down its business capabilities into smaller and distributed services. Often, practitioners commonly use frameworks and platforms to provide generic functionalities to address recurring quality attribute concerns on microservices-based systems. Nevertheless, in practical settings, the information of frameworks and platforms is imprecise and changing as well as requirements. More realistically deployable approaches combine the exploration of candidate architectures with their evaluation regarding requirements satisfaction and the fuzziness and incompleteness available frameworks and platforms information. This doctoral thesis outlines a novel technique called μAzimut, whose purpose is to generate, evaluate, and compare frameworks and platforms assemblies using potentially imprecise and changing descriptions of non-functional requirements, frameworks and platforms. The frameworks and platforms assembly's evaluation is based on a support score which allows modeling imperfect architectural knowledge. The technique is evaluated in three empirical studies. The results point out that μAzimut generates solutions close to those solutions that architects select for designing and implementing microservices architectures.

Table of Contents

List of Tables

List of Figures

Part I

Problem Statement and Background

Chapter 1

Introduction

This chapter aims to describe the principal elements and concepts of this doctoral thesis. Section 1.1 describes the main concepts used in this doctoral thesis; Section 1.2 details the general research question and hypothesis addressed by this doctoral thesis; Section 1.3 describes the solution that addresses the research problem; Section 1.4 describes the main research objectives; In the same way, Section 1.5 describes the main research questions; Section 1.6 introduces the main contributions of this doctoral thesis; Section 1.7 describes the published works: And Section 1.8 describes the general structure of the thesis.

1.1. Background

This section introduces the most relevant concepts related to the proposal of this thesis. This section describes (i) software architecture, (ii) non-functional requirements and quality attributes, (iii) architectural patterns, (iv) architectural tactics, (v) frameworks and (vi) microservices.

1.1.1. Software architecture

The software development process is structured around a set of disciplines often represented as requirements engineering, software design, implementation, testing and maintenance [23]. Each of these disciplines' practices produces a software product from an initial request from the client and stakeholders. While the requirements engineering discipline helps analysts understand the clients' concerns, the design discipline leads to the development of abstract solutions to map from such requirements to implementations and eventually to a software product. In this regard, software architecture design occurs at the overlap of the two disciplines mentioned above. It includes principles and practices of both requirements engineering and software design [100].

Over the last decade, software architecture design has frequently been considered a key activity to influence software qualities such as security, reliability, performance, etc. Consequently, it has been the focus of a wide range of research and practice. There are several definitions for the architecture of a software system. Traditional definitions consider software architecture as the structure or skeleton of a system. In this definition, architecture is a collection of components building a software system [21]. Nevertheless, an essential criticism of this definition is that they address only the physical infrastructure of a system and fail to capture other architectural decisions' importance. Other definitions (such as [22], [81] and [112]) refer to software architecture as a set of interrelated design decisions that work together to shape the structure, behavior, properties, processes, technologies and governance of the delivered solution. Additionally, some definitions also consider that software architecture is performed by reusing proven solutions to recurring design problems. These proven solutions can be conceptual, such as architectural/design patterns or concrete, such as application frameworks [31]. From these perspectives, architectural quality is achieved not only through engineering practices but also through managing and maintaining a broad set of architectural decisions, conceptual and concrete solutions.

1.1.2. Non-functional requirements and quality attributes

To design software architectures, architects generally rely on non-functional requirements (NFRs) [36]. These requirements are not related to the specific functions provided by the system, but rather to system properties such as performance, security, availability, and others. More precisely, they do not talk about "what" the system does, but "how" it does it. Alternatively, NFRs define system constraints such as the capacity of the input/output devices and the representation of the data used in the system interface. NFRs are originated from user needs, due to budgetary restrictions, organizational policies, the need for interoperability with other software or hardware systems, or external factors such as security regulations, privacy policies, among others.

There are different types of NFRs, and they are classified according to their implications:

- *Product requirements.* They specify product behavior, such as performance requirements on system execution speed and the amount of memory needed, reliability requirements that set the failure rate for the system to be acceptable, portability requirements, and usability requirements.
- *Organizational requirements.* These requirements are derived from existing policies and procedures in the client organization and the developer organization such as standards in the processes to be used, implementation requirements such as programming languages or the design method to be used, and delivery requirements that specify when the product and its documentation will be delivered.
- *External needs.* These requirements derive from external factors to the system and its development process. They include interoperability requirements that define how the system interacts with other systems in the organization, legal requirements that must be followed to ensure that the system operates within the law and ethical requirements. The latter is imposed on the system to ensure that the user will accept it.

According to Cysneiros *et al.* [38], NFRs can be translated into *quality attributes*, which

represent restrictions or the qualities that the system must have, such as precision, usability, security, performance, reliability, performance, among others. They represent non-functional characteristics considered desirable in a software system [21]. Although quality attributes are essential to determine the qualities of the systems, some of them will be more important than others (depending on the system), and certainly cannot be maximized all at once. A difference is made between qualities and requirements because some of them can be incorporated as input to the design by a different path than the analysis (for example, as architectural constraints or environmental influences).

1.1.3. Architectural patterns

Architectural patterns represent systematic solutions to recurring architectural problems [21]. More specifically, architectural patterns are used to express a base organizational structure or scheme for software. Furthermore, they inherit much of the design pattern terminology and concepts but focus on providing reusable models and methods specifically for general system architecture. In other words, this means that unlike design patterns, these are incomplete templates and cannot be directly applied to code with merely contextual modifications.

Usually, architectural patterns have four elements [4]:

- *Name*
- *Context*: A recurring, common situation in the world that gives rise to a problem.
- *Problem*: The situation that arises in the given problem, appropriately generalized.
- *Solution*: A successfully architectural resolution to the problem, appropriately abstracted.

For example, the well-known Broker architectural pattern (Figure 1.1) can be described as follow:

- *Name*: Broker
- *Context*: Distributed environments.
- *Problem*: How to structure the system to facilitate service invocations among remote components?
- *Solution*: Introduce a broker component that decouples clients and servers by making the latter's services available to the former via requests to the broker itself.

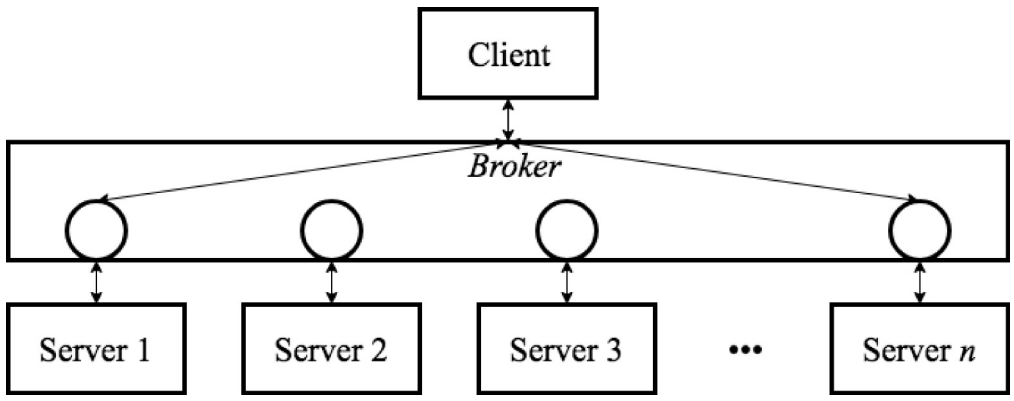


Figure 1.1: The Broker architectural pattern

1.1.4. Architectural tactics

For architectural patterns to meet quality attributes, they must be complemented by architectural tactics. Architectural tactics are design decisions that influence the control of the response of a quality attribute. Unlike design patterns, architectural tactics are more specific solutions and are focused on specific quality attributes [21].

For example, we illustrated the use of architectural tactics to build *availability* (the QA) into a system using as catalog the Availability tactics taxonomy of [21] (see Figure 1.2). Architects must make three decisions (each a category in architectural tactics taxonomies) on how the system will *act* to address a *fault* (the stimulus): (1) how to *detect* it (e.g. monitor a heartbeat), (2) how to *react* to it (e.g. rollback), and (3) how to *recover* from it (e.g. shadow state). Finally, additional decisions may concern how to *prevent* the stimulus itself in the future.

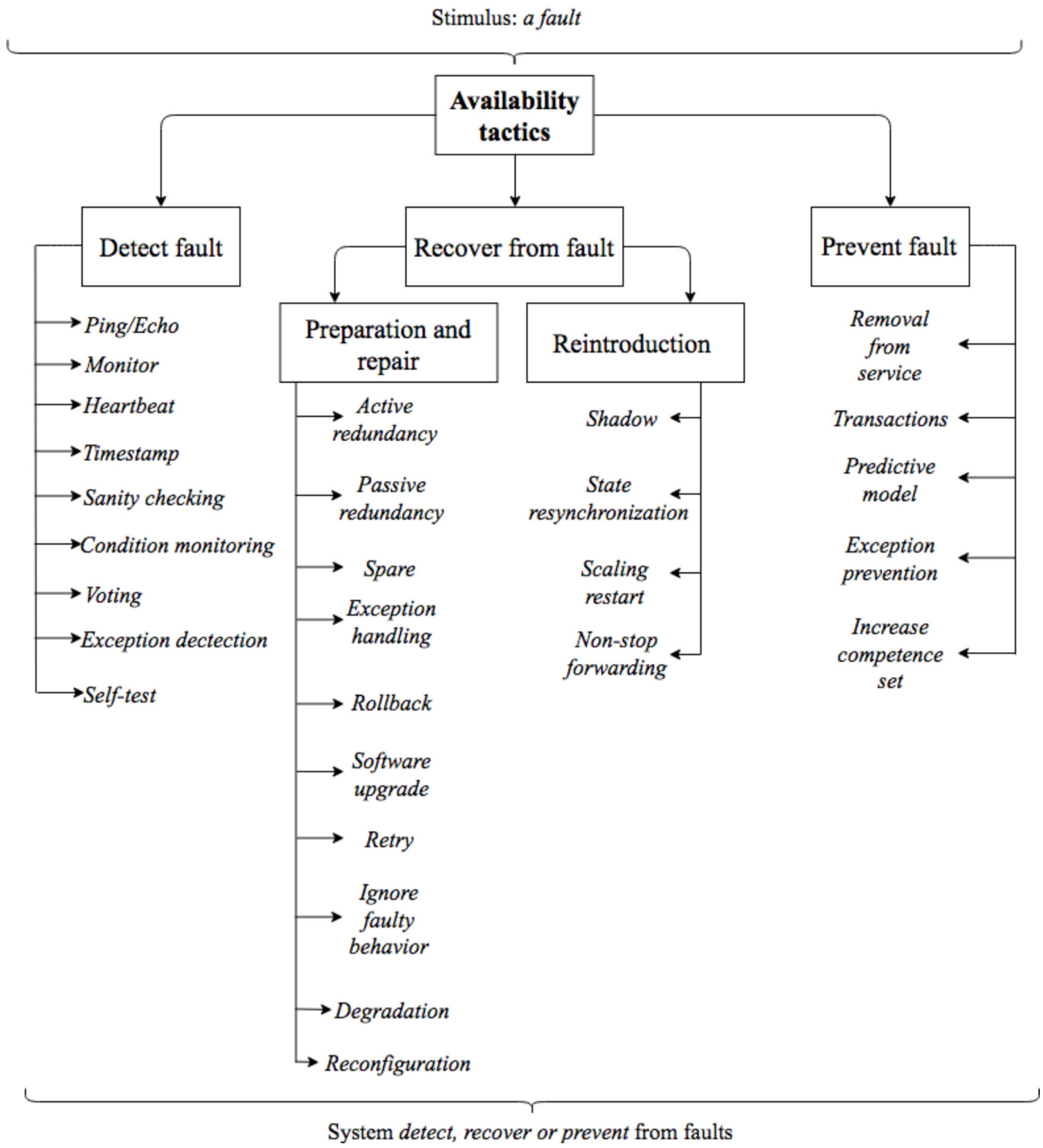


Figure 1.2: Example tactics taxonomy for Availability [21]

According to Harrison *et al.* [70], each architectural tactic is a design option for the architect. For example, using the same Figure 1.2, one of the architectural tactics introduces redundancy to increase the availability of a system. This is one option the architect has to increase availability, but not the only one. Usually, achieving high availability through redundancy implies a simultaneous need for synchronization (to ensure that the redundant copy can be used if the original fails). We see two immediate ramifications of this example.

- *Architectural tactics can refine other architectural tactics.* We identified redundancy as a architectural tactic. As such, it can be refined into active or passive

redundancy. Both types are also architectural tactics. There are further refinements that a designer can employ to make each type of redundancy more concrete. Consequently, for each quality attribute that we discuss in this doctoral thesis, we organize the architectural tactics as a hierarchy.

- *Patterns package architectural tactics.* A pattern that supports availability will likely use both a redundancy architectural tactic and a synchronization architectural tactic. It will also likely use more concrete versions of these architectural tactics.

Regarding this second point, an illustrative example is described. Let us consider a performance attribute scenario such as the following: "A user submits an offer at a time when there are 500 other users online, the offer is successfully processed and less than 20 ms elapses between the receipt of the request and the sending of the updates".

In order to achieve a scenario like this, we can refer to the catalog of architectural tactics. In this case, the main concern is the management of resources, which in this case are the processor and the data corresponding to the state. Two architectural tactics can support us in solving this problem: the *introduction of concurrency* and the *maintenance of multiple copies*. This way, instead of processing each request sequentially and waiting for all client programs to be notified before processing the next offer, we can perform this process concurrently. On the other hand, the maintenance of multiple copies can be achieved, for example, by introducing a cache so that each time the client programs have to be notified, there is no need to access the database to recover the status. In this way, we can see how patterns and architectural tactics combine to solve design problems thus allowing to structure the system and satisfy the quality attributes.

1.1.5. Frameworks

While architectural patterns and tactics are fundamental in the creation of software architectures, these concepts are abstract, and sometimes it is not clear how to connect

them to the technology used for development and deployment. So other key elements are frameworks. These are reusable elements of software that provide generic functionalities focused on solving recurrent issues [74]. More precisely, they automate many processes and also facilitate the whole programming [73]. They are useful, for example, to avoid having to repeat code to perform common functions in a range of tools, such as accessing databases or making calls to the Internet. All these tasks are much easier to perform when working within a framework. They offer many advantages and, in addition, are capable of making even more complex tasks that would be impossible to do even when programming something. Yet, its usefulness depends on the type of program and the context in which it is to be used.

A framework serves to make writing code or developing an application easier. It is something that allows a better organization and control of all the code developed, as well as a possible reuse in the future. Because of this, it guarantees a greater productivity than the most conventional methods and a minimization of the cost by speeding up the working hours devoted to development. On the other hand, its action is something that also affects the errors, minimizing them considerably. In short, it is something that provides general and more than considerable help to the programmer and developer, making their tasks much easier.

Frameworks are typically based on patterns and tactics [32]. An example of this is the Swing framework for creating user interfaces in Java, which applies patterns such as *Model-View-Controller (MVC)*, *Observer* and *Composite*; and incorporates modifiability and usability tactics such as *module generalization* and *separation of user interface elements*[21].

In this doctoral thesis, we included *platforms* within the global concept of *frameworks* [21]. We define platforms a set of frameworks that provides a complete set of functionality for implementing an application in a particular domain (such as MongoDB[1], Kubernetes, Apache Kafka[2], among others). These tools can be composed of one or several frameworks that help to achieve the defined objectives.

[1]https://www.mongodb.com
[2]https://kafka.apache.org

Frameworks can range from small and straightforward (such as ones that provide a set of standard and commonly used data types to a system) to large and sophisticated. A framework amounts to a substantial (in some cases, enormous) piece of reusable software, and it brings with it all of the advantages of reuse: saving time and cost, avoiding a costly design task, encoding domain knowledge, and decreasing the chance of errors from individual implementers coding the same thing differently and erroneously. On the other hand, frameworks are difficult to design and get correct. Adopting a framework means investing in a selection process as well as training, and the framework may not provide all the functionality that it is required. Additionally, the learning curve for a framework is often extremely steep.

1.1.6. Microservices

Traditionally, systems has been designed under the paradigm of *monolithic architectures* where the software is often structured in such a way that all functional aspects are coupled to the same program. In this type of architectures, all the information is hosted on a server, so there is no separation between modules, and, in turn, the different parts of the software are closely coupled. This architectural design creates a long-term problem, as monolithic systems cannot be easily scaled. For this reason, the *microservices architectural style* emerges as an option (see Figure 1.3).

Microservices represent a architectural style for developing and deploying software. In this regard, a microservices architecture is an approach to developing a software application as a series of small, distributed services, each running autonomously and communicating with each other, for example, through HTTP requests to their Application Programming Interfaces (APIs) [85]. Usually, there are a minimum number of services that handle common procedures to all others (such as database access), but each microservice is small and corresponds to a business area of the application. In addition, each service is independent, and its code must be able to be deployed without affecting the others. Each one can even be written in a different programming language, since they only expose the API (a common interface, which does not care

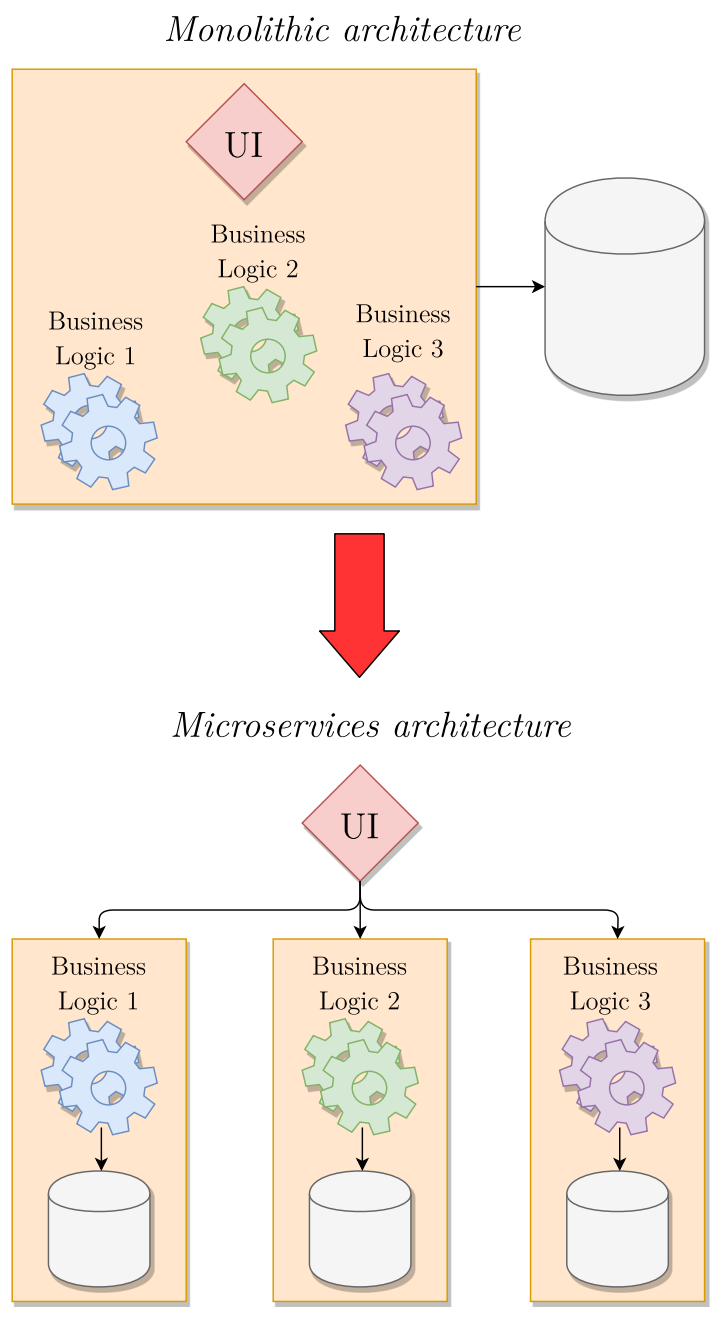


Figure 1.3: Monolithic and microservices architectures

which programming language the microservice is programmed in underneath) to the rest of microservices. There are no rules about how big each microservice has to be, nor about how to divide the application into microservices.

The popularity of microservices is related to the benefits that this architectural style has brought to large companies, such as Amazon and Netflix (see Figure 1.4). These are two of the world's most valuable and rapidly growing enterprises, and their leadership in adopting new architectures and infrastructure models helped teach the world what it means to create native cloud applications. An essential but underappreciated factor is that skilled developers thrive in environments with flexible coupling and personal autonomy.

Figure 1.4: Some companies that are adopting microservices architecture in their business

Additionally, microservices have created more adaptable and flexible IT infrastructures [42]. For example, if it is required to modify only one service, it does not need to alter the rest of the infrastructure. Each service can be deployed and modified without affecting other services or functional aspects of the application. Thanks to containers (such as Docker[3]) and orchestrators (such as Kubernetes[4]), it has been possible to change traditional web servers for virtual containers that are much smaller and more adaptable. It is clear that for leading companies such as Google, Amazon, Netflix, or Uber, with large systems and heavy computing loads, microservices are the best option. But before deciding to implement a microservices architecture, several issues must be considered, for example, the number of end-users, the volume of requests, demand peaks, the size of the company and the equipment available, whether a monolithic architecture is currently in place, whether there is someone on the team with the necessary expertise or whether services must be contracted out, among others.

1.2. Problem and general hypothesis statements

In general, finding the perfect match between frameworks to satisfy NFRs is often a challenge due to the imperfect nature surrounding the matching process. On the one hand, NFRs are continually changing and evolving, and, on the other hand, the

[3]https://www.docker.com
[4]https://kubernetes.io

information in the frameworks, software components, and platforms are *imperfect*. In this regard, Noppen *et al.* [106] describe that, although there are slight differences in the terminologies used, *uncertainty* and *impreciseness* are considered as two subcategories of imperfection. The term uncertainty refers to a transient case, where imperfect information becomes eventually perfect (well known) in due time. On the contrary, imprecise information will always remain imperfect to some degree. The authors also classify imperfection in software design in two ways:

- *Imperfection in contextual information*: An important source of contextual information in software development is derived from the related business context and formulated as stakeholders' requirements. In addition, different kinds of contextual information can be collected during the software development process, such as updates of requirements, skills of the available people, available budget and so on. Impreciseness in contextual information generally manifests itself in non-functional requirements. For example, in the requirement "The system must complete the function F in less than T seconds otherwise the user will be annoyed", it is very difficult to precisely define the threshold value of going from "not annoyed" to "annoyed".

- *Imperfection in design processes*: Software engineers have to deal with many kinds of uncertainties, especially in the early phases of software development. For example, software engineers may be forced to decompose a system into a certain modular structure to manage complexity, already in the early phase of software development. On the other hand, it may be preferable to defer this decision to a later phase when the interactions among components are known. This will allow grouping the densely interacting components into the same module for the purpose of improving performance and cohesion.

Concerning frameworks, several new frameworks are overwhelming the market with no end in sight partly from grassroots open-source movements, and partly from the IT companies who commoditize their software. This growing market of frameworks tends to produce an increment of imprecise frameworks documentation.

Regarding the software development context, developers realized that adding a framework is simple, but removing it again later is difficult. Frameworks that enter in the project codebase, often, stay in. If one doesn't work well or gets obsolete, the entire system and project stuck with it. That said, frameworks selection becomes a fundamental task. Part of framework selection is processing candidates as fast as possible, and only investigate candidates that are worth of time. For some use cases, there are hundreds of viable alternatives to consider. In general, both software developers and architects try to select the best framework(s) that make as much functionality as possible. It is tempting to choose the biggest and most successful framework; but it might be a very poor fit for scenarios related to specific projects.

According to the specialized website Dzone [65] [66], frameworks implement architectural styles (such as microservices) in a domain. Because of this, it is possible to characterize the services offered by these frameworks in terms of the properties they help optimize, such as security, performance, scalability, and others. Therefore, a bad selection of frameworks severely compromises the objectives that a system tries to achieve. This not only affects the business, but it also incurs in building bad architectures. In this regard, the main consequences of bad architecture are [125] [14]:

- *Hard to maintain*: At the beginning, bad architecture causes minor problems. Simple maintenance is difficult. Adding seemingly "basic" features are impossible. A change in one area breaks other parts of the application. The annoyances are fairly minor, but begin to build.
- *Poor Integration*: Applications don't work well with other software. Users will wonder why they have to use multiple applications to access related information. Basic integration is only possible with sloppy workarounds, which adds to the application complexity.
- *Limited scalability*: As the business grows, the applications must support more users and include more features. In some situations, requested features are impossible, while others require even more workarounds. With all of these workarounds, the applications are getting even harder to maintain.

- *Limited options*: As the business continues to grow, another problem arises: What if the business requires a new enterprise software package? What if it needs a new database? The bad architecture severely limits options, and may force the company to choose an inferior option or even stick with the current setup.

- *Short life span*: Finally, it is reached the point where the applications are nearly impossible to maintain. They're so limited, that they can no longer support the business effectively, and have even begun to hold it back. Now, it is necessary to face the most difficult task of all: How to tell management that your relatively new applications need replacing?

In the context of microservices, one of the significant challenges corresponds to the complexity of the design and implementation of microservices-based systems [119]. From an architectural point of view, we have defined four levels of complexity (see Figure 1.5):

- *Level 0*: At this level, there are several challenges related to the definition of each "microservice", that is, the definition of the boundaries of each microservice and its corresponding business rules.

- *Level 1*: This level addresses how to satisfy systemic properties in a microservices-based system. In general, quality attribute scenarios are evaluated at this level.

- *Level 2*: This level discusses what architectural patterns and design decisions should be made in the microservices architecture.

- *Level 3*: Once the microservices architecture has been designed, the frameworks and platforms that support the established design are discussed.

On the other hand, in microservices architectures, each service is more straightforward, but the entire system is more complex, given many platforms, frameworks, and technologies that interact with each other. This requires development teams to maintain

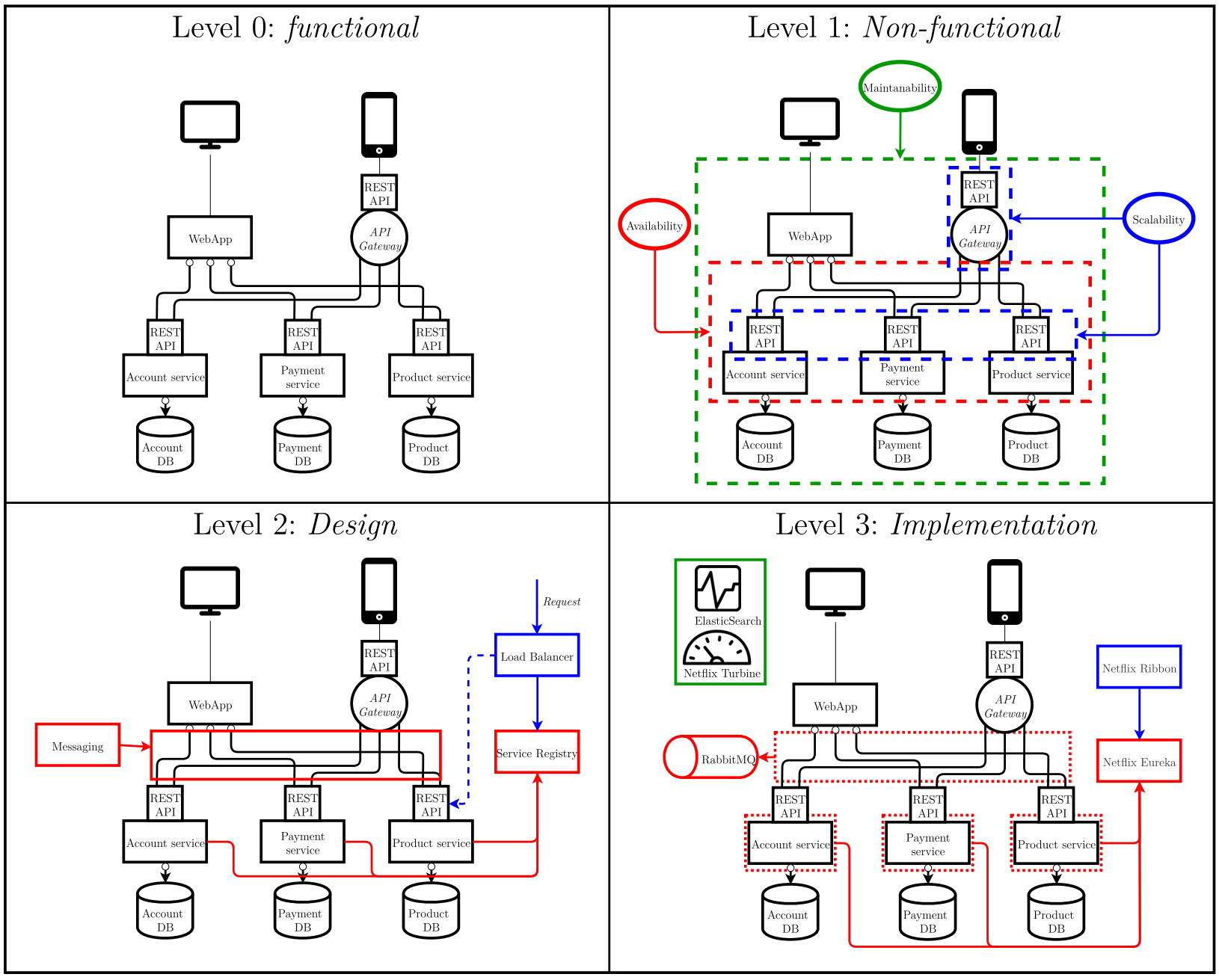


Figure 1.5: Different levels of complexity in the design of microservices architectures

an excellent understanding of the entire application architecture as well as the selected architectural decisions. It also requires permanent verification of the application architecture to ensure that all microservices are working well together.

That said, framework and platform selection becomes one of the most critical activities in the development of microservices-based systems. If this selection is not related to the main NFRs of the project, a series of critical errors and shortcomings can be incurred that can compromise the entire microservices-based system. Some of these shortcomings are:

- Choosing a different technological stack for different components leads to non-uniform application design and architecture.

- Endless documentation in the form of updated schemas and interface documents for every individual component application.
- The costs of maintenance, operational costs, and production monitoring are much higher, and the latter also suffers from a dearth of available tools.
- Microservices, when implemented incorrectly, can make poorly written applications even more dysfunctional.

Additionally, microservices involve many more moving parts than traditional applications, requiring enormous effort, careful planning and strategically applied automation to ensure that communication, monitoring, testing and deployment processes run smoothly. Consequently, if the selection of frameworks and platforms is not adequate, microservices-based systems tend to fail both in terms of NFRs' and stakeholders' satisfaction. Therefore, the main problems that will be addressed in this doctoral thesis are:

Main research problem

There are not enough techniques that allow architects to systematically evaluate and compare frameworks (with imperfect information) to satisfy NFRs in microservice architectures.

Sub research problem 1

There is no explicit description of which recurring solutions (architectural patterns) are implemented by frameworks and platforms widely in microservices-based systems.

Sub research problem 2

It is unclear what design decisions (architectural tactics) are used to address different quality attribute concerns in microservices-based systems.

Given the background related to the problem discussed by this doctoral thesis, the description of the general hypothesis addresses the main components that have been mentioned in the previous sections, which are: architectural tactics, architectural patterns and frameworks. Therefore, the general hypothesis and corresponding sub-hypothesis of this thesis are the following:

General hypothesis

The systematic analysis of architectural patterns, architectural tactics, and frameworks outlined in imprecise-information-based multidimensional catalogs improves the architect's capability for collect, refine, and evaluate solutions for addressing NFRs in microservices-based systems.

Sub - hypothesis 1

The intersection of imprecise-information-based multidimensional catalogues generates acceptable solutions close to the genuine solutions that an architect selects and evaluates to address NFRs.

Sub - hypothesis 2

The evaluation and selection of frameworks and platforms, supported by architectural patterns and architectural tactics, requires less effort from stakeholders.

Sub - hypothesis 3

Frameworks and platforms supported by architectural patterns and architectural tactics are better suited to address NFRs.

1.3. Proposed solution: μAzimut

This doctoral thesis propose μAzimut (see Figure 1.6), a novel technique which uses architectural knowledge represented by multi-dimensional catalogs of quality properties, architectural tactics, architectural patterns, and frameworks characterizations to

support the architect in the analysis, evaluation, and comparison of framework assemblies for the purpose of satisfying NFRs [91] in microservices-based systems. The technique deals with the fuzziness of information using imprecise "characterization" of architectural tactics, architectural patterns and frameworks. In the following sections, we describe each μAzimut components.

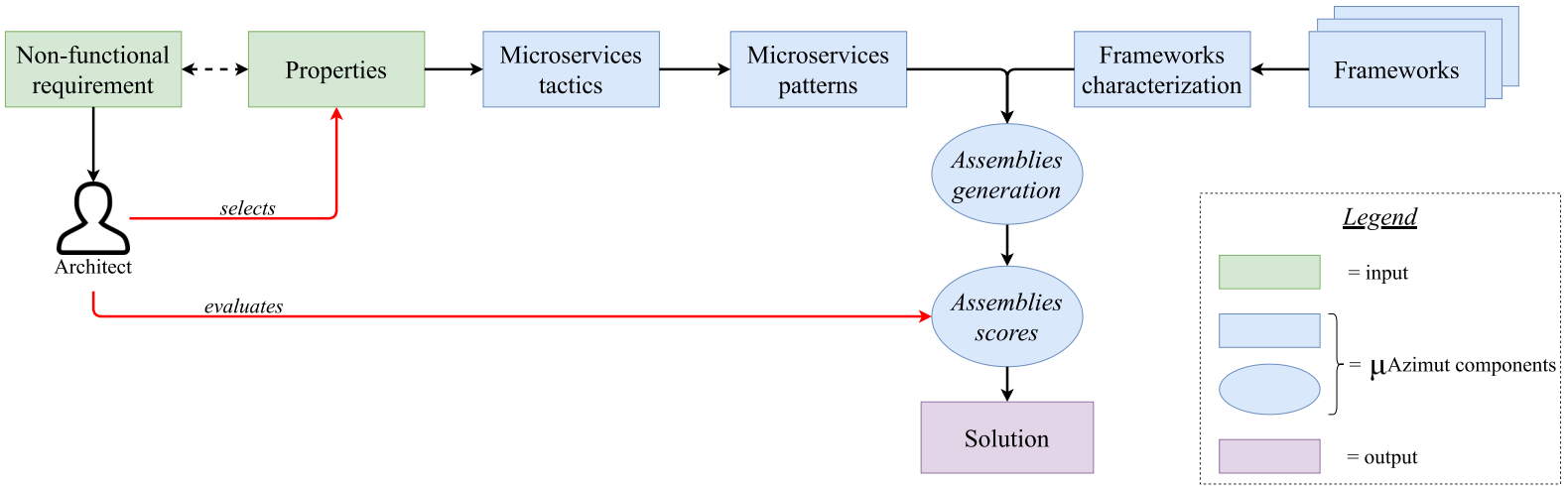


Figure 1.6: The μAzimut approach

1.3.1. Input

μAzimut uses **properties** which characterize quality attributes of microservices development. The primary purpose of properties is to encapsulate specific characteristics that represent quality attributes. Furthermore, properties allow the architect to focus on particular perspectives of an NFR rather than analyzing it as a whole.

1.3.2. Processing

μAzimut uses **architectural tactics** for microservices ("microservices tactics" in Figure 8.4) as a method of addressing the properties that the architect selects for each NFR. Likewise, μAzimut utilizes **architectural patterns** for microservices ("microservices patterns" in Figure 8.4) [128] [92] related to microservices tactics. μAzimut assumes that a given architectural pattern may be related to architectural tactics for the same quality concern or architectural tactics across several quality concerns; similarly, a given architectural tactic may be related by several architectural patterns. Along

with architectural tactics and patterns, μAzimut uses frameworks to translate generic functionalities in order to satisfy quality attributes.

Aiming at obtaining frameworks assemblies, μAzimut uses **catalogues** that store microservices-related architects' knowledge about microservices patterns, microservices tactics, and frameworks, as well as rules of satisfaction among them. Hence, they are the key to reusing information about improving the quality of knowledge concerning design spaces and frameworks and to support the architect in the process of exploring these design spaces. In practical deployment contexts, the catalog preparators might not know or not be certain whether a microservice pattern supports a certain microservice tactic. Furthermore, NFRs may be imprecise, incomplete, or uncertain, and consequently, it would be difficult to know which microservices pattern fits NFRs. The catalogues considers **credibility degrees**, which correspond to 1 (*does support*), 0.6 (*probably supports*), 0.3 (*possibly does not support*), and 0 (*does not support*).

1.3.3. Output

μAzimut uses credibility degrees to intersect the dimensions of architectural tactics/architectural patterns and architectural patterns/frameworks catalogs in order to compute a **support score** that ranks framework assemblies based on the degree of importance that the architect determines for each NFR. The support score counts dimensions in favor of the statement "framework assemblies satisfies the tactic t". Likewise, the support score measures the credible arguments for a given framework or assembly as a solution to a given tactic, where "credible" means having a credibility index above the credibility threshold.

1.4. Research objectives

The main research objective that characterize this doctoral thesis is the following

Main research objective
Propose a technique that generates frameworks assemblies to support the satisfaction of NFRs in microservices-based systems using architectural patterns, architectural tactics and imprecise frameworks information.

Additionally, this objective can be disseminated into the following sub-objectives:

Sub research objetive 1
Explore the state of the art and practice regarding architectural patterns, architectural tactics and quality attributes used by microservices architectures.

μAzimut uses, as a source of knowledge, architectural patterns, architectural tactics and quality attributes that have been referenced in both academic and grey literature. This exploration allows establishing the bases that support the multidimensional catalogs that μAzimut uses to generate framework assemblies.

Sub research objetive 2
Define, characterize, and populate multidimensional catalogs of patterns, tactics, and frameworks with credibility degrees.

This objective aims to create datasets that represents the relationship between (i) tactics and patterns and (ii) patterns and frameworks using incomplete and imprecise information. This type of information is characterized by credibility degrees.

Sub research objetive 3
Validate, through empirical studies, the μAzimut technique.

Two case studies and one experiment are used to validate μAzimut. In the first case study, the technique is evaluated by comparing the μAzimut-generated framework assemblies and an architect-generated assembly in a microservices-based system. In the second case study, μAzimut is evaluated by adding the stakeholders in the decision of the microservices architecture design of an Ambient-Assisted Living system. Finally,

the experiment consists of evaluating if architectural tactics influence the final result of the technique.

1.5. Research questions

It is clear that frameworks selection using imprecise information becomes a latent challenge when it comes to software (and eventually microservices) architecture design. Converging concepts such as NFRs, quality attributes, architectural patterns, architectural tactics and frameworks to achieve architecture designs that meet the needs of stakeholders requires addressing the problem in different aspects. Therefore, the main research questions (RQs) of this doctoral thesis are:

RQ1
Which is the current state of the art and practice concerning architectural patterns for microservices?

The goal of RQ is to explore, detect, and describe the architectural patterns reported in the context of microservices. The characterization of architectural patterns allows understanding which architectural problems recurrently emerge in the development and deployment of microservices.

RQ2
Which is the current state of the art and practice concerning architectural tactics for microservices?

This RQ invites for exploring, detecting, and describing architectural tactics reported in the context of microservices. Current architectural tactics catalogues provide a set of design decisions that enable quality attributes to be addressed. Yet, new functionalities in microservice architectures allow to define tentative new architectural tactics.

RQ3
Which are the emerging properties that characterize quality attributes in the context of microservices architectures?

There are transversal properties that define quality attributes for any system. Yet, since microservices architectures use different types of architectural elements (such as networks, servers, infrastructure, among others), new properties may emerge. That said, this RQ aims to identify properties for the quality attributes used by μAzimut.

RQ4
Can a systematic technique, using architectural artifacts (such as properties, architectural patterns, and architectural tactics) as well as imprecise information, be used as a method of supporting decision-making in the design of microservice architectures?

This RQ aims to evaluate μAzimut using two empirical strategies: case studies and an experiment. The purpose of conducting these empirical studies is to assess whether the results generated by the technique are sufficient for an architect to use as a support in making decisions on the design of microservice architectures.

1.6. Research contributions

The main contributions of this doctoral thesis are described below:

- A technique that uses (i) microservices patterns, (ii) microservices tactics, and (iii) frameworks to help the architect to evaluate framework assemblies to satisfy NFRs.
- A technique for recovering microservice architectures, which illustrates a (i) pattern-based view, (ii) functional microservices and (iii) infrastructure microservices.

- The description of multidimensional catalogues (represented by datasets) composed of microservices patterns, microservices tactics, and frameworks that represent one of the first efforts to characterize and describe the architectural knowledge that microservices architectures are generating.

- The characterization and description of properties that represent quality attributes related to microservices architectures.

- New architectural tactics used in microservice architectures that complement the current catalog of availability and scalability tactics.

1.7. Published work

Based on the work performed in this doctoral thesis, the following articles have been published:

Articles directly related to the doctoral thesis:

- **Gastón Márquez**, Felipe Osses, Hernán Astudillo: *Review of Architectural Patterns and Tactics for Microservices in Academic and Industrial Literature.* IEEE Latin America Transactions, 16(9), 2321-2327, 2017. Doi: `10.1109/TLA.2018.8789551` (WoS)

- **Gastón Márquez**, Yoslandy Lazo, Hernán Astudillo:*Evaluating Frameworks Assemblies In Microservices-based Systems Using Imperfect Information.* IEEE International Conference on Software Architecture Companion (ICSA-C), Brazil, 2020. Doi:`10.1109/ICSA-C50368.2020.00049`

- **Gastón Márquez**, Jacopo Soldani, Francisco Ponce, Hernán Astudillo: *Frameworks and High-Availability in Microservices: An Industrial Survey.* CIbSE, *In press*, Brazil, 2020.

- **Gastón Márquez** and Hernán Astudillo: *Identifying Availability Tactics to Support Security Architectural Design of Microservice-based Systems*, Proceedings of

the 13th European Conference on Software Architecture-Volume 2, pp. 123-129, Paris, France, 2019. Dou: https://doi.org/10.1145/3344948.3344996

- Anelis Pereira, **Gastón Márquez**, Hernán Astudillo, Eduardo B. Fernández: *Security Mechanisms Used in Microservices-Based Systems: A Systematic Mapping In press.* XLV Latin American Computing Conference, Panama City, Panama, 2019. Doi: 10.1109/CLEI47609.2019.235060

- **Gastón Márquez**, Hernán Astudillo: *Actual Use of Architecture Patterns in Microservices-based Open Source Projects.* 25th Asia-Pacific Software Engineering Conference (APSEC 2018), 31-40. Nara, Japan. Doi: 10.1109/APSEC.2018.00017

- **Gastón Márquez**, Mónica M. Villegas, Hernán Astudillo: *An Empirical Study of Scalability Frameworks in Open Source Microservices-based Systems.* 37th International Conference of the Chilean Computer Science Society (SCCC) (pp. 1-8).10.1109/SCCC.2018.8705256

- **Gastón Márquez**, Mónica M. Villegas, Hernán Astudillo: *A pattern language for scalable microservices-based systems.* ECSA (Companion) 2018: 24:1-24:7, Madrid, Spain. Doi: https://doi.org/10.1145/3241403.3241429

- Felipe Osses, **Gastón Márquez**, Hernán Astudillo: *Exploration of academic and industrial evidence about architectural tactics and patterns in microservices.* ICSE (Companion Volume) 2018: 256-257. Gothenburg, Sweden. Doi: https://doi.org/10.1145/3183440.3194958

- Felipe Osses, **Gastón Márquez**, Hernán Astudillo: *An Exploratory Study of Academic Architectural Tactics and Patterns in Microservices: A systematic literature review.* CIbSE 2018. Bogota, Colombia.

- **Gastón Márquez**, Hernán Astudillo: *Helping Novice Architects to Manage Architectural Technical Debt in Microservices Architecture.* XIII Jornadas Iberoamericanas de Ingeniería de Software e Ingeniería del Conocimiento. Copiapo, Chile.

- Diego Gatica, **Gastón Márquez**, Hernán Astudillo: *Systematic Selection of Software Components through Architectural Tactics. Is a Relationship between Tactics and NFRs Possible?* CIbSE 2017: 183-195. Buenos Aires, Argentina.

- **Gastón Márquez**: *Selection of software components from business objectives scenarios through architectural tactics.* ICSE (Companion Volume) 2017: 441-444. Buenos Aires, Argentina. Doi: 10.1109/ICSE-C.2017.35

- **Gastón Márquez**, Hernán Astudillo: *Selecting components assemblies from non-functional requirements through tactics and scenarios.* SCCC 2016: 1-11. Valparaiso, Chile. Doi: 10.1109/SCCC.2016.7836020

New research approaches emerged from this doctoral thesis:

- **Gastón Márquez**, Carla Taramasco: *Using Dissemination and Implementation Strategies to Evaluate Requirement Elicitation Guidelines: A Case Study in a Bed Management System.* IEEE Access, p.p 145787 – 145802, 2020. Doi: 10.1109/ACCESS.2020.3015144 (Wos)

- **Gastón Márquez**, Hernán Astudillo, Carla Taramasco: *Security in Telehealth Systems from a Software Engineering Viewpoint: A Systematic Mapping Study.* IEEE Access, p.p 10933 - 10950, 2020. Doi: 10.1109/ACCESS.2020.2964988 (Wos)

- Vincent Zalc, Matthias Pideri, Dan Istrate, Carla Taramasco, **Gastón Márquez**: IoT Data Assembly and Recording, Smart Health International Conference (SHeIC), 2020, Troyes, France. (*In press*).

- **Gastón Márquez**, Carla Taramasco, Hernán Astudillo: *Defining Security Metrics To Evaluate Electronic Health Records Systems: A Case Study in Chile*, IEEE International Conference on Software Architecture Companion (ICSA-C), p.p 173-180, 2020, Salvador, Brazil. Doi: 10.1109/ICSA-C50368.2020.00038

- **Gastón Márquez**, Hernán Astudillo, Carla Taramasco: *Exploring Security Issues in Telehealth Systems*, IEEE/ACM 1st International Workshop on Software

Engineering for Healthcare (SEH), p.p. 65-72, 2019. Doi: 10.1109/SEH.2019.00019

- Juan Brito, Felipe Beroiza, **Gastón Márquez**, Marcello Visconti, and Hernán Astudillo: *Evaluating Impact of Experience in Architectural Design Decision-Making Techniques: An Experimental Study*, 38th International Conference of the Chilean Computer Science Society (SCCC), 2019, Concepción, Chile. Doi: 10.1109/SCCC49216.2019.8966395

- Francisco Ponce, **Gastón Márquez**, and Hernán Astudillo: *Migrating from monolithic architecture to microservices: A Rapid Review*, 38th International Conference of the Chilean Computer Science Society (SCCC) 2019, Concepción, Chile. Doi: 10.1109/SCCC49216.2019.8966423

- Felipe Osses, **Gastón Márquez**, Mónica M. Villegas, Cristian Orellana, Marcello Visconti, Hernán Astudillo: *Security tactics selection poker (TaSPeR): a card game to select security tactics to satisfy security requirements.* ECSA (Companion) 2018: 54:1-54:7. Madrid, Spain. https://doi.org/10.1145/3241403.3241459

- Felipe Osses, **Gastón Márquez**, Cristian Orellana, Hernán Astudillo: *Towards the selection of security tactics based on non-functional requirements: Security tactic planning poker.* SCCC 2017: 1-8. Arica, Chile. 10.1109/SCCC.2017.8405144

Master thesis related to this doctoral thesis:

- Felipe Osses: *Técnica para seleccionar tácticas de arquitectura de software en forma consensuada: Tactics Selection Poker (TaSPeR)*, UTFSM.

1.8. Thesis structure

The doctoral thesis is structured in five main parts (see Figure 1.7). Each part and its corresponding chapters are described below:

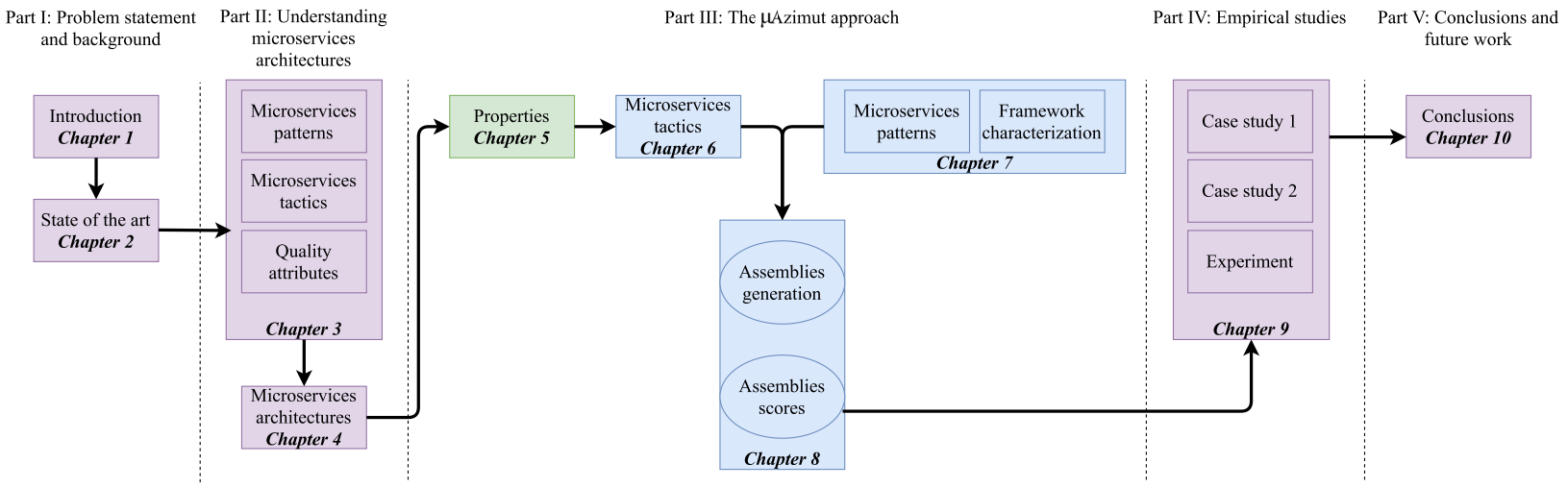


Figure 1.7: Thesis structure

- Part I: Chapter 1 introduces the main concepts surrounding this doctoral thesis as well as the problem statement, general hypothesis description, research objectives and contributions; Chapter 2 describes the state of the art.
- Part II: Chapter 3 shows an overview of microservices patterns, microservices tactics, and quality attributes in academic and grey literature; Chapter 4 illustrates the Pekenum, a technique that allows recovering microservices architectures. This technique helps to understand how a microservices architecture is structured, how microservices communicate with each other and the objectives of frameworks in microservices-based systems.
- Part III: Chapter 5 details the procedure for defining quality attribute properties; Chapter 6 describes the microservices tactics used in the technique; Chapter 7 describes the state of practice of microservices patterns; Chapter 8 illustrates μAzimut.
- Part IV: This part describes the case studies and the experimental study conducted to evaluate μAzimut (Chapter 9).

- Part V: Summarizes the research done in this doctoral thesis, mentions main findings, and describes future work (Chapter 10).

Chapter 2

State of the Art

The proposal of this thesis is based on concepts that have been used in several software architecture aspects over the years. Concepts such as architectural patterns, architectural tactics, and frameworks have been used for different purposes, in order to build the body of knowledge of software architecture and design. Nevertheless, microservices have incited to update these concepts and, in turn, to propose new methods and techniques. In this chapter, we discussed the state of the art of studies that address the problem of selecting frameworks and platforms to design and implement software architectures. Given that the selection of frameworks, platforms or software components is a problem that can have diverse edges, we focus on studies that address, to a certain extent, the main concepts that surround this doctoral thesis, which are (i) quality attributes, (ii) architectural patterns, (iii) architectural tactics, (iv) frameworks/platforms, (v) imprecise, imperfect or uncertain information and (vi) microservices.

Astudillo *et al.* [10] present an evaluation approach to support software architects in exploring design spaces, by enabling evaluation and comparison of whole component assemblies. The approach deals with the fuzziness of information using incomplete, uncertain or imprecise "characterizations" of policies, mechanisms or components and allowing the evaluation of assemblies when better information is obtained.

The approach support architects in generating component assemblies for a given set of

requirements that rely on multi-dimensional catalogues of architectural policies, mechanisms and components, and derivation rules among constructs of these levels. The approach's goal is enabling architects to gather component characterizations, but imprecise and incomplete, and derive component assemblies for the specific requirements at hand.

The authors mention that architects often may reason about the overall solution properties using architectural policies, and later refine them (perhaps from existing policy catalogs) into artifacts and concepts that serve as inputs to software designers and developers, such as component models, detailed code design, standards, protocols, or even code itself. Thus, architects define policies for specific architectural concerns and identify alternative mechanisms to implement such policies. For example, an availability concern may be addressed by fault-tolerance policies (such as master-slave replication or active replication) and a security concern may be addressed by access control policies (such as identification-, authorization- or authentication-based).

Each reification yields more concrete artifacts; thus, architectural decisions drive a process of successive reifications of NFRs that end with implementations of mechanisms that do satisfy these NFRs. To characterize such reifications, the authors used a vocabulary taken from the distributed systems community, duly adapted to the software architecture context:

- *Architectural Policies*: The first reification from NFRs to architectural concepts. Architectural policies can be characterized through specific concern dimensions that allow describing NFRs with more details.
- *Architectural Mechanisms*: The constructs that satisfy architectural policies. Different mechanisms can satisfy the same architectural policy, and the differences between mechanisms is the way in which they provide certain dimensions.

López *et al.* [86] present an approach to extend Model-Driven Architecture (MDA) through the concepts of architectural policies and mechanisms (same concepts used by Astudillo *et al.* [10]) in order to select software components. The key ideas are

representations of NFRs through architectural concerns using architectural policies, systematic reification of policies into mechanisms, and multi-dimensional description of components as implementations of mechanisms.

In the approach (see Figure 2.1), the authors distinguished two Platform-Independent Models (PIM) levels, Platform-Independent Architecture Model ($PIAM$) and Platform-Independent Model for Concern v (PIM^v). The $PIAM$ characterizes platform-independent architectural policies and their dimensions, and the PIM^v characterize platform-independent mechanisms satisfying the required architectural policies. The $PIAM$'s elements are transformed to PIM's and PIM^v's using the Architectural Reification Model (ARM), which provides guidelines to go from architectural policies to architectural mechanisms. The ARM indicates which combinations satisfy each policy, and may have rules about mechanism combinations (e.g. potential restrictions). The transformation process determines feasible sets of mechanisms that provide specific architectural policies, and propose them to the architect for validation or correction; supported mechanisms are presented according to the policies they satisfy (possibly several), and are grouped into PIM^v for each concern v.

Petersen *et al.* [113] investigate how practitioners choose among the component sourcing options concerning stakeholders involved, criteria used for making the decision, and the decision-making method used. Moreover, they focused on determining the decision-making outcome with respect to the chosen component sourcing options, the effort invested in decision making, the criteria being most important in the final decision, and the evaluation of the perceived success degree of the choice.

The authors realized in their study that the most frequently considered options to develop software intensive systems were In-house, COTS, and Outsourcing. In-house development was considered in 17 of 22 cases, which indicates that the most common trade-off is to either make (in-house) or obtain the components externally (COTS, Outsource, OSS or Services). In-house development was most commonly traded off with Outsourcing and COTS. Furthermore, in most cases, the management initiated the decision-making process.

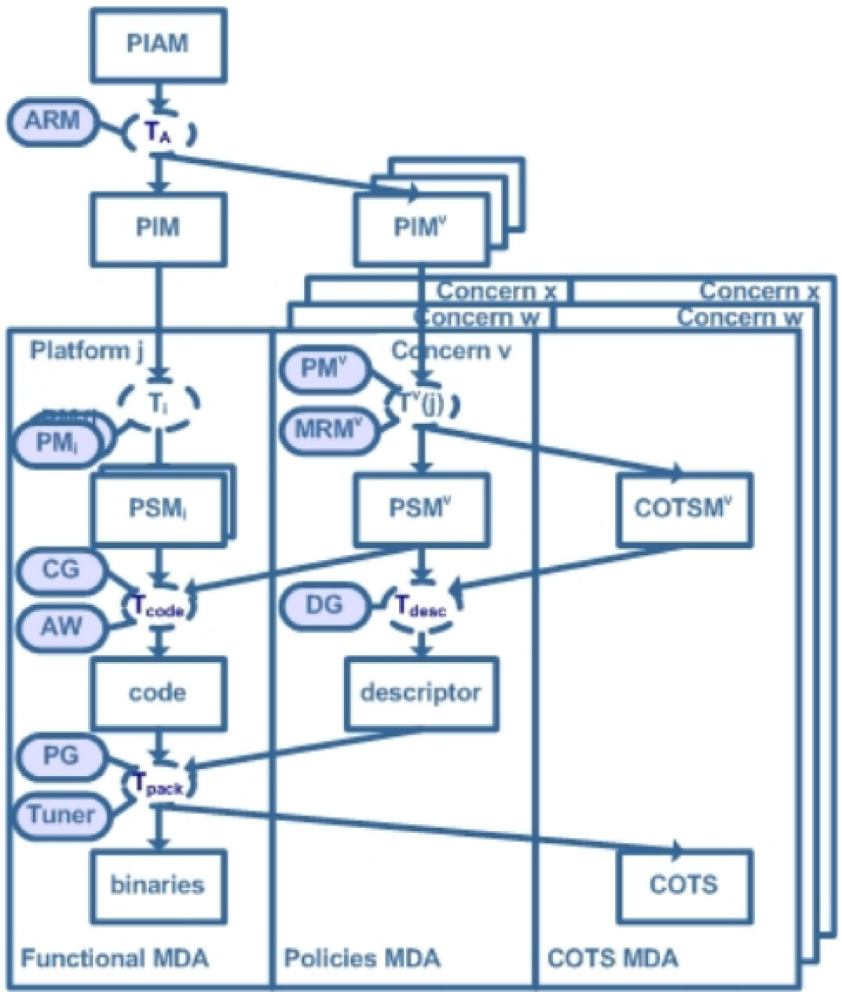


Figure 2.1: Component selection using MDA

Regarding stakeholders, the identification of the need for making the decision mostly originates from a single role and is mostly decided by a single role. A group of stakeholders does decision preparation. The most commonly involved roles in preparing the decision were software management, software design/construction, external decision support, and sales. The decision for the CSO primarily lies with the managers. Decision-makers are commonly involved in the initiation and preparation activities.

The authors conclude that cost, performance, and system reliability stand out as the three most commonly considered criteria to choose component sourcing options. When reliability is considered a criterion, it is highly likely that it is considered along with performance. Similarly, it is highly likely that performance and reliability are also considered when the cost is considered but these results may be biased. Commonly more than two criteria are considered; thus, solutions aiding in decision making have to provide opportunities for multi-criteria analysis and decision support.

Kwong *et al.* [82] formulate an optimization model of software components selection for

components development. The model has two objectives: maximizing the functional performance of the component selection and maximizing the cohesion and minimizing the coupling of software modules (see Figure 2.2). Additionally, the authors introduce a genetic algorithm to solve the optimization model for determining the optimal selection of software components.

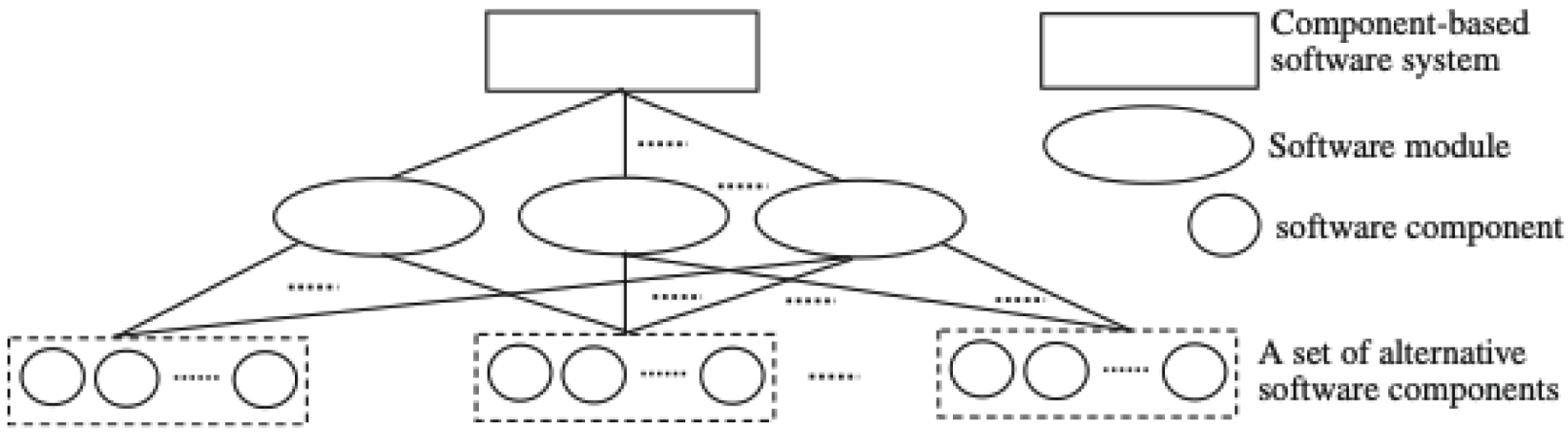


Figure 2.2: Formulation of a CBSS

According to the authors, in comparison with other studies related to component-based development, a modified way of measuring the cohesion and coupling of software modules and considering the function ratings of various software components for component-based software system development are introduced in their study. Based on that proposed methodology, an optimization model can be formulated to perform the software components' selection for the software modules of a component-based software system. An example of financial system design was used to illustrate the proposed methodology in the study. However, the proposed methodology involves some subjective judgments from software development teams, such as the determination of the scores of interaction and the function ratings. In this regard, the fuzzy set theory could be introduced to deal with the fuzziness caused by subjective judgments. In addition, the information theory and the complexity theory could be explored to measure the cohesion and the coupling of software modules.

Tang *et al.* [131] discussed a practical software engineering problem, i.e., a software developer undertakes multiple development tasks of enterprise application concurrently

(see Figure 2.3). The authors introduce an optimization model in order to assist software developers in selecting components. The proposed model considers the components' compatibility and costs simultaneously.

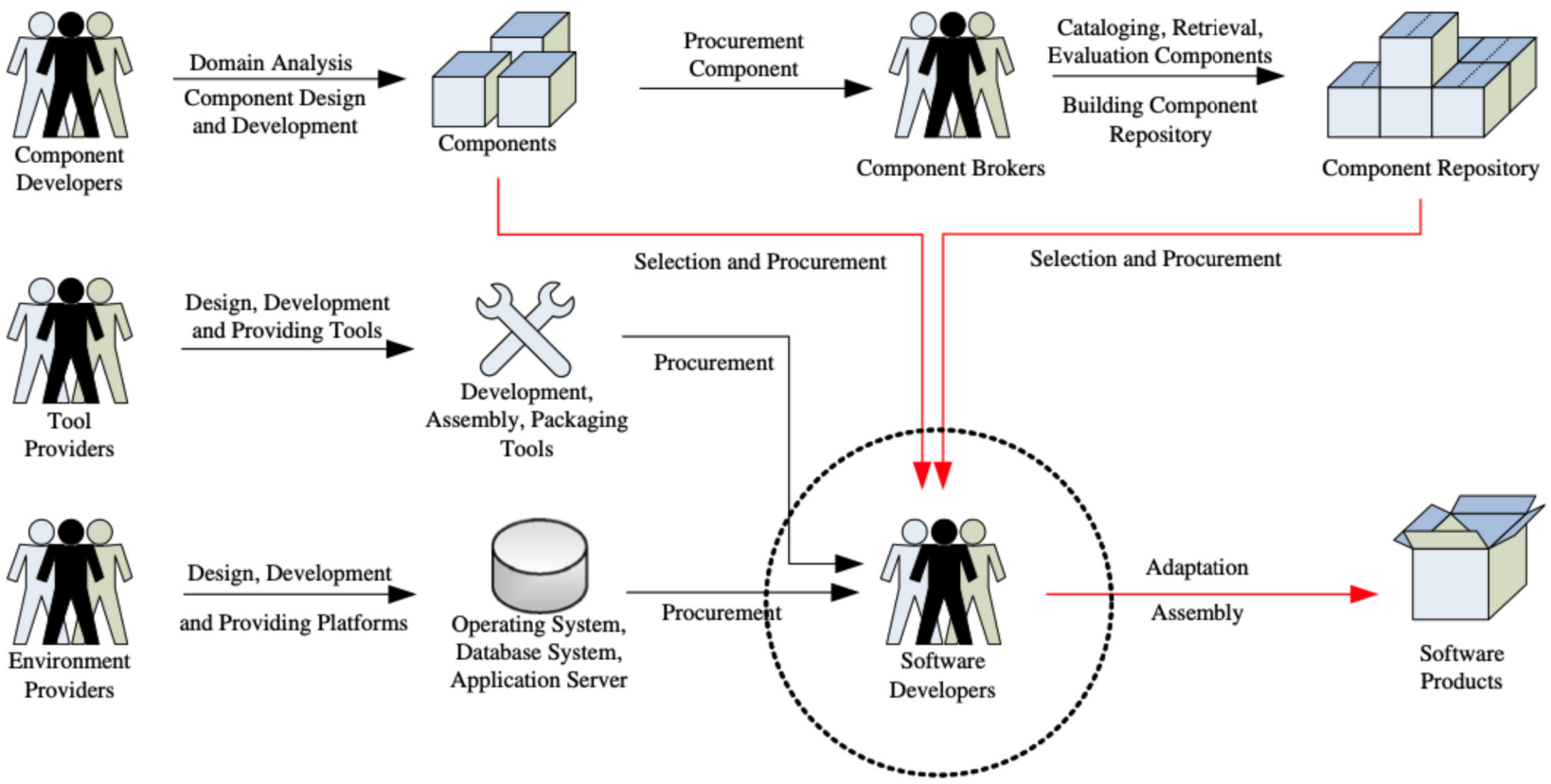


Figure 2.3: A flowchart of CBSD process

In the article, several management implications are provided through experiments and sensitive analysis. In the first experiment, a simulation case is used to illustrate and discuss how to use the proposed model to estimate the different effect of component reusability and the number of available components upon average development cost. In can be concluded that average cost decrease while the effect of component reusability ($P(r)$) is increasing and the number of available components is decreasing simultaneously, vice versa, cost will increase. In the second experiment, the influence upon average cost comes from the ratio of adaptation cost to procurement cost and the number of available components is discussed. The influence of ratio of adaptation cost to procurement cost $R(a/p)$ of available components (ACs) has a more substantial effect than the volume of ACs in a market. In the third case, how to estimate average cost while the number of compatibility set and available components changing is explained.

The authors established additionally that the software product development process is a complex one involving many factors. Functional or non-functional requirements, e.g.,

development times cycle, quality, reliability, etc. should be considered simultaneously, as objectives or constraints, which will be addressed soon as an extension of this paper.

Ernst *et al.* [49] introduce a reasoning framework, approach, and tool suite for rapidly identifying suitable software components (e.g., libraries, APIs, frameworks) within a continuous software engineering setting. The authors aim to increase both speed and confidence, key goals for most procurement. The author's approach is to provide component scorecards based on project health measures and quality attribute indicators that enable the partially- or fully automated assessment of external components with greater developer confidence, supporting rapid software delivery. To do this, the authors applied existing automated analysis techniques and tools (e.g., code and software-project repository analyses) mapping extracted information to common quality attribute indicators. Additionally, the authors show how the creation of aggregation functions aids the selection process. The purpose of aggregation functions is to enable not only indicator-specific scores but also aggregate scores so that components can be easily compared using a single value.

The author discussed that the approach has promise as a rapid triage for new and updated components before detailed analysis is undertaken. This is especially important given the many components and frameworks most projects use and the pace with which these components change. Some improvements that will be added to the approach are:

- Integrate the measurement tools into a continuous integration platform. Then upgrade the components and automatically re-run the measurement tools, testing our continuous evaluation approach.
- Create statistical models of the aggregation approach.
- Interview architects to assess the suitability of the scorecard approach in their context and how the results match their intuitions and needs. Questions that authors will ask include their current approach to assessment, stopping criteria for assessments, and level of confidence in the selection.

Cervantes *et al.* [31] investigate the criteria used by practicing software architects

in selecting security frameworks. They also propose how information associated with some of the criteria that are important to architects can be obtained manually or in an automated way from online sources such as GitHub.

More precisely, the authors performed a study of criteria that are useful to architects in the selection of application frameworks. The study has allowed to understand which criteria are important to practitioners and how data associated with some of these criteria can be gathered from online sources.

The authors mentioned that architects generally select frameworks as part of the design process, and selecting an appropriate framework is an important design decision which may be costly to revert. The goal of the author's research is to help software architects make better and more informed design decisions, particularly regarding the selection of application frameworks during architectural design. In their study they identify a list of criteria that can be used in the selection of software security frameworks. Using this list of criteria they surveyed practicing software architects to understand how important these criteria are to them. They then investigate how data associated with a subset of these criteria can be obtained from online sources such as GitHub to provide relevant information to the architects.

While the goals of this study are rather narrow (looking at the decisions affecting adoption of security frameworks for Java applications) the methodology that the authors have applied is not specific to either Java or security. The authors state that that the reasoning, criteria, and tools that we they used to collect data in this paper are generic. Thus they claim that this research represents a first step towards creating scorecards for third-party components, supporting the rapid selection of such components.

Lenarduzzi *et al.* [84] describe a work-in-process approach to support developers in selecting alternative components in case the component is not working anymore or future versions cannot be used in the future. The approach is composed by the following steps:

- Component Identification: In the first step, it is defined a selection process to

support developers in identifying the most suitable components and possible alternatives.

- Exploratory Survey: To understand how developers select components for their microservices-based systems, it will conduct an exploratory survey among different companies. The goal is to understand the different motivations behind and factors considered for the components adoption.

- Component adoption model: Based on the results obtained in the literature review, and based on the factors highlighted in the survey carried out in the second step , it will propose an initial component adoption model. The model will support developers in understanding in which context using container images is more effective than (a) customizing components, (b) deploying them in containers and (c) identify strategies for the replacement of existing components in case the current component breaks the system.

Di Noia *et al.* [44] they propose to structure the knowledge associated with NFRs via a Fuzzy Ontology. The authors describe a declarative approach that makes it possible to represent and maintain the architectural knowledge by keeping the flexibility and fuzziness of modeling thanks to the use of fuzzy concepts such as high, low, fair, and others. More precisely, the authors present a decision support system based on (i) a Fuzzy OWL 2 ontology that encodes 109 design patterns, 28 pattern families and 37 NFRs and their mutual relations, (ii) a novel reasoning service to retrieve a ranked list of pattern sets able to satisfy the Non-Functional Requirements within a system specification.

The study allowed to build a theoretical framework of the authors' approach by defining a fuzzy ontology in which they catalog the NFRs, the families, and the patterns according to catalogs and categorizations found in the state-of-the-art descriptions. By defining an algorithm called "Covering Answer Set" for retrieval of patterns from the fuzzy ontology, the authors solved the problem of facilitating the decision-making problem in architectural design.

Watada *et al.* [141] conducted a comprehensive experimental study to compare the

performance of VMs, containers, and unikernels in terms of CPU utilization, memory footprints, network bandwidth, execution time, and technological maturity using standard benchmarks and observed containers to deliver satisfactory performance in almost all aspects. The experiment results about containerization provide many promising features like super lightweight, faster spin-up/down, efficient energy and resource utilization, impressive workload distribution capabilities, achieving server consolidation, and many more, but at the same time, it has few major problems such as weaker isolation, higher chance of container sprawl, lack of capable tools for container orchestration and cross-platform supports and container portability limitations. On the other hand, VMs provide strong isolation; it has a reach set of available tools to provide cross-platform supports and management, offer better portability than containers though bigger memory footprints, and slow booting remains their major issues. Unikernels provide VM-like isolation with significantly small footprints along with extremely fast booting.

Analyzing architectural features and frameworks in microservices, Montesi *et al.* [101] conducted a comparative study between architectural patterns for microservices and their respective frameworks. The authors reviewed three mainstream mechanisms found in Microservice Architectures (MSAs): Circuit Breaker, Service Discovery, and API Gateway. They discussed different strategies for their implementation, and elicited the interplay between deployment topologies and circuit breakers. These patterns are emerging as essential for the reliability, ease of access, and flexibility of MSAs. Since microservices is in its early development, the authors mentioned that it is possible to expect more patterns to appear in the future. It is interesting that these patterns are structural, in the sense that they do not change the operations that services provided by developers offer, which are more custom to the specific MSA at hand. However, their adoption also makes MSAs more complicated, and they influence the communication structures that will be enacted in a system. This suggests that methods for the programming and verification of communications among services should keep patterns such as these into account.

2.1. Summary

The studies mentioned in this section address μAzimut-related work from different perspectives. Several studies address architectural patterns and tactics for microservices, as well as studies that investigate properties and frameworks that are used to build and deploy microservices-based systems.

Some studies, on the other hand, use optimization techniques and genetic algorithms to obtain software components. This is because the selection of components can become an optimization problem given the variables that can be considered, such as cost, version of the component, and documentation. Table 2.1 summarizes the related work.

Table 2.1: Summary of the state of the art

	Microservices	Quality properties	Architectural tactics	Architectural patterns	Frameworks/Platforms	Imprecise info.	Imperfect info.	Uncertain info.
López *et al.* (2005)	No	Partially	No	No	No	Yes	Yes	No
Astudillo *et al.* (2006)	No	Partially	No	No	No	Yes	Yes	Yes
Kwong *et al.* (2010)	No	No	No	No	Partially	No	No	No
Tang *et al.* (2011)	No	Partially	No	No	Partially	No	No	No
Montesi *et al.* (2016)	Yes	Yes	No	Yes	No	No	No	No
Pertersen *et al.* (2017)	No	Partially	No	No	No	No	No	No
Lenarduzzi *et al.* (2018)	Yes	Partially	No	No	Yes	No	No	No
Di Noia *et al.* (2019)	No	Partially	No	Yes	Partially	Yes	No	No
Watada *et al.* (2019)	Yes	Partially	No	No	Yes	No	No	No
Cervantes *et al.* (2019)	No	No	Yes	Yes	Yes	No	No	No
Ernst *et al.* (2019)	No	No	No	No	Yes	No	No	No
This doctoral thesis	Yes	Yes	Yes	Yes	Yes	Yes	No	No

To the best of our knowledge, we did not find sufficient evidence on studies that address the problem of framework selection using quality properties, architectural tactics, architectural patterns, frameworks/platforms and imprecise information in microservices architectures. About consider imperfect or uncertain information as variables, these

variants will be considered as future work.

Part II

Understanding Microservices Architectures

Chapter 3

Microservices patterns, architectural tactics, and quality attributes

This chapter describes a multivocal review [61] of both academic and industrial sources regarding architectural patterns and tactics for microservices. This review aims to describe state of the art and practice of architectural patterns and architectural tactics in the context of microservices.

3.1. Introduction

The microservice architectural style is becoming [85] a preferred industrial approach to develop applications as suites of small services, each running in its process and communicating with lightweight mechanisms, often an HTTP (Hypertext Transfer Protocol) resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery; there is a minimum of centralized management; and may be written in different programming languages and use different data storage technologies.

This approach to development using microservices allows to identify existing functionalities in monolithic systems that are both non-critical and loosely coupled with the rest of the application. For example, in an e-commerce system "events" and "promotions" are often ideal candidates for a microservices proof-of-concept [104].

One of the *avant-garde* companies in adopting the microservice architecture approach is Netflix[1] [124]. The Netflix application has an architecture featuring an API Gateway that handles about two billion API edge requests every day, and consists of approximately 500 microservices. On the other hand, Uber[2] has about 1300 microservices to improve reliability and scalability [83], developing global standards that apply to all microservices. The number of companies that are adopting microservices is increasing, making it attractive to investigate their relevance.

Microservices have become a dominant approach in the service oriented software industry and attracted great interest [43] [139] [6] [47]. Yet, there is still no systematic categorization of emerging recurring solutions (architectural patterns) or design decisions (architectural tactics) for microservices.

In this chapter, we review academic and industrial sources regarding architectural patterns and architectural tactics regarding microservices. The review is organized in three phases: Phase I explores the academic evidence, Phase II complements academic evidence with other sources of information, and Phase III adds the industrial evidence and contrasts the of both areas.

3.2. Review planning

This section describes the research method executed in this study, based on procedures for performing systematic reviews of Kitchenham's [76] [77] guidance for systematic reviews for software engineering researchers. Furthermore, we used the 12 guidelines proposed by Garousi *et al.* [61] to conduct grey literature reviews. We focused the study

[1]https://www.netflix.com/
[2]https://www.uber.com/

in three parts: Phase I surveys the academic evidence regarding architectural pattern and tactics; Phase II complements the academic evidence with proposals obtained from other sources of information; and Phase III adds industrial evidence and compares results (see Figure 3.1).

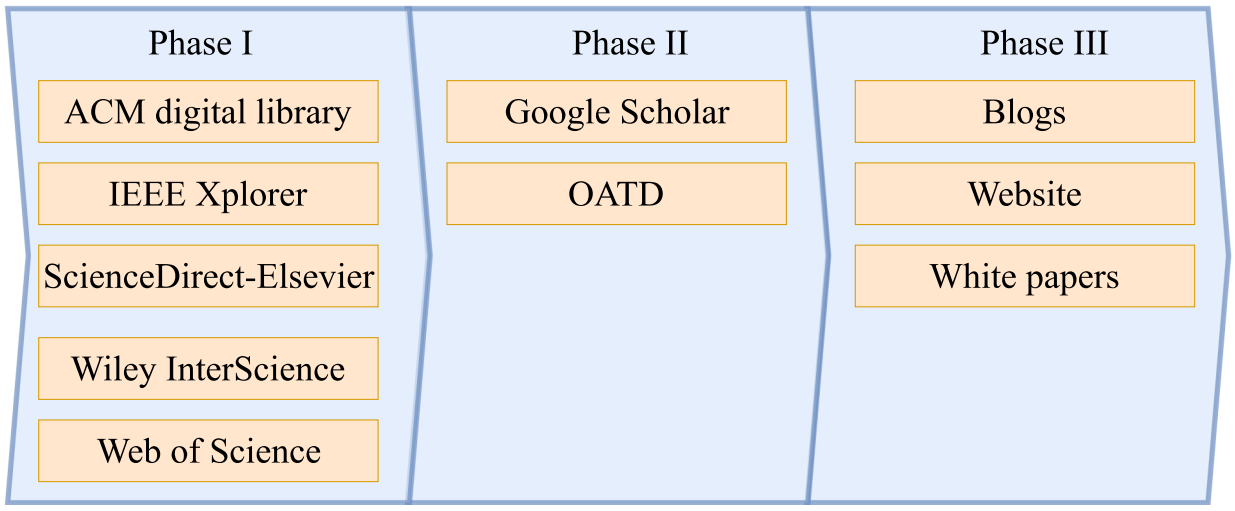


Figure 3.1: Review planning

3.2.1. Review protocol

For this study, we established a protocol [77] specifying the method used for the systematic review. The method begins with three research questions: the search process; the selection criteria; and the data extraction and execution review protocol (see Figure 3.2). To reduce the possibility of bias, the study was developed by three researchers. The study was conducted between March and August 2017.

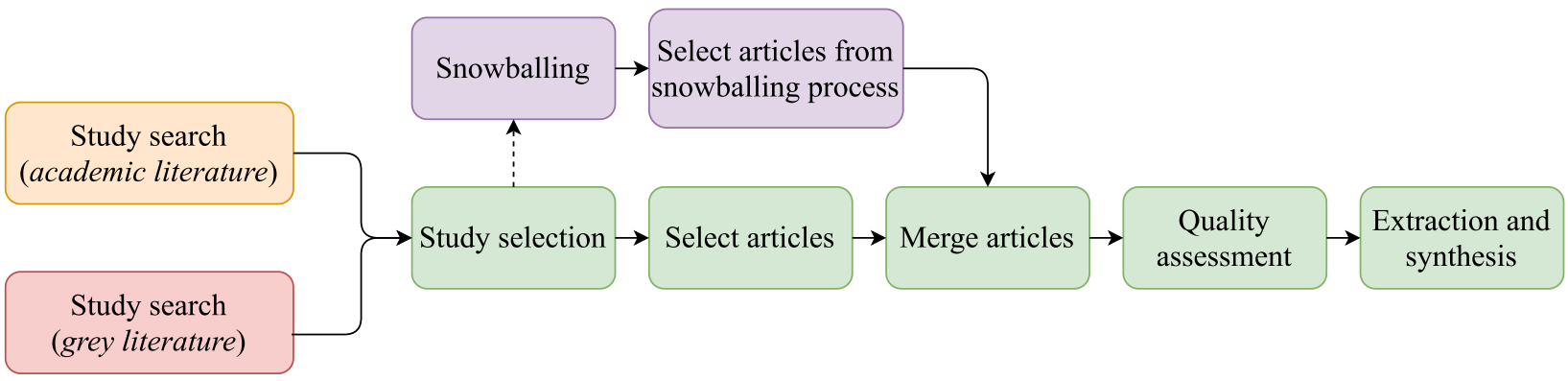


Figure 3.2: Review execution

3.2.2. Research questions

To establish the research questions, we set up meetings with experts using brainstorming[3] focused on the following keywords: "software architecture", "microservices", "quality attributes", "patterns" and "tactics". From those meetings, we established the following research questions:

RQ1.1
What architectural patterns have been proposed for microservices?

RQ2.1
What architectural tactics have been proposed for microservices?

RQ3.1
What are the quality attributes associated with architectural patterns and tactics in microservices?

3.2.3. Phase I and II: academic review protocol

We used several terms to find relevant studies in the academic literature, combining them with AND - OR. The search string has two word bases: *software architecture* and *microservices*, which are included in all queries. Therefore, it was decided not only use "tactic" and "patterns" words, but also we decided to expand the search string using the words "decision" and "design." For microservices, we included in the string the words "micro servi", "microservi" or "micro-servi" (see Table 3.1).

Table 3.1: Search string

"software architecture"	**AND**	("micro servi" **OR** "microservi" **OR** "micro-servi")	**AND**	"tactic"	**Q1**
				"pattern"	**Q2**
				"decision"	**Q3**
				"design"	**Q4**

[3]https://www.mindtools.com/brainstm.html

In the academic review protocol, we performed the search in two phases. In the first phase we obtained 744 papers on the most important databases of journals and conferences in the community of software engineering. The databases consulted in the first phase were: ACM Digital Library[4], IEEE Xplore[5], ScienceDirect - Elsevier[6], and Wiley InterScience[7].

In the second phase, with the aim to expand the search, we obtained 323 papers. The databases consulted in the second phase were: Google Scholar[8] and Thesis Databases[9].

Selection criteria

Each study was analyzed using inclusion and exclusion criteria, in order to verify whether paper is suitable to respond the research questions. The inclusion and exclusion criteria used in this study were:

- Academic inclusion criteria:
  - Papers related to software engineering and software architecture
  - All papers published in journals and conferences proceedings in the field of software architecture and software engineering; and
  - All papers in English.
- Academic exclusion criteria:
  - Studies which address only commercial-off-the-shelf (COTS) software
  - When the same study is reported by more than one paper, we only consider the most complete study
  - Papers that do not explicitly related to microservices, patterns or tactics or not relate to the research questions

[4]http://portal.acm.org
[5]http://www.ieee.org/web/publications/xplore/
[6]http://www.elsevier.com
[7]http://www3.interscience.wiley.com
[8]https://scholar.google.com/
[9]https://oatd.org/

- Papers that no are in English

Selection Process

The selection of studies was performed in three steps: (1) search and identify relevant studies in the selected databases using the search terms, (2) apply the inclusion and exclusion criteria, (3) manually exclude studies based on the title, abstracts, introduction, and conclusion, and (4) a full reading, yielding a final list of papers.

3.2.4. Phase III: industrial review protocol

In the grey literature, we established a search process including blogs, websites and whitepapers (we will refer to these sources as *documentation*), using the words from the string search. To performed this search, we use two searching browser: Google[10] and Bing[11]. The search process was manual because the amount of documentation was acceptable for three researchers.

Selection criteria

Each study was analyzed using inclusion and exclusion criteria, in order to verify whether it was suitable to respond the research questions. The inclusion and exclusion criteria used in this study were:

- Industrial inclusion criteria:
  - Documentation related to software engineering and software architecture
  - Documentation regarding trends in microservices
  - Documentation regarding microservices and technology

[10] www.google.com
[11] www.bing.com

- All documentation in English

- Industrial exclusion criteria:

  - Documentation which address only commercial off-the-shelf (COTS)
  - When the same documentation is reported by more than once, we only consider the most complete documentation
  - Documentation that do not explicitly related to microservices, patterns or tactics or not relate to the research questions
  - Documentation that no are in English

3.2.5. Data extraction - Phases I, II and III

Once we apply the selection criteria and process, we use templates to extract relevant data. For each primary study, we used the next fields: Paper ID, Publication year, Paper title, Authors, Type of study (conference, journal, and others), and Keywords. The mentioned fields allowed us to visualize, in a general manner, if the obtained articles were coherent and according to the research questions. For patterns and design, the fields were considered: Paper ID, Pattern Name, Context, Problem, and Solution. And, for tactics and decisions, we considered the following fields: Paper ID, Tactic Name, Architectural decision, Consequence.

3.2.6. Snowball process

In our research it is fundamental that selected studies are representative of the population in terms of their ability to answer our research questions. In this context, in order to mitigate a potential bias with respect to the construct validity of the study, we complemented the review with the snowballing process [143]. The main goal of this step is to enlarge the set of potentially relevant studies by considering each study selected in the previous stages, and focusing on those papers either citing and cited by it.

More technically, we performed a closed recursive backward and forward snowballing activity.

3.3. Results

In this section, we proceed to describe the results obtained after executing the research process. Tables 3.2 and 3.3 depict the year, title and issue of primary studies concerning architectural patterns and tactics.

Table 3.2: Primary studies related to architectural patterns

Ref.	Year	Title	Venue
[15]	2015	Microservices Migration Patterns	Automated Software Engineering Group, Sharif University of Technology
[26]	2016	Microservices Approach for the Internet of Things	IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)
[116]	2016	IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications	First International Conference on Internet-of-Things Design and Implementation
[24]	2016	Implementation Patterns for Microservices Architectures	Pattern Languages of Programs conference
[99]	2016	A Simplified Database Pattern for the Microservice Architecture	The eighth International Conference on Advances in Databases, Knowledge, and Data Applications
[98]	2016	The Database-is-the-Service Pattern for Microservice Architectures	ITBAM 2016: Information Technology in Bio- and Medical Informatics

Table 3.3: Primary studies related to architectural tactics

Ref.	Year	Title	Venue
[33]	2015	Architectural Support for DevOps in a Neo-Metropolis BDaaS Platform	34th Symposium on Reliable Distributed Systems Workshops
[148]	2016	Reflections on SOA and Microservices	4th International Conference on Enterprise Systems

RQ1.1: Architectural patterns for microservices

We have decided to summarize architectural patterns for microservices using as reference the schema proposed in [4], which are: reference, ID (MSP, MicroService Pattern

and IMSP, microservices patterns obtained from industrial sources), name, context, problem and solution (architectural patterns are detailed in Appendix C).

Patterns from MSP1 to MSP17

Brown *et al.* [24] proposes the patterns from **MSP1** to **MSP17**. In the paper, the authors divide the patterns into four parts: (1) Modern Web Architecture Patterns, (2) Microservice Architecture Patterns, (3) Scalable Store Patterns, and (4) Microservices DevOps Patterns. In (1) its described **MSP1** as a root pattern. Adopting a modern web architecture will lead different potential implementation choices such as a **MSP2** or **MSP3**. The **MSP4** pattern is commonly implemented in modern web applications to improve performance and to allow for some functioning when the client is disconnected from back-end systems. Related to patterns to building microservices and making them run efficiently, (2) deal with the concerns of identifying and implementing microservices. **MSP5** is the basic root pattern that the design journey by following, leading implementing multiple **MSP6**. **MSP8** can be used if is required to communicate with existing back-end systems. The **MSP7** pattern is a key component for implementing the microservices architecture in the context of modern web architecture in connecting different client types to the business microservices. Finally, **MSP9** and **MSP10** are commonly used approaches to improve microservice performance. About decisions of data storage that are necessary to build systems that are both performance and highly available, in (3) is described that a key part of the microservices architecture is that wherever possible each microservice should control or "own" its data. However, microservices are also expected to be scalable and stateless. This combination of requirements leads to the need for **MSP11**. A scalable store can come in many different types, such as a **MSP12** or a **MSP13**. The choice depends upon the type of data is storing. Finally, (4) highlight that while not required to implement a microservices architecture, teams soon find that the greatly increased number of runtimes needed in a microservices architecture lead to new challenges in managing that proliferation. This leads to the need for **MSP14**. Microservices DevOps will lead to implementing patterns such as a **MSP15** and **MSP17**. Debugging long chains of

microservices may lead to implement **MSP16**.

Patterns from MSP18 to MSP32

Balalaie *et al.* [15] propose migration patterns (from **MSP18** to **MSP32**) that can be used by a method engineer to select guidelines and construct a bespoke migration methodology. Migrating current on-premise software systems to microservices introduces many benefits including, but not limited to, different management of availability and scalability for different parts of the system, ability to utilize different technologies and avoid technology lock-in, reduced time-to-market, and better comprehensibility of the code base. Furthermore, the migrated system can make use of the elasticity and better pricing model of the cloud environment, and therefore, provide a better user-experience for its end-users. Although microservices presents many benefits, it introduces distribution complexities to the system for which new supporting components are needed. The difficulties around the decomposition of a system into small services, and monitoring and managing these services are among the other important factors that make migrating to microservices a non-trivial and cumbersome task.

Is important to mention that are factors used to select the patterns in the paper under study. They used two guidelines for selection: architectural and operational. Next, we will detail the factors associated with each factor [15]:

- Architectural factors
  - Scalability: increasing Scalability of an application by scaling out its services (**MSP23**, **MSP24**, **MSP25**, **MSP26**)
  - High Availability: increasing Availability of an application by replicating its services (**MSP23**, **MSP24**, **MSP25**, **MSP26**, **MSP27**)
  - Fault Tolerance: decreasing the chance of failure in an application and providing means for handling failures effectively (**MSP27**)
  - Modifiability: increasing the ability to change an application with the least

side effects and without affecting its end-users (**MSP20**, **MSP21**, **MSP22**, **MSP28**, **MSP29**, **MSP32**)

- Polyglot-ness: enabling an application to use different programming languages and data stores (**MSP20**, **MSP21**)
- Decomposition: re-architecting an application to a set of services (**MSP20**, **MSP21**, **MSP22**)
- Understanding: perceiving the current situation of an application (**MSP19**)
- Visioning: deciding on the final situation of an application after migration (no pattern in this category)

- Operational factors:

  - Dynamicity: enabling an application to change in runtime without affecting its end-users (**MSP23**, **MSP24**, **MSP25**, **MSP26**, **MSP28**, **MSP29**)
  - Resource-efficiency: decreasing the amount of resources which are needed for an application's deployment (**MSP30**, **MSP31**)
  - Deployment: facilitating an application's deployment process and removing deployment anomalies (**MSP18**, **MSP23**, **MSP24**, **MSP28**, **MSP30**, **MSP31**)
  - Monitoring: enabling an application to be monitored in runtime effectively (**MSP32**)

Patterns from MSP33 to MSP39

In the paper of Butzin *et al.* [26] is possible to appreciate an attractive study on patterns and best practices that are used in the microservices approach and how they can be used in the Internet of Things (IoT) (from **MSP33** to **MSP39**). Since the companies using microservices have made considerations on how services have to be designed to work together properly, IoT applications might adopt several of these design decisions to improve the ability to create value added applications from a multitude

of services. Although in this paper the patterns are not directly described, the authors mention recurring solutions in microservices for typical IoT situations. **MSP33** describe that the property of self-containment is one of the core aspects. The term *self-containment* is used to detail that services should contain everything they need to fulfill their task on their own. This concept includes not just its business logic but also its front and back-end, as well as required libraries. Keeping it this way the services can be scaled individually by starting multiple instances. Furthermore, dependencies to other services are kept small, and thus, services can evolve independently. Nevertheless, services should not become too big to stay maintainable.

The **MSP34** pattern has evolved from practice. In short, the circuit breaker checks health status or remember the number of unsuccessful calls and trips if a certain threshold is reached. If the circuit breaker is triggered, it will return an error instead of sending the request to the remote service to prevent the broken service to be penetrated with additional requests. After a certain amount of time, the circuit breaker tries to reach the service again to test if the services have recovered or check for the health status and enter a half open state if the test is successful again. It is not completely open immediately to prevent the called service to become unavailable again due to high incoming traffic. The **MSP34** pattern furthermore perfectly works together with the **MSP35** pattern. In this case, the load balancer distributes workload on a set of equal services. The circuit breaker enables the load balancer to put work only on services that are in a good health state. Services to which the circuit breaker is only half open the number of routed requests is lowered. Broken services will, for the time being, not be used.

About the pattern **MSP36**, is helpful that the individual services can be hosted as a single container. Those containers enclose the microservice itself, including all required libraries and data, which also support the requirement of self-containment. This pattern states that if container technology is used, deploying applications is greatly simplified (**MSP37**). After an application was tested and put into operation this specific artifact is not altered anymore. This can be emphasized by not providing any user credentials to the container. Instead, when something needs to be changed one just

replaces the application-container with a new version of it. Another pattern that can be used about multiple versions of a service is **MSP38**. This pattern deals with the problem of replacing applications by new releases in place, which would cause a down-time of the service. Instead, when deploying a new version of a service the old and new versions are running in parallel. At the beginning all requests are routed to the old version of the service and the new service can be started and configured. Finally, **MSP39** is a slightly modified version of the blue green deployment. Instead of routing any traffic immediately to the new service, the fraction of traffic that is routed to the new services is iteratively raised. This approach keeps the impact of a faulty new service even smaller as not everyone is immediately affected by the new version.

Pattern MSP40

The pattern presented by Messina *et al.* [99] [98] (**MSP40**) is the only one in our review that has a validation based on proof of concepts. The pattern aboard the problem related to: *if each scalable service has its database (cluster), is there any way to reduce the complexity of the architecture and the related risks, while also gaining more improvements regarding speed and scalability?* Whenever the database has an open architecture and provides the necessary hooks to extend its capabilities, and then it can embed the business logic that implements the desired service. Finally, the **MSP40** pattern has been tested adding ebXML registry capabilities to a NoSQL database. Experimental tests have shown improved performances of the proposed simplified microservice architecture compared with SQL-based ebXML registry implemented as traditional Java web service.

Patterns from MSP41 to MSP44

The article presented by Qanbari *et al.* [116] was the most discussed by the research team. Although the context of the article is IoT, the proposed patterns are strongly linked to microservices. Having said this and after discusses with experts, the review of this article was more detailed and will be detailed the IoT patterns that are most

related to microservices (from **MSP41** to **MSP44**). **MSP41** shows that IoT devices are usually scattered geographically, sometimes hard to reach and large in number. Operation managers and developers must be able to reconfigure devices or provision new ones in an efficient way and have pre-configured nodes. In these cases, container-based virtualization is a good choice for provisioning resources, as they contain not only the code but also all other software dependencies, configurations, and the whole runtime environment. By transferring the containerized image to a new machine and running it, we have a pre-configured environment with required applications installed along with any software dependencies. The pattern described in **MSP42** mention that maintainability is the main factor while deploying a piece of code to some remote IoT devices. As developers enhance and improve the code or fix some critical bugs, they expect to deploy the updated code to their several remote IoT devices quickly. This mechanism grants distributing functionality between devices. Also, at some point, developers need to re-configure the application's environment. So, as developers are familiar with version control systems, it is best to utilize it for deployments too. Nowadays, Git has become the *de facto* standard for developers to share their code and maintain versioning. It can be used as the starting point to trigger the build system and then the deployment process.

The pattern **MSP43** resolve the problem related to orchestrate IoT devices by their tightly scripted configurations as nodes of a cluster remotely. Edge infrastructure toolkits treat, and provision edge devices with limited compute resources (CPU, memory, and power) as constrained nodes of a cluster. Nodes to find each other, and the services they provide can leverage service discovery mechanisms, such discovery can also be achieved via device pairing. Once paired, devices trust each other and start sharing data or trigger functionality over a constrained network. **MSP44** describe that metering mechanisms can vary based on applied business models. These mechanisms range from different usage patterns such as invocation basis (event-based) and usage over time (time-based) to subscription models such as prepaid and pay-per-use models. This yields to the need for defining some metrics for service and resource usage, which in turn, can be used to measure the consumption of the service and to price it.

Patterns from IMSP44 to IMSP49

This whitepaper [123] highlights a taxonomy of microservices architectural patterns that have been practiced in the wild. The pattern **IMSP44** is arguably the "big bang" of microservices. Fine-grained SOA is somewhat self-explanatory (at least, to SOA practitioners): reduces the issues experienced with SOA and apply the same principles allowing to **IMSP45** gives some structure to a fine-grained API approach. **IMSP46** is usually the first architectural pattern implemented as a way to avoid the side-effects of accessing and mutating state. The description of **IMSP47** is nothing new, but when overlaid on microservice patterns they provide some powerful abstractions. **IMSP48** offer the option of coalesce the internal consistency of each microservice. Finally, **IMSP49** is essentially the antidote to the problems that emerge from isolating state: specifically, that consistency is required.

Patterns from IMSP50 to IMSP55

This blog [67] proposes to use design patterns [57] in microservices from an architectural point of view.**IMSP50** describe the situation related a simple web page that invokes multiple services to achieve the functionality required by the application. The pattern **IMSP51** is a variation of aggregator. Regarding responses, **IMSP52** produce a single consolidated response to the request. In the same way as **IMSP51**, the architectural pattern **IMSP53** extends aggregator design pattern and allows simultaneous response processing from two, likely mutually exclusive, chains of microservices. One of the design principles of microservice is autonomy and the **IMSP54** it represent and asynchrony can be achieved but that is done in an application specific way (**IMSP55**).

Patterns from IMSP56 to IMSP58

The blog [27] describes three architectural patterns related to the following areas: (1) structural patterns (**IMSP56**) and (2) distribution (**IMSP57- IMSP58**). In (1), a solution to user experience is to have different APIs for mobile and web. On the other

hand in (2), is described that URIs are documented and specified in countless RFCs and other documents that deal with all sorts of use-cases and impedance mismatches that happen when you govern a standard used in such open ecosystem as the Internet.

Patterns from IMSP59 to IMSP61

In this website [89], the author discusses different integration architectural patterns for synchronizing information between microservices. Through an illustrative example, the author describes his architectural patterns from different points of view: fastest way of sharing data across multiple services (**IMSP59**), integration of services using REST (**IMSP60**), and calling services by putting a message broker in the middle (**IMSP61**).

Patterns from IMSP62 to IMSP65

This white paper [41] discusses some of the key performance challenges, which can impact the performance of microservices based systems. The pattern **IMSP62** describes a technique which can be used to avoid any misbehaving or rouge application, from overloading or bringing down our application, by sending more requests than what our application can handle. Following the same idea, **IMSP63** remarsk if a microservices being invoked, is responding slow, this can cause our application to take longer time to complete a request. **IMSP64** reveals that it is always a good practice to have dedicated pool for individual service. Finally, **IMSP65** is used to minimize the impact of any of the downstream being not accessible or down (due to planned or unplanned outages).

Patterns from IMSP66 to IMSP74

Microsoft has decided to contribute to the knowledge of microservices through its Azure technology. Due to the above, [140] have proposed the following architectural patterns they have discovered: **IMSP66** can be used to offload common client connectivity tasks such as monitoring, logging, routing, and security (such as Transport Layer Security)

in a language agnostic way. **IMSP67** implements a façade between new and legacy applications, to ensure that the design of a new application is not limited by dependencies on legacy systems. **IMSP68** creates separate backend services for different types of clients, such as desktop and mobile. **IMSP69** isolates critical resources, such as connection pool, memory, and CPU, for each workload or service. **IMSP70** aggregates requests to multiple individual microservices into a single request, reducing chattiness between consumers and services. **IMSP71** enables each microservice to offload shared service functionality, such as the use of SSL certificates, to an API gateway. **IMSP72** routes requests to multiple microservices using a single endpoint, so that consumers don't need to manage many separate endpoints. **IMSP73** deploys helper components of an application as a separate container or process to provide isolation and encapsulation. **IMSP74** supports incremental migration by gradually replacing specific pieces of functionality with new services.

Patterns from IMSP75 to IMSP80

This blog [37] highlights that the refactoring of a monolithic program into a microservices-based application considers decomposition (**IMSP75-IMSP76-IMSP77**) and underlying patterns for microservices deployment, along with a few variations (**IMSP78-IMSP79-IMSP80**).

RQ2.1: Architectural tactics for microservices

The architectural tactics found in this review are not directly related to microservices. Nevertheless, there are architectural tactics that complement each other with other disciplines, such as DevOps (Development and Operations) [33]. For DevOps to be successful, an architectural approach must be taken to ensure that system-wide requirements are consistently realized. To realize DevOps strategies, architectural tactics must be carefully chosen and applied [33]. The relationship between microservices and DevOps is related to continuous deployment, because this approach requires architectural support in: (1) deploying without requiring explicit coordination among teams,

(2) allowing for different versions of the same services to be simultaneously in production, and (3) rolling back a deployment in the event of errors; allowing for various forms of live testing. Microservices architectures support many of these requirements [20].

Another article found in the review dealing with architectural tactics indirectly is [148]. The authors discuss that the API and microservice paradigm has emerged as the *next big thing* for delivering IT outcomes to support the modern enterprise, with many technology vendors and service jumping on the bandwagon. The article undertakes a critical investigation of the key concepts around SOA, API, and microservices, identifying similarities and differences between them and dispelling the confusion and hype around them. In the investigation, they describe two indications to what may be architectural tactics: (1) *scaling* and (2) *independence.* From one side, (1) detail that a microservice must be independently deployable with each microservice having its deployment architecture; and not share its resources like containers, caches, datastores with other enterprise software components. For instance, where the microservice stores and retrieves information from a database, the database schema should be part of the microservice. On the other side, (2) is complemented by (1) and detail that scaling of one microservice is independent of other microservices; and increasing the capacity of a microservice, by moving its hosted target or by increasing the number of instances, should not have any impact on its consumers. Both (1) and (2) are not architectural tactics directly. Yet, the authors refer to these words as decisions in microservice architecture. Therefore, we consider that they may be potential of unexplored architectural tactics.

RQ3.1: Quality attributes associated with architectural patterns and tactics in microservices

We found the following quality attributes (QAs):

- *Scalability*: This QA is observed in most of the revised patterns, and its philosophy lies in that scaling a particular microservice instance in the application is not a hard task: if a particular microservice in a flow becomes a bottleneck due to slow execution, that microservice can be run on more powerful hardware for increased performance if required, or one can run multiple instances of the microservice on different machines to process data elements in parallel. Nevertheless, in contrast to the easy microservices scalability with monolithic systems, scaling is non-trivial; if a module has a slow internal piece of code, there is no way to make that individual piece of code run faster. To scale a monolithic system, one has to run a copy of the complete system on a different machine and even doing that does not resolve the bottleneck of a slow internal-step within the monolith.

- *Flexibility*: Some patterns describe that using microservices makes it easier to organize an application's architecture. Fowler notes many enterprises create teams based on the business capability for a microservice. This means each cross-functional team includes personnel responsible for the UX, database, middleware, etc. According to [146], for achieving the necessary flexibility in microservices, each microservice can be easily changed and brought into production. Microservices emphasize isolation: ideally, a user interaction is completely processed within the need to call another microservice.

- *Testability*: This QA is associated with those patterns that within its recurrent solutions are supported by technologies. The services may be written in different languages and may use different data storage techniques. While this results in the development of systems that are scalable and flexible, it needs a dynamic makeover for the testing team. This QA requires that the testing team builds a deep architectural and design understanding and re-invent traditional test techniques.

- *Elasticity*: Although this QA is mentioned little, it is imperative to consider it, since it goes hand in hand with scalability. Each microservice should be elastic so that it can quickly scale up or down the number of containers used for each

microservice. Some microservices may only require one container; others may require many. Many times when considering elasticity, it has to consider whether the application is stateless or stateful. Stateless apps should be trivial to scale up and down. However stateful apps require more care.

- *Performance*: The patterns found directly associate this QA to scalability. For example, to build a scalable application, some patterns describe that is necessary to design for concurrency and partitioning: concurrency allows each task to be broken up into smaller pieces, while partitioning is essential for allowing these smaller pieces to be processed in parallel. So, while scalability is related to how we divide and conquer the processing of tasks, performance is the measure of how efficiently the application processes those tasks. At this point, the evidence found in the patterns coincides with the view of Sam Newmann [104].

- *Availability*: Ability to guarantee that both the system and the data will be available to the user at all times. In addition, this concept is applied when it is necessary to have a contingency plan on any component that has an abnormal situation in order to continue providing the service of the same. If a user cannot access the system, it is said that it is not available. The term downtime or offline time is used to define when the system is not available. So high availability is when the system is almost always up because all the elements of servers, network, and storage were duplicated in advance, eliminating the single points of failure. At the software level, intelligent and automated systems are used to restore the service.

- *Interoperability*: Capacity of two or more systems (or components) to exchange information and use the information that has been exchanged. Interoperability typically takes place at two levels; semantic and technical. Semantic interoperability allows the parties involved to describe the requirements without considering the technical implementation. With respect to software, the term interoperability is used to describe the technical ability of different programs to exchange data through a common set of exchange formats, to read and write the same file formats, and to use the same protocols.

- *Security*: Capacity to ensure that the system's resources are used as expected and that those who can access the information in the system are those who are accredited to do so. In addition, Security considers sets of systems, organizational means, human means, and actions designed to eliminate, reduce or control the risks and threats that may affect a person, an entity, a facility, or an object.

3.4. Summary

This chapter presents a summary of architectural patterns and architectural tactics proposed for microservices in academia and industry. To perform the revision, we developed a multivocal literature review.

For architectural patterns in academic and industrial field, the review reported 124 patterns for microservices. On the side of architectural tactics in academia and industry, we did not find architectural tactics related to microservices, leading us to believe that it is an unexplored area. About quality attributes, we found that architectural patterns and architectural tactics are associated with eight quality attributes: Availability, Flexibility, Testability, Elasticity, Performance, Scalability, Interoperability, and Security.

Chapter 4

Discovering microservices patterns in microservices-based systems

Microservices allow building and maintaining systems by breaking down the business capabilities of systems into smaller (reusable) services, which work together to achieve the overall business goal. To design microservices-based systems, developers may reuse solutions known as *microservices patterns*. However, in practice, the documentation of microservices is often an afterthought and leaves out essential design decisions, reducing the maintainability of systems built with them. This chapter introduces Pekenum, a technique to recover microservice-based architectures based on rules that characterize microservices patterns, allowing to visualize a system's microservice-based architecture and to identify the microservices patterns used to implement it. Pekenum has been evaluated in an industrial case study at an Ambient-Assisted Living system, where it could recover the microservice-based architecture and identify most instances of microservice patterns. This is a significant step towards understanding existing microservice-based systems to support their systematic evaluation and evolution.

4.1. Introduction

The microservices architectural style suggests separating large monolithic systems into smaller manageable independent services, where each service focuses on specific tasks. The independent services communicate with each other via lightweight protocols [85] [104].

When developing and maintaining microservices-based systems, architectural patterns for microservices [128] (*microservices patterns*) offer solutions to reoccurring design problems. Such design problems include how to implement typical features of microservices-based systems, such as service registration, control, monitoring, and communication channels between services [42]. However, microservices patterns are rarely documented in practice [119]. This means that the knowledge about the design decisions (which eventually result in a microservices-based architecture) is only available to few developers directly involved on those decisions [21]. Although several studies address the challenge of recovering microservices-based architectures [96] [48] [7] [18] [80] [64], few identify and describe design decisions in form of pattern instances in specific microservices-based systems. Therefore, information about quality attributes and how a system may achieve them, as well as decisions that led to a microservices-based architecture, are often poorly understood.

This chapter presents *Pekenum*[1], a technique to recover microservice-based architectures using rules that characterize microservice patterns instances and which were extracted from the specification and documentation of several frameworks and platforms that are known to implement microservice patterns. Pekenum is useful during architecture maintenance to help developers understand the architecture and related design decisions (in particular if there is no documentation of the architecture and when development artefacts such as configuration files are the main source of information). We support the practical application of Pekenum by implementing a tool[2]. To evaluate Pekenum, we conducted a case study on an industrial Ambient Assisted Living (AAL) system. Results indicate that Pekenum is able to identify most of the microservices

[1] A Mapudungun word for "to discover things".
[2] https://github.com/gmarquez87/Pekenum

patterns that also exist in a "ground truth" architecture for that system.

4.2. Pekenum

Pekenum has three process components (see Figure 4.1), each one yielding different architectural insights (e.g., dependency information, runtime information) by examining the architectural elements of microservices architectures: build dependencies, ports used by microservices, image ID, IP addresses used by microservices, endpoints, logs, and traces. Pekenum uses these elements and information obtained from these elements to define microservices patterns and pattern rules.

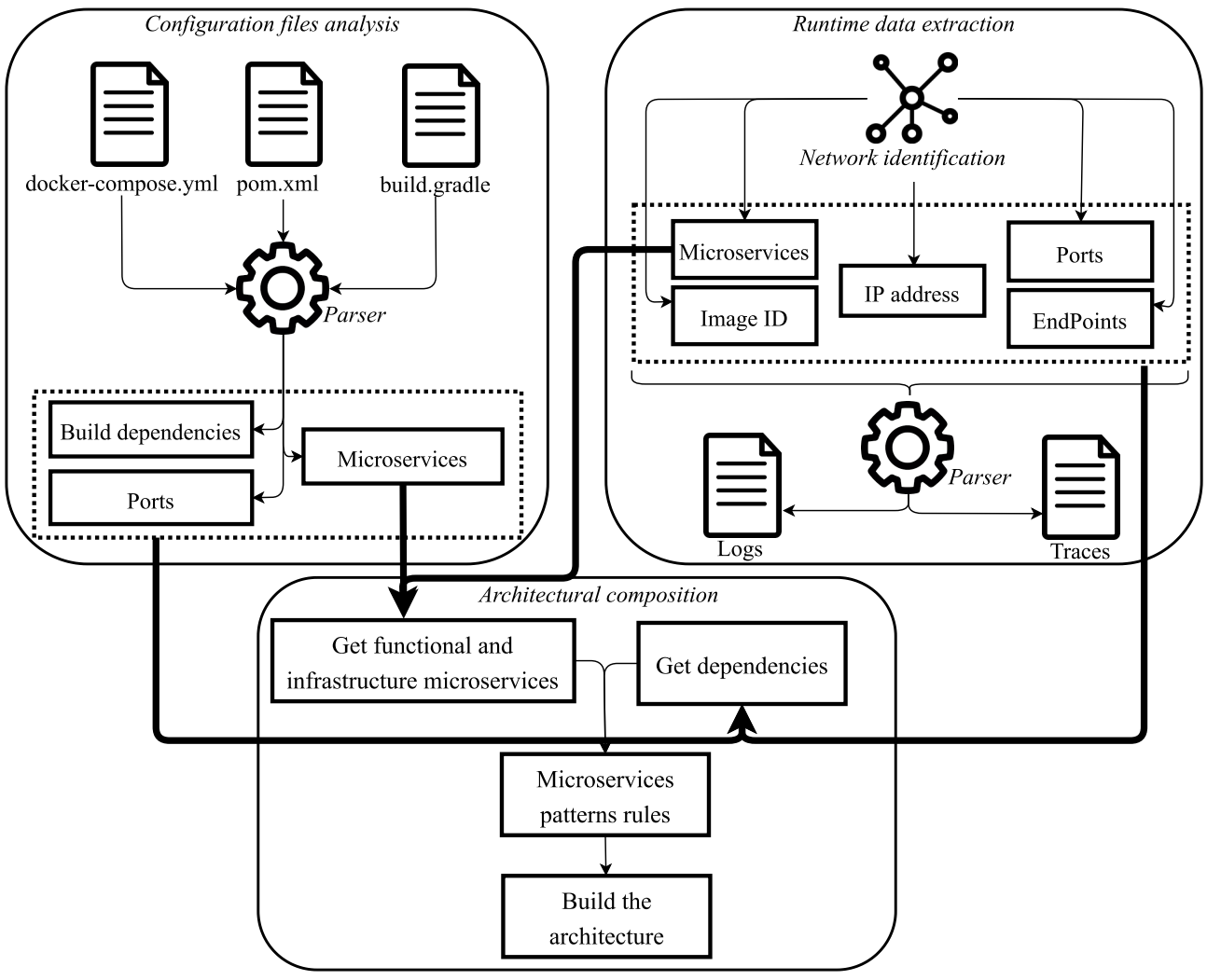


Figure 4.1: Overview of Pekenum

Pekenum distinguishes functional and infrastructure microservices. *Functional microservices* represent business-critical purposes; any change or modification to them affect not only the entire microservices-based system, but potentially also its business

goals. *Infrastructure microservices* belong to the virtualization and deployment environments of the microservices-based system. The following sections describe each component of Pekenum.

4.2.1. Configuration Files Analysis

This component obtains data about microservices from the configuration files of a microservices-based system, which can be:

- *Docker compose*: The docker-compose file corresponds to the Compose tool, which allows defining and running multi-container Docker applications. Using Docker along with docker-compose files is possible to use a YAML (YAML Ain't Markup Language) file to configure the system's services [1]. Docker-compose files define how to configure and run a collection of containers, which virtual networks they should be linked to, which volumes should be mounted, and more features.

- *POM*: Maven defines POM (Project Object Model) files as XML files that contain information about configurations and technologies to build a project [9]. Microservices can be structured as "modules" using POM files because modules provide advantages when declaring microservices build dependencies.

- *Gradle*: Gradle is an automation tool that relies on Groovy and a DSL (Domain Specific Language) to work with a simple and clear language for declaring the project configuration [63]. This tool provides the flexibility to work with other languages (not only Java) and it has a robust dependency management system.

These configuration files yield the names, ports and build dependencies of the application's microservices. Ports (along with IP addresses) help discover service instances that are dynamically assigned to network locations. Likewise, build dependencies reveal how microservices references and use other artifacts, frameworks or microservices.

4.2.2. Runtime Data Extraction

This component obtains runtime data of the microservices-based system, to identify the network ID (where the application is running), microservices' names, image ID, ports, endpoints, and IP addresses. The Network ID allows efficient discovery of each microservice's IP address. The Image ID gives access to the container log to check logs created at runtime. Furthermore, it enables to get the container log to check batch-retrieves logs present at runtime. Ports and endpoints facilitate the analysis of network traces among microservices.

4.2.3. Architectural Composition

This component creates a complete view of the recovered architecture, identifying functional and infrastructure microservices, and dependencies among them (either static or dynamic). To recover the architecture, Pekenum creates tuples that combine the output of the configuration files analysis and runtime data extraction. For example: the tuple (*microservice*, *file*) specifies that *file* is associated with *microservice*; the tuple (*microserviceA*, *microserviceB*, *buildDep*) indicates that *microserviceA* has a build dependency with *microserviceB*; etc. The tuples are stored in a XGMML file (eXtensible Graph Markup and Modeling Language).

4.2.4. Microservices Patterns Rules

Microservices patterns rules describe the dependencies that must be satisfied to identify microservices patterns instances in actual microservices-based systems. These rules were developed based on the conceptual description of microservices patterns [90] [128] [17], analyzing existing platforms and application frameworks used in the development and deployment of microservices-based systems.

Since *patterns* (including microservices patterns) represent reusable architectural and design knowledge, they also represent *pre-defined designs* to be used in the development

and/or implementation of specific systems [73]. Indeed, frameworks often implement these pre-defined designs.

To establish microservices pattern rules that relate frameworks and platforms in actual microservices-based systems, we build on previous empirical work [90] that identified the microservices patterns implemented by well-known frameworks and platforms. Pekenum uses a core set of microservices patterns (see chapter 7 for more detail on these microservices patterns), which are described in Table 4.1 along some frameworks and platforms that implement each.

Table 4.1: Microservices patterns, frameworks and platforms

Name	Description	Examples
Messaging	Suggests asynchronous messaging for inter-service communication; message queue allows the state to be asynchronously and reliably sent to different locations	RabbitMQ, gRPC
Service Discovery	Stores service instances addresses removal of an instance from the registry can be triggered through either not receiving a periodic heartbeat from the instance or by the instance itself during termination	Apache Zookeeper, Kubernetes, Netflix Eureka
Database is the Service	Suggests that each service has its own database server; when the service has to be scaled, the database can also be scaled in a database cluster, no matter the service	Apache Cassandra, MongoDB
Service Registry	Maps a unique identifier and the current address of a service instance in order to decouple the physical address of service from the identifier	Netflix Eureka, Consul
Load Balancer	Distributes a workload on a set of equal services; the circuit breaker enables the load balancer to put work only on services that are in a good state of health	Netflix Ribbon, Kubernetes, Apache Zookeeper

For each microservices pattern, Pekenum, defines a set of rules to recognize structures and dependencies of microservices patterns (see Table 4.2). Figure 4.2 shows the "Messaging" and "Service Discovery" pattern rules.

These rules allow Pekenum to recognize the functional and infrastructure microservices that implement a microservice pattern in a especific application.

Table 4.2: Microservices patterns rules

Messaging
1) There must exist a service (or services) that acts as messing service
2) There must be one runtime dependency between the service described in 1) and another service.
Database is the Service
1) There must exist a service that acts as database instance
2) Only one service should have a runtime dependecy with the service described in 1)
Load Balancer
1) There must exist a service (or services) that acts as load balancer
2) There must be dependencies (either build or runtime traces) which allow linking the project's services with the load balancer and a service registry (optional).
Service Registry
1) There must exist a service (or services) that acts as service registry
2) There must be dependencies (either building or runtime traces) which allow linking the project's services with the service registry
Service Discovery
1) There must exist a service (or services) that acts as load balancer
2) There must exist a service (or services) that acts as service registry
3) There must be dependencies (either build or runtime traces) which allow linking the project's services with the load balancer and a service registry

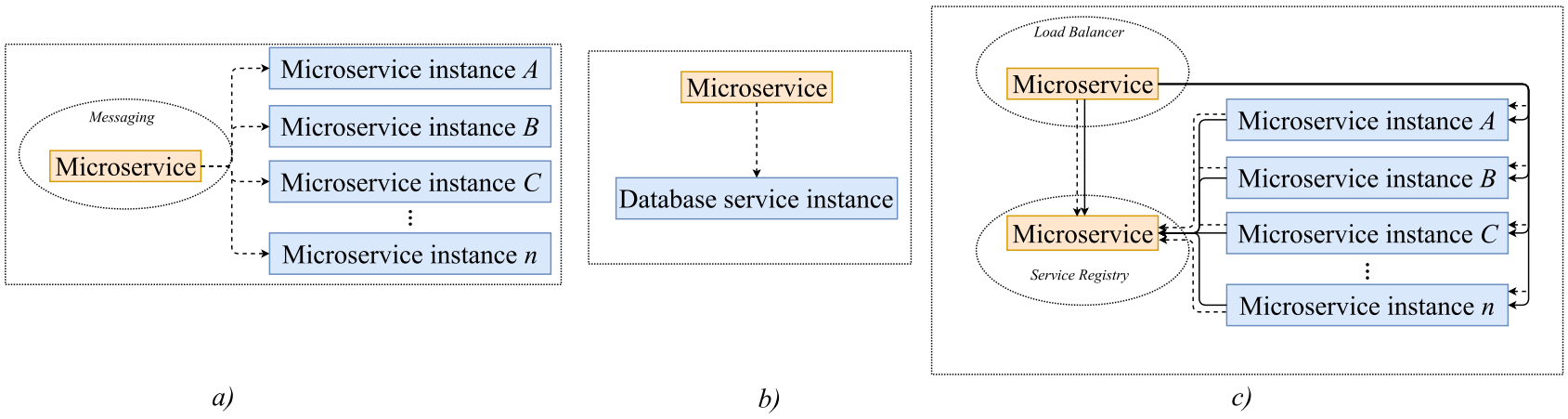
4.2.5. Current limitations of Pekenum

Before presenting a case study to evaluate Pekenum's usefulness in Section 4.3, we acknowledge some limitations of its current incarnation.

First, currently it only uses three types of configuration files to extract data. It can be extended to consider additional types of configuration files, and we are currently extending its ability to recognize some emerging technologies (e.g. Go[3]).

Second, the number of microservice patterns that Pekenum uses to recover an architecture is limited. The microservices patterns in Table 4.1 were chosen because typically there is enough information on how to detect them and to use them in microservices projects. Many microservice patterns have been published already [128] [119] [17] and we are adding rules to identify their instances.

[3]https://gokit.io



a) *b)* *c)*

Figure 4.2: Design of Messaging (a) and Service Discovery (b) patterns. The continuous arrows describe build dependencies. The dashed arrows depict runtime traces.

Third, runtime analysis in Pekenum is limited to detecting microservice communication on networks using primarily IP addresses and endpoints. We are currently working on extending runtime analysis by detecting the type of message sent among microservices and incoming/outgoing API data, and we are incorporating runtime techniques to retrieve more specific data (e.g. the content of messages among microservices, REST messages, etc).

4.3. Case Study

An industrial case study (see Appendix A) was conducted to evaluate Pekenum, by measuring how well it could identify instances of microservice patterns in an actual microservices-based system.

4.3.1. Context

This case study was part of a project to provide health care providers with real-time monitoring of elderly patients activities in a home environment. The project involved the design of an Ambient-Assisted Living (AAL) system using an Internet of Medical Things (IoMT), thus accounting for different devices. A microservices-based system was explicitly requested, to facilitate devices' integration with other systems, and ensure scalability and availability of services that will capture, process, and analyze

devices data.

The AAL system uses various kind of sensors and devices (see Figure 4.3), such as OMRON D6T-8L-06 and Melexis MLX90640. To capture and process data, and send alerts to some microservices, it uses a single-board ODROID-C1+ computer. This computer is also connected to an ATMEGA328P microcontroller, which reads sensor data and sends it to the ODROID-C1+ via UART interface with a baud rate of 115,200 and a sample rate of $5[Hz]$.

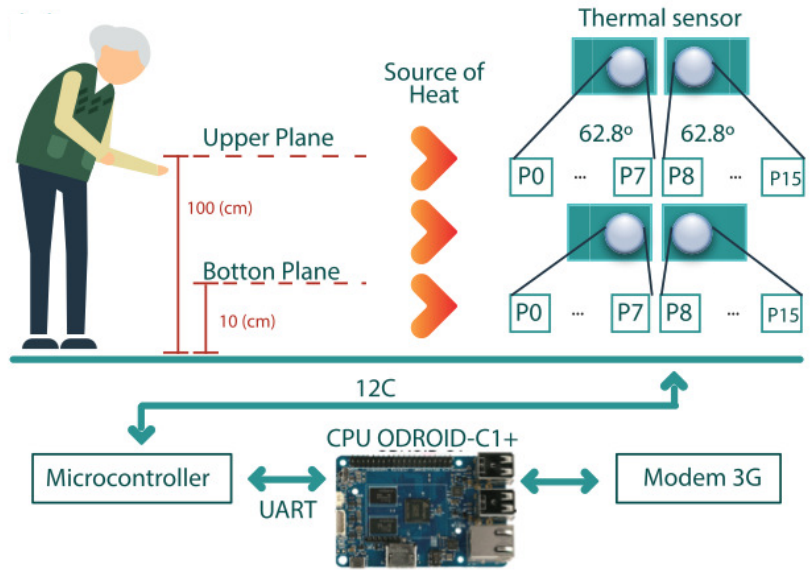


Figure 4.3: Sensor data capture and processing

This AAL system was originally designed using Attribute-Driven Design (ADD) [145], which requires documenting both architectural decisions and patterns used. Thus, the design documentation offers a ground truth for instanced microservices patterns and for which microservices participate in them.

4.3.2. Planning

This case study compares the architecture description generated by Pekenum with the ground truth architecture obtained from the ADD documentation. The main research question is:

RQ1.2

What is the performance of Pekenum, in terms of precision and recall, in identifying microservices pattern instances?

Precision is the ratio between the correctly identified patterns instances and the total number of pattern instances in the application. *Recall* is the ratio between the pattern instances correctly identified and the total number of pattern instances in the application.

Figure 4.4 shows a high-level description of the AAL architecture, with the main functional and infrastructure microservices of the system; other microservices in the system were omitted for space reasons. The ground truth architecture identified from the ADD documentation was validated by the two main project architects.

4.3.3. Data Preparation and Collection

First, Pekenum scanned all files in the project directory, yielding an XGMML file; Figure 4.5 depicts the AAL application's microservices data and system architecture, as produced by Cytoscape[4] (very reduced for space). The orange rectangles represent functional microservices; the light blue ones represent infrastructure microservices; and the purple ones represent infrastructure microservices related to databases. The green lines represent runtime communication among microservices. The blue arrows represent static dependencies.

4.3.4. Case Study Results

Table 4.3 shows the number of microservice instances in the ground truth for each microservices pattern, the number of microservice instances identified by Pekenum, and the precision and recall for each microservices pattern.

[4]https://cytoscape.org

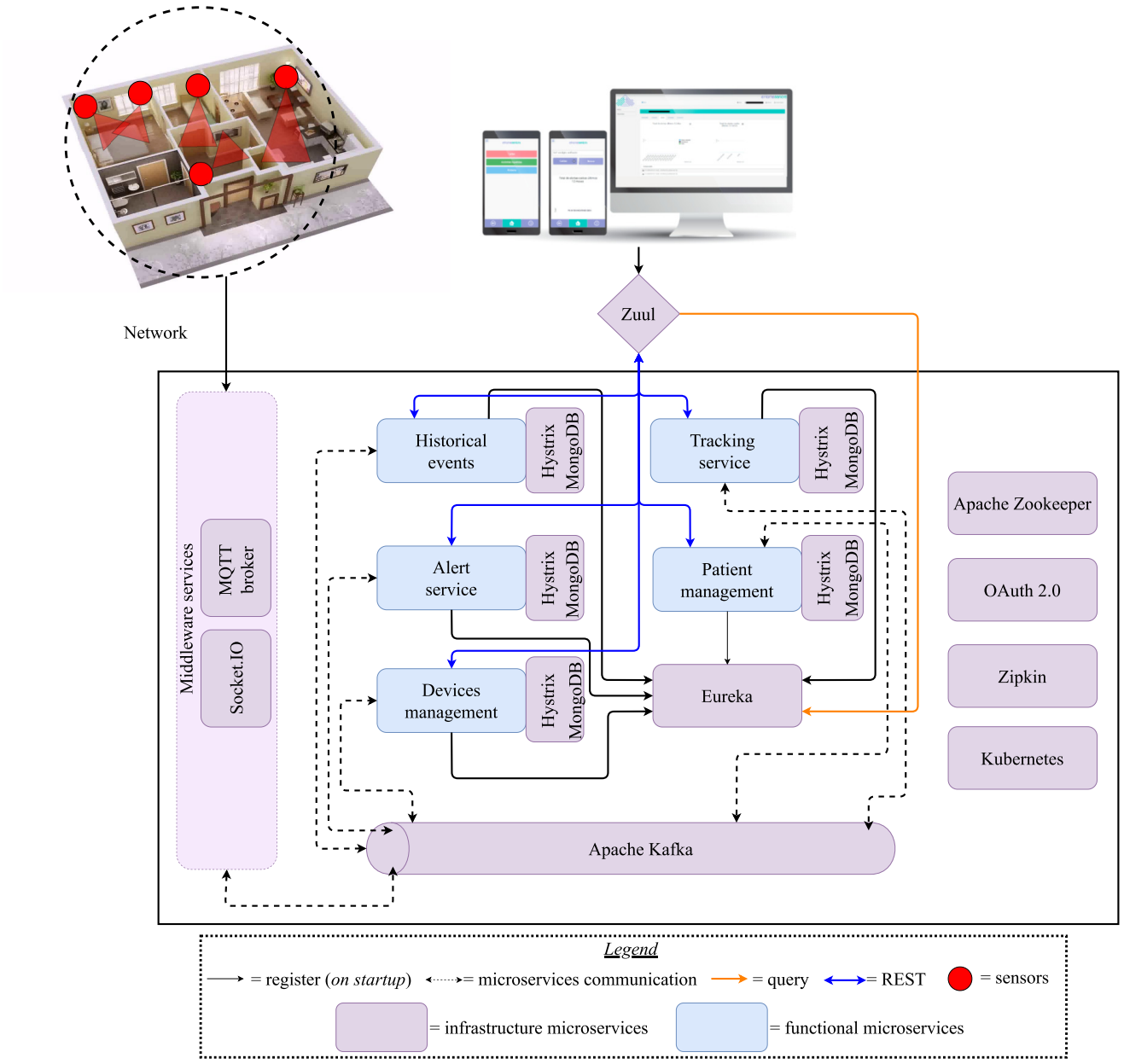


Figure 4.4: Microservices-based AAL system architecture

In this case study, Pekenum achieved a precision between 93.75% and 100% (with a mode of 100%) and a recall between 88.2% and 100%, with almost all in the 88%-89% range; i.e. most (but not all) pattern instances are identified, and almost all identifications are correct.

The analyzed results per pattern are:

- *Messaging*: all identified pattern instances were correct, but Pekenum was unable to link three infrastructure microservices that are related to a Messaging pattern instance; it identified them, but was unable to identify their dependencies or runtime communication.

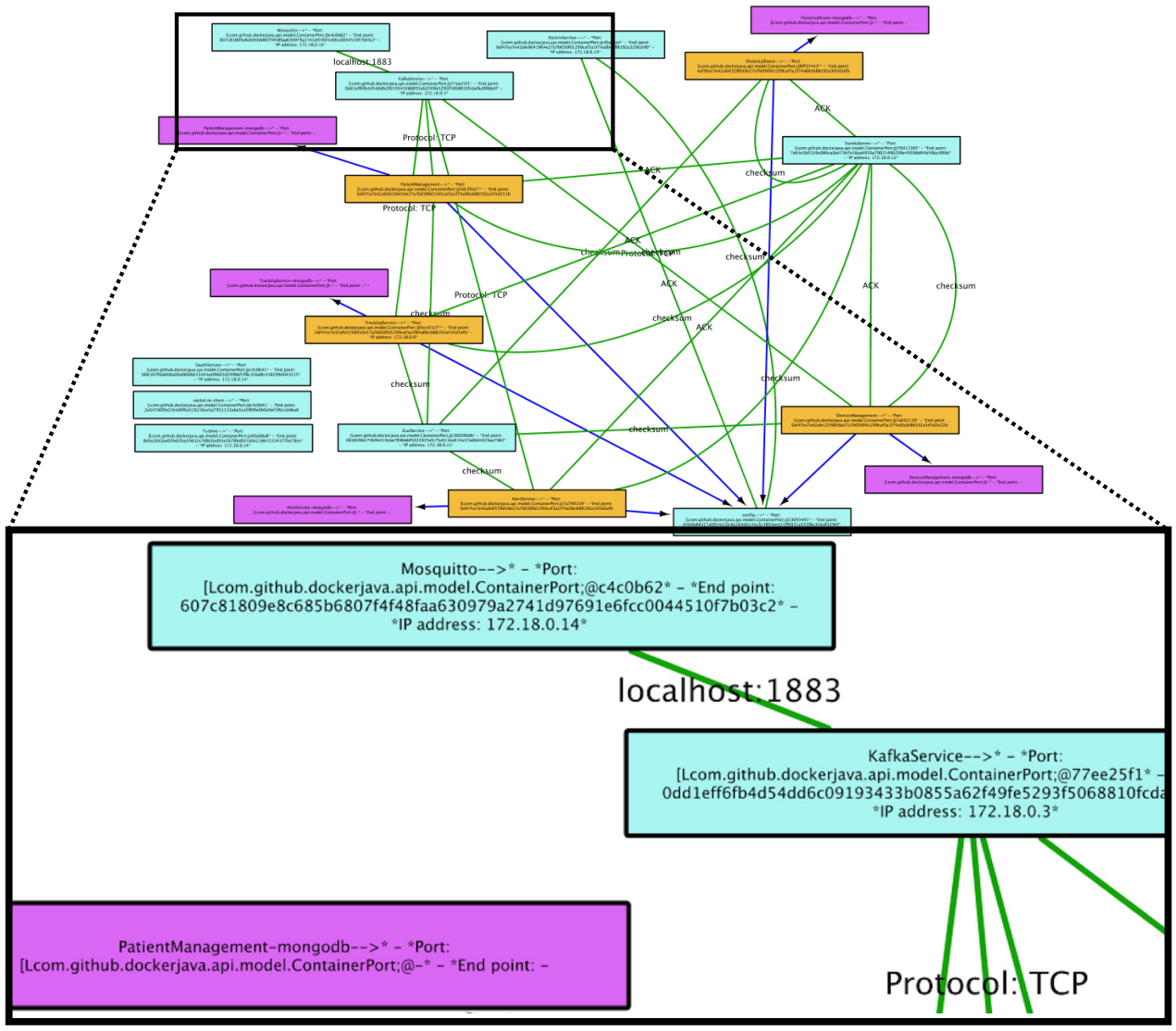


Figure 4.5: Architecture of microservices-based AAL system generated by Pekenum

- *Service Discovery and Service Registry*: Pekenum identified 18 instances of Service Discovery and 16 instances of Service Registry, but in both cases associated wrongly an infrastructure microservice (Netflix Turbine). Netflix Turbine uses Netflix Hystrix and Netflix Eureka to monitor the status of microservices in real-time, and the communication between seems to have "fooled" Pekenum.

- *Database is the Service*: Pekenum did a perfect job of identifying microservices instances, which probably happens because most database-related infrastructure microservice dependencies are explicitly described in the configuration files.

- *Load Balancer*: like in the *Messaging* case, Pekenum identified two infrastructure microservices, but could not link them to a microservices pattern instance.

Table 4.3: Case study results

	Messaging	Service Discovery	Database is the Service	Service Registry	Load Balancer
Pattern instances in the ground-truth	25	19	14	17	18
Patterns instances identified by Pekenum	22	18	14	16	16
False positives identified by Pekenum	0	2	0	2	0
Precision	100%	94.4%	100%	93.75%	100%
Recall	88%	89.4%	100%	88.2%	88.8%

4.3.5. Discussion

The recovery of microservices-based architectures is complex. Since microservices-based systems are a form of distributed systems, complexity comes from representing (a) the communication between distributed microservices in a network, and (b) which services surround a system. Furthermore, this complexity is often increased by (i) interpreting the architectural decisions made in the system, and (ii) describing to stakeholders the current state of a microservices-based system.

Pekenum goal is to address these issues. Regarding (i), Pekenum illustrates the architecture of a microservices-based system but also gives the possibility for an architect to easily interpret what kind of microservices patterns were implemented in the system. Concerning (ii), architecture visualization helps to involve stakeholders in the discussion of architectural decisions.

Regarding the visualization of microservices patterns, it is helpful for stakeholders to know which microservices are part of microservice patterns instantiations. Thus, they can decide about system maintenance and evolution towards other aspects of IoMT, such as the incorporation of new medical devices, the logic of the middleware layer microservices, among others.

For example, Pekenum identified an instance of the Service Registry pattern, showing graphically in Figure 4.6 that the communication among functional and infrastructure microservices are ACK and checksum messages. A figure like this allows explaining to stakeholders the importance of Netflix Eureka in their AAL application, and the impact that any changes or updates to this service and its functional microservices may have on the system, e.g. on its failover capability.

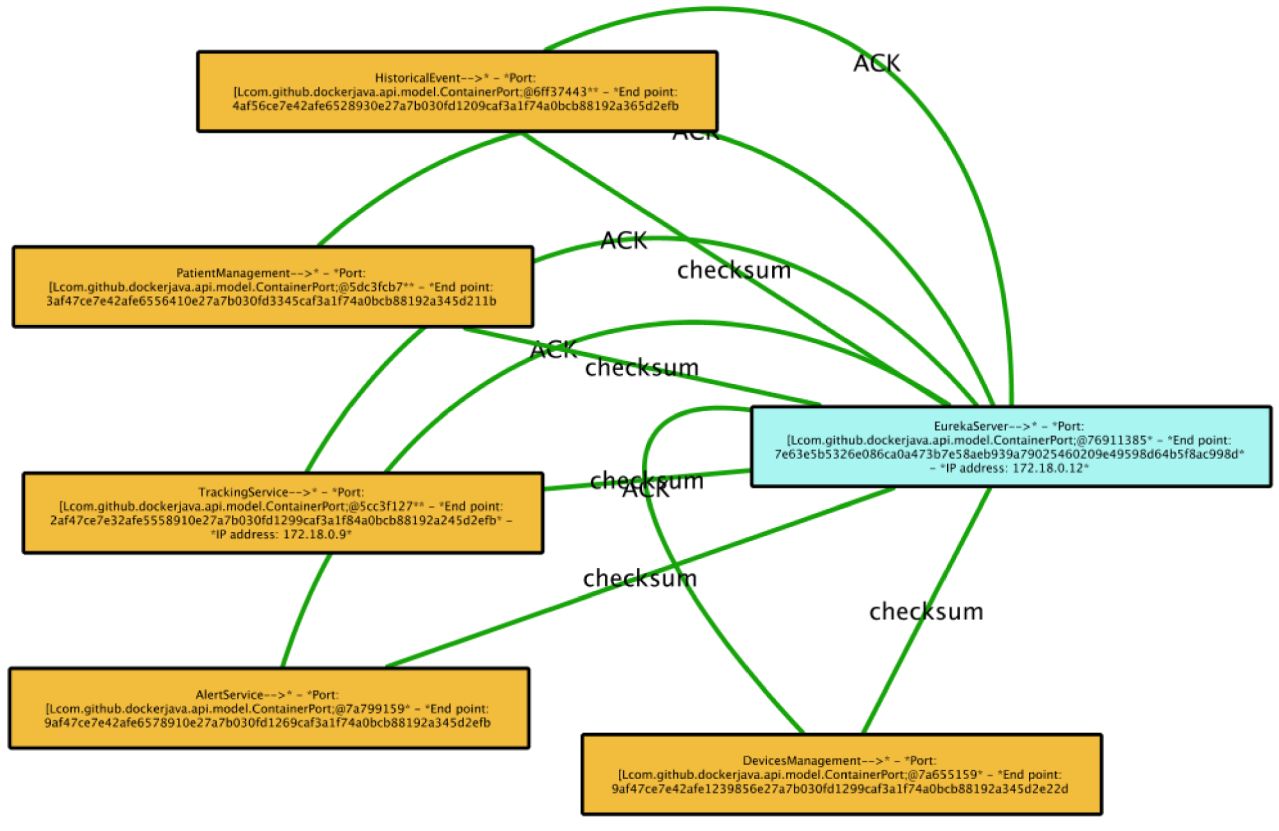


Figure 4.6: The Service Registry pattern recovered by Pekenum

4.3.6. Threats to the validity of the case study

Like all empirical studies, this case study design must address the threats to validity, which we organize following Wohlin's taxonomy [144].

Threats to internal validity describe factors that could affect the results obtained from the study. The main threat we identified is the bias regarding the ground truth. To mitigate it, we used the ADD method documentation to identify the main functional microservices of the AAL system and the used microservices patterns. Additionally, we had working sessions with the two project lead architects to clearly identify the infrastructure microservices. Finally, we iterated this ground truth with the stakeholders

several times, to refine and detail the microservices architecture.

Threats to external validity are conditions that limit our ability to generalize the results. The main threat is whether the case study is indeed representative of certain type of systems. We cannot claim our results would be directly generalizable to a large population of microservices-based systems, but since our case study shows promising results in an industrial microservices project, we believe that the results would not be too different with other projects or systems that show similar characteristics (e.g. IoMT or AAL applications).

Threats to conclusion validity concern issues that affect the ability to draw the correct conclusion about relations between the treatment and outcome of a study. The main threat we identified is the reliability of measures; to mitigate it, instead of evaluating software architecture identification *per se*, we used well-known metrics from Information Retrieval (i.e. precision and recall) [13] to compare true and false positives of identified microservices with a ground truth.

4.4. Summary

This Chapter has described Pekenum, a technique to recover microservices-based architectures using rules that characterize microservices pattern instances. Pekenum identifies key architectural microservice elements from static sources (mainly configuration files) and runtime information (communication between microservices detected in the network), and combines them into a single global architecture picture.

The key underlying feature of Pekenum are microservice patterns rules, which describe the dependencies that must be satisfied to identify microservices patterns instances in actual microservices-based systems. The initial (and current) set of rules builds on recent empirical work that identified the microservices patterns implemented by some well-known frameworks and platforms.

The approach was evaluated in a industrial case study with a microservices project

in the ambient assisted living domain. We compared the pattern instances identified by Pekenum with a ground truth created with the application architects, and evaluated precision and recall regarding five microservices patterns. The results show that Pekenum could identify correctly most implemented microservices patterns.

This is a significant step towards understanding existing microservice-based systems to support their systematic evaluation and evolution.

Part III

The μAzimut approach

Chapter 5

Microservices properties

While building microservice-based applications, architects need to choose among different frameworks to provide generic functionalities to address quality attribute concerns. Although using frameworks brings various benefits, it is not clear how they actually impact on the *properties* characterising quality attributes.

In this chapter, we discussed the importance of identifying properties that represent quality attributes in the context of microservices. The properties allow locating the architectural decisions that must be applied to satisfy NFRs.

5.1. Introduction

In chapter 1 we described the importance of NFRs in software architecture design (and eventually, in microservices architectures). Additionally, we discussed the relationship between NFRs and QA to represent systemic properties. To design software architectures, architects use NFRs descriptions and QA specifications as a reference for decision making. Regarding microservices, there are studies such as the review conducted by di Francesco *et al.* [42] that describes those quality attributes that are significant for developers of microservices-based systems, such as maintainability, evolvability, availability, among others.

Nevertheless, when referencing these quality attributes (and others), there is little work that specifically mentions what specific guidelines or properties of these quality attributes an architect wishes to address in a system. This lack of clarity of properties or guidelines can be a problem in practice. To say, for example, that the services of a microservices-based system must be "highly available" may be a bit ambiguous, as in microservices, availability can be characterized in different ways (e.g., fault tolerance, service monitoring, among others).

Since microservices-based systems are distributed systems that communicate with lightweight protocols, new properties that characterize quality attributes may emerge. Therefore, in this chapter, we define the most relevant properties that constitute the main quality attributes mentioned in research related to microservices.

According to the results of a worldwide survey we conducted, in which 106 microservices practitioners participated[1], we identified 15 quality attributes that are popular in the microservices context. For each quality attribute, the participants were asked to rank their importance for microservices systems using a Likert-scale question (rated as "very important", "important", "somewhat important", "not important", and "not sure"). Table 5.1 indicated 55.7% (59 out of 106) of the respondents ranked security as very important, followed by availability (58 out of 106 54.7%) and performance (56 out of 106, 52.8%). On the other hand, only 26.4% (28 out of 106) and 23.5% (25 out of 106) respondents ranked resilience and portability as very important QAs, respectively. We also found several QAs that are ranked as "somewhat important" by the participants. Such QAs are reusability (25 out of 106, 23.6%), compatibility (24 out of 106, 22.6%), and monitorability (23 out of 106, 21.7%).

From the set of quality attributes described in Table 5.1, we have selected for μAzimut availability, scalability, interoperability, and security. This decision is based on the fact that these quality attributes have architectural tactics taxonomies that have been researched in the academic literature. Although we do not consider other significant quality attributes, such as maintainability, performance, and monitorability, we will

[1]The results of this survey were recently submitted to the Journal of Systems and Software. Muhammad Waseem, Peng Liang, Mojtaba Shahin and Amleto Di Salle participated in this work.

Table 5.1: Importance of QAs (in %) for designing microservices systems. VI- Very Important, I- Important, SI Somewhat Important, NI- Not important, NS- Not Sure

ID	Quality attributes	VI	I	SI	NI	NS
QA1	Security	55.7	27.4	9.4	3.8	3.8
QA2	Availability	54.7	29.2	14.2	1.9	0.0
QA3	Performance	52.8	33.0	9.4	2.8	1.9
QA4	Scalability	48.1	38.7	6.6	3.8	2.8
QA5	Reliability	40.6	38.7	12.3	2.8	5.7
QA6	Usability	35.8	40.6	14.2	2.8	6.6
QA7	Maintainability	35.8	42.5	20.8	0.0	0.9
QA8	Compatibility	34.0	34.0	22.6	1.9	7.5
QA9	Testability	33.0	41.5	19.8	0.9	4.7
QA10	Monitorability	32.1	38.7	21.7	1.9	5.7
QA11	Functional suitability	30.2	42.5	15.1	4.7	7.5
QA12	Reusability	30.2	35.8	23.6	5.7	4.7
QA13	Resilience	26.4	30.2	22.6	5.7	15.1
QA14	Portability	23.6	35.8	26.4	7.5	6.6
QA15	Interoperability	21.7	42.5	17.0	5.7	13.2

include some of them in the next version of the technique.

5.2. Obtaining microservices properties

In our previous study [90], we conducted an empirical study aiming at identifying frameworks used to develop microservices-based systems, and their relation with microservice patterns and quality attributes.

Figure 5.1 recaps the process we conducted to obtain frameworks. We firstly (i) explored 18 open-source microservices projects chosen using the benchmark requirements for microservices research proposed in [3]. We then (ii) analysed documentation and configuration files of each project into elicit the frameworks used in such projects used, and we (iii) analysed the documentation of each framework to identify which microservice patterns are implemented by each framework and which quality attributes are addressed by such framework. Finally, we (iv) created a matrix recapping the results of our analysis by relating framework to quality attributes and microservices patterns.

The main purpose of the process described in Figure 5.1 is to explore the state of

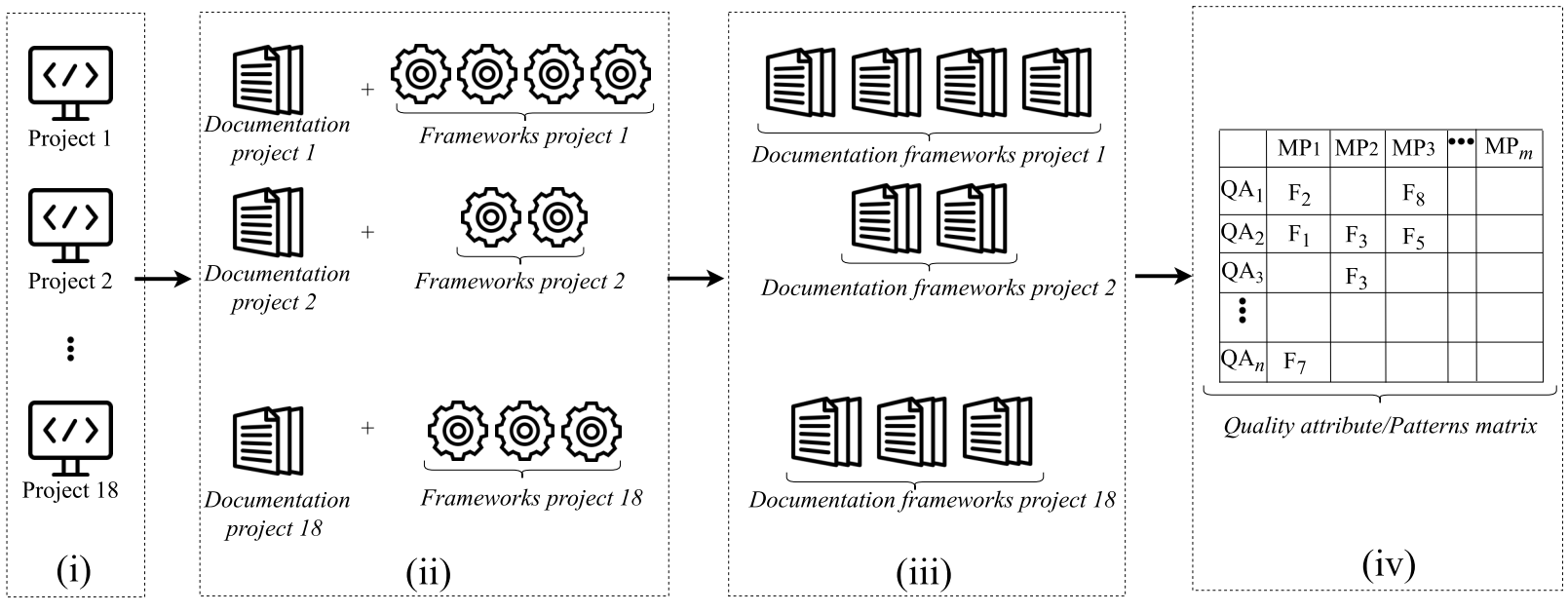


Figure 5.1: Process to obtain frameworks. "QA", "MP", and "F" represents quality attributes, microservices patterns, and frameworks, respectively.

practice regarding the use of frameworks in open-source microservices-based projects. Often, open source projects developers do not start from scratch but improve, renew or innovate on a common already existing base. The difference is that traditional software providers create and develop their products from scratch and "in-house", while in open source, it is possible to rely on a solution to improve it and tailor it to specific needs. Then, the state of the practice is provided by the issues or logs of each open source project. For example, GitHub assigns to each open source project its corresponding open and closed issues. These types of issues store the main concerns that the community writes about a certain project (open issues), and, at the same time, they also store those issues that were solved by the leading developers and architects of each project (closed issues). Therefore, open and closed issues become an essential and fundamental source of architectural knowledge.

Consequently, at this point, we proceed to study the open and closed issues of the projects described in Figure 5.1 with the aim of identifying the main issues associated with availability, scalability, interoperability, and security (see Figure 5.2).

Table 5.2 summarizes the total of open and closed issues reviewed.

From a total of 4,393 issues, we collected 958 relevant issues where developers describe

Table 5.2: Number of issues per project

Project	Name	URL (https://github.com/)	Latest commit	Open	Closed	Total	Relevant	Filtered
PT1	Gizmo	/nytimes/gizmo	2020-05-21	17	46	63	21	8
PT2	Magda	/magda-io/magda	2020-08-04	207	1,053	1,260	235	14
PT3	Acme Air	/acmeair/acmeair-nodejs	2016-08-25	6	1	7	3	0
PT4	Socks Shop	/microservices-demo/microservices-demo	2019-03-28	62	276	338	45	13
PT5	Piggy Metrics	/sqshq/PiggyMetrics	2020-02-02	85	120	205	30	0
PT6	Apolo	/ctripcorp/apollo	2020-08-01	139	1,033	1,172	332	14
PT7	Share bike	/JoeCao/qbike	2019-01-03	4	0	4	1	0
PT8	eShop	/dotnet-architecture/eShopOnContainers	2020-08-05	40	575	615	89	13
PT9	Warehouse	/HieJulia/warehouse-microservice	2018-03-03	0	0	0	0	0
PT10	Micro. Reference	/mspnp/microservices-reference-implementation	2020-07-26	17	17	34	5	5
PT11	Vehicle tracking	/mohamed-abdo/vehicle-tracking-microservices	2019-04-19	0	0	0	0	0
PT12	EnterprisePlanner	/gfawcett22/EnterprisePlanner	2018-04-05	5	1	6	0	0
PT13	Micro company	/idugalic/micro-company	2020-07-03	0	10	10	6	5
PT14	Freddy's bbq joint	/william-tran/freddys-bbq	2017-06-04	1	2	3	1	0
PT15	Photo uploader	/nginxinc/mra-ingenious	2018-09-04	1	5	6	2	0
PT16	WeText	/daxnet/we-text	2017-11-15	0	1	1	0	0
PT17	Pitstop	/EdwinVW/pitstop	2020-07-27	1	12	13	7	5
PT18	SiteWhere	/sitewhere/sitewhere	2020-06-29	33	623	656	211	10
Total						4,393	988	87

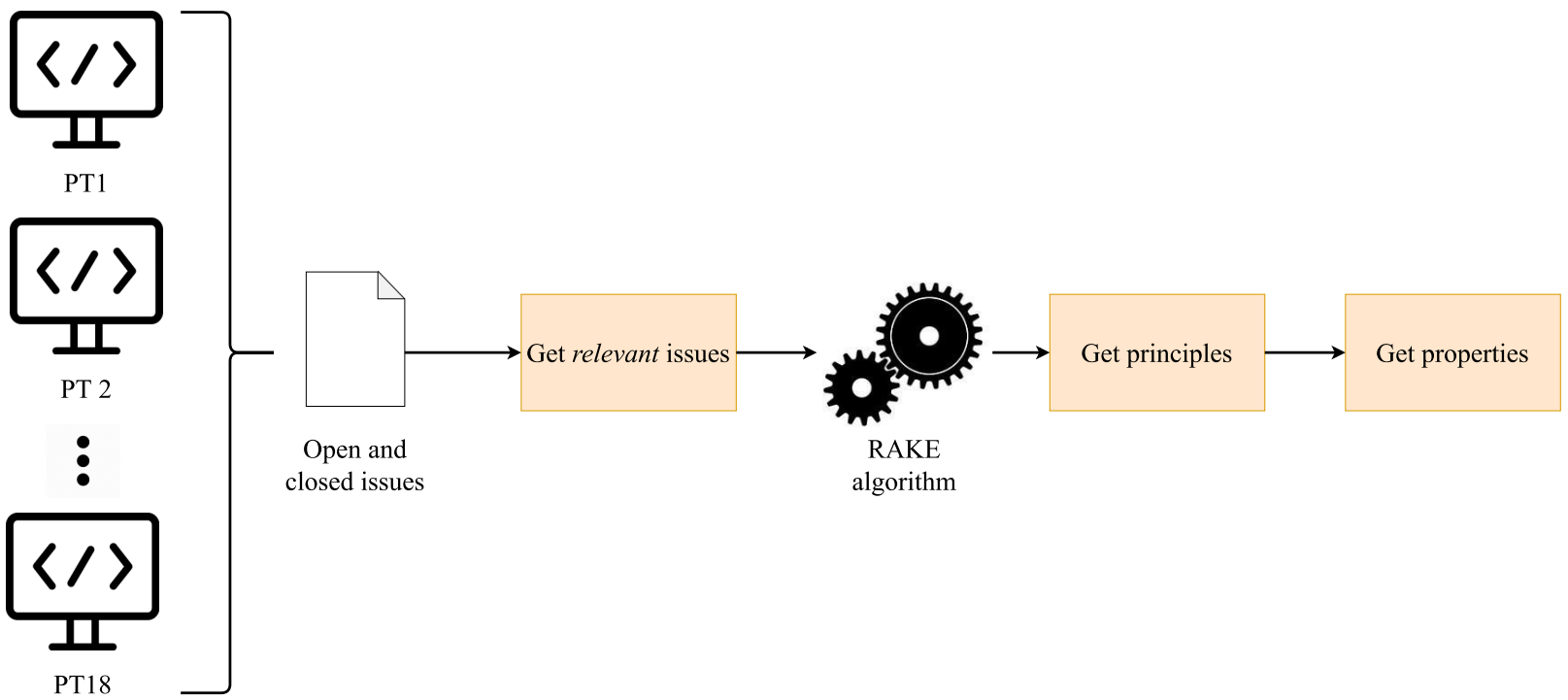


Figure 5.2: Process for obtaining quality properties.

their rationale to select frameworks in the context of availability, scalability, interoperability, and security. To obtain *relevant* issues, we omitted technical, configuration issues and bugs.

Subsequently, we proceed to apply the following steps in order to obtain the final set of filtered issues:

- *Delete redundant issues*: In this step, we eliminate those issues that are similar or that were focused on the same topic.
- *Select issues only written by the project's developers/architects*: We decided to use this filter because we realized that users outside the project mostly consulted operational as well as technical questions. Although we detected some issues written by these types of users that recommended a relevant aspect of the project, we decided to lean towards the developers/architects of the project, since they answered most of the issues.
- *Omit those issues not related to the QAs addressed in this doctoral thesis*: Although the projects use a large number of frameworks, we limit the issues to those related to QAs related to this doctoral thesis.

After the filtering process, we obtained 87 frequent reasons categorized by three principles: *instances*, *configuration*, and *communication*. Additionally, some issues related to these principles are described.

- **Instances**: In general aspects, developers and architects refer to instances when they described issues related to fail-over, uptime and scalability concerns. Instances are one of the most important benefits of microservices and it is possible to scale services independently.

- **Configuration**: A large number of issues expressed that the use of frameworks is largely due to the facilities that they provide, regarding configurations. Many of these, such as Netflix Hystrix, Apache Turbine, RabbitMQ and other frameworks provide facilities for configuration management.

- **Communication**: In microservices architecture, two types of communication protocols are mentioned; synchronous and asynchronous. Synchronous communication refers to when service A sends a request to service B and waits for a response from the service B. Asynchronous communication means service A habitually does not wait for a response [104]. The benefit of representing the communication type is to provide insight into how to include microservices asynchronously to maintain the independence of microservices. Several issues described the use of frameworks to satisfy one of these two types of communication.

Instance issue

...Rather than requiring access to tenant management via gRPC, allow each microservice to host the tenant management API directly. Now that tenant management is completely held in Zookeeper and considering the fact that all microservices already have a Zookeeper connection, there is no need for the gRPC overhead for tenant operations. The tenant management gRPC API is still hosted by the instance management microservice, but the underlying API will be made available to all microservices. This will speed up system bootstrapping and avoid the extra network access for functions all of the microservices rely on...

Configuration issue

...We noticed the same thing while testing the 2.1 release and are in the process of adding a fix. The cache is currently wrapped around the client API made available for communicating with remote microservices, but this is a problem when using the REST API. For instance, calls to get after update should always return the updated data. Our solution for the short term is to offer both a cached and uncached version of the APIs and the client microservice can choose which version to use. For the REST API, we will skip the cache for the primary entity you are interacting with, but use it for enrichment data. For instance, in the REST call for getting a device, the device data bypasses the cache, but the optional enrichment data for device type is loaded from cache...

...We'll also look into adding Redis so that we can support distributed caching. In the monolithic version of the system (before we used K8s and Helm) the fact that we had to run a separate server for Redis was a non-starter, but with the Helm install it should be doable. We can look at running it as a sidecar for the services which require access to cached data...

Communication issue

... They're not all running on the same IP. Each service runs in a container and, depending on the orchestrator (Docker, Kubernetes, etc), is provided with its own IP. You can use Weave Scope to visualise what happens when traffic passes through the system...

These categories were obtained using the RAKE algorithm [122]. This algorithm is an unsupervised and domain-independent method for extracting keywords from individual documents based on a score. Therefore, as each issue corresponds to different context as well as different microservices-based systems, this algorithm helps us to obtain keywords not depending on a specific domain. Briefly, the main steps of the algorithm are described below:

1. *Candidate keywords*: RAKE begins keyword extraction on a document by parsing

its text into a set of candidate keywords. First, the document text is split into an array of words by the specified word delimiters. This array is then divided into sequences of contiguous words at phrase delimiters and stop word positions. Words within a sequence are assigned the same place in the text and together are considered a candidate keyword.

2. *Keywords score*: After every candidate keyword is identified, a score is calculated for each candidate keyword and defined as the sum of its member word scores. The procedure of calculating the words (w) score is obtaining word frequency ($freq(w)$), word degree ($deg(w)$), and the ratio of the degree to frequency ($deg(w)/freq(w)$).

3. *Adjoining keywords*: RAKE looks for pairs of keywords that adjoin one another at least twice in the same document and in the same order. A new candidate keyword is created as a combination of those keywords and their interior stop words. The score for the new keyword is the sum of its member keyword scores.

4. *Extract keywords*: After candidate keywords are scored, the top scoring candidates are selected as keywords for the document.

The principles (keywords) "Instances", "Configuration", and "Communication" obtained the highest scores. In Table 5.3 we describe the categories and the reasons in concise sentences. Regarding instances, one of the advantages of microservices-based systems is to manage multiple instances of services. In this context, Table 5.3 illustrates that developers and architects select frameworks intending to facilitate to run multiple service instances on each one. Concerning configuration, developers and architects use framework for various tasks. For example, Table 5.3 describes that frameworks help (i) to manage service registries, (ii) to aggregate streams, (iii) to change configurations dynamically, among other features. About communication, frameworks help in two main tasks: communication between services and orchestration of them.

Table 5.3: Principles and reasons to choose frameworks

Principle	Reason
Instances	*Protect controllers from external access*
	Locate service instances
	Determine locations of available service instances
	Receive heartbeat messages from each instance
	Control over latency and failure from dependencies
	Generate metrics on execution outcomes and latency
Configuration	*Make core services work*
	Simplify loads config files from the local classpath
	Change app configuration dynamically
	Provides convenient annotations
	Store static content
	Registry services
	No additional hop for every over-the-wire invocation
	Stop cascading failures
	Add a fallback method
	Aggregate streams
	Centralized approach to configuration management
Communication	*Secure machine-to-machine communication*
	Route requests to appropriate microservices
	Control over the behaviour of HTTP and TCP clients
	Create a declarative HTTP client
	Event-driven communication
	Orchestation of services
	Persist the events to a datastore

Then, using as reference the reasons described in Table 5.3, we proceed to identify the properties that characterize QAs manually. To do this, analytical reading of each issue is made in order to identify properties that characterize sets of issues. For example, if we find that 20 issues talking about "high isolation between services" (in general), it can be established that a potential property is "high isolation," and this is related to availability. Consequently, Tables 5.4, 5.5, 5.6, and 5.7 describe the properties found for availability, scalability, interoperability, and security, respectively.

To validate the identified properties, different types of empirical studies are executed. We have decided to execute different empirical strategies according to the amount of academic literature for each quality attribute (using μAzimut) to validate the quality

Table 5.4: High-availability properties in microservices-based systems

Id	Name	Description
P1	*High detection of failed host*	This property defines the capability of detect failed hosts in order to a load balancer can stop requests to them.
P2	*Intermittently asynchronous data transmission*	Communication property where a message sender does not wait for a response.
P3	*Regular snapshots*	Property related to recovering the system from planned host maintenance owing to hardware upgrade, soft reboot, among others.
P4	*Efficient duration of timeouts periods*	Property that prevents remote procedure calls from waiting indefinitely for a response.
P5	*High isolation*	The property where each microservices is its own encapsulated application.
P6	*Effective load balancing*	Efficiently distributing incoming network traffic among groups of backend servers.
P7	*Quick broken state recovery*	Capability to restart states when they are broken for a more extended period.
P8	*High control of failure propagation*	Property which indicates the capability of isolate failures through a good definition of service boundaries.
P9	*High service monitoring visibility*	Property that allows visibility into the health of the microservice architecture.
P10	*Periodic heartbeat signal*	Property related to the periodic signal to check the status of services.
P11	*Low application restarting*	This property refers to the low rate of restart services when a failure occurs.
P12	*Efficient resources consumption*	Property related to the impact on the resources consumption (hardware/software) of a system.

properties. In the following points, we detail the studies conducted on each quality attribute.

- *Availability*: We conducted a Rapid Review [29] with the aim of obtaining a set of primary studies related to high availability in microservices architectures. The primary studies obtained from this review have allowed us to validate the properties obtained. More precisely, we did an exhaustive work to identify the properties in the primary studies reviewed. Additionally, to complement the results obtained from the review, we decided to conduct a survey with practitioners to validate the properties. In section 5.3 we describe in more detail the survey and the results obtained [94].

- *Scalability*: As with the quality attribute above, we conducted a Rapid Review

Table 5.5: Scalability properties in microservices-based systems

Id	Name	Description
S1	*Effective technical duplication*	This property focuses on the need to execute multiple identical copies of an application behind a load balancer, to improve its capacity and availability.
S2	*High functional decomposition*	This property focuses on separating services and data along noun or verb boundaries, allowing segmentation of teams and ownership of code and data.
S3	*Effective data partitioning*	This property focuses on grouping related items.

Table 5.6: Interoperability properties in microservices-based systems

Id	Name	Description
I1	*High cooperation among components*	The platform demands the capability of exchange information among services and devices (such as sensors) to use data that has been exchanged for research and patient monitoring purposes.
I2	*High coordinated orchestration among components*	This property points to a control mechanism to coordinate, manage and sequence the invocation of particular components (which could be ignorant of each other).

to find academic evidence regarding the properties obtained. The primary studies obtained from the review allow us to find evidence of the properties in the academic literature. Given the number of properties of this quality attribute, we do not conduct a survey. Instead, to complement the results obtained from the survey, we conducted a case study with a real system where we used the scalability properties [95].

- *Security*: This is an emerging quality attribute in microservices. For this reason, we have decided to conduct a systematic mapping study to collect primary studies related to security mechanisms used in microservices-based systems [137].

- *Interoperability*: It was possible to validate the properties of Interoperability based on the theory of this quality attribute in software architecture. In the definition of Interoperability described in [104] it is possible to verify that the properties identified in Table 5.6 effectively correspond to the context of Interoperability.

Table 5.7: Security properties in microservices-based systems

Id	Name	Description
C1	*High level of individuals, groups, or systems authorization*	Capacity of control users to grant them limited access to platforms systems resources without having to expose their credentials.
C2	*High-security authentication*	Capacity of verifying the identity of a person or device.
C3	*Effective credentials management*	Property related to the management of credentials, making them available to less or high privileged users for authentication to other systems without giving them access to the credentials themselves.
C4	*Effective access control*	Property that allows verifying if an entity requesting access to a resource has the necessary rights to do so.

5.3. Study design - Availability

Using the same microservice-based project reported in Figure 5.2, we adapted the properties of Table 5.2 to the context in which they were obtained. More precisely, the issues analyzed in Table 5.2, in general, describe *positive* or *negative* situations where developers or architects expressed design principles. For example, the following sentences describe a positive and negative situation concerning RabbitMQ.

- Positive issue: "*the Event bus is based on async communication based on RabbitMQ [...] It is used for integration events derived from transactions that happened in any of the microservices and when other microservices need to be aware of those events*".
- Negative issue: "*If RabbitMQ is unreachable when the service tenant starts up, the existing connection is lost for whatever reason, it won't reconnect*".

Considering the positive or negative situations of each issue, we proceeded to identify the frameworks related to the high availability issues to describe each framework's positive and negative sentences. The frameworks where we found evidence of high availability properties are RabbitMQ, Apache Zookeeper, Docker, Kubernetes, Apache Cassandra, Netflix Eureka and MongoDB (see Table 5.8). Finally, these sentences will be used in the survey. We believe that contextualizing the high availability properties in framework-based contexts can facilitate the evaluation of practitioners in the survey.

Table 5.8: Statements on the positive/negative impact of frameworks on high-availability properties

Statement	Impact	Prop.
S1: RabbitMQ permits implementing asynchronous communication among microservices	Positive	P2
S2: RabbitMQ can affect the number of long-lived connections if a connection is lost	Negative	P1
S3: Apache Zookeeper supports session timeouts	Positive	P4
S4: Apache Zookeeper can significantly impact on the resources consumption (hardware/software) of a system	Negative	P12
S5: Docker is a good solution for packaging and isolating microservices into containers	Positive	P5
S6: Docker can significantly impact on the resources consumption (hardware/software) of a system.	Negative	P12
S7: Docker can hamper the heartbeat-based monitoring	Negative	P10
S8: Kubernetes supports services monitoring	Positive	P9
S9: Kubernetes can significantly impact on the resources consumption (hardware/software) of a system	Negative	P12
S10: Apache Cassandra supports database service replication	Positive	P6
S11: Apache Cassandra can significantly impact on the resources consumption (hardware/software) of a system	Negative	P12
S12: Netflix Eureka helps monitoring microservices, by allowing to locate them and their logs	Positive	P9
S13: Netflix Eureka can significantly impact on the resources consumption (hardware/software) of a system	Negative	P12
S14: MongoDB supports the isolation of database services	Positive	P5
S15: MongoDB can hamper the heartbeat-based monitoring of database services	Negative	P10
S16: MongoDB automatically routes requests to the appropriate databases	Positive	P2
S17: MongoDB does not support fanning simple transactions out to different database partitions	Negative	P8

5.3.1. Evaluating high-availablity issues, based on a survey

To evaluate the statements described in Table 5.8 among industrial practitioners, we conducted an online survey (following the guidelines described in [78] and [79]). We hereafter present the survey setting.

Scope and target audience

The scope of the survey falls within the IT industry, and in particular that regarding the usage of microservices to deliver core businesses. Within such scope, the focus was

on the usage of open-source frameworks to develop and deploy microservices.

The target audience hence entailed industrial practitioners daily working with microservices, either directly part of the authors' relationship networks or reachable through professional networks, frameworks-oriented social networks and technology-related portals. Further practitioners were also reached by exploiting the snowballing sampling approach [79]. The target audience was given the possibility to participate in the survey in the period lasting from the 1st of March 2019 to the 31st of October 2019.

Goal and research questions

The goal of the survey was to evaluate the impact of open-source frameworks on the high-availability of microservices, as per its actual perceiving by industrial practitioners daily working with microservices. In particular, we aimed at evaluatiing their level of agreement with the statements we elicited in Table 5.8. To this end, we established the following research questions:

RQ3.2
Which is the level of agreement of respondents concerning positive contributions related to specific frameworks and high-availability properties?

By answering this research question, we aimed at identifying to which level frameworks are considered to positively impact on high-availability properties.

RQ3.3
Which is the level of agreement of respondents concerning negative contributions related to specific frameworks and high-availability properties?

As for RQ3.2, the goal of this research question was to understand to which level frameworks are considered to negatively impact on high-availability properties.

Survey questions and response format

The survey questions were built by directly submitting the statements in Table 5.8. Respondents were then given the possibility to make their level of agreement with such statements by marking one of the following three possible responses: *Strongly agree*, *Agree*, *Not agree*.

We used an online questionnaire as survey methodology since we intend to ask precise questions in the context of availability. Furthermore, such a methodology is known to allow to search and explore widespread opinions on a specific topic. For specific domains (including that considered by our study), online questionnaires offer several other advantages, e.g., the effort to handle the questionnaire is reduced (for the participants), it is possible to navigate easily through the questionnaire [115].

Data analysis

For each response, we proceed to perform a descriptive analysis based on the frequency of responses in order to better understanding of results. In addition, with the aim of further elaborating the key findings regarding the survey, we performed a series of brainstorming sessions. Obtained results are presented and discussed in the following section.

5.3.2. Results and discussion

We hereby illustrate the outcomes of our survey. More precisely, we first show the data about the recognised positive and negative impacts of frameworks, and we then discuss the outcomes of the study.

In doing so, we shall establish the agreement on a statement by defining a threshold based on the absolute majority of respondents (as in [46]). The latter means that a candidate statement must obtain over a half of the votes to get agreed, i.e., given n the total amount of respondents, it must get $(n+1)/2$ votes if n is odd, or $(n+2)/2$

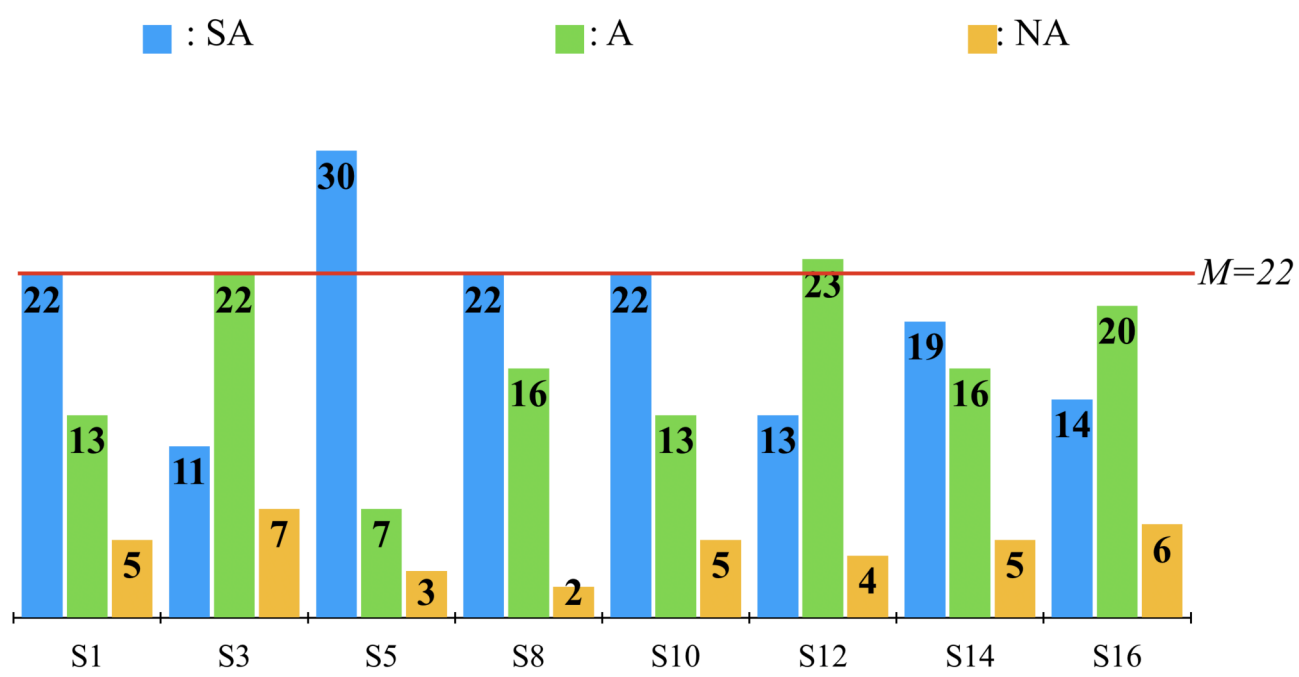


Figure 5.3: Survey results regarding positive contribution of frameworks. “SA” correspond to *Strongly agree*, “A” to *Agree*, “NA” to *Not agree*, and “M” to the absolute majority

votes if n is even. As a total of 40 practitioners participated to our survey, this means that a candidate statement is agreed if at least 21 respondents agree with it.

Practitioners’ recognition on considered impact of frameworks

Figures 5.3 and 5.4 plot the frequencies of answers on the statements concerning the impact of frameworks on high-availability properties.

In both cases, we can observe that the vast majority of respondents agrees on the proposed statements (if we considers the sums of *Strongly Agree* and *Agree* answers), hence confirming that our statements on the impact of frameworks on high-availability properties are reflected by the community.

Focusing on Figure 5.3, we observe that, according to the responses, there is a ”strong agreement” on statements S1, S5, S8, S10 and S14, and an “agreement” on statements S3 and S12. Statements S14 and S16 instead do not have a majority of *Strongly Agree* or *Agree* answers, but their sum farly outgoes the threshold.

Focusing on Figure 5.3, we observe that respondents where lighter in agreeing with statements, as their majority ”agrees” (not strongly) on all statements but S6 and S17.

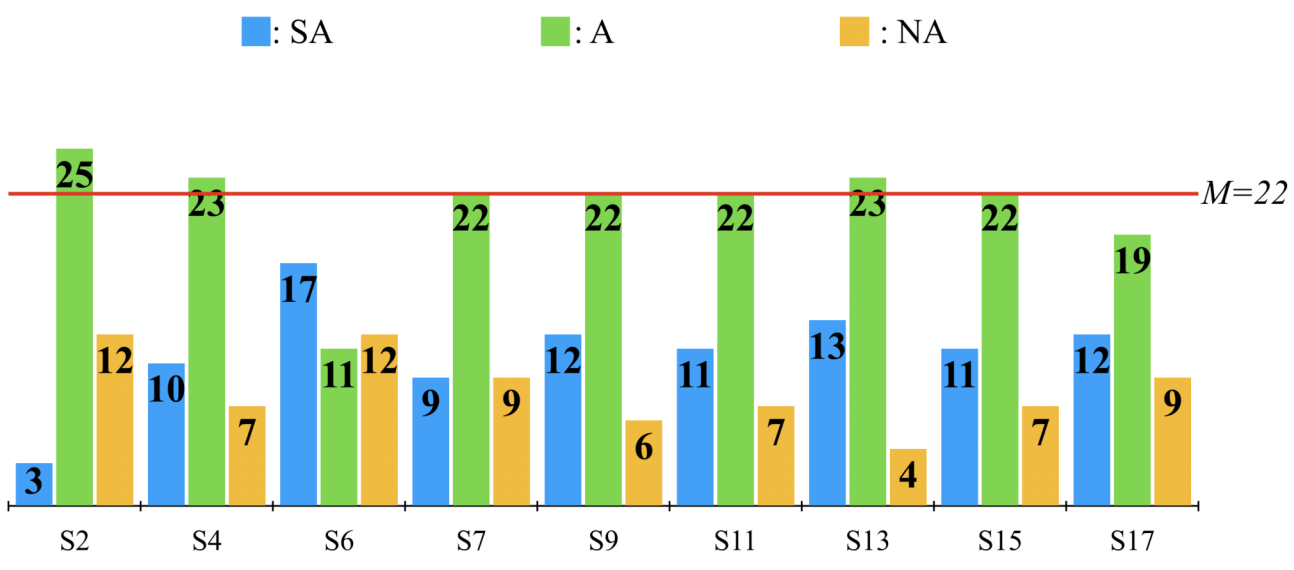


Figure 5.4: Survey results regarding negative contribution of frameworks. "SA" correspond to *Strongly agree*, "A" to *Agree*, "NA" to *Not agree*, and "M" to the absolute majority

However, if we consider the sum of *Strongly agree* and *Agree* answers, we observe an overall agreement on all statements.

Discussion

On the positive compensations side, the survey results indicate that RabbitMQ, Docker, Kubernetes, and Apache Cassandra positively satisfy (*Strongly agree* selection) properties P2, P5, P9, and P6, respectively. Regarding Apache Zookeeper and Netflix Eureka, although respondents indicate that these frameworks are a positive contribution to properties P4 and P9, results indicate that there is no unanimous agreement (*Agree* selection) regarding the positive effect of these frameworks for the P4 and P9.

About negative compensations, the most affected property is P12. This indicates that Apache Zookeeper, Docker, Kubernetes, Apache Cassandra, and Netflix Eureka compromise resource consumption. However, the level of agreement illustrates that the respondents are not categorical with this situation since everyone selected the *Agree* option. This is because, according to what we could investigate, property P12 will be affected depending on the size of the microservices project. For properties P1 and P10, the same situation occurs.

According to the results described in Figures 5.3 and 5.4, in the statement S6, S14,

S16, and S17 it is not possible to identify the level of agreement. 3 of the 4 statements correspond to MongoDB and 1 to Docker. To investigate why there is no agreement in these statements, we conducted a rapid review [28] to establish the final decision regarding which level of agreement should be assigned for these statements.

Regarding statements of MongoDB, Ueda *et al.* [136] conducted an empirical study with attention to understand the characteristics to design an infrastructure optimized for microservices. The authors mention that, since MongoDB implements a transactional ACID model (Atomicity, Consistency, Isolation, and Durability) on different documents, the way on how the model reference documents to ensure integrity is done by nesting documents, promoting the isolation of the data. Likewise, the same authors describe that one of the significant disadvantages of MongoDB is that it does not support transactions at the moment of updating more than one document or collection. Gadea *et al.* [56] also mention this observation in the conclusions of his experience of using microservices architecture in a collaborative document editing system. Nevertheless, new MongoDB releases (such as 4.0[2]) provide the ability to perform multi-document transactions against replica sets.

Other empirical studies, such as [138], mention that MongoDB has the capability to aggregate, route and indexes the unique document ID automatically, improving the performance and scalability in queries against indexed and non-indexed collections, for various (virtual) server hardware configurations [58].

About the Docker resource consumption statement, although some authors (such as [8] and [126]) describe that this situation depends on the project's characteristics, Casalicchio *et al.* [30] provide an extensive empirical study related to docker performance. One of the conclusions emerged from the authors' experiments is related to the tendency of Docker to use high disk I/O overhead. The authors mentioned that Docker heavily penalizes the execution time when the number of container increase.

Therefore, the studies' results corroborate the answers of sentences S6, S14, S16, and S17. This means that we could take into account the answers with the highest tendency

[2]https://docs.mongodb.com/manual/release-notes/4.0/#multi-document-transactions

for these sentences (S6→SA, S14→SA, S16→A, and S17→A) in the first instance. Nevertheless, although the studies provide significant research related to the statements' frameworks, there is not sufficient data and evidence to conclude irrefutably the level of agreement of the aforementioned sentences. As future work, we plan to further investigate this finding in order to establish a level of agreement to these sentences.

5.4. Study design - Scalability

This case study is related to the application of a scalability pattern language [95] (see Table 5.9) to document the architectural design of an actual real-time vehicle positioning capture system for a public transportation Chilean company. The main idea of using this pattern language is that it is mainly composed of the properties described in Table 5.5. In this case study, the main research question is the following:

RQ3.4

Can a scalability pattern language support architectural decision making in order to satisfy scalability requirements and properties in microservices-based systems?

Table 5.9: The structure of the patterns language

Dimension	Patterns category	Architecture patterns	Architecture decomposition
X-axis scaling	Load balancer patterns	Internal load balancer pattern	Communication components
		External load balancer pattern	
Y-axis scaling	Decomposition patterns	Decompose the monolithic pattern	Services
		Change code dependecy pattern	
		Circuit breaker pattern	
		Database per service pattern	
Z-axis scaling	Grouping patterns	Service discovery pattern	Service discovery components
		Deploy cluster pattern	
		Container pattern	

The system under study has 3 types of components: (1) messaging component: responsible for distributing and gluing the data packages for further processing. These data come from active vehicles. (2) cache component: due to the fact that it is necessary to represent data in real time, it is important to have intermediate storage to optimize recurring queries to data sources, therefore, this component is in operation, in order to enhance such procedure. (3) data analytics component: this component is in charge of processing large volumes of data and delivering information to support decision making in real time. The workflow of the system is the following: a vehicle reports its positioning to the system, which is received by the messaging component that processes it and delivers it to the data analytics component, considering that the data can go through the caching component depending on the the resources required. Finally, the processed data is displayed on a web platform and available to the end user.

In order to use the pattern language, 5 steps must be executed, which allow analyzing at different levels how to approach scalability in microservices-based systems. In the following paragraphs, the steps and their corresponding description are mentioned.

Step 1. *Identify and describe scalability requirements.* In this step, we proceeded to elicit and identify scalability requirements. Once the requirements were identified, we used the properties described in Table 5.5 to contextualize the scalability requirements.

Step 2. *Build a high-level microservices architectural description.* The real-time bus position capture system provides continuous capture of positioning for moving buses. Data is sent to a central server and saved in a database (see Figure 5.5). This system will grow depending on the number of vehicles that will be added over time, and the infrastructure will adapted to meet the scalability-related requirement described for the system.

Step 3. *Identify patterns related to the X-axis property.* We identified one load balancer pattern: Internal load balancer pattern. *Rationale*: This pattern allow to have a method of balancing the load in the central server, with the aim of capturing and processing the data and information from the vehicles in a microservices style for the

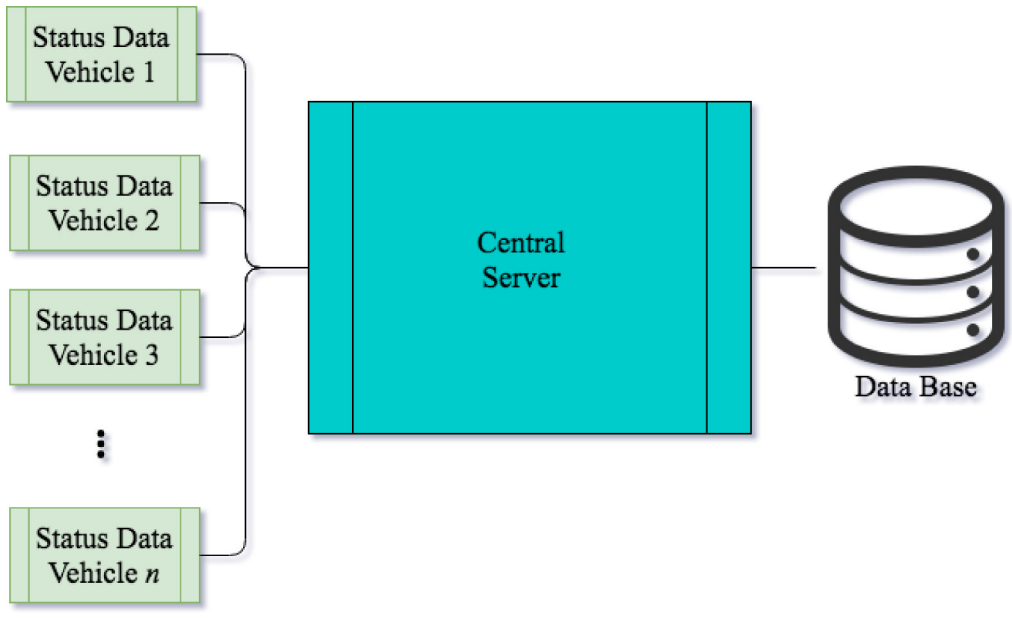


Figure 5.5: Real-time bus position capture high-level architecture

services running in the server. This pattern satisfied the availability and performance NFRs.

Step 4. *Identify patterns related to the Y-axis property.* We identified one decomposition pattern: Circuit breaker pattern. *Rationale*: This pattern allow to handle failures while using small services in a microservices architectural style, preventing malfunctioning in the system by using thresholds for the failures detections. This pattern satisfied the availability, reliability and performance NFRs.

Step 5. *Identify patterns related to the Z-axis property.* We identified one grouping pattern: Container pattern. *Rationale*: This pattern allow to decompose the software system in small services by using containers running in the central server, and together with the circuit breaker pattern, satisfied the availability, reliability and performance NFRs.

Discussion

After executing these steps, we created an architectural design following based on a microservices architectural style, focused in the scalability of the system and satisfying its NFRs through the proper patterns. At present, the system is up and running capturing and processing data constantly.

The findings of this case study point out that the properties of scalability are highly related to the theory of scalability. Abbott and Fisher [2] related scalability to three major problems that technology companies face: (1) systems downtime may be costly, (2) failures need to be quickly corrected, and (3) failure to correct outages results in more time being spent repairing failures and less on new products and features. They proposed a scale cube with three scaling dimensions: cloning entities or data and distributing work evenly across workers ("layers", property S1); separating work evenly by activity or data ("components", property S2); and separating work according to the work requester ("shards", property S3). In step 1, using the scalability properties was relevant to making design decisions for designing the architecture. In addition, contextualizing the scalability requirements allowed for a better understanding of the problem.

Therefore, since the scalability properties are directly related to the features described in the scalability theory an the proposed scalability pattern language [93], we can establish that the properties mentioned in Table 5.5 characterize scalability in microservices-based systems.

5.5. Study design - Security

In order to evaluate the security properties, we executed a systematic mapping study related to security in microservices. The main research question of this literature review is the following:

RQ3.5
Which security mechanisms have been reported in microservices-based systems research?

We obtained a total of 37 primary studies published between 2015 and the first half of 2019. We observed an increment in the number of publications of around 50% per year. Publications in 2019 decreased, but we must consider that we only review half

of the year. Most publications (54%) appeared in conferences, followed by journals (35%), and symposia (11%).

In Table 5.10 we present the security mechanisms that we searched for in the primary studies selected. We choose these security mechanisms because they are recognizable and recurring standard software structures resulting from the use of systematic design, like the security patterns referenced in Fernandez [53]. However, we did not discard other security mechanisms different to the ones listed in Table 5.10.

Figure 5.6 presents the security mechanisms reported in microservices-based systems studies. The most reported are authorization (62% of studies), authentication (54%) and credentials and control Access (46%).

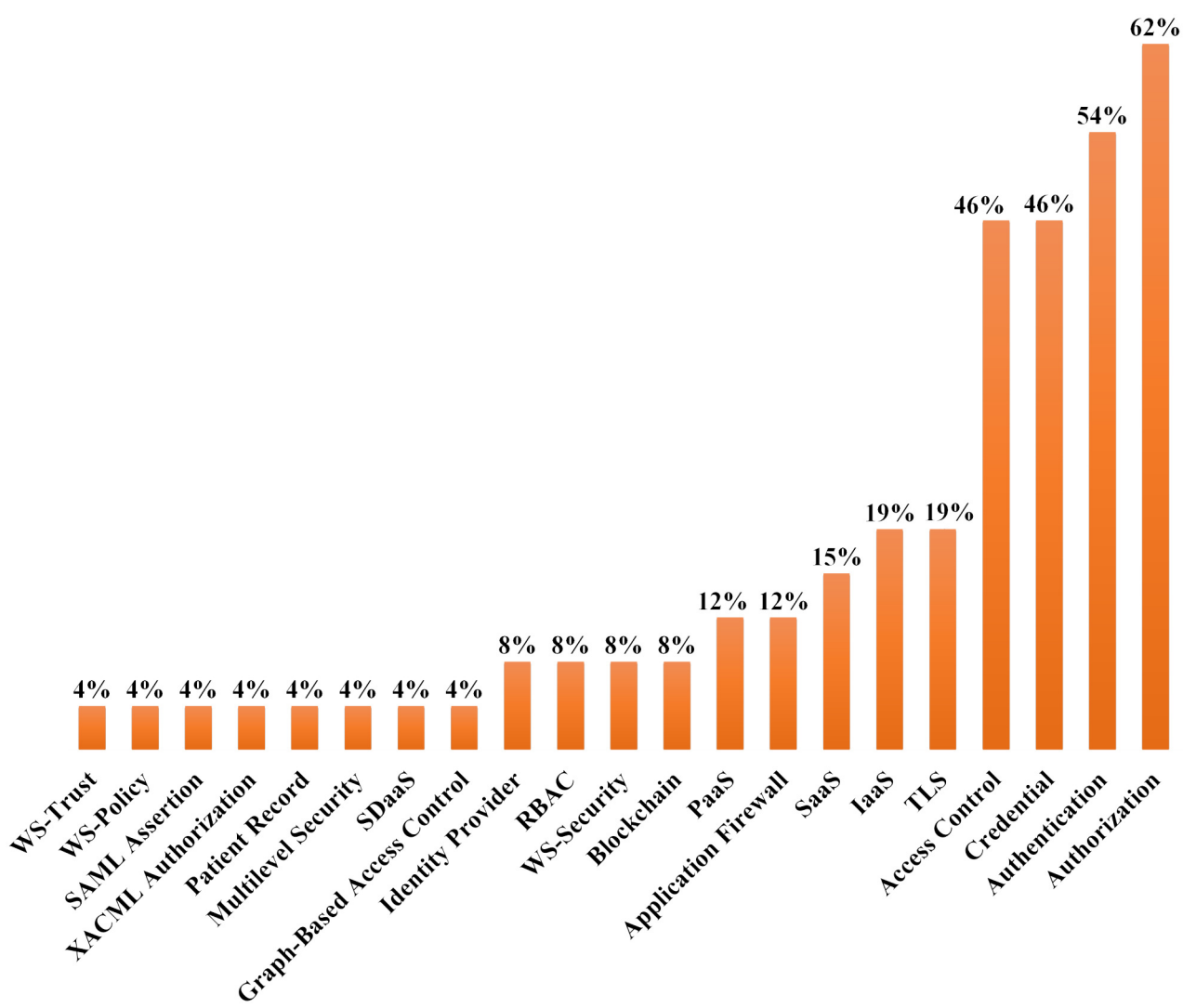


Figure 5.6: Security mechanisms reported in primary studies

These four mechanisms are closely related, and describe (i) who is authorized to access specific resources in a system and in what way, (ii) how to verify that the subject

Table 5.10: Detailed security mechanisms list obtained from primary studies

Security mechanisms	Security mechanisms
Abstract Virtual Private Network	Secure Adapter
Access Control	Secure Blackboard
Administrator Hierarchy	Secure Broker
Application Firewall	Secure Distributed Publish/ Suscribe
Asymetric Encryption	Secure Enterprise Service Bus
Authentication	Secure Handling of legal cases
Authorization	Secure Model-View-Controller
Behavior Based IDS	Secure Pipes and Filters
Circle of Trust	Secure Process Thread
Controlled-Object Monitor	Software As A Service
Credential	Systematic Encryption
Digital Signature with Hashing	Transport Layer Security
Execution Domain	TLS Virtual Private Network
Identity Provider	Virtual Address Space Structure Selection
Infrastructure As A Service	Virtual Machine Operating System Architecture
IPsec VPN	Web Services Policy Language
Multilevel Security	WS-Policy
Patient Record	WS-Security
Plataform As A Service	WS-Trust
Protection Rings	XACML Access Control Evaluation
Role-Based Access Control	XACML Authorization
Protected Entry Points	XML Encryption
Remote Authentication/Authorizer	XML Firewall
SAML Assertion	XML Signature

that intends to access the system is who it claims, and (iii) how to perform secure authentication and authorization registration, respectively. Besides, Credentials are an authorization and authentication mechanism. The remaining reported security mechanisms appear in between 19% and 4% of studies.

Two of the selected primary studies report the use of blockchain as a security mechanism used in microservices-based systems. Thanks to its decentralized structure and immutability, blockchain technology has the potential to address the relevant security and privacy challenges [50]. Nagothu *et al.* [103] use the blockchain technology to securely synchronize databases between microservices and provide proof of data manipulation in an untrusted network environment. The access authorization strategy enabled by smart contract prevents any unauthorized user from accessing the microservices and offers a scalable, decentralized access control solution.

Another relevant primary study is that of Pahl *et al.* [110], where an access control based

on graphs is proposed. This solution intercepts and firewalls between communication services and automatically creates a model of legitimate communication relationships.

The results of the review coincide with what we found in the open and closed issues regarding security concerns. The microservices community is mainly concerned with satisfying access controls, delivering secure credentials, providing authentication and authorization mechanisms. That is why we can establish that the properties described in the Table 5.7 do characterize security in microservices.

Regarding the security properties, we could see that most of them are exhibited in security patterns. These patterns are generally used to achieve specific quality objectives. The primary studies that use security patterns as security mechanisms attempt to satisfy the security properties that we find in open and closed issues. The case studies using security patterns reported in primary studies clearly describe which security properties are important in microservices-based systems and how some security patterns are related to them.

5.6. Summary

In this section, the main properties that characterize the quality attributes addressed by this doctoral thesis have been described. Through an exhaustive analysis process in microservices-based projects, open and closed issues have been investigated to identify potential quality properties manifested by developers. Subsequently, the defined properties are described, and different types of empirical studies are executed to validate the properties. The empirical studies' results suggest that the identified properties are suitable for representing quality attributes concerns in microservices context. Therefore, these properties can be used in μAzimut to systematize the architectural knowledge of microservices.

Chapter 6

Microservices tactics

In this chapter, we present the definition and description of microservices tactics related to availability, scalability, security, and interoperability. Additionally, we conducted an empirical study to investigate the use of some of these architectural tactics in microservices-based systems.

6.1. Introduction

According to [104], the complexity of microservices-based systems is determined partly by its functionality and partly by the global requirements on its development. In order to map availability, scalability, security and interoperability properties into microservices architectures, architectural tactics emerge as an alternative. These are design decisions that influence the achievement of a quality attribute response [21], allowing the evolution of the software architecture or removing obsolete decisions to satisfy changing requirements.

In this chapter, we investigated which architectural tactics for microservices (*microservices tactics*) related to availability, scalability, security and interoperability emerge to support architectural design decisions related to microservices architectures. We described a template to characterize and illustrate microservices tactics. In order to

investigate the usage of these microservices tactics, we conducted an empirical study in 17 open microservices-based systems.

6.2. Architectural tactics in microservices-based systems

In chapter 10, we identified 21 microservices patterns that address microservices architectural concerns. Then, using these patterns and at the same time, the open microservices-based systems described in Table 5.2, we proceed to read and examine documentation focusing in to identify *design principles* implemented in the frameworks [1]. We defined design principles to those design guidelines which satisfy the following attributes:

- *Stimuli*: Something that incites to action or exertion or quickens action.
- *Environment*: The aggregate of surrounding things, conditions, or influences; surroundings; milieu.
- *Response*: An answer or reply, as in words or in some action.
- *Architectural pattern*: The design principles should describe in which pattern(s) they are contextualized.
- *Quality attribute*: The documentation should illustrate which QA's the design guidelines help to achieve.
- *Parameters*: This field describes which variables are used to measure architectural tactics response.
- *Architectural elements*: This field complements the above one. It defines which elements are used to measure an architectural tactic response.

[1]The complete list of revised frameworks can be found at the following link: `https://github.com/gmarquez87/microAzimut`

From the microservices patterns and their respective frameworks, we obtained design principles that are interpretable as *tentative* microservices tactics.

Concerning security and interoperability, we have noticed that architectural tactics are evidently used to satisfy NFRs in microservices-based systems; yet, architectural tactics used for these two quality attributes do not differ from those that already exist. Regarding interoperability, the evidence of architectural tactics found is quite similar to those described in [21]. The architectural tactics for interoperability are the following:

- *Discover service*: This tactic suggests locating services by searching for a known directory service. There may be multiple levels of indirection in this location process, i.e., a known location points to another location that, in turn, can be searched for the service. The service can be located by type of service, by name, by location, or by some other attribute.

- *Orchestrate*: This tactic considers a control mechanism to coordinate and manage the invocation of particular services that could be ignorant of each other. Workflow engines are an example of the use of this tactic.

- *Tailor interface*: This tactic adds or removes capabilities to an interface. Capabilities such as translation, adding buffering, or smoothing data can be added. Capabilities may be removed. An example of removing capabilities is to hide particular functions from untrusted users.

Regarding security, in our previous work [137] we noticed that the most used architectural patterns are authentication, authorization, access Control and credentials. From these architectural patterns, we were able to identify the same architectural tactics proposed by Fernandez *et al.* [53] The architectural tactics for security are the following:

- *Authenticate Actors*: Authentication is ensuring that a user or remote computer is actually who it purports to be. Passwords, one-time passwords, digital certificates, and biometric identifications provide authentication.

- *Identify Actor*: Identifying "actors" is really about identifying the source of any external input to the system. Users are typically identified through user IDs. Other systems may be "identified" through access codes, IP addresses, protocols, ports, and so on.

- *Authorize Actor*: Authorization is ensuring that an authenticated user has the rights to access and modify either data or services. This is usually managed by providing some access control patterns within a system. Access control can be by user or by user class. Classes of users can be defined by user groups, by user roles, or by lists of individuals.

- *Limit Access*: Firewalls restrict access based on message source or destination port. Messages from unknown sources may be a form of an attack. It is not always possible to limit access to known sources. A public Web site, for example, can expect to get requests from unknown sources. One configuration used in this case is the so-called demilitarized zone (DMZ). A DMZ is used when access must be provided to Internet services but not to a private network. It sits between the Internet and a firewall in front of the internal network. The DMZ contains devices expected to receive messages from arbitrary sources such as Web services, e-mail, and domain name services.

- *Limit Exposure*: Attacks typically depend on exploiting a single weakness to attack all data and services on a host. The architect can design the allocation of services to hosts so that limited services are available on each host.

Regarding availability and scalability, we were able to find evidence of new architectural tactics related to microservices. We found an initial set of 15 tentative microservices tactics. In order to filter out microservices tactics, we proceeded to interview developers to validate and/or omit microservices tactics as a first instance. We interviewed 5 practitioners intending to know which decisions they make regarding availability/scalability issues and microservices availability/scalability tactics. For this, we analyzed scenarios corresponding to projects performed by them. Then, we presented to them the tentative microservices tactics, and for each scenario, we evaluated the advantages

and disadvantages of each microservices tactic. After brainstorming sessions between the practitioners and our research team, we obtained 10 microservices tactics in total, which are described in Tables 6.1 and 6.2.

We omitted "Quality attribute" in Tables 6.1 and 6.2 due to space limitation. Additionally, regarding availability, we find well-know architectural tactics [21], which are the following:

- *Ping/Echo*: This tactic refers to an asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path. But the echo also determines that the pinged component is alive and responding correctly. The ping is often sent by a system monitor. Ping/echo requires a time threshold to be set; this threshold tells the pinging component how long to wait for the echo before considering the pinged component to have failed ("timed out"). Standard implementations of ping/echo are available for nodes interconnected via IP

- *Monitor*: A monitor is a component that is used to monitor the state of health of various other parts of the system: processors, processes, 1/0, memory, and so on. A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack. It orchestrates software using other tactics in this category to detect malfunctioning components. For example, the system monitor can initiate self-tests, or be the component that detects faulty time stamps or missed heartbeats.

- *Heartbeat*: This tactic is a fault detection mechanism that employs a periodic message exchange between a system monitor and a process being monitored. A special case of heartbeat is when the process being monitored periodically resets the watchdog timer in its monitor to prevent it from expiring and thus signaling a fault. For systems where scalability is a concern, transport and processing overhead can be reduced by piggybacking heartbeat messages on to other control messages being exchanged between the process being monitored and the distributed system controller. The big difference between heartbeat and ping/echo

Table 6.1: Microservices availability tactics

Id	Tactic	Stimulus	Environment	Response	Arch. pattern	Parameters	Arch. elements
A1	*Preventing single dependency*	Latency in transmitting or processing data in a network	Communication over a network	Prevent single dependencies by wrapping all calls to external systems (or dependencies) in an object which typically executes within a separate thread	Circuit breaker	Number of dependencies	Services, dependencies
A2	*Set timeouts*	Timing-out calls that crash services	Microservices communication	Set timing-out calls that take longer than thresholds arbitrarily defined by the architect. There is a default, but for most dependencies, it is possible to set custom timeouts utilizing "properties" so that they are slightly higher than the measured 99.5th percentile performance for each dependency	Circuit breaker	Number of timing out calls	Dependencies
A3	*Providing fallbacks*	A client service repeatedly suffers dependency faults	Microservices communication	Maintain a small thread-pool for each dependency. If it becomes full, requests destined for that dependency will be immediately rejected instead of queued up	Circuit breaker	Number of dependency failures	Dependencies
A4	*Self-preservation*	An unexpected number of registered service clients fail in their connections and are pending eviction at the same time	Catastrophic network events	Execute an explicit unregister action when service clients are permanently going away. Any service client that fail 3 consecutive heartbeat renewals is considered to have an unclean termination and will be evicted by the background process	Service Registry	Servers operations, operation failures	Services
A5	*Asynchronous messaging*	A service fails to respond to an client request	Microservices communication	Apply asynchronous messaging protocol. The client code or message sender usually does not wait for a response. It just sends the message as when sending a message to a queue or any other message broker and waits for an acknowledgment that he broker has received the message	Messaging	Number of messages exchanged	Services

Table 6.2: Scalability microservices tactics

Id	Tactic	Stimulus	Environment	Response	Arch. pattern	Parameters	Arch. elements
S1	*Data Store Separation*	It is necessary to establish database instances	Microservices architecture	When choosing the database for a microservice, single data store simplify database structure sharing by reducing work duplication. It requires adding a tool to perform data management by operating in the background to find and fix discrepancies.	Database is the Service	Number of dependencies in database	Services
S2	*Build Separation*	A client requires separate building procedures	Microservices infrastructure	Scaling microservice architectures involves separating the building procedures. This separation helps to pull configuration files from the repository at the appropriate revision levels, making easy to add new files when new microservices are built.	Container	Number of configuration files	Infrastructure services
S3	*Container Deployment*	A system needs to group different functionalities into one set	Microservices infrastructure	Deploying microservices in containers gather deployment tasks as a whole. As long as a microservice is in a container, the container platforms know how to implement it. The information in each container is independent.	Container	Number of container instances	Containers
S4	*Network Location*	A system needs to discover new network instances	Network	Network location allows discovering services instances registering the service when it starts up by assigning dynamically network locations. When the instance ends, the service is removed. The service instance's registration is refreshed periodically using a heartbeat mechanism.	Circuit Breaker	Number of network operations	Dependencies, services
S5	*Balancing Scale*	A system triggers several services disruptions	Microservices architecture	Proactive monitoring is critical before many applications become affected and produce business disruptions. Cloud monitoring adds a view to controlling balancing metrics on the entire production environment in one place; by identifying and tracking services, notifications can be issued before impact becomes widespread.	Service Discovery	Number of services disrupted	Services

Table 6.3: Profile and experience (years) of practitioners

	Domain	Work exp.	Microservices exp.
Practitioner 1	Retail sales	20+	6
Practitioner 2	Retail sales	10	6
Practitioner 3	Retail sales	3	2
Practitioner 4	Retail sales	3	2.5
Practitioner 5	Banking	8	3

is who holds the responsibility for initiating the health check the monitor or the component itself.

- *Sanity Checking*: This tactic checks the validity or reasonableness of specific operations or outputs of a component. This tactic is typically based on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny. It is most often employed at interfaces, to examine a specific information flow.

- *Transactions*: The most common realization of the transactions tactic is "two-phase commit" (a.k.a. 2PC) protocol. This tactic prevents race conditions caused by two processes attempting to update the same data item.

- *Removal from Service*: This tactic refers to temporarily placing a system component in an out of-service state for the purpose of mitigating potential system failures. One example involves taking a component of a system out of service and resetting the component in order to scrub latent faults (such as memory leaks, fragmentation, or soft errors in an unprotected cache) before the accumulation of faults affects service (resulting in system failure).

- *State Resynchronization*: Is a reintroduction partner to the active redundancy and passive redundancy preparation-and-repair tactics. When used alongside the active redundancy tactic, the state resynchronization occurs organically, because the active and standby components each receive and process identical inputs in parallel. In practice, the states of the active and standby components are periodically compared to ensure synchronization.

6.3. Study design

6.3.1. Goal and research questions

In the previous section, we described new microservices tactics that we found in our research, as well as well-known architectural tactics. In this empirical study, we focused on validating the microservices tactics described in Tables 6.1 and 6.2. To this end, we analysed the documentation and source code of the microservices-based systems described in Table 5.2[2].

Subsequently, the research questions are:

RQ3.6
How many microservices availability and scalibility tactics is possible to recognize in source code?

By answering this research question, we aim at illustrating the frequency of availability and scalability tactics usage in source codes corresponding to the projects set described in Table 5.2.

RQ3.7
How many microservices availability and scalibility tactics is possible to recognize in projects' documentation?

Following the same idea of RQ3.6, the goal of this research question is to describe the frequency of availability tactics identification in the projects' documentation. When we refer to *the documentation*, we include README files, projects' open and closed issues, wikis, and others.

[2]We omitted a project because its source code was written using the Go programming language. Since this language is emerging, we decided not to consider it in this study.

6.3.2. Scope

The scope of this study is related to microservices projects in the context of Open Source Software (OSS). These kinds of projects have followed recent years of growth among users and companies that integrate open source software into their activities. Its multiple advantages and dynamism make OSS's an attractive work tool for all types of companies and developers [129].

6.3.3. Analysis

To conduct the analysis, (i) we reviewed the documentation and (ii) we examined the structure and source code (see Figure 6.1).

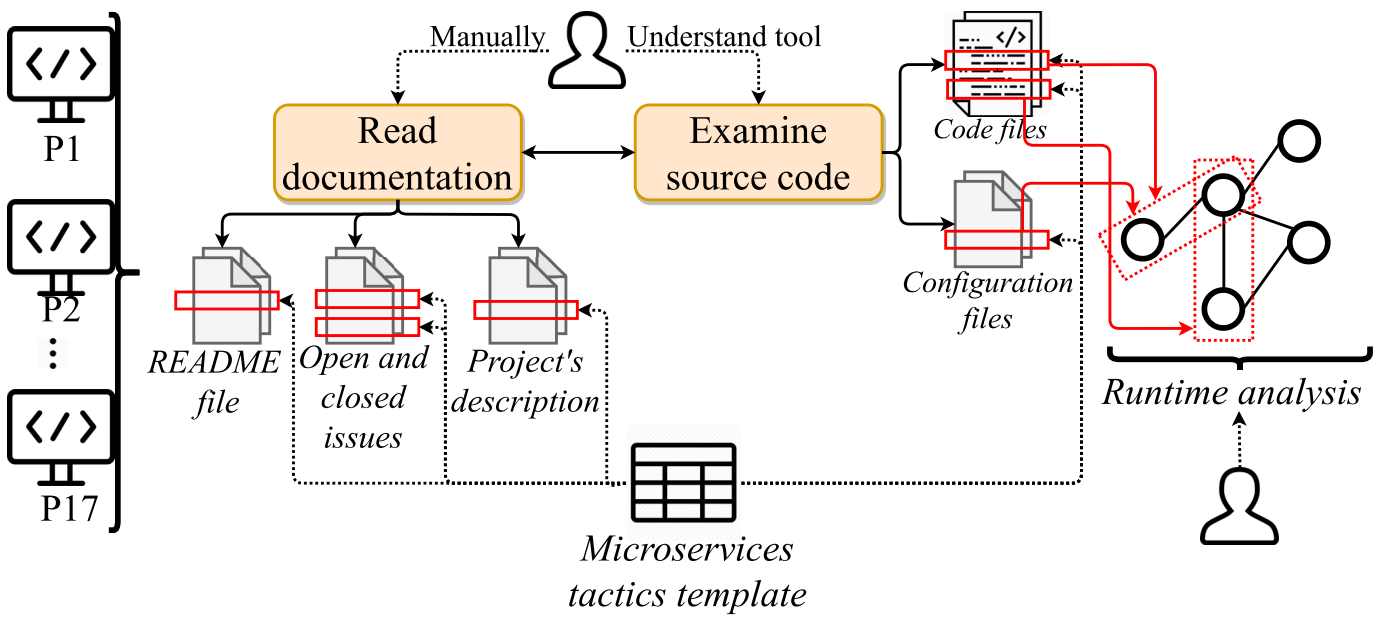


Figure 6.1: Analysis overview

With regard to documentation, we examine for keywords that indicate signs of microservices tactics. Each indication found was analysed manually in order to establish whether it corresponds to microservices tactics. On the code side, some proposals automatically identify architectural significant code [102]. Nevertheless, this proposal uses previously known architectural tactics. Given that we are looking for a new proposal of microservices tactics in this case study, we have decided to look for signs of microservices tactics by searching for keywords in the code and analysing the traceability

of the source codes using the Understand tool[3]. This tool allows us to explore code snippets dynamically and, at the same time, generate graphs and documentation of each project's source codes.

6.4. Results and Discussion

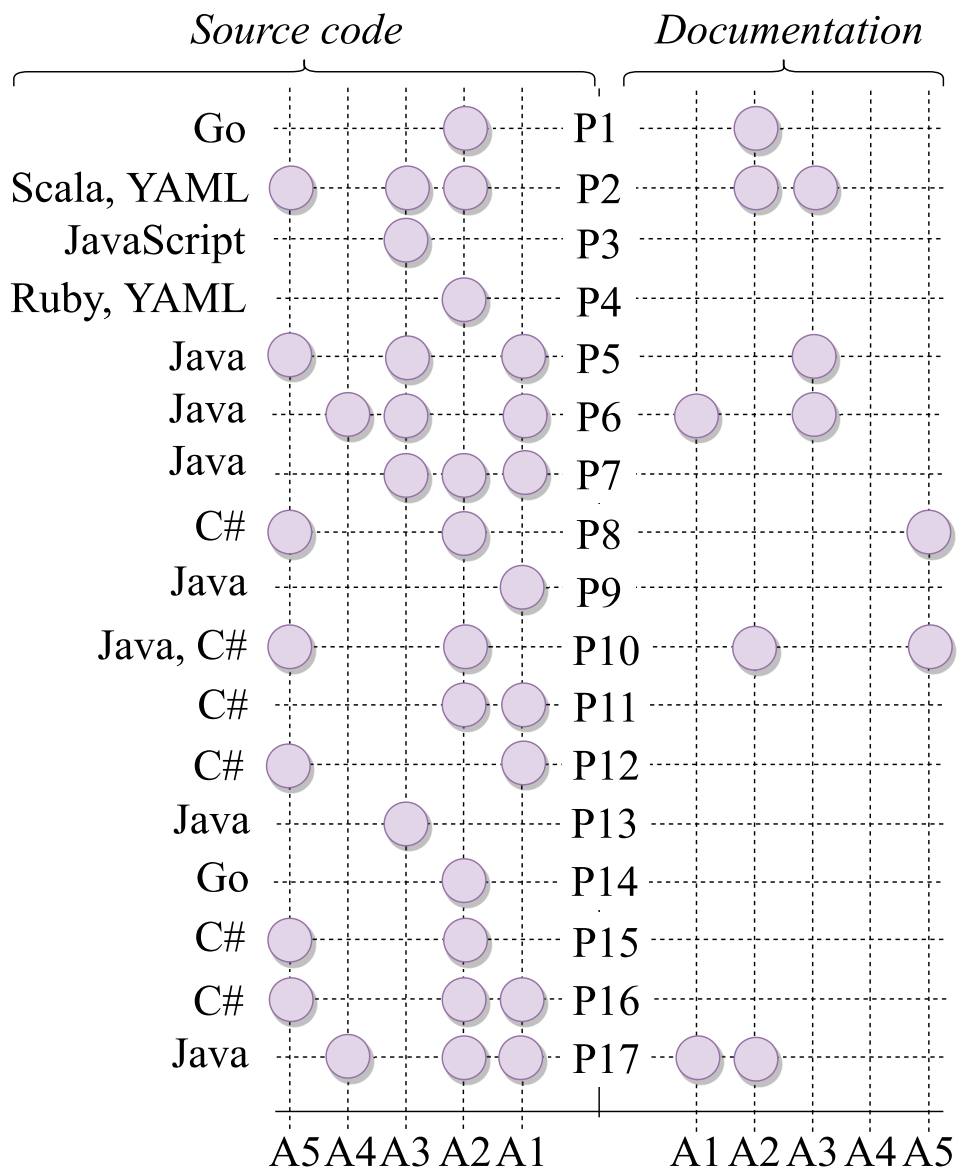


Figure 6.2: Presence of microservices availability tactics in source code and documentation

This section answers the research questions and discusses key findings.

[3]https://scitools.com

6.4.1. Code analysis

Figure 6.2 describes the frequency of microservices availability tactics' presence in the open projects as well as the programming languages which the evidence of microservices tactics is found, where the tactic T2 is the most used (65%), succeeded by T1 (47%), T5 (41%), T3 (35%), and T4 (12%).

The goal of this microservice tactic is to provide mechanisms that allow stopping waiting for a response that will not come. At the time when we reviewed the source code, we realized that open microservices projects developers use this microservice tactic because networks are fallible (projects P1, P5, and P17 emphasize this). Even when communication can be established between microservices in a network, any of the elements that intercede in the network (computers, routers, and others) could be broken at any time. When this happens, a service cannot wait forever for an answer that will never come. Although T2 is the most used microservice tactic, it is essential to set which will be the timeout value because confusions on this parameter may produce an *anti-pattern* [118]. An example of this microservice tactic is described in Figure 6.3. In order to avoid this, two solutions are proposed. The first is to calculate the database timeout within the service and use that as a base for determining what the service timeout should be. The second solution, which is by far the most popular technique, is to calculate the maximum time under load and double it, thereby giving you that extra buffer in the event it sometimes takes longer.

Timeouts provide mechanisms that allow stopping waiting for a response that will not come. In turn, provide failure isolation; if there is a problem in some other system, subsystem, or device, this should not compromise the microservices-based system as a whole.

On the other hand, the microservice tactic with less presence is T4. The aim of this tactic to avoid poor network connectivity failure. Nevertheless, although it is reasonable to associate the usage of well-known architectural tactics to one or more application frameworks [74], this microservice tactic is generally used by Netflix Eureka. The relationship between T4 and the aforementioned framework is that Eureka servers

```
1  private HttpComponentsAsyncClientHttpRequestFactory
2   getCustomClientHttpRequestFactory(
3   CloseableHttpAsyncClient client) {
4
5   HttpComponentsAsyncClientHttpRequestFactory
6   cHttpReqFact =
7    new HttpComponentsAsyncClientHttpRequestFactory();
8   cHttpReqFact.setConnectionRequestTimeout(30*Second);
9   cHttpReqFact.setConnectTimeout(30*Second);
10  cHttpReqFact.setBufferRequestBody(false);
11  cHttpReqFact.setReadTimeout(0);
12
13  cHttpReqFact.setHttpAsyncClient(client);
14   return cHttpReqFact;
15  }
```

Figure 6.3: Example of T2 (implemented in the `ServiceCaller.java` class) used in P10 in order to establish timeouts in asynchronous clients requests

stop terminating instances from the service registry when they do not receive heartbeats beyond a certain threshold. Therefore, the usage of T4 depends on if the microservices-based system uses Netflix Eureka.

Figure 6.4 depicts the frequency of microservices scalability tactics' presence in the open projects as well as the programming languages which the evidence of microservices tactics is found.

We realized that some microservices scalability tactics compromise the performance of the microservices-based system. For example, the Piggy Metric project (P5) requires eight spring boot applications, four MongoDB instances, and one messaging server; thus, when we tried to add more microservices to this project, its scalable framework demanded yet more hardware and software resources. Fortunately, other microservices scalability tactics (such as S5) have procedures to avoid these performance issues. Thus, depending on the microservices-based system goal, it seems recommendable to check the pros and cons before selecting microservices scalability tactics.

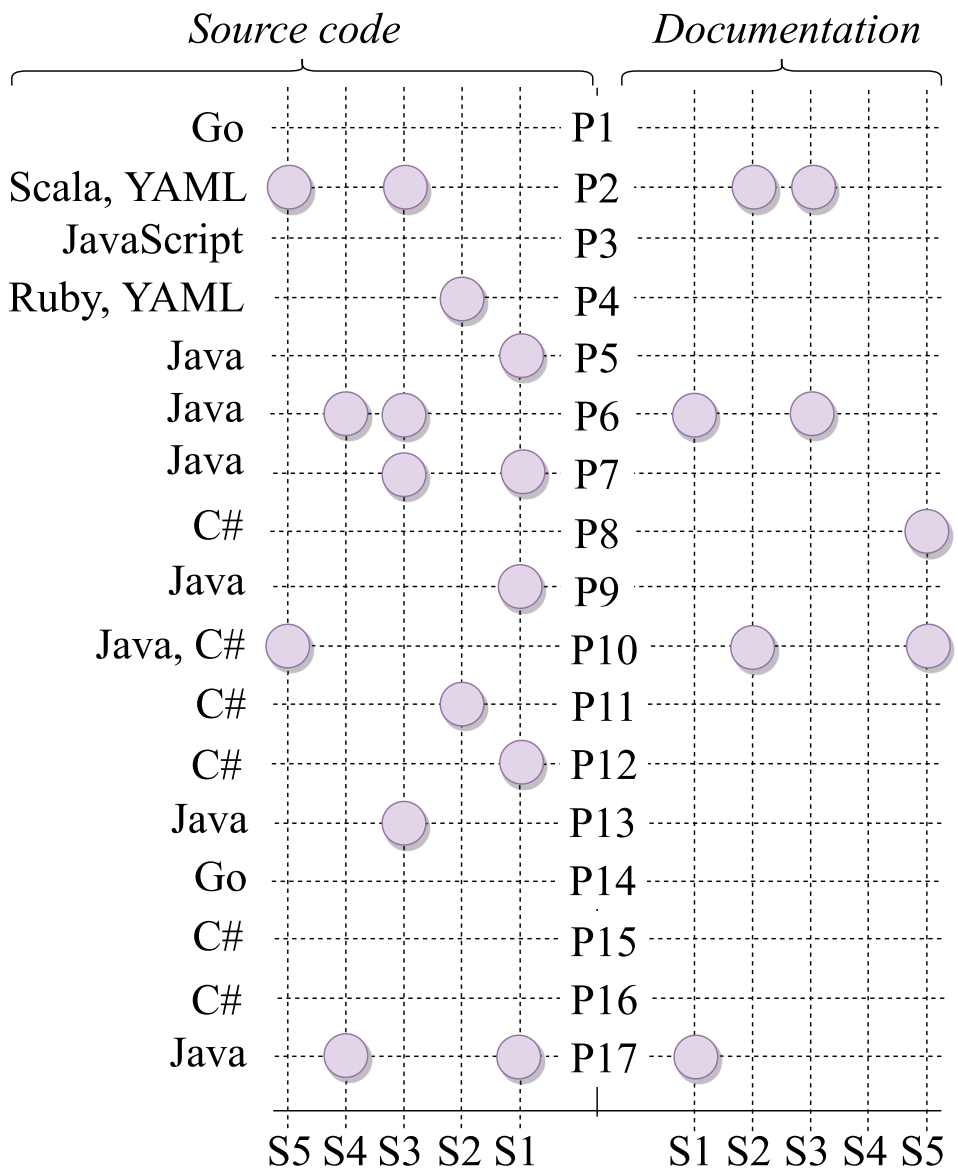


Figure 6.4: Presence of microservices scalability tactics in source code and documentation

6.4.2. Documentation analysis

Regarding the projects' documentation, Figures 6.2 and 6.4 describe which microservices availability and scalability tactics is possible to recognize in documentation. We realized that the most important source to identify microservices availability and scalability tactics are open and closed issues.

From a total of 2,103 issues, we collected 450 ($\approx 10.87\%$) relevant issues (we considered issues posted by developers or architects, where keywords that define each tactic are manifested). Then, we omitted irrelevant issues and we obtained 122 issues where we find hints of microservices tactics. Subsequently, we manually organized these issues by tactics and projects, and after merging similar issues, we obtained a final set of 21 issues from 7 open microservices-based systems (41% from the initial set of projects).

Principally, most issues focused on describing the rationale of using A2,A3, S3, and S1 in specific scenarios.

6.4.3. Discussion

The results of our empirical study reveal that microservices availability tactics are concerned first and foremost with preventing faults, and secondarily with detecting them, instead of reacting or recovering from them. The previous observation coincides with the results obtained in our previous work regarding security mechanisms in microservices-based systems [111].

In order to investigate why there is not enough analysis in the context of reacting and recovering from attacks as well as failures in microservices architectures, we examined academic [45] as well as industrial sources (InfoQ[4] and StackOverflow[5]). According with this review, there are three key availability concerns with respect to microservices architectures:

Code reuse: The use of shared code and libraries can help migration to microservices, but it can also introduce lock-in problems and the need for propagation of patches to included components, possibly up to and including all the system's microservices.

Traffic between microservices: Microservices exchange information. If traffic happens in a segregated part of the network, it can be assumed that the risk of having a spy is reduced since it is usually behind a corporate firewall, hence less susceptible to man-in-the-middle attacks. However, when the microservices-based system is in an open cloud environment, this assumption is no longer valid, and besides adding microservices capabilities to handle encrypted traffic, which may affect the performance of the composing microservices.

In the furtherance to address this gap, Toffetti *et al.* [134] discussed the possibility to manage recovery and/or react from failures strategies in microservices-based systems

[4]https://www.infoq.com
[5]https://stackoverflow.com

by introducing *self-healing* properties in the system. This feature is related to which steps should be executed to recover from a broken state. Generally, self-healing is implemented by an external system that watches the instances health and restarts them when they are in a broken state for a more extended period. Moreover, by placing components across different failure domains, microservices architectures can be resilient to failure and are able to guarantee that failed components will be restarted within seconds.

Concerning scalability, Fowler [85] argues that in a microservices ecosystem, each microservice needs to be able to scale the whole system without suffering from performance problems. We notice that the three axis match the scalability issues in microservices-based systems, as shown in our previous study [93] that found scalability a key distinguishing feature in microservices research. The analogy between scale cube and microservices [55] is illustrated in Figure 6.5.

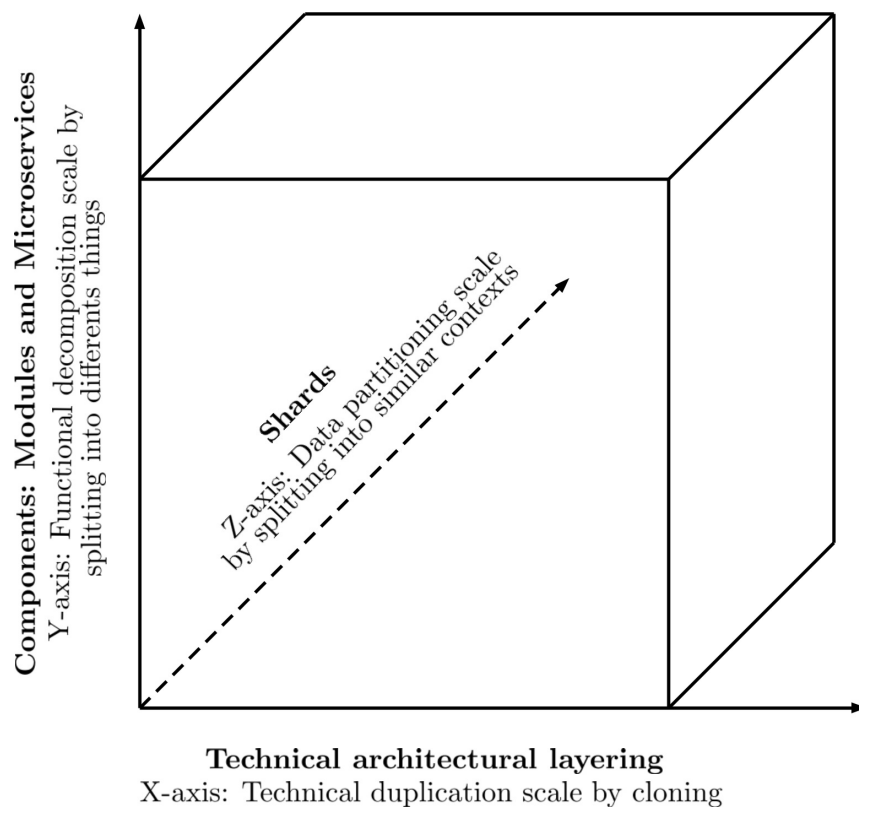


Figure 6.5: Analogy of scale cube and microservices-based systems

In this regard, the Z-axis isolates data across different servers based on routing criteria, but when demand increases the whole application must be replicated, which implies multiplying the component instances, and hence compromising their availability. For

example, the Customers-Stores system [6] (a microservices-based system which uses Netflix Eureka and microservices scalability tactics) increases its number of microservices (sometimes mere invocations), the framework demands that we deploy and manage more service registries clusters, requiring yet more resources. This project also uses Circuit Breaker frameworks (Y-axis), whose failure rate tends to grow as the number of microservices increases, compromising the whole system availability. Although this example does not imply that *all* frameworks in the Z-axis compromise availability, clearly this issue must be considered; in fact, developers have already reported issues with this architectural trade-off[7].

6.5. Summary

This chapter describes the procedure to define and idenitfy of microservices tactics related to availability, scalability, security and interoperability to support design decision in microservices-based systems. Concerning security and interoperability, the evidence of microservices tactics corresponds to previously defined architectural tactics taxonomies. Nevertheless, regarding availability and scalability, we found new evidence.

We proposed a template to characterize architectural tactics (independently of the context), and we examined 17 open microservices-based systems aiming at distinguishing microservices availability and scalability tactics in source code as well as in the documentation. Key findings are (i) most microservices architectural tactics are focused on preventing faults, but we do not find tactics with the aim of recover or react from faults, and (ii) source code provide more evidence of microservices availability and scalability tactics than the documentation.

[6] https://github.com/spring-cloud-samples/customers-stores
[7] https://github.com/Netflix/Hystrix/issues/1633

Chapter 7

Microservices patterns

In this chapter, we explored which microservices patterns are used in microservice-based systems, by subjecting thirty well-known open source projects to a comprehensive multi-criteria code and design review. We found that (1) open projects use only a few architectural patterns broadly; and (2) most projects use the same few frameworks. This study shows that microservice systems developers do use architectural patterns, but only a few of them. It remains to be determined whether additional patterns would be productively used to build microservice systems, or the few ones currently used are the only ones actually necessary.

7.1. Introduction

The knowledge acquired in industrial and academic experiences [54] [75] shows that using *patterns* (specifically, architectural patterns) in systems development allows (1) discerning the domain where the system is being developed, (2) satisfying systemic properties (quality attributes [36]), and (3) creating large-scale reuse design techniques (frameworks [73]).

We have already conducted studies [108] [93] to determine whether architectural patterns have been used in the development of microservices-based systems, and we found

that some well-known architectural patterns are indeed used to develop microservices architectures, but also new patterns are emerging as design alternatives. Thus, this chapter expands our previous study [92] to explore whether microservices architectural patterns are actually used to develop microservices-based systems, focusing on three aspects: microservices architectural patterns, quality attributes, and frameworks.

7.2. Research methodology

In this section, we describe the research methodology used in this study, which includes the following stages: the research scope; the overall strategy; the goals and research questions; and (in some detail) the study steps.

7.2.1. Scope

We limited the study to the Github repository because it is one of the primary repositories for open source projects, but future studies may include other repositories.

7.2.2. Strategy

We defined three phases, each one containing several activities:

Phase I - Exploration We explored and synthesized existing characterizations of microservices architectural patterns from academic and industry sources since the variety in proposed patterns suggests that there is not yet an agreed upon definition of *microservices architectural pattern*.

Phase II - Refinement We established a set of criteria to filter microservices architectural patterns and establish the patterns corpus as the basis for this study.

Phase III - Verification We analyzed a set of microservices-based open source projects to (1) identify the architectural patterns used in their development and (2) compare these patterns with those found in our previous study [93] [108].

7.2.3. Goals and research questions

We established the goal of this study, based on the Goal-Question-Metric [19] approach, which is *distinguish which microservice architectural patterns identified in academic and industry sources are actually used in well-known microservices-based open source systems.*

To achieve this goal, we defined the following research question:

RQ1.3
Which microservices patterns reported in academic and industry sources are actually used in microservices-based open source systems?

This RQ intends to verify if the developers of open source projects use some of the architectural patterns reported in chapter 3.

7.2.4. Process

The research process (see Figure 7.1) is divided into three phases, each one with a set of activities and an artifact as a final result. The steps of Phase I are addressed in chapter 3.

In Phase II, we performed the refinement tasks, and filtered the architectural patterns using the following criteria:

- **C1**: Patterns to migrate from monolithic to microservices This criterion is to filter those patterns that represent solutions to migrate from monolithic architectures to microservices. Our research is focused on architectural patterns for systems

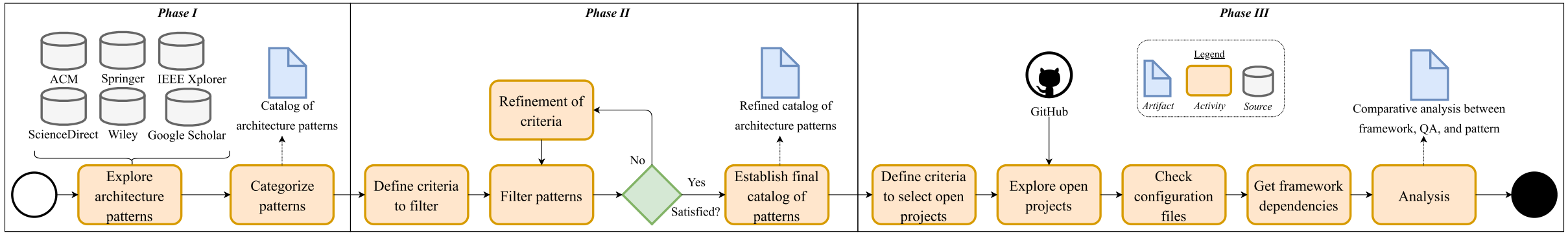


Figure 7.1: Research process

already built using the microservices approach.

- **C2**: Redundancy In Phase I, we realized that some patterns with different names describe the same context, problem, and solution; some cases, such as the Containerize the Services [15] and Container [26], are quite similar indeed. In such cases, we gathered identical patterns into a single one.

- **C3**: Poor documentation All reported architectural patterns have descriptions, but sometimes it is quite weak. In such cases, we omitted these patterns from the analyzed corpus since the available documentation did not allow an adequate study.

These criteria were executed sequentially. Finally, the remaining architectural patterns were subjected to quality assessment using three quality criteria (QC) questions:

- **QC1**: Does the architectural pattern describe the *context* in which it is applied? Y (yes), it clearly describes the context; P (partly), it partially describes the context; N (no), it vaguely describes the context

- **QC2**: Does the architectural pattern describe the *problem* it is trying to solve? Y, it clearly describes the problem; P, it does not directly describe the problem; N, we could not determine which problem it tries to solve

- **QC3**: Does the architectural pattern describe the *solution* for this context and problem? Y, it clearly describes the solution; P, it partially describes the solution; N, we could not determine what solution it proposes

Each pattern was evaluated with the QCs using the scores $Y = 1$, $P = 0.5$, $N = 0$.

Finally, in Phase III we explored the open source projects to obtain the architectural pattern used in them. We used the same projects described in Table 5.2.

Since each selected project may use one or more frameworks, we gathered three types of configuration files to determine frameworks and dependencies:

- *Docker compose*: The docker-compose file corresponds to the Compose tool, which allows defining and running multi-container Docker applications. Using Docker along with docker-compose files, it is possible to use a YAML (YAML Ain't Markup Language) file to configure the system's services [1]. Docker-compose files define how to configure and run a collection of containers, which virtual networks they should be linked to, which volumes should be mounted, and more features.

- *POM*: Maven defines POM (Project Object Model) files as XML files that contain information about configurations and technologies to build a project [9]. Microservices can be structured as "modules" using POM files because modules provide advantages when declaring microservices build dependencies.

- *Gradle*: Gradle is an automation tool that relies on Groovy and a DSL (Domain Specific Language) to work with a simple and clear language for declaring the project configuration [63]. This tool provides the flexibility to work with other languages (not only Java) and it has a robust dependency management system.

Additionally, we developed a Java parser that reads the files and shows framework dependencies, using the following libraries: `jackson-dataformat-yaml`[1] to read `docker-compose.yml` files, `MavenXpp3Reader` [2] to read `POM.xml` files and a custom tool (developed by us) to read `build.gradle` files.

To complete the frameworks extraction, we also study the source code through a manual process. The goal to perform this task is to analyze how these frameworks work (functionality and dependency) at source code level. This analysis allowed us to refine the framework extraction from configuration files (see Figure 7.2). For this purpose, we used the tools Understand[3]and Cytoscape[4] in four steps:

[1]https://github.com/FasterXML/jackson-dataformat-yaml
[2]https://maven.apache.org/ref/3.3.1/maven-model/apidocs/org/apache/maven/model/io/xpp3/MavenXpp3Reader.html
[3]https://scitools.com/features/
[4]http://www.cytoscape.org/

1. Generate the architectural dependency diagram of the project, allowing to identify clusters (as potential microservices)
2. Identify the files related to each cluster
3. Code analysis of these files to find their frameworks
4. Traceability analysis to locate framework references in the source code

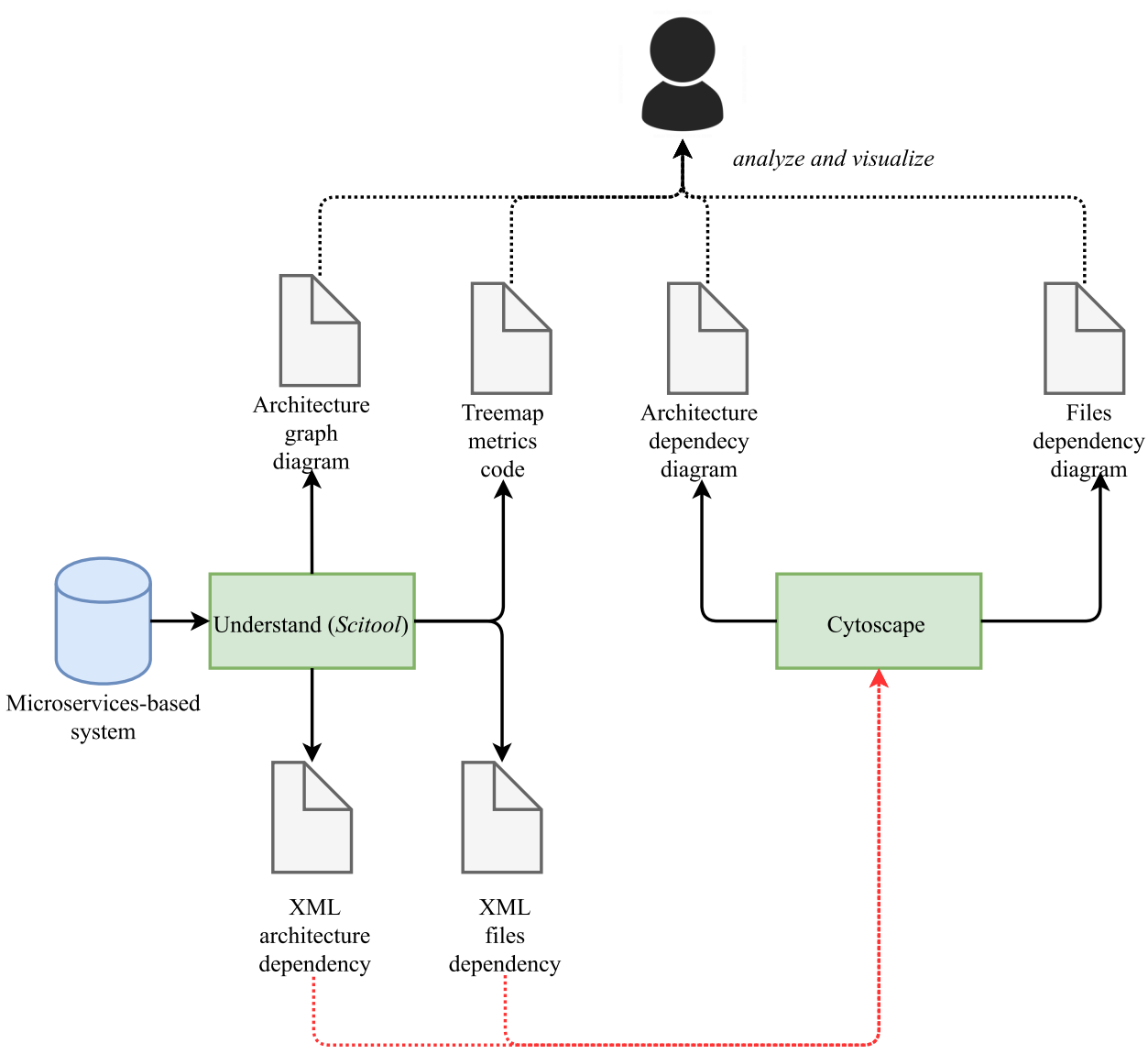


Figure 7.2: Manual steps overview

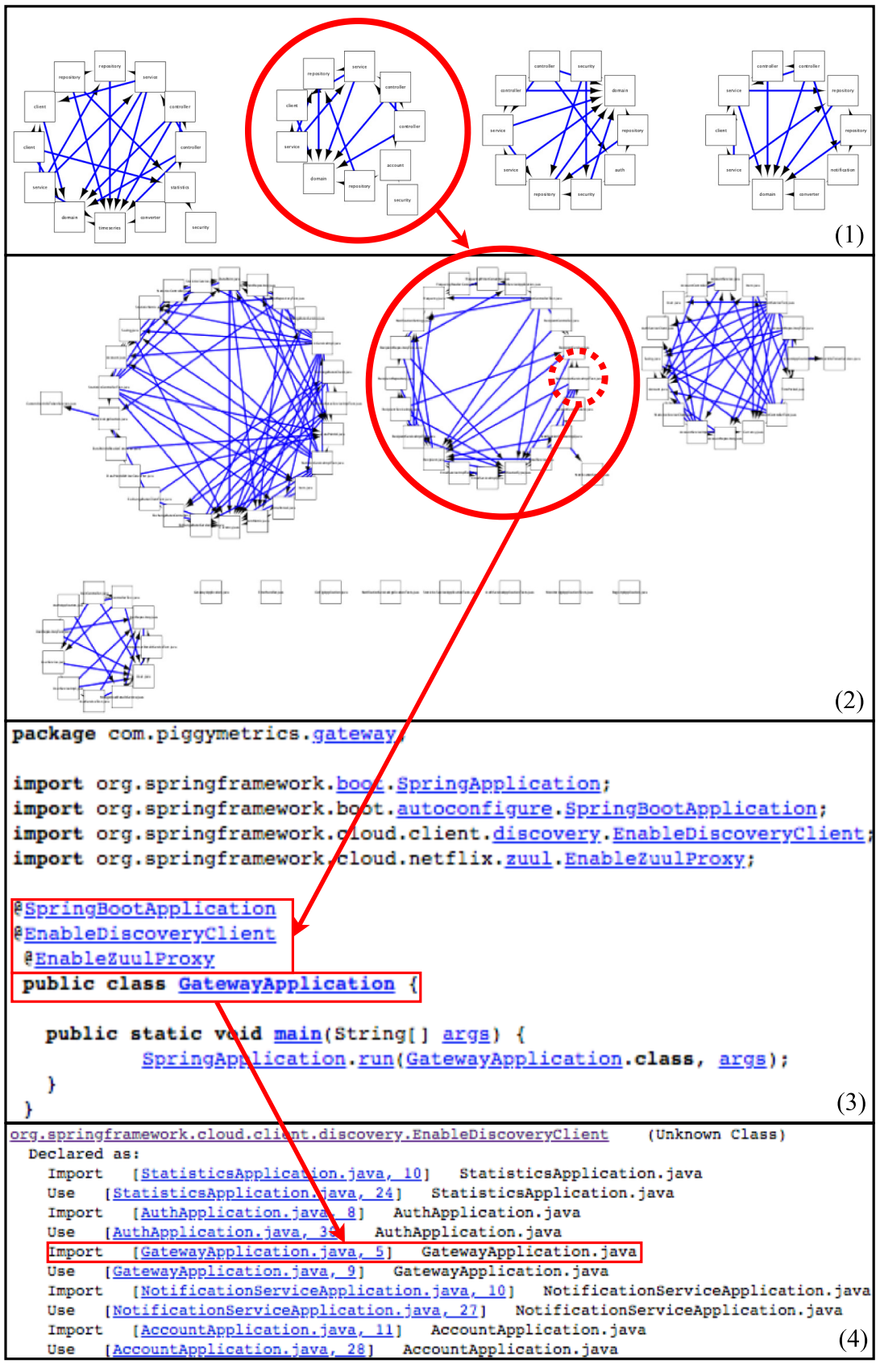


Figure 7.3: Source code analysis' steps

For example, consider Figure 7.3. This Figure illustrates the four steps executed in the project P5. In step (1), we identified four clusters composed by the architectural dependencies of the project. The tool Understand represent architectural dependencies through the identification of folders and sub-folder of the project and the dependencies among folders are illustrated by blue lines. In step (2), we obtained each source file corresponding to each cluster. In this example, we examined the second cluster.

Following in the context of this example, in step (3) we selected a random file called `GatewayApplication.java`.

The goal of choosing this source code is to check which framework instances are invoked (red box) in this class in order to identify frameworks. In our study, we executed step (3) in all files. Subsequently, in step (4) we used the features of the tool Understand in order to review which other classes are linked with `GatewayApplication.java`. We realized that `StatisticsApplication.java`, `AuthApplication.java`, `NotificationServiceApplicatio` and `AccountApplication.java` depend on `GatewayApplication.java`. According to the project's documentation, `GatewayApplication.java` use the annotations `@EnableZuulProxy` and `@EnableDiscoveryClient`, which means that this class is using the pattern AMP15 and AMP9 through the frameworks Netflix Zuul and Spring Cloud Config.

7.3. Results

We obtained 21 microservices pattern from the quality assessment. For each microservices pattern the context, problem, and solution are described.

BackEnd for FrontEnd (AMP1) **Context**. Business microservices that encapsulate individual business functions do not map cleanly to the channel-specific needs of client applications. All of the server-side functionality a client application needs should be accessible through a single API. **Problem**. How to represent a channel-specific service interface that is consistent with an overall microservices architecture, but which allows enough uniqueness that it can be adapted to the needs of a specific client type? **Solution**. Build a "Backend for Frontend" (BFF) that acts as a single API for a client. Implement different BFF's for different types of clients, each with an API customized to what that client type needs (see Figure 7.4).

Result Cache (AMP2) **Context**. Making network calls to remote databases or services is expensive when it is necessary to build clients for services. On the other

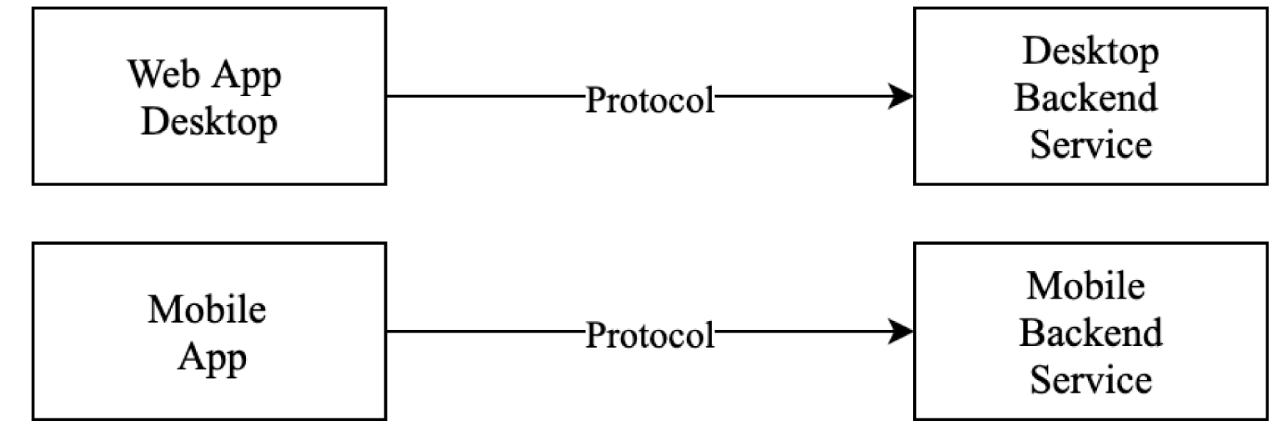


Figure 7.4: BackEnd for FrontEnd pattern illustration

hand, latency can be added by distance, layers or simply by long processing in a service.

Problem. How to improve the performance of an application or microservice when it makes many repeated calls to services? **Solution**. Use a Results Cache that shortcuts the need to make repeated calls to the same service (see Figure 7.5).

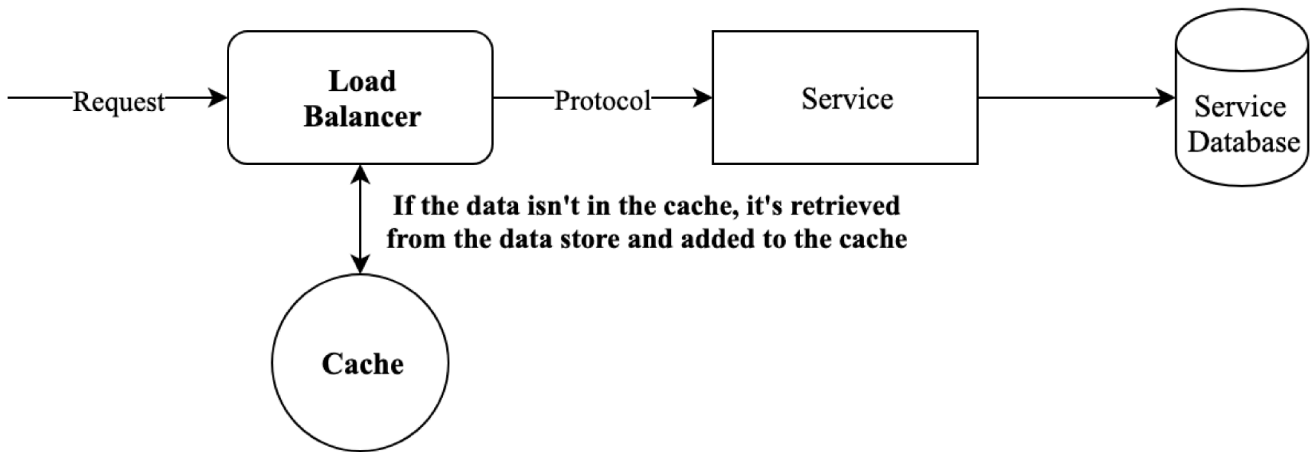


Figure 7.5: Result Cache pattern illustration

Page Cache (AMP3) **Context**. In the context of a web or mobile application using a microservices architecture, it is necessary to apply the Backend for Frontend pattern in order to build dispatchers for a web or mobile application. **Problem**. How a single microservice that represents a business entity or concept can return more information that can be easily displayed, particularly on a mobile application, without a great deal of scrolling? **Solution**. Use a Page Cache with Backend for Frontend. The Backend for Frontend will present an interface that allows the client application to request a limited subset of a much larger set of data. The information can be indexed by page numbers based on the total size of the dataset and the number of elements per page that can be displayed on the device (see Figure 7.6).

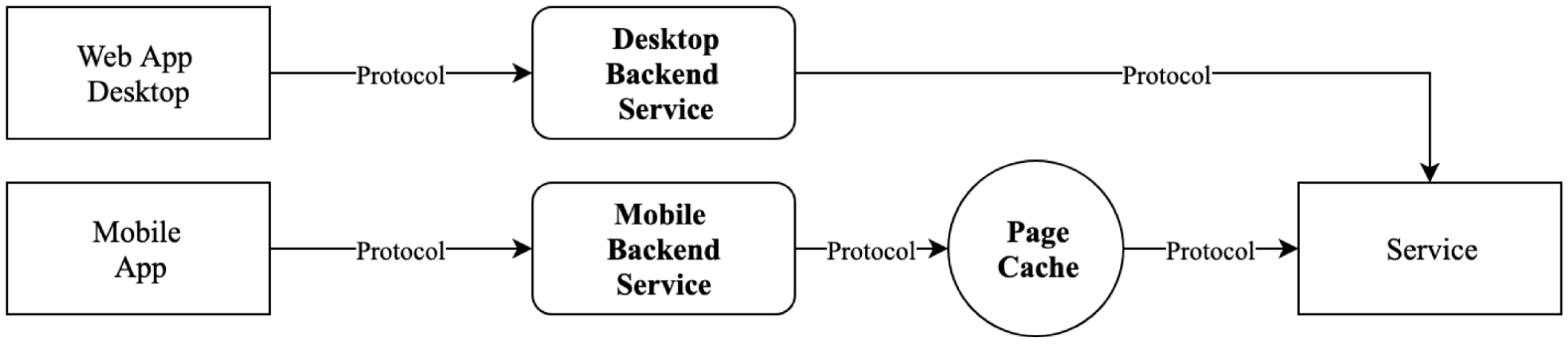


Figure 7.6: Page Cache pattern illustration

Scalable Store (AMP4) **Context**. When an application is using business microservices, it will need a persistent state to represent current and previous user interaction. **Problem**. How to represent a persistent state in an application? **Solution**. Put all states in a Scalable Store where they can be available for and shared by any number of application runtime. The key here is that the database must be naturally distributed and able both to scale horizontally and to survive the failure of a database node.

Key Value Store (AMP5) **Context**. Microservices-based systems development. **Problem**. How to store data that is naturally accessed through a simple key look-up, such as cache entries? **Solution**. Store your data in a scalable Key-Value Store. The principal advantage of a key value store over other types of Distributed Store is its simplicity. Most Key-Value Stores act, in principle, as a hash map. For example, Redis has simple GET and SET commands to store and retrieve string values.

Log Aggregator (AMP6) **Context**. In a microservice-based system, it is required to be able to debug problems that cross the different components of that architecture. **Problem**. How to view and search all the different log files that emerge from all the differently distributed runtime that comprise collections of microservices? **Solution**. Use a Log Aggregator that pulls all the files together into a single, searchable database. The Log Aggregator will "listen for" or tail each individual log file and forward the log entries to an aggregated collection point as they are made (see Figure 7.7).

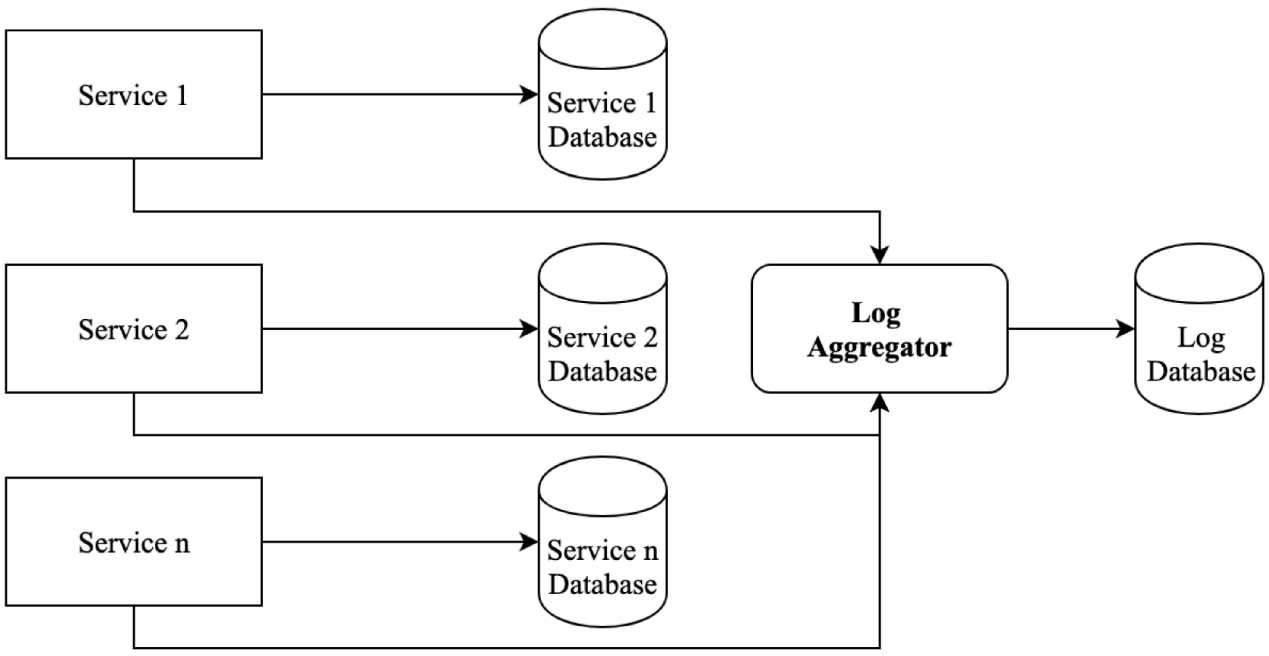


Figure 7.7: Log Aggregator pattern illustration

Service Registry (AMP7) **Context**. The current microservices-based system has different business microservices comprising the application. **Problem**. How to decouple the physical address (IP address) of a microservice instance from its clients so that the client code will not have to change if the address of the service changes? **Solution**. Use a Service Registry to map between a unique identifier and the current address of a service instance in order to decouple the physical address of service from the identifier (see Figure 7.8).

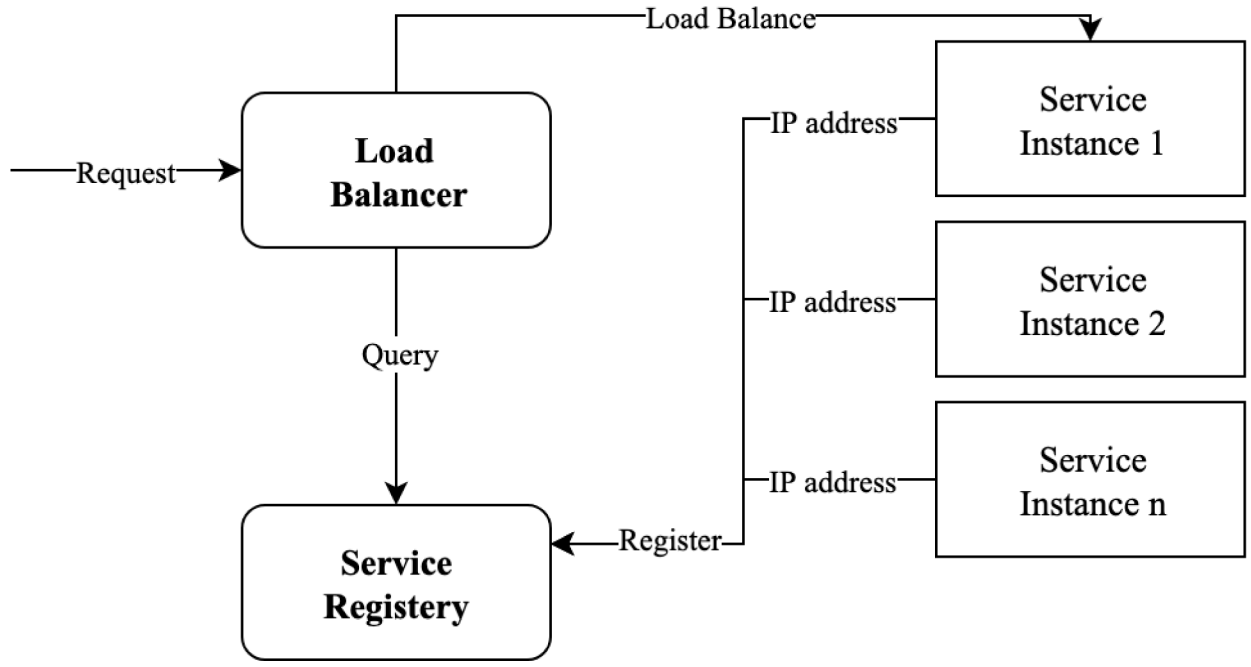


Figure 7.8: Service Registry pattern illustration

Enable Continuous Integration (AMP8) **Context**. The number of services is increasing in a microservices-based system. **Problem**. How to have available production-ready artifacts; how to prepare the system for introducing Continuous Delivery? **Solution**. The first step towards Continuous Delivery is to set up Continuous Integration. Continuous Integration automates the build and test process and helps to always have production-ready artifacts. Usually, a Continuous Integration pipeline contains a code repository, an artifact repository and a Continuous Integration server. First, each service should be placed in a separate repository which helps to have a clearer history and separates the build life cycle of each service. Then, a Continuous Integration job should be created for each service. Each time a service's code repository changes, the job should be triggered. The job's responsibility includes fetching the new code from the repository, running the tests against the new code, building the corresponding artifacts and pushing these artifacts to the artifact repository. Failure in doing each of these steps should terminate the job from proceeding and informs the corresponding development team of the occurred errors. This team should not do anything else until addressing the reported errors. One simple rule in Continuous Integration is that new changes should break the system's stability and should pass all the predefined tests (see Figure 7.9).

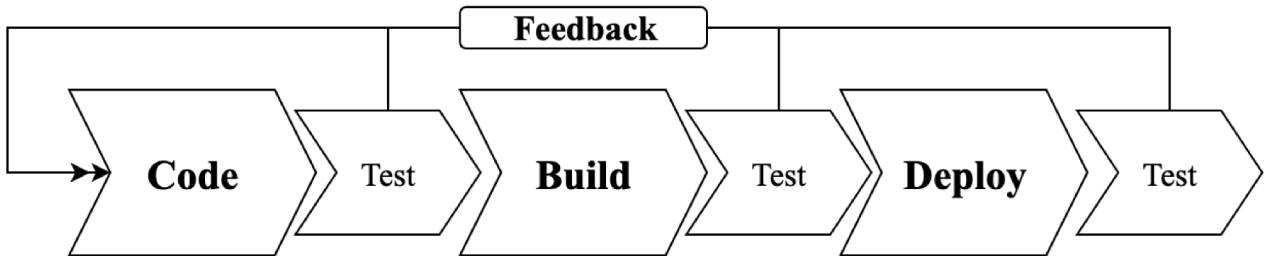


Figure 7.9: Enable Continuous Integration pattern illustration

Service Discovery (AMP9) **Context**. Each service has one or more instances deployed in the production environment. The number of instances can be changed dynamically, and each of them can be deployed in different places. **Problem**. How services can locate each other dynamically? **Solution**. Setup a Service Discovery which stores service instances addressed. Each service registers itself during initiation. The removal of an instance from the registry can be triggered through either not receiving a periodic heartbeat from the instance or by the instance itself during termination.

Having a list of available service instances, an Edge Server, a Load Balancer or other services can locate their desired services dynamically through Service Discovery.

Container (AMP10) **Context**. There is a microservices-based system, and a Continuous Integration pipeline is in place and is working. Each service needs a specific environment to run correctly, which is set up manually or through a configuration management tool. The differences between the development and production environments cause some problems like the same code producing different behaviors in these two environments. Overall, the deployment of so many services in production has become either a cumbersome or complex task. **Problem**. How to make development and production achieve the same results for the same code; how to remove the complexity of configuration management tools? **Solution**. The use of containers encloses the microservice itself, including all required libraries and data, which also supports the requirement of self-containment. This, in turn, provides several advantages: better testability, ease of service deployment and better scalability.

Circuit Breaker (AMP11) **Context**. In order to provide monitoring capabilities, each service should provide an interface to hand over monitoring information. In particular, the health status (e.g. "ok" and "broken"), is of high importance. **Problem**. How to monitor services and prevent cascading faults? **Solution**. The Circuit Breaker checks the health status or remembers the number of unsuccessful calls and trips if a certain threshold is reached. If the circuit breaker is triggered, it will return an error instead of sending the call to the remote service, to prevent the broken service to be penetrated with additional requests. In the microservice architecture, it is recommended to use one logging format throughout all services. This helps to aggregate individual logs to get a whole picture of the overall system (see Figure 7.10).

Load Balancer (AMP12) **Context**. It is required to scale the client and microservices independently of each other. **Problem**. How does a client interacts with a microservices-based application without knowing the instances that are serving it?

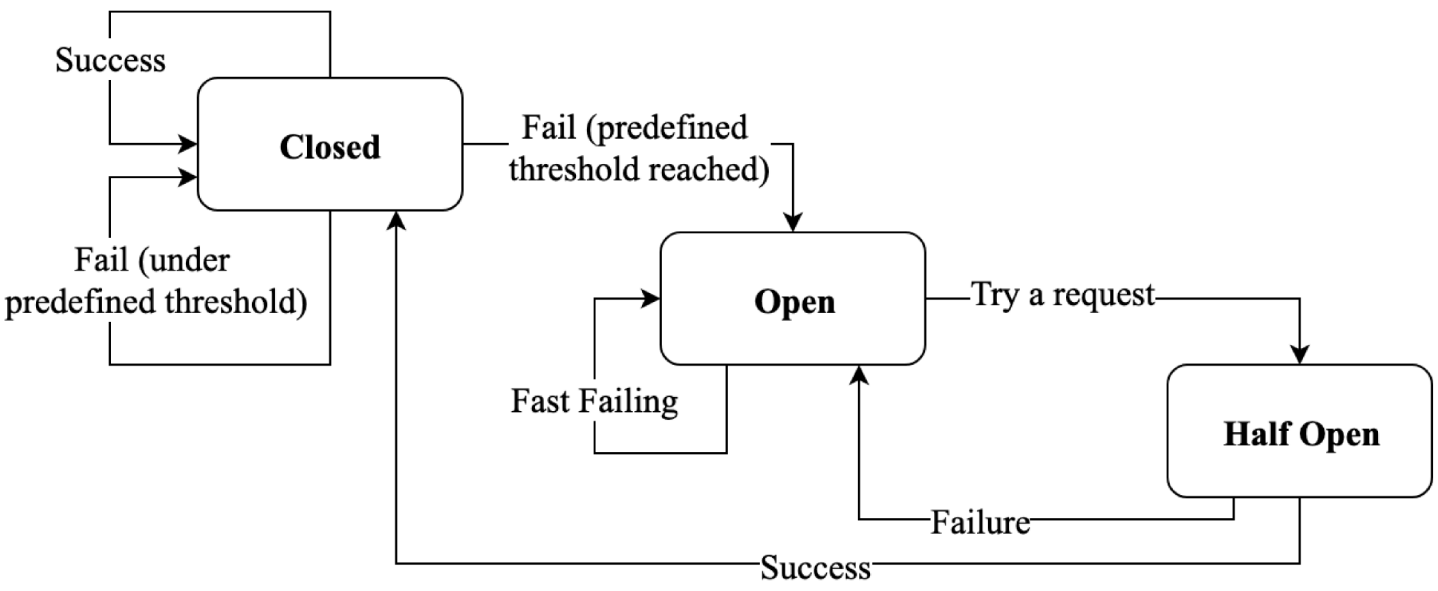


Figure 7.10: Circuit Breaker pattern illustration

Solution. A load balancer distributes a workload on a set of equal services. The circuit breaker enables the load balancer to put work only on services that are in a good state of health. The number of routed requests is lowered for services to which the circuit breaker is only half open. Broken services will, for the time being, not be used (see Figure 7.11).

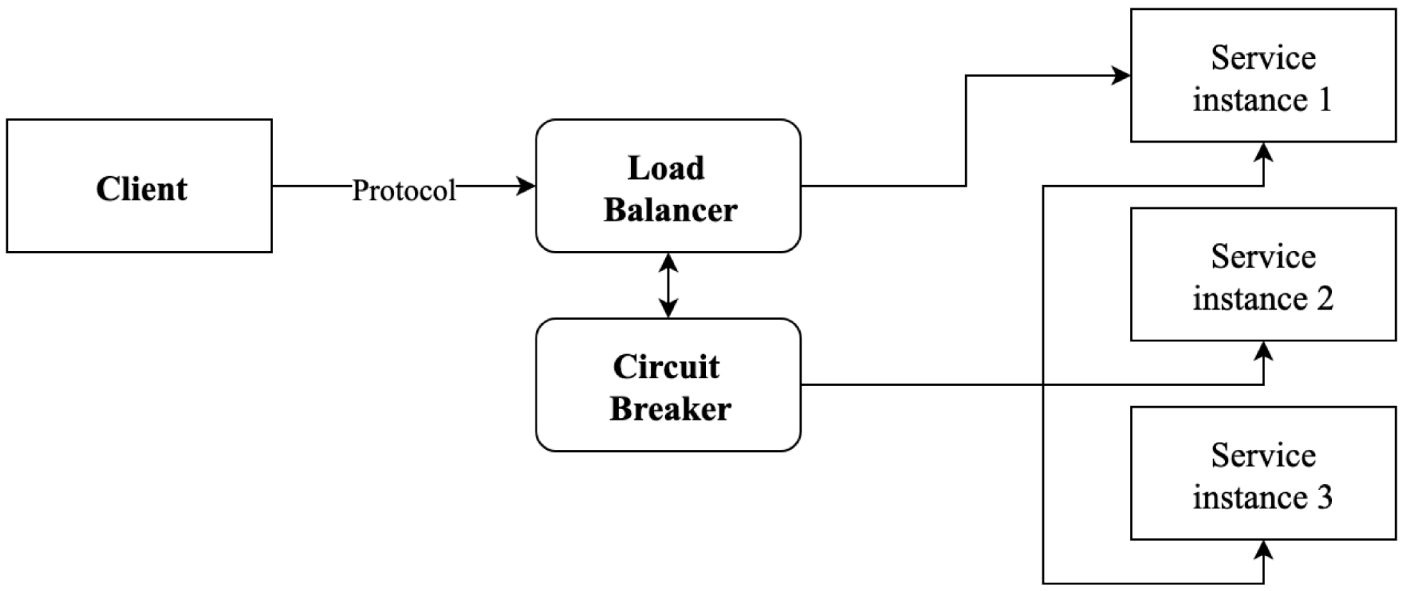


Figure 7.11: Load Balancer pattern illustration

Database is the Service (AMP13) **Context**. The addition of new behaviors and business logic at the database level may be a possible approach to reduce complexity, and thus the related risks, and also to gain improvements in terms of speed and scalability. **Problem**. If each scalable service has its own database (cluster), how to reduce the complexity of the architecture and the related risks, while also gaining

more improvements in terms of speed and scalability? **Solution**. Database-server-per-service: each service has its own database server. When the service has to be scaled, the database can also be scaled in a database cluster, no matter the service (see Figure 7.12).

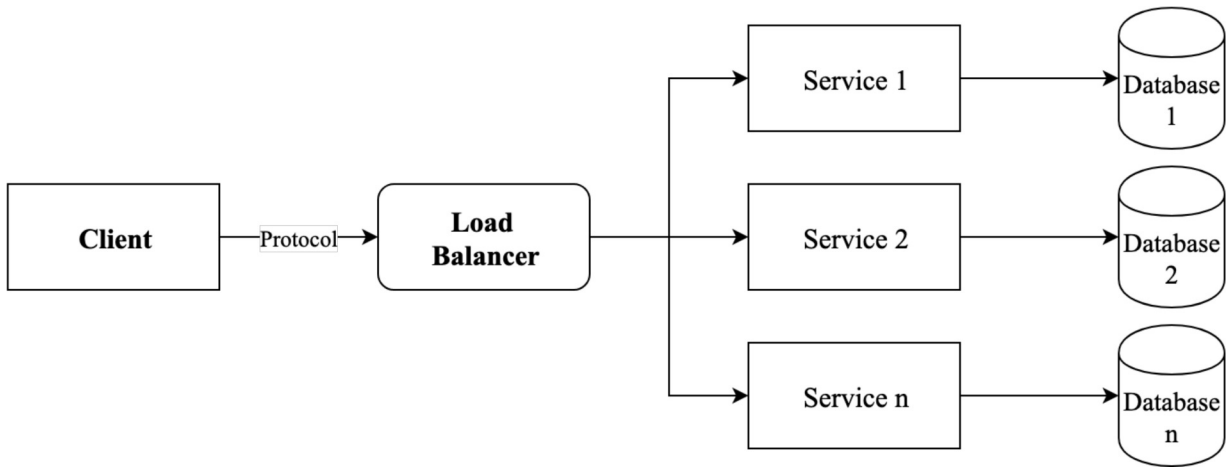


Figure 7.12: Database is the Service pattern illustration

Messaging (AMP14) Context. Services must handle requests from the application's clients. Furthermore, services must sometimes collaborate to handle those requests. They must use an inter-process communication protocol. **Problem**. Which communication mechanisms do services use to communicate with each other and their external clients? **Solution**. Use asynchronous messaging for inter-service communication. Message queue allows state to be asynchronously and reliably sent to different locations (see Figure 7.13).

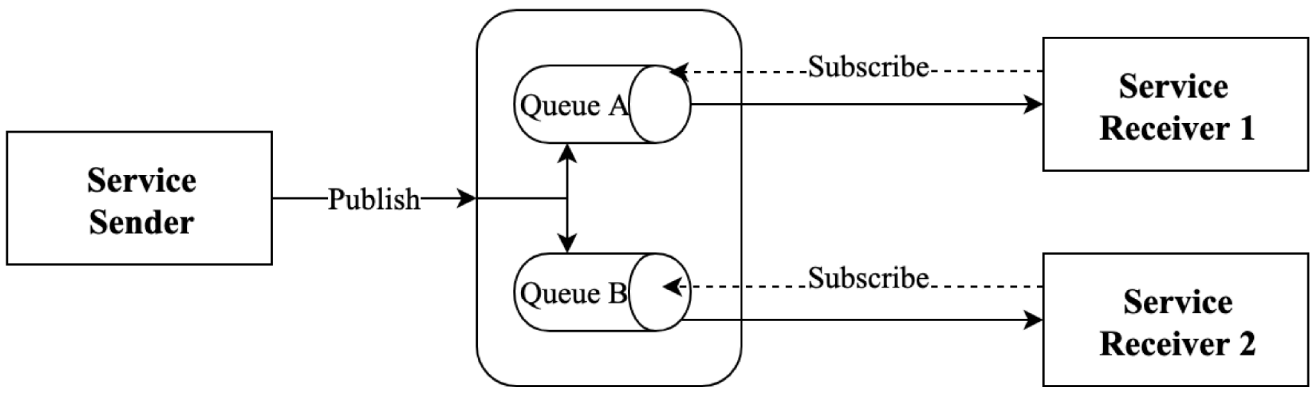


Figure 7.13: Messaging pattern illustration

API Gateway (AMP15) Context. Developing multiple versions of the product's user interface is required. Code that displays the product details needs to fetch information from various services. **Problem**. How do external clients communicate with

the services? **Solution**. Use an API Gateway which allows a service to provide each client with a unified interface to services (see Figure 7.14).

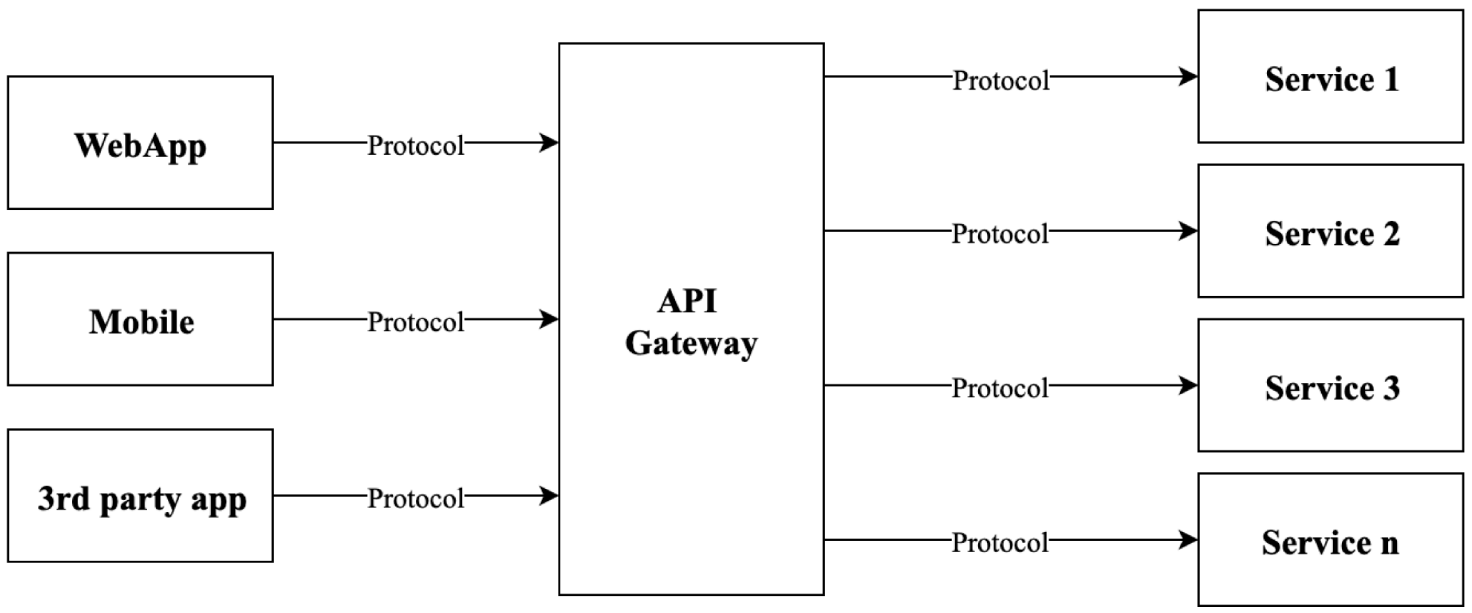


Figure 7.14: API Gateway pattern illustration

Health Check (AMP16) Context. The monitoring system should generate an alert. Also, the Load Balancer or Service Registry should not route requests to the failed service instance. **Problem**. How to understand the behavior of an application and troubleshoot problems? **Solution**. Create a service API (e.g. HTTP endpoint) that returns the health of the service and can be pinged, for example, by a monitoring service, service registry or load balancer (see Figure 7.15).

Monitoring (AMP17) Context. The microservices-based system is being run on a cluster of containers with each service having many instances in production. **Problem**. How to monitor the underlying infrastructure; how to use the gathered data to re-architect the system by providing feedback to the development team? **Solution**. Each service should have its independent monitoring facility owned by the Ops part of the services team. These facilities include the required components to gather the monitoring information, e.g. CPU and RAM usage, and to send them to a monitoring server. Therefore, the following components should be added to each service container image and be configured during either the creation of container images or the creation of actual containers. Then, in the monitoring server, this information should be parsed

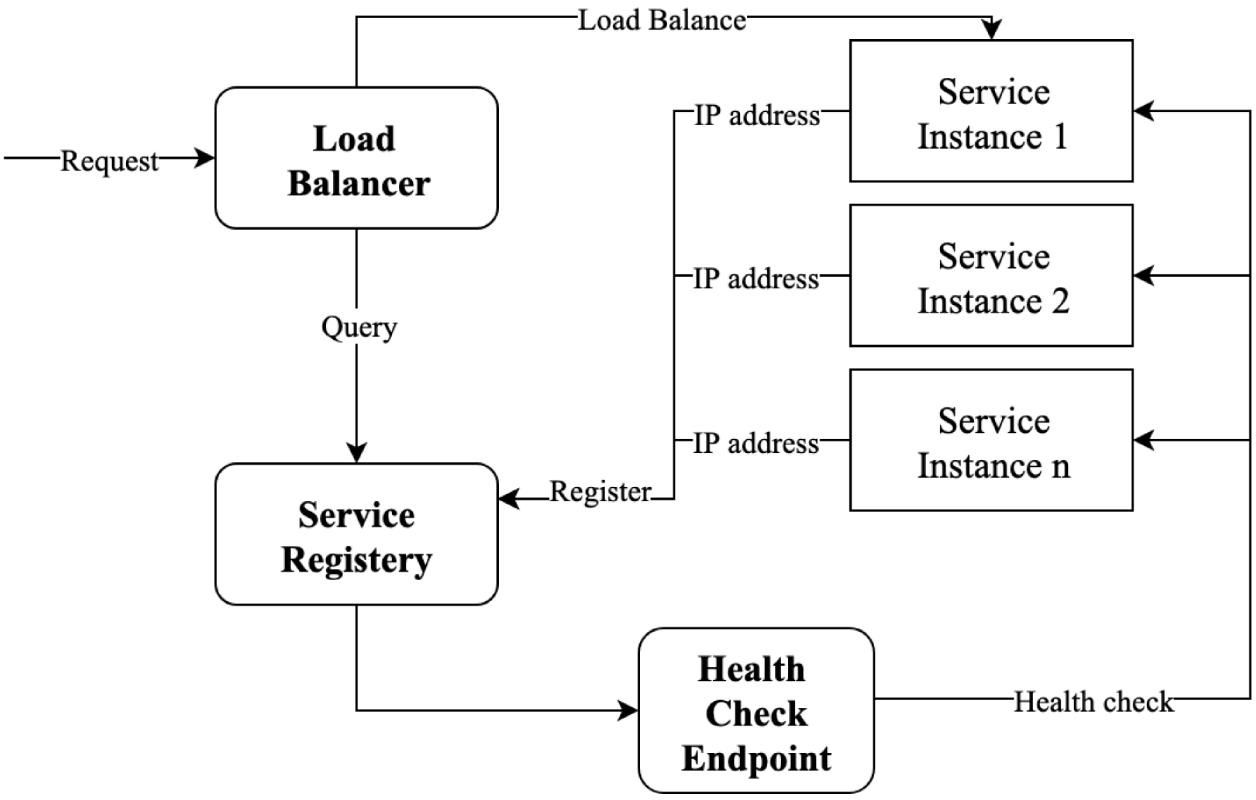


Figure 7.15: Health Check pattern illustration

and aggregated into a piece of structured data and stored somewhere that can be queried.

Broker (AMP18) **Context**. Distributed environments. **Problem**. How to structure the system to facilitate service invocations among remote components? **Solution**. Introduce a broker component that decouples clients and servers by making the latter's services available to the former via requests to the broker itself.

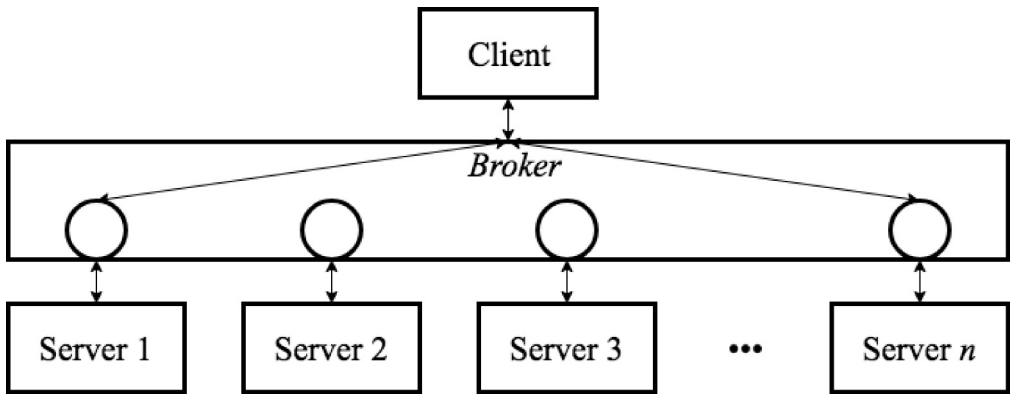


Figure 7.16: The Broker architectural pattern

Authenticator (AMP19) **Context**. Computer systems that contain resources that may be valuable because they include information about business plans, user medical

records and so on. We only want subjects that have some reason to gain access to our system to be able to do so. **Problem**. How can we prevent imposters from accessing our system? **Solution**. Use a single point to access to receive the interactions of a subject with the system and apply a protocol to verify the identity of the subject. The protocol used may be simple or complex, depending on the needs of the applications.

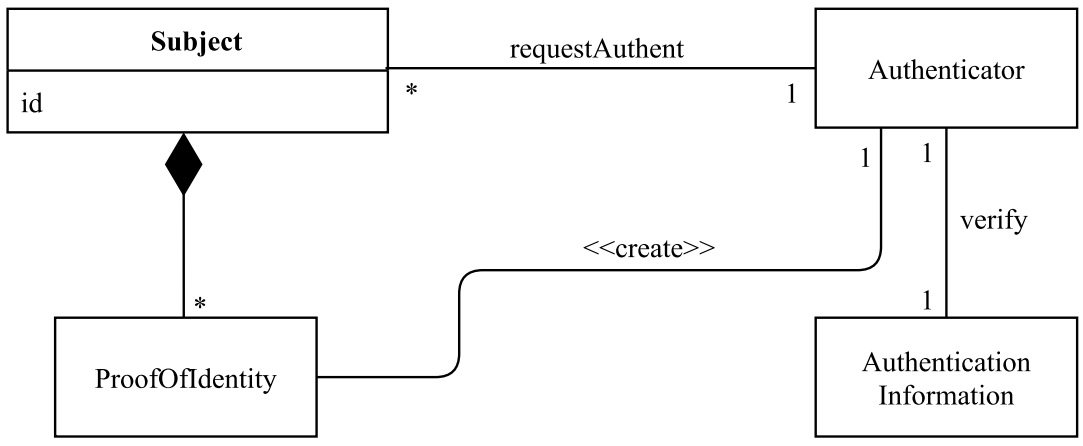


Figure 7.17: The Authenticator pattern

Credential (AMP20) **Context**. System which the users of one system may wish to access the resources of another system, based on a notion of trust shared between the systems. **Problem**. How then do we allow external users to access some of our resources? **Solution**. Store authentication and authorization data in a data structure external to the systems in which the data is created and used. When presented to a system, the data (credential) can be used to grant access and authorization rights to the requester. For this to be a meaningful security arrangement, there must be an agreement between the systems which create the credential (credential authority) and the systems which allow their use, dictating the terms and limitations of system access.

Authorization (AMP21) **Context**. A computing environment that has resources that have value for its users or their institution. **Problem**. How can we describe who is authorized to access specific resources in a system? **Solution**. Indicate, for each active subject that can access resources (objects or protection objects) which resources it can access and how (access type).

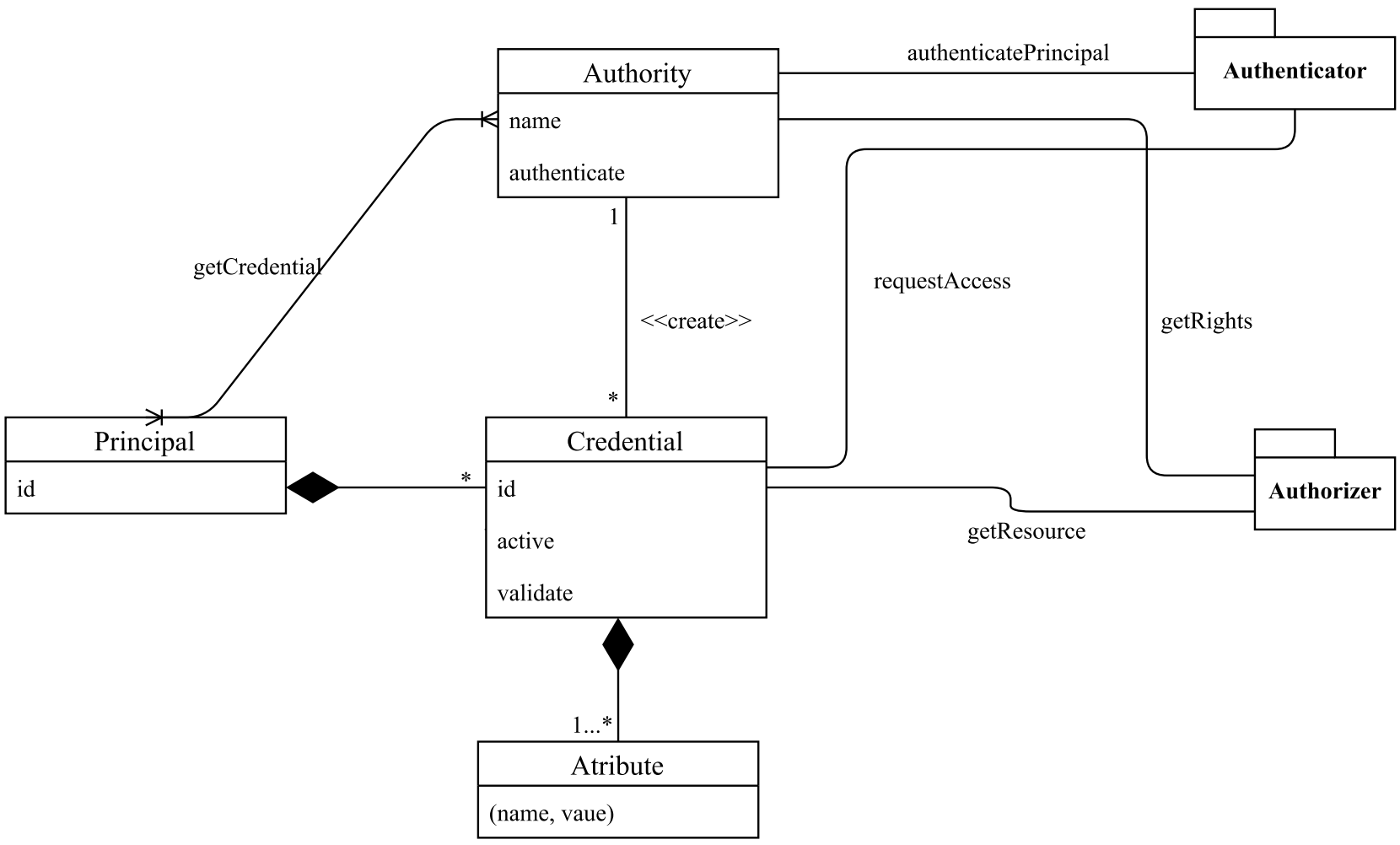


Figure 7.18: The Credential pattern

7.3.1. Discussion

The results obtained for RQ1.3 show that there is a wide range of proposed architectural patterns for microservices. We have no doubt that the authors who have contributed with architectural patterns have done so with the aim of describing recurring solutions that they have used for a particular type of problems. However, we realize there is redundancy concerning the patterns found.

Taking that observation, only a few patterns satisfy recurrent problems in microservices-based systems, and of these, some microservices patterns are taken from other approaches (e.g., SOA). We point this out not to make a gratuitous difference, but to illustrate new knowledge found about microservices patterns.

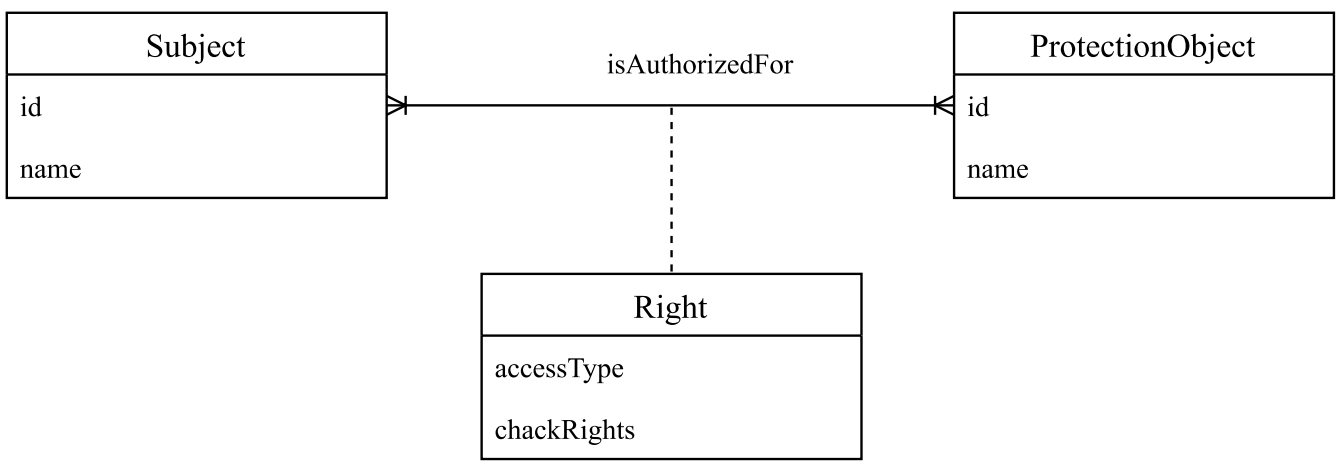


Figure 7.19: The Authorization pattern

7.4. Summary

This chapter depicted a study of the actual use of microservices patterns in open microservices-based projects. The study was conducted through a research process with three steps: (1) gathering evidence on architectural patterns reported in academic and industrial sources, (2) filtering architectural patterns through criteria, and (3) analyzing the use of these architectural patterns in open source projects, to evaluate adoption of microservices patterns. We identified 21 patterns that fit these criteria.

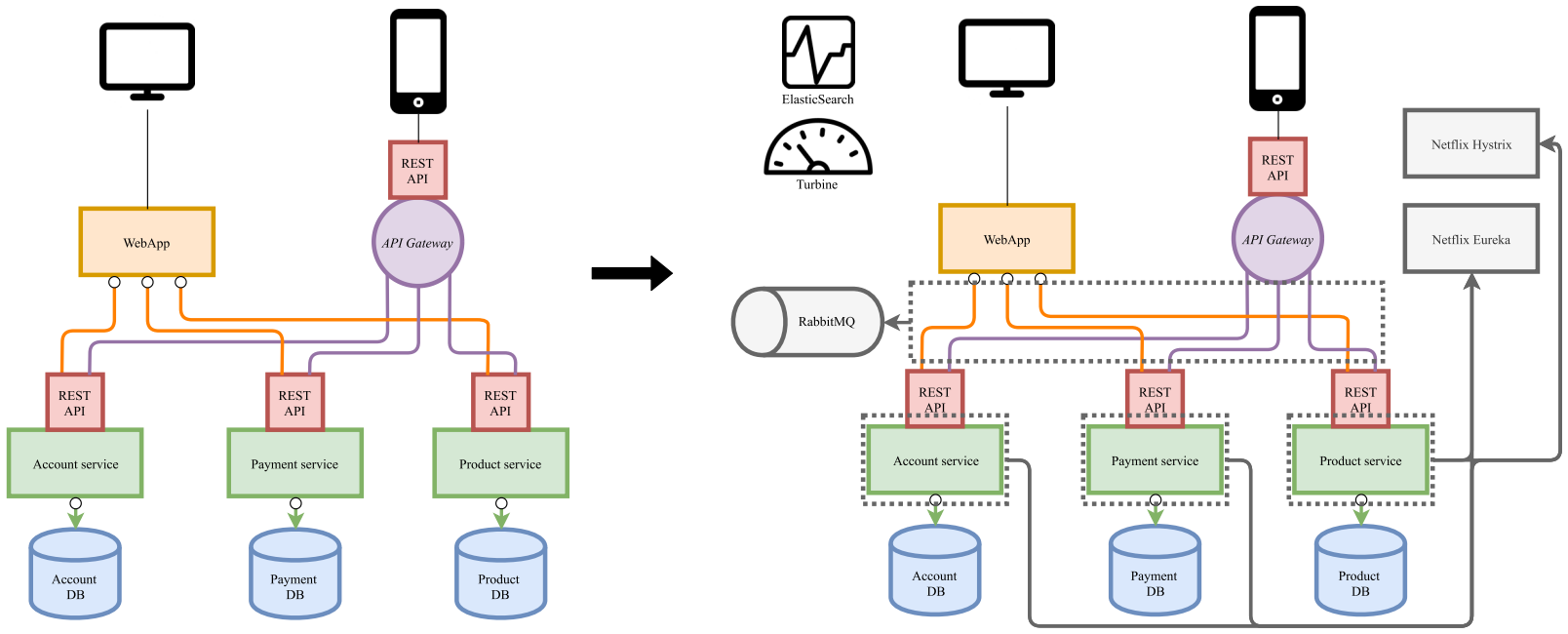
Our findings point to that, the growing use of microservices to develop systems increases software architectural knowledge. Microservices patterns are emerging as a real alternative for architects to evaluate microservice architecture designs. This situation produces many advantages, such as rapid solutions of recurring problems in microservices-based systems, more precise evaluation of frameworks and platforms, among others.

Chapter 8

Systematic Selection of Microservices Frameworks

The development of microservices-based systems requires a complete and robust description of the behavior of the components, connections, and prerequisites. Microservices architectures eliminate the challenges associated with monolithic applications, offering the ability to implement and benefit from the most modern technologies at the web-scale, reducing the use of resources, quickly add new functions without interrupting the service, and easily integrate the solutions developed by third parties. In general, a microservices-based system is cohesion between business services and infrastructure services. Both types of services are based on frameworks that help and facilitate the systematic development of these systems (see Figure 8.1).

The proper selection of frameworks to build microservices-based systems will depend on which NFRs must be satisfied. However, in practice, framework information is imprecise. On the other hand, NFRs are constantly evolving. This chapter describes the fundamentals of the proposed technique, called μAzimut. This technique is a semi-systematic approach that generates, evaluates, and compares sets of frameworks using imprecise information to satisfy continually changing NFRs.



a) *b)*

Figure 8.1: Microservices-based systems in theory (*a)*) and practice (*b)*)

8.1. The μAzimut approach

The μAzimut approach (see Figure 8.4) helps architects in the generation of framework assembly for a set of NFRs[1] (see Figure 8.2).

μAzimut is based on multi-dimensional catalogues of microservices patterns, microservices tactics, and frameworks. These catalogues consider incomplete and imprecise information, i.e., misinformation conditions under which architects usually must generate, evaluate, compare, and select architectural artifacts. The goal of μAzimut is to enable architects to gather frameworks characterization and derive assembly of them in order to satisfy specific NFRs (see Figure 8.3).

The μAzimut catalogues store architects' knowledge about microservices patterns, microservices tactics, and frameworks, as well as rules of satisfaction among them. Hence, they are the key to reusing information about previous selection processes; improve the quality of knowledge about design spaces and frameworks; and to support the architect in the process of exploring these design spaces.

[1]A more detailed description of the microservices tactics, microservices patterns, frameworks, and catalogues of the technique can be found in the following link: `https://github.com/gmarquez87/microAzimut`

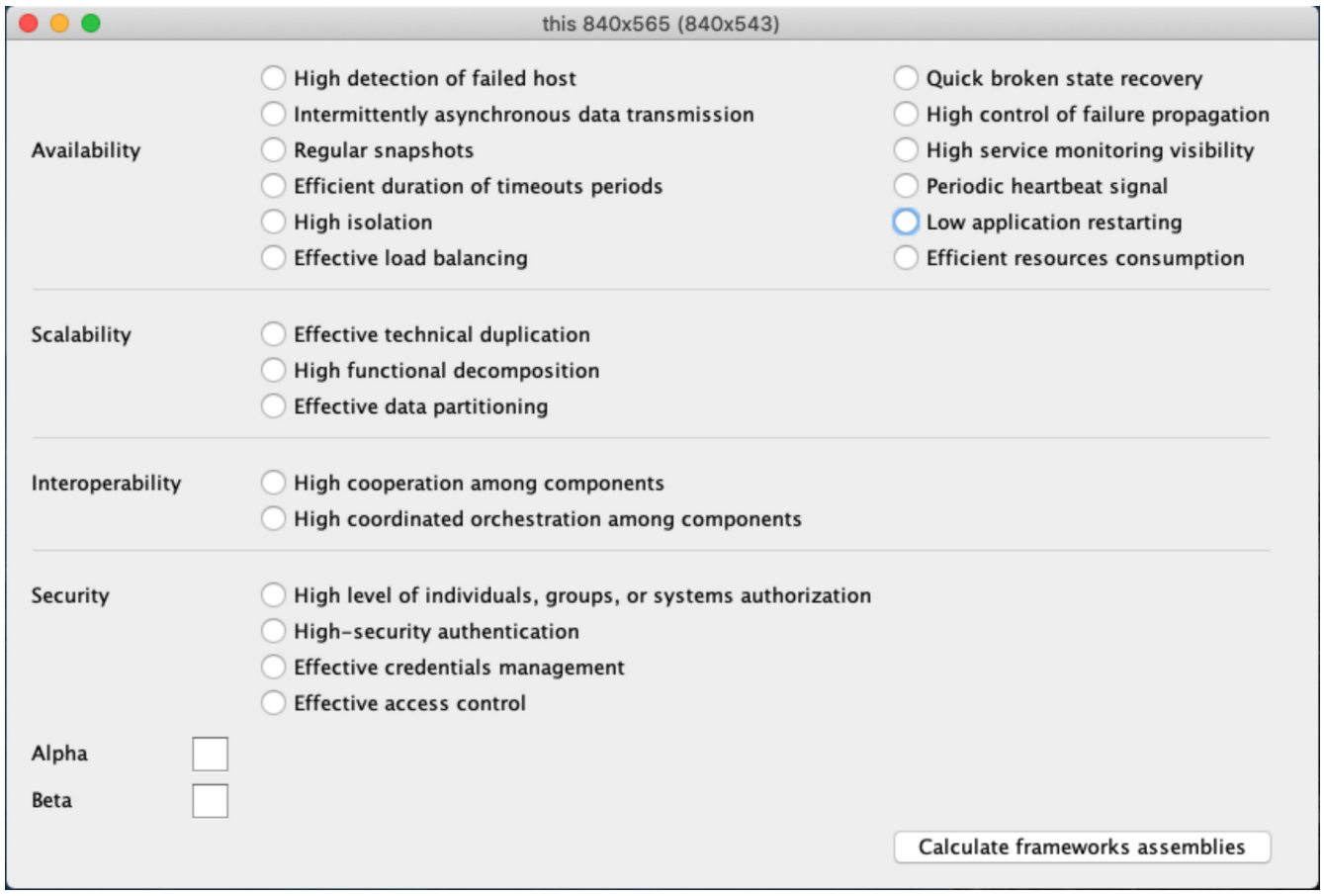


Figure 8.2: The input of μAzimut

Figure 8.6 partially shows the catalogue of microservices tactics and properties. For example, a microservices-based system may require high fault propagation control as property. To achieve this, it is possible to prevent single dependencies, set timeouts and/or providing fallbacks. The same situation occurs for scalability. If the property "efficient execution of multiple identical copies of services" is required, then the following microservices tactics support this property: build separation, container deployment, network location, and balancing scale.

The microservices patterns catalogue records architectural patterns that potentially are related to microservices tactics (see Figure 8.1). A given pattern may be related to tactics for the same concern or tactics across several concerns; similarly, a given tactic may be related by several patterns. In this regard, according to [21] and [70], there is an implicit relationship between patterns and tactics, *patterns are built from tactics*.

In real-world deployment contexts, the catalogue preparators might not know or not be certain whether a microservice pattern supports a certain microservice tactic. Furthermore, NFRs may be imprecise, incomplete, or uncertain, and consequently, it would be difficult to know which microservices pattern fits NFRs. Therefore, the microservices

Table 8.1: Partial catalogue of microservices tactics and patterns

		Microservices tactics						
		Build Separation	Container Deployment	Network Location	Balancing Scale	Preventing Single Dependencies	Set Timeouts	Providing Fallbacks
Microservices patterns	Change Code Dependency	1	0	0	0			
	Container	0.6	1	0	0			
	Load Balancer	0.6	0	1	1			
	Service Discovery	0	0	1	0			
	Circuit Breaker					1	0	1
	Health Check					0	1	0
	Messaging					0.6	0	0
	API Gateway					0	0	0
	Service Registry					0	0	0

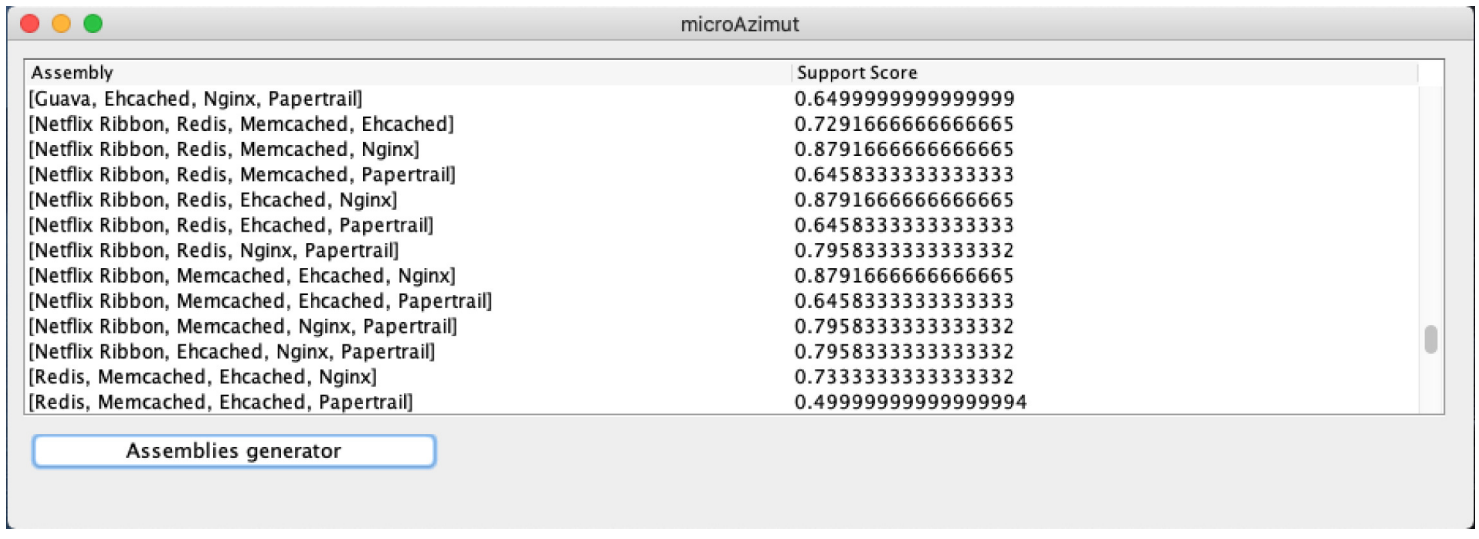


Figure 8.3: The output of μAzimut

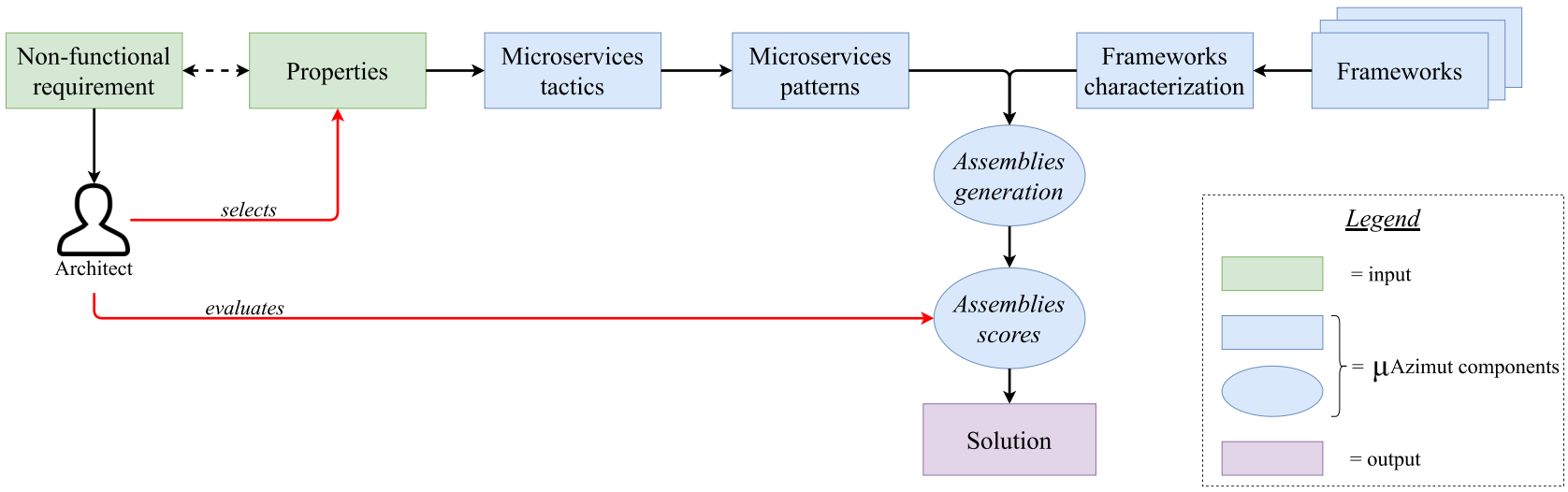


Figure 8.4: The μAzimut approach

pattern catalogue considers credibility degrees regarding support for a given microservices tactic. The values corresponding to credibility degrees are 1 (*does support*), 0.6 (*probably supports*), 0.3 (*possibly does not support*), and 0 (*does not support*). The assignment of credibility degrees was based on the experience described in Chapters 5 and 6. Mainly, to assign the credibility degree to the catalogues, we used almost the same steps that were used to obtain properties and microservices tactics. In short, the steps are:

1. Review the documentation of the framework(s) in GitHub.
2. Look for evidence of the pattern(s) in the framework documentation.
3. If there is explicit information on the implementation of the pattern(s) in the documentation, assign credibility degree=1. It is desirable to look for code files

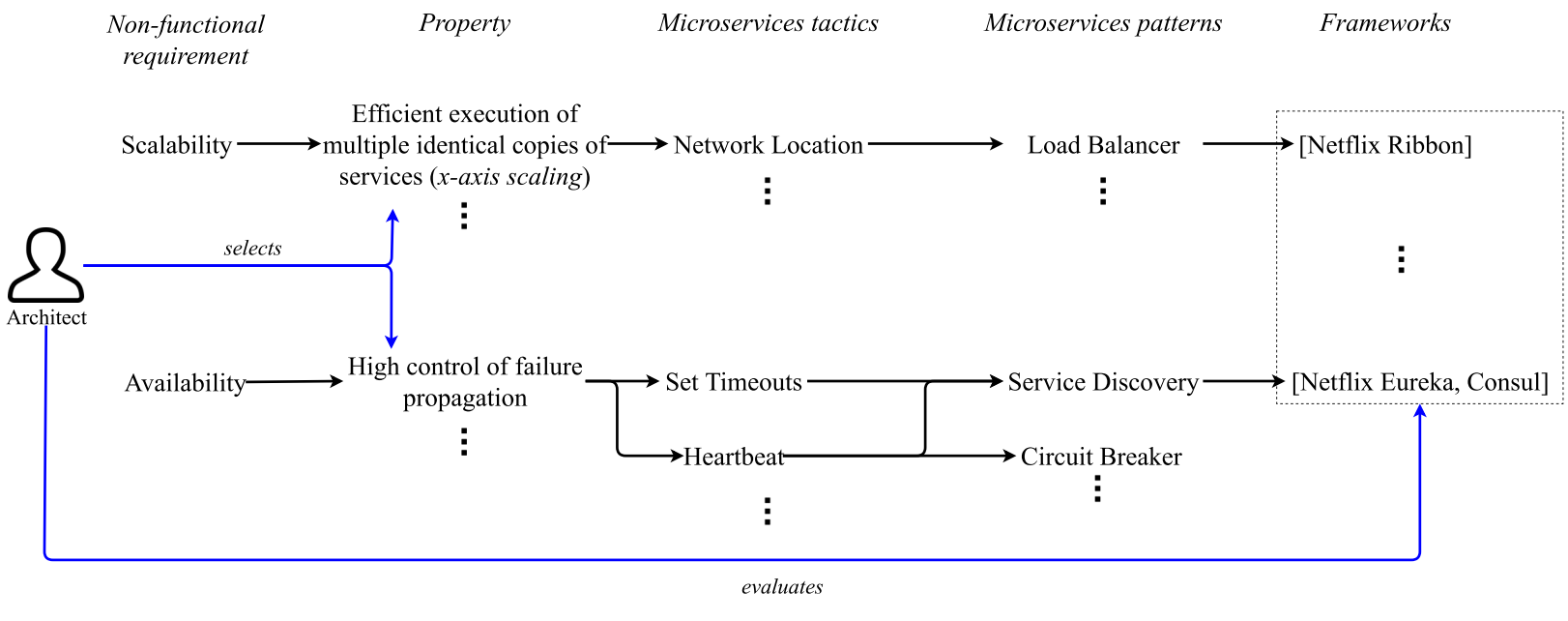


Figure 8.5: Systematic generation of frameworks

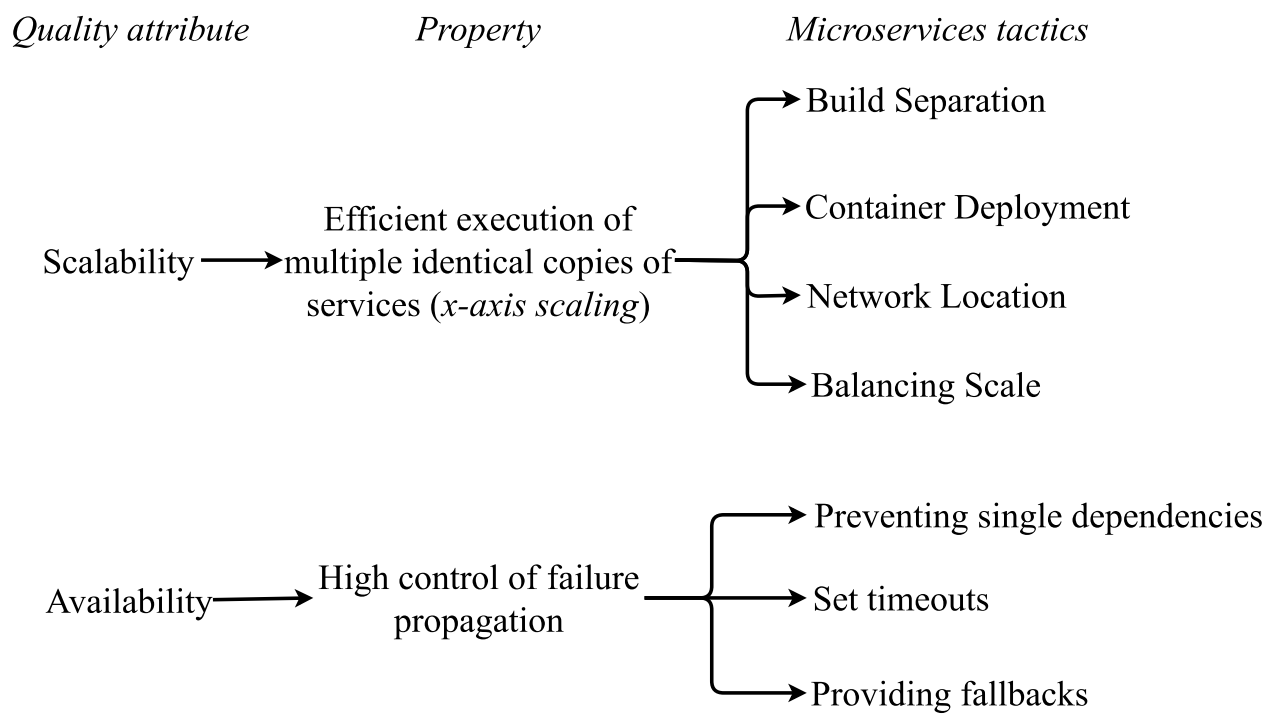


Figure 8.6: Partial content of microservices tactics catalogue

that support the pattern-framework relationship.

4. If the documentation of the framework(s) does not clearly describe the pattern-framework relationship, look in the grey literature (especially in specialized sources such as InfoQ, DZone, among others) for evidence of cases where the framework has been used. The goal of this search is to look for references of the use of the pattern(s).

5. When the information is found, assign credibility degree=0.6 or credibility degree=0.3.

6. If no evidence is found, assign credibility degree=0.

7. Desirable: Analyze the values with microservices developers.

The relationship between architectural tactics and architectural patterns has been widely discussed in academic literature. Bass *et al.* [21] explain that tactics are *architectural building blocks* from which architecture patterns are created; that patterns package tactics [21]. They also illustrate, through example, how the selection of tactics helps determine the selection of the architecture patterns used, which then leads to a further collection of tactics. Thus it is possible to appreciate that patterns, while they may be largely composed of tactics, also provide the starting point for incorporation of tactics. For example, Harrison *et al.* [69] describe a case study of the use of Attribute-Driven-Design [147] shows how the client-server pattern is selected very early in the architecture. One of the reasons for the selection of the early structure is to help satisfy reliability requirements by providing a persistent storage service with state information. Subsequently, tactics are incorporated into the structure provided by the pattern that further satisfy reliability and availability requirements through duplication and other tactics.

On the other hand, Harrison *et al.* [70] also mentioned that not all software development is greenfield. It is often necessary to enhance quality attributes of existing systems. This is probably much more common than designing a new system. This means that tactics must be added to the existing architecture patterns, regardless of how difficult it is. Knowledge of the interaction of patterns and tactics helps the system maintainer understand how to implement a given tactic in a given architecture.

For example, Table 8.2 describes the μAzimut partial frameworks catalogue for scalability. This Table illustrates that several microservices patterns and frameworks combinations (assemblies) are possible.

Table 8.2: Partial catalogueue of frameworks and microservices patterns for scalability

		Microservices patterns						
		Change Code Dep.	Container	Load Balancer	Service Disco.	Result Cache	Scalable Store	Key Value Store
Frameworks	Netflix Eureka	0	0	1	1	0	0	0
	Netflix Hystrix	0	0	1	0	0	0	0
	Memcached	0	0	0	0	0.6	1	0
	Ehcache	0	0	0	0	1	0	0
	Nginx	0.6	0	1	0	0	0	0
	Redis	0	0	0	0	1	1	0.6
	Apache Zookeeper	0.6	0	1	0.6	0	0	0
	Netflix Ribbon	0	0	1	0	1	0	0

8.1.1. Support score for frameworks assemblies

At this point, we assume stakeholders reason in terms of *uncertainty* (from credibility) and *indetermination* (from incompleteness). Therefore, fuzzy statements are acceptable and adequate representations of imprecise knowledge concerning microservices tactics establishment and microservices patterns or frameworks descriptions. For example, "microservices pattern X supports microservices tactic Z with credibility 0.6" or "framework Y implements microservices pattern X with credibility 0.8" will be used as the basis for derivation of frameworks assemblies from requirements.

Indetermination arises in catalogues when there is no credibility value designated to a statement, either because no information is available or because architects have not defined a credibility value yet. The remainder of this section only deals with imprecision as fully modelled and illustrated. Any indetermination is considered as a case of null credibility. The remainder of this section only deals with imprecision as fully modeled and illustrated. Any indetermination is considered as a case of null credibility.

Let us suppose a specific representational situation where microservices patterns are described by dimensions and they are implemented by frameworks. Let D, MP, FW, MT be the set of dimensions, microservices patterns, frameworks, and microservices tactics. In addition, n_d, n_{mp}, n_{fw}, n_{mt} the corresponding sizes. We establish that a tactic $mt \in MT$ is represented by dimensions in D if the exist a set $D_{mt} \subset D$ helping to describe it.

The number $\mu\left(d, mp\right) \in [0, 1]$ is defined as the credibility level from which a pattern $mp \in MP$ supports a tactic mt on the dimension $d \in D_{mt}$. In turn, $\forall d \in D - D_{mt}, \mu\left(d, mp\right) = 0$. Similarly, let a framework $fw_i \in FW \mid (i = 1, \ldots, n_{fw})$ be the set $fw_i = \left\{mp_{i,1}, mp_{i,2}, \ldots, m_{i,n_{mp}} \in MP\right\}$ of microservices patterns, we establish that a pattern is implemented by a framework with credibility level $\mu\left(fw_i, mp_{i,j}\right) \in [0, 1]$.

We define the **aggregate support score** of a tactic mt in relation to pattern mp, at

minimum credibility level of a pattern $\alpha \in (0, 1]$, is described as

$$S_1 (MT, mp, \alpha) = \frac{1}{n_{mt}} \sum_{f(mp, D_{mt})} 1 \tag{8.1}$$

where $f (mp, D_{mt}) = \{d \in D_{mt} \mid \mu (d, mp) \geq \alpha\}$ and n_{mt} is the number of microservices tactics in MT. This formula takes into account the arguments in favor of the statement "mp satisfies the tactic mt". Thus, if a pattern has enough credibility on a dimension to support the tactic, then it provides a favorable argument.

The **support score of a framework** fw, in relation to the tactic mt, at minimum credibility level of a pattern and a framework $\alpha, \beta \in (0, 1]$ respectively, is described as

$$S_2 (D_{fw}, fw, \alpha, \beta) = \sum_{f\left(g\left(fw, D_{fw}\right)\right)} \frac{1}{n_{mt}} \tag{8.2}$$

where $g (fw, D_{fw}) = \{d \in D_{mt} \mid \mu (d, mp) \geq \alpha, \mu (fw, mp) \geq \beta, mp \in MP\}$ and $f (g (fw, D_{fw}))$ is the function composed of the functions $g (fw, D_{fw})$ and $f (mp, D_{mt})$. Parameters α and β are defined by the architect and show how significant a credibility must be in order to be considered a support. This score counts for the argument in a favor of fw faced to the statement "fw satisfies the tactic mt".

The **support score for a framework assembly**, in relation to the tactic mt, is described as

$$S_3 (D_{fw_n}, fw_n, \alpha, \beta) = \sum_{fw_i=1}^{fw_i=n} \left(\sum_{f\left(g\left(fw_i, D_{fw_i}\right)\right)} \frac{1}{n_{mt}} \right) \frac{1}{\mid fw_n \mid} \tag{8.3}$$

where fw_n is a set of frameworks, D_{fw_n} is the set of frameworks of set dimensions fw_n, and D_{fw_i} is the dimension of the framework fw_i. This support score counts dimensions in favor of the statement "c_n satisfies the tactic mt". In summary, the **support score** measures count the credible arguments for a given framework or assembly as a solution

to a given tactic, where "credible" means having a credibility index above the credibility threshold.

8.1.2. Example

Suppose we want to calculate the value of S_1 with $\alpha = 0.6$ for the Load Balancer pattern described in Table 8.1. Using formula 8.1, the calculation proceeds as follows:

$$S_1\left(MT, mp, \alpha\right) = \frac{1}{n_{mt}} \sum_{f(mp, D_{mt})} 1 \implies$$

$$S_1\left(MT, \text{Load Balancer}, 0.6\right) = \frac{1}{4} \sum_{f(mp, D_{mt})} 1 \implies \quad a)$$

$$S_1\left(MT, \text{Load Balancer}, 0.6\right) = \frac{1}{4} \sum_{f(mp, D_{mt})=1}^{3} 1 \implies \quad b)$$

$$S_1\left(MT, \text{Load Balancer}, 0.6\right) = \frac{1}{4}\left(1 + 1 + 1\right) \implies \quad c)$$

$$S_1\left(MT, \text{Load Balancer}, 0.6\right) = \frac{3}{4} = 0.75 \quad d)$$

In a) the following values are assigned in formula 8.1:

$$mp = \text{Load Balancer}$$

$$\alpha = 0.6$$

$$MT = [0.6 \quad 0 \quad 1 \quad 1]$$

$$n_{mt} = \mid MT \mid \implies n_{mt} = 4$$

Then, in b) the following calculations were made:

Finally, in c) and d) we proceed to calculate the mathematical operations and obtain the support score, which is 0.75. This value means that if an architect is analyzing a medium priority NFR ($\alpha = 0.6$), the microservices pattern Load Balancing has an acceptable support score and can be an alternative to satisfy the NFR. However, for a critical NFR ($\alpha = 1$) the support score decreases to 0.5, as it is not acceptable for this

$$S_1\left(MT, \textsc{Load Balancer}, 0.6\right) = \frac{1}{4} \sum_{f(mp,D_{mt})=1}^{3} 1$$

$$\sum_{f(mp,D_{mt})} 1 = \sum_{f(mp,D_{mt})=1}^{3} 1$$

$$f\left(mp, D_{mt}\right) = \left\{d \in D_{mt} \mid \mu\left(d, mp\right) \geq \alpha\right\}$$

D_{mt}	$\mu\left(d, mp\right) \geq \alpha$	$f\left(mp, D_p\right) = d$
1	True	1
2	False	-
3	True	2
4	True	3

requirement to make decisions with imprecise information.

In the following examples (next pages), we show how to calculate S_2 and S_3 using dummy catalogues. For S_2, the catalogue is given by D_fw, and the framework is Nginx. On the other hand, for S_3, the catalogues for Netflix Eureka and Netflix Zuul are described in the same example.

8.2. Summary

In this chapter, we have described μAzimut, a technique that allows satisfying NFRs to design microservices-based systems through the analysis of frameworks assemblies. The technique converges the key findings of chapters 5, 6, and 7 as architectural knowledge and multi-dimensional catalogues. On the other hand, we presented illustrative examples of how to evaluate framework assemblies in order to explain how support scores work.

$$S_2\left(D_{fw}, fw, \alpha, \beta\right) = \sum_{f\left(g\left(fw, D_{fw}\right)\right)} \frac{1}{n_{mt}} \quad \Rightarrow \quad S_2\left(D_{fw}, \text{Nginx}, 1, 0.3\right) = \sum_{f\left(g\left(fw, D_{fw}\right)\right)} \frac{1}{n_{mt}} \quad \Rightarrow$$

$$\begin{array}{l} fw = \text{Nginx}\ ; \quad \alpha = 1; \quad \beta = 0.3 \\ D_{fw} = \left[\begin{array}{cccccccccccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.4 & 0.4 & 0.4 & 0.6 & 0.6 & 0.4 & 0 & 0 \end{array}\right] \end{array}$$

$$\Rightarrow \quad S_2\left(D_{fw}, \text{Nginx}, 1, 0.3\right) = \sum_{f\left(g\left(fw, D_{fw}\right)\right)=1}^{7} \frac{1}{n_{mt}} \quad \Rightarrow$$

$$\sum_{f\left(g\left(fw, D_{fw}\right)\right)} \frac{1}{n_{mt}} = \sum_{f\left(g\left(fw, D_{fw}\right)\right)=1}^{7} \frac{1}{n_{mt}} = \frac{1}{11} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{9} + \frac{1}{9} + \frac{1}{6} \quad \Rightarrow$$

D_{fw}	$g(fw, D_{fw})$	$f(mp, D_p)$											
1	True	1	0	1	0	0	1	0	1	0	1	0	$n_{mt} = 11$
0	False												
0	False												
0	False												
0	False												
0	False												
0	False												
0.4	True	0	0	1	0	0	1						$n_{mt} = 6$
0.4	True	0	0	1	0	0	1						$n_{mt} = 6$
0.4	True	0	0	1	0	0	1						$n_{mt} = 6$
0.6	True	1	0	0	1	0	1	0	0	1			$n_{mt} = 9$
0.6	True	1	0	1	0	0	1	0	1	0			$n_{mt} = 9$
0.4	True	0	0	1	0	0	1						$n_{mt} = 6$
0	False												
0	False												

$$\Rightarrow \quad = \frac{18 + 33 + 33 + 33 + 22 + 22 + 33}{198} = \frac{194}{198} = 0.979797$$

$$\Rightarrow \quad S_2\left(D_{fw}, \text{Nginx}, 1, 0.3\right) = 0.979797$$

$$S_3\left(D_{fw_n}, fw_n, \alpha, \beta\right) = \sum_{fw_{i=1}}^{fw_{i=n}} \left(\sum_{f\left(g\left(fw_i, D_{c_i}\right)\right)} \frac{1}{n_{mt}} \right) \cdot \frac{1}{\mid fw_n \mid} \quad \Rightarrow$$

$fw_n = [\text{Netflix Eureka, Netflix Zuul}]\ ; \quad \alpha = 1; \quad \beta = 0.5$

$$\Rightarrow \quad S_3\left(D_{fw_n}, [\text{Netflix Eureka, Netflix Zuul}], 1, 0.5\right) = \sum_{c_{i=1}}^{c_{i=n}} \left(\sum_{f\left(g\left(fw_i, D_{c_i}\right)\right)} \frac{1}{n_{mt}} \right) \cdot \frac{1}{\mid fw_n \mid} \quad \Rightarrow$$

$$\left(\sum_{f\left(g\left(fw_i, D_{c_i}\right)\right)} \frac{1}{n_{mt}} \right) \cdot \frac{1}{\mid fw_n \mid} = \left(\frac{1}{11} + \frac{1}{6} + \frac{1}{6}\right) \cdot \left(\frac{1}{2}\right)$$

$fw_i = \text{Netflix Eureka}; \quad \mid fw_n \mid = 2$

D_{c_i}	$g(fw_i, D_{c_i})$	$f(mp, D_p)$											
1	True	1	0	1	0	0	1	0	1	0	1	0	$n_{mt} = 11$
0	False												
0	False												
0	False												
0	False												
0	False												
0	False												
1	True	0	0	1	0	0	1						$n_{mt} = 6$
0	False												
0	False												
1	True	1	0	0	1	0	1	0	0	1			$n_{mt} = 9$

$$\left(\sum_{f\left(g\left(fw_i, D_{c_i}\right)\right)} \frac{1}{n_{mt}} \right) \cdot \frac{1}{\mid fw_n \mid} = \left(\frac{1}{9} + \frac{1}{9}\right) \cdot \left(\frac{1}{2}\right)$$

$fw_i = \text{Netflix Zuul}; \quad \mid fw_n \mid = 2$

D_{c_i}	$g(fw_i, D_{c_i})$	$f(mp, D_p)$									
0	False										
0	False										
0	False										
0	False										
0	False										
0	False										
0	False										
0	False										
0	False										
0	False										
1	True	1	0	0	1	0	1	0	0	1	$n_{mt} = 9$
1	True	1	0	1	0	0	1	0	1	0	$n_{mt} = 9$

$$\Rightarrow \quad S_3\left(D_{fw_n}, [\text{Netflix Eureka, Netflix Zuul}], 1, 0.5\right) = \left(\frac{6+11+11}{66}\right)\cdot\frac{1}{2}+\left(\frac{2}{9}\right)\cdot\frac{1}{2} \quad \Rightarrow$$

$$\Rightarrow \quad S_3\left(D_{fw_n}, [\text{Netflix Eureka, Netflix Zuul}], 1, 0.5\right) = \left(\frac{28}{66}\right)\cdot\frac{1}{2}+\frac{2}{18} \quad \Rightarrow$$

$$\Rightarrow \quad S_3\left(D_{fw_n}, [\text{Netflix Eureka, Netflix Zuul}], 1, 0.5\right) = \frac{28}{132}+\frac{1}{9} \quad \Rightarrow$$

$$\Rightarrow \quad S_3\left(D_{fw_n}, [\text{Netflix Eureka, Netflix Zuul}], 1, 0.5\right) = \frac{3\cdot 28+44\cdot 1}{396} \quad \Rightarrow$$

$$\Rightarrow \quad S_3\left(D_{fw_n}, [\text{Netflix Eureka, Netflix Zuul}], 1, 0.5\right) = \frac{84+44}{396} \quad \Rightarrow$$

$$\Rightarrow \quad S_3\left(D_{fw_n}, [\text{Netflix Eureka, Netflix Zuul}], 1, 0.5\right) = \frac{128}{396} \quad \Rightarrow$$

$$\Rightarrow \quad S_3\left(D_{fw_n}, [\text{Netflix Eureka, Netflix Zuul}], 1, 0.5\right) = 0.32323$$

Part IV

Empirical studies

Chapter 9

Empirical validation

In this Chapter, we illustrated three empirical studies that were conducted to evaluate and test μAzimut. The first empirical study corresponds to a case study whose primary focus is to evaluate frameworks assemblies with respect to an architect's decision in a real microservices-based system. The second empirical study corresponds to another case study where we involved stakeholders in evaluating μAzimut-generated microservices architecture designs for an Internet of Medical Things (IoMT) platform. Finally, the third empirical study consists of a controlled experiment where we evaluated the impact of the essential architectural construct of μAzimut, microservices tactics. The three empirical studies address three different angles of μAzimut's effectiveness and usefulness to be used as a decision support tool in microservice architectures.

9.1. Case study: Salary Payment System

9.1.1. Context

Currently, a Chilean forestry company (from now on, it will be referenced with the fictitious name of "ABC") meets the objective of promoting and protecting forest resources to generate goods and services for the restoration and recovery of forests. Given the

versatility of services, processes, and the different profiles of employees working in the company, one of the most complex systems it has is the salary payment system. The current payment system is 20 years old. Because of this, the company's owners have decided to update the salary payment system.

The business management software market does not meet the expectations of the stakeholders, since adapting such systems to the ABC company's business is costly. For this reason, it is necessary to develop a salary payment system that facilitates and minimizes the time of the salary payment calculation process. In turn, the system must interoperate with other internal and external government systems.

Based on the scenario described above, the team in charge of developing the salary payment system has opted for a microservices-based system, since this type of system will help meet the needs of stakeholders.

9.1.2. Case study design

Research objective

In this case study, we evaluated μAzimut based on the frameworks that were selected by the development team to satisfy NFRs. More precisely, we want to evaluate 2 NFRs related to Availability and Scalability, respectively[1]. We selected these quality attributes because these attributes are the most relevant in the salary payment system and concentrate the most significant amount of development so far.

Research questions

Based on the objective of the case study, the research question is the following

[1]By confidentiality agreement, the detailed description of each NFR are omitted.

RQ4.1

How close the solution, generated by μAzimut, is concerning the selection of frameworks, taken by the project leader architect, for each NFR?

This RQ intends to evaluate the precision (the ratio between the correct predictions and the total predictions) and recall (the ratio of the correct predictions and the total number of correct items in the set) of that assembly generated by μAzimut and selected by the subject participating in the case study (*the solution*) with respect to the decision made by the lead architect related to the framework selection that was conducted to satisfy the same NFRs in the project in practice.

9.1.3. Preparation

The first activity to conduct the case study is to update the catalogues. To this end, we requested meetings with the development team to obtain the list of frameworks that were used to satisfy the NFRs. Subsequently, we researched the grey literature and executed the activity mentioned in Section 8.1 to update the catalogues. Those frameworks in which we did not find evidence were omitted. The second activity is to present the technique and goal of the case study to the lead architect. We omitted details of μAzimut so as not to bias the architect's decision. The time required for the analysis of the two NFRs was one hour.

9.1.4. Collection of data

The data was collected through a form that is filled out by the architect. For each NFR, the form asks for the selected properties, the selected assembly and the rationale of the selection. For the sake of simplicity, α and β of μAzimut are set to 1. In addition, we adjusted μAzimut to show the top three assemblies with the highest support scores.

9.1.5. Analysis of collected data

The results of the case study are described in Table 9.1

Table 9.1: Case study results

NFR1	*Availability*
Expectation	[Netflix Eureka, RabbitMQ, Swagger, Netflix Ribbon, Kubernetes]
Assembly selected	[Netflix Eureka, Swagger, Netflix Ribbon]
Precision	100%
Recall	60%
NFR2	*Scalability*
Expectation	[Netflix Eureka, Netflix Hystrix, MongoDB]
Assembly selected	[Netflix Hystrix, MongoDB]
Precision	100%
Recall	67%

For NFR1, the properties selected by the architect in order to execute μAzimut are related to the following objectives: (i) detecting bad hosts for the purpose of load balancing in order to send requests to those hosts, (ii) asynchronous communication between services, and (iii) efficient distribution of incoming network traffic among groups of back-end servers. The architect estimated that these objectives are necessary to satisfy NFR1; therefore, he selected those properties closer to these objectives. As a result, the architect chose the assembly described in Table 9.1 for NFR1. The architect mentions that he is satisfied with the result shown by μAzimut, but disagrees about the fact that the technique has not shown frameworks related to asynchronous communication (such as RabbitMQ) in the assembly. Furthermore, for this NFR1, the architect would have expected to have more frameworks related to workload management between services, given the nature of NFR1.

Regarding NFR2, the only concern of the architect is to have a good load balance to manage the interaction of new services corresponding to companies and external suppliers. Having said this, the architect selected those properties related to the described objective and chose the assembly described in Table 9.1 for the NFR2. For this requirement, the architect is quite happy that μAzimut has shown MongoDB. Prior to the case study, the development team had been working hard to apply sharding (distribution of information between different machine clusters) to satisfy the NFR2.

After months of searching, they discovered that MongoDB had the capability to work with this technique.

The feedback provided by the architect suggests that μAzimut is quite useful for comparing decisions. Since μAzimut stores and structures architectural knowledge regarding microservices architectures, it turns out to be a good ally to at least have a basis to start making architectural decisions.

9.1.6. Lessons learned

Architectural decisions in microservices-based systems are a complex task. According to feedback from the principal architect, although microservices architectures provide many advantages, the most difficult thing is to achieve the satisfaction of stakeholder concerns. Unlike other types of architectures, microservices architectures have features (such as scalability, flexibility, maintainability, among others) that make it attractive to developers and architects; however, operationalizing such features is not trivial.

In this case study, we realized that μAzimut, despite having a limited set of microservices patterns, microservices tactics and frameworks, was very useful for the architect to make more informed decisions. Both the developers and the architect valued μAzimut's responses because they know that the framework assemblies are generated through knowledge representation between microservices patterns, microservices tactics and frameworks. Furthermore, since μAzimut inputs are QA properties, they help conduct the decisions that the architect must address in the system. Thus, the selection of frameworks can be based on the properties that the architect must address in the system.

9.1.7. Study limitations

μAzimut was used to evaluate two NFRs from the salary payment system. Although we cannot extrapolate the results of the case study to the whole project, the encouraging

results and the positive feedback from the lead architect inspire the further improvement of the technique with the aim of being a decision support in the evaluation of frameworks for developing microservices-based systems.

9.2. Case study: Microservices-based IoMT platform and stakeholders

9.2.1. Background

People who are interested in the system are often referred to as *stakeholders*. More precisely, a stakeholder is anyone who has a stake in the success of the system: the customer, the end-users, the developers, the project manager, the maintainers, and even those who market the system [21]. But the stakeholders, although they all share the desire for the success of the project, have different specific needs that they want the system to guarantee, optimize, or deliver to a target audience. Depending on the context in which the system is involved, these needs encompass different types and levels of concern. In this regard, some studies, such as [11] [114], investigate the role of stakeholders in decision making in system development. The results of the studies point out that stakeholders have a significant influence on decisions in the early stages of system development, specifically in the capture and elicitation of requirements. Nevertheless, as system development progresses, stakeholders reduce their participation, and design and implementation decisions fall on the judgment of experts, architects, technical leaders, among others.

In the context of microservices-based systems, some studies identify the different profiles and roles of stakeholders who participate in microservices-related projects. For example, Haselböck *et al.* [71] identify six types of stakeholders, which are: software architect, developer, application engineering, quality assurer, and manager. On the other hand, Rademacher *et al.* [117] describe that from the point of view of microservice architecture modeling, there are three types of stakeholder roles, which are: domain

expert, service developer, and service operator. When exploring practical experiences of the use of microservices-based systems (such as [25] [16] [87]), we have noticed that the participation of stakeholders is high in the modeling, design and analysis phases of microservices development, but low in the implementation phase. Although decisions in the implementation phase are often led by technical experts and architects, one of the main issues in our knowledge regarding microservices development is the lack of stakeholder participation in decision-making to select the appropriate technological tools to develop microservices-based systems. More precisely, technical experts and architects evaluate frameworks and platforms that will be used to implement the microservices-based system design; nevertheless, there is not always clarity and determination if such frameworks and platforms address the needs of stakeholders.

9.2.2. Stakeholders and μAzimut

μAzimut conducts the systematic generation of framework assemblies through the selection of properties. To incorporate stakeholders in the use of μAzimut and eventually, in the evaluation of implementation alternatives, we modify μAzimut so that the selection of properties is made collaboratively, i.e., different perspectives of stakeholders are considered to select properties through consensus (see Figure 9.1).

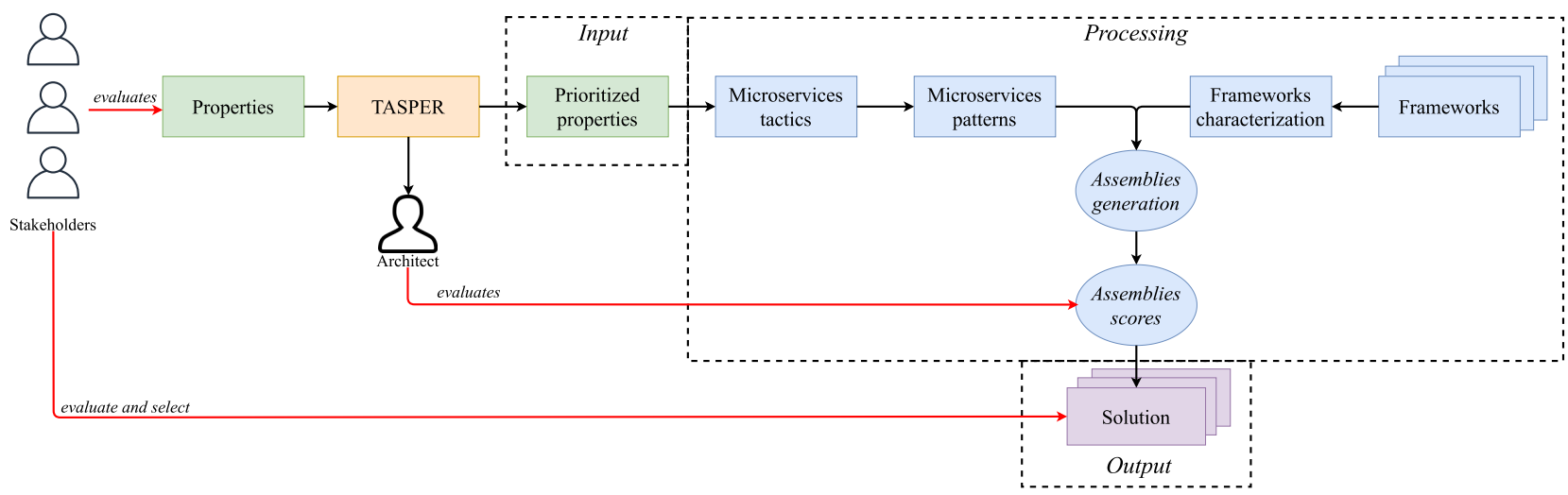


Figure 9.1: The extended μAzimut approach

To conduct the consensual selection of properties among stakeholders, we used part of the TaSPeR technique to define the steps of discussion and decision by consensus [109]. TaSPeR is a card game-based technique and consensus-building technique (based

on Planning Poker) that allows development team members to identify, argue for, and choose among security architectural design decisions according to objectives and priorities. For this μAzimut extension, we use neither cards nor security architectural designs; properties replace these artifacts.

During the consensus decision process, stakeholders must prioritize properties. In this context, several studies address some challenges regarding the prioritization of quality attributes. Dabbagh *et al.* [39] propose an approach to prioritize quality attributes under two approaches. The first is to prioritize based on the importance of customers and the second aims to apply an eliminatory approach to ensure consistency in the list of prioritized quality attributes. In the same line, Dabbagh *et al.* [40] conduct an empirical study in which they evaluate different techniques of prioritizing NFRs. The results of the empirical study describe that the integrated prioritization approach (IPA) technique obtains better results than the other techniques used. On the other hand, Thakurta [133] proposes a negotiation algorithm in order to facilitate the selection of quality attributes through the satisfaction of business objectives. Gupta *et al.* [68] describe a prioritization technique based on the collaborative dependence of requirements, which takes into consideration multiple criteria to obtain individual preferences.

For our proposal, we inspired by the practical experiences reported by McGee *et al.* [97] and M. Svahnberg [127] to prioritize quality attributes because these studies consider software architecture aspects as prioritization variables. Each stakeholder must prioritize properties using the letters H (high, value=1), M (medium, value=0.6), and L (low, value=0.3). Likewise, the prioritization is based on two dimensions (i) business goals and (ii) the maintainability and performance of the system. The business objectives define the actions to be taken in order to fulfill the mission and vision of the organization. On the other hand, maintainability and performance are some of the most relevant systematic properties in microservices-based systems [42]. Maintainability is defined as the probability of performing a successful repair action within a given time. At the same time, performance measures how effective is a software system with respect to time constraints and allocation of resources. We have selected

these dimensions so that stakeholders can select priorities through tradeoff analysis between business and microservices architecture aspects. Under those circumstances, the TaSPeR technique suggests the following steps aiming at achieving stakeholders' consensus:

1. Each stakeholder argues their choice and prioritization
2. A moderator[2] records the rationales manifested by stakeholder
3. If one or more properties are selected by all stakeholders, it became in a selected property (o properties)
4. If there is no consensus on a property, stakeholders can argue their rationale and try to make a new common choice. If they still do not reach an agreement, the property is rejected.
5. Repeat steps 1), 2), 3) and 4) until no NFR remains to be analyzed.

Once the prioritized properties are selected, they become the input for μAzimut. The calculation of the support scores considers two inputs by the architect, the degree of importance of the quality attribute property (hereafter, α) and the degree of importance of the frameworks that will be used (hereafter, β). Both alpha and beta determine the framework assemblies that the architect must evaluate. For example, if the architect is analyzing a critical NFR, both α and β should be 1. Thus, the μAzimut calculation algorithm generates a set of assemblies that meet this condition.

However, what we propose in this research is α should be the result of the consensus decision and α=β. Additionally, instead of the architect selecting the framework assembly and designing the architecture on her/his own, the architect should create comparison matrix using the framework assemblies generated by μAzimut and the priorities. The objective of this matrix is to enable the architect to argue how the prioritizations defined by the stakeholders are satisfied in each assembly. The satisfaction

[2]By moderator, we mean the lead architect, project manager, or anyone who has the capacity to make architectural decisions on a project.

level is classified using the following labels: Yes (Y), Partially (P), and No (N). Using this classification, the stakeholders quickly will know which framework assembly is the most convenient for them.

Limitations

μAzimut uses two datasets composed of 26 microservices tactics, 18 microservices patterns and 25 frameworks. Although the market for microservices frameworks is wide, μAzimut only uses frameworks in whose documentation it is possible to identify both microservices patterns and microservices tactics. This constraint eventually limits the analysis capacity of architects and stakeholders to satisfy only certain NFRs. With the intention of further increasing the capacity of μAzimut, we are extending the spectrum of frameworks through a collaborative platform that allows to gather and describe frameworks that practitioners use to develop microservices-based systems. To add a framework to this platform, practitioners must describe what problem the framework solves and what design decisions the framework realises. These data describe potential microservices patterns and microservices tactics, respectively.

Illustrative example

Let us consider the following case: an architect is evaluating the design and implementation of an IoT architecture. She or he evaluates four NFRs corresponding to the following quality attributes: Availability, Scalability, Interoperability and Security.

The first step of our proposal is the prioritization of the properties that should satisfy the microservices architecture. To this end, the architect executes the TASPER technique to involve the stakeholders in evaluating all properties and prioritizing them. Table 9.2 describes the selected properties.

Since the properties described in Table 9.2 are significant for stakeholders, then $\alpha = 1$, consequently β=1. Once the properties are selected, μAzimut proceeds to calculate the support scores for each QA in order to obtain a list of framework assemblies that satisfy

Table 9.2: Selected properties for the illustrative example

QA	ID	Property	Description
Availability	A1	High detection of failed host	This property defines the capability of detect failed hosts in order to a load balancer can stop requests to them.
	A2	High isolation	The property where each microservices is its own encapsulated application.
	A3	Effective load balancing	Efficiently distributing incoming network traffic among groups of backend servers.
	A4	Periodic heartbeat signal	Property related to the periodic signal to check the status of services.
Scalability	S1	Effective technical duplication	This property focuses on the need to execute multiple identical copies of an application behind a load balancer, to improve its capacity and availability.
	S2	High functional decomposition	This property focuses on separating services and data along noun or verb boundaries, allowing segmentation of teams and ownership of code and data.
Interoperability	I1	High coordinated orchestration among components	This property points to a control mechanism to coordinate, manage and sequence the invocation of particular components (which could be ignorant of each other).
Security	SC1	Strong level of individuals, groups, or systems authorization	Capacity of control users to grant them limited access to platforms systems resources without having to expose their credentials.
	SC2	Effective credentials management	Property related to the management of credentials, making them available to less or high privileged users for authentication to other systems without giving them access to the credentials themselves.

the properties selected by the stakeholders. For simplicity of this example, Table 9.3 describes the first three framework assemblies generated by μAzimut.

Once the assemblies for each NFR are generated, the architect then proceeds to assemble them and obtain a final list of assemblies. For each assembly, the architect must evaluate the level of satisfaction of the properties defined by the stakeholders (see Table 9.4). In this way, the architect argues the pros and cons of each assembly and thus selects, together with the stakeholders, the most appropriate assembly.

Table 9.3: Ranked framework assemblies

Ranking	Availability	Scalability	Interoperability	Security
1	A_{av}=[Netflix Eureka, Netflix Zuul, Netflix Ribbon]	A_{sc}=[Kubernetes, Docker]	A_{inter}=[Mosquitto]	A_{sec}=[OAuth, Netflix Zuul]
2	A_{av}=[Netflix Eureka, Netflix Zuul]	A_{sc}=[Kubernetes]	A_{inter}=[Mosquitto]	A_{sec}=[Netflix Zuul]
3	A_{av}=[Netflix Eureka, Netflix Ribbon]	A_{sc}=[Docker]	A_{inter}=[RabbitMQ]	A_{sec}=[OAuth]

Table 9.4: Framework assemblies comparison matrix

Frameworks assemblies options	A1	A2	A3	A4	S1	S2	I1	SC1	SC2
A_{opt1}=[Netflix Eureka, Netflix Zuul, Netflix Ribbon, Kubernetes, Docker, Mosquitto, OAuth, Netflix Zuul]	Y	Y	Y	Y	P	Y	Y	Y	P
A_{opt2}=[Netflix Eureka, Netflix Zuul, Kubernetes, Mosquitto, Netflix Zuul]	Y	N	Y	Y	P	Y	Y	P	P
A_{opt3}=[Netflix Eureka, Netflix Ribbon, Docker, RabbitMQ, OAuth]	Y	P	Y	Y	P	Y	P	Y	P

9.2.3. Case study

Context

In the last few years, the population of adults over 60 years has presented a global expansion. A significant group of older adults is forced to live alone in their homes, implying an increased likelihood of suffering distress situations related to home accidents. In this regard, falls are especially relevant to patients and health systems because approximately one-third of adults older than 65 that live in a community suffer a fall each year [72].

Aiming at facing current challenges concerning the care of aged patients, sensor-based technologies arises as an alternative. Using modern technologies, such as smart devices, it is possible to collect health-related sensor data from elderly patients, such as physiological, actimetric, and others, which need complex analysis and interpretation [132].

The three main project stakeholders requested the design of an AAL system using an IoMT environment to visualize a technological solution for monitoring elderly patients with different types of devices (not only sensors). Additionally, to facilitate the integration of devices and other systems and, in turn, ensure the scalability and availability of services, stakeholders require a microservices-based system.

Objectives and research questions

The objective of this case study is to evaluate the support, from the stakeholders' viewpoint, of μAzimut to generate solutions that can be evaluated by them in order to select the best alternative to implement the AAL system. Consequently, the main research question of the case study is the following

RQ4.2
What effect does the use of μAzimut produce on the project's stakeholders concerning the effort of evaluating implementation alternatives?

The purpose of this research question is to evaluate the consequences on stakeholders of using μAzimut as a means of decision making to analyze implementation alternatives of the AAL system.

Data collection

Although μAzimut has an initial core of multidimensional catalogs created from previous experiences, as the μAzimut technique suggests, for this case study, we investigated other studies (such as [105] [5] [35] [130]) to gather frameworks that have been used in AAL and IoMT/IoT-based systems. Using inclusion and exclusion criteria defined by μAzimut to include new frameworks, we updated the catalogs and calibrated the corresponding credibility degrees. In turn, we used the information gathered in the system design phase regarding architectural patterns and tactics (explained in the following section) in order to update μAzimut catalogs.

Preparation

Before evaluating implementation options, we first created the platform architecture using the Attribute-Driven (ADD) method [145]. ADD allows determining a software architecture in which the design process is based on the software's quality attribute requirements (see Figure 9.2). ADD follows a recursive design strategy that decomposes a system (or system element) by applying architectural tactics and patterns that satisfy its driving requirements and quality attributes.

The ADD inputs are three: functional requirements, design constraints, and quality attribute requirements. With respect to the first, functional requirements specify the functionalities that the system must provide. In turn, the second input describes the

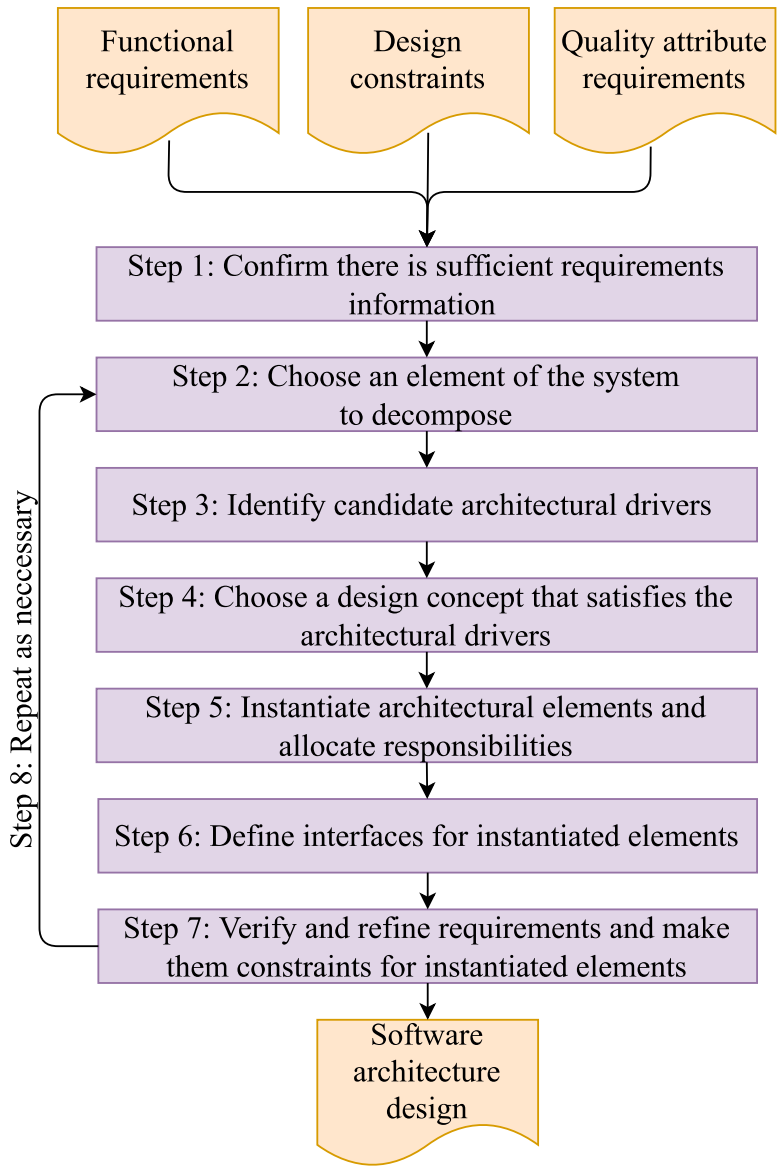


Figure 9.2: The ADD method

decisions that must be incorporated into the final version of the system design. Finally, the third input is related to requirements that indicate the degree to which a system must satisfy its properties. Subsequently, the output of ADD is a system design in terms of the roles, responsibilities, properties, and relationships among software elements.

In the following points, we briefly introduce the steps of ADD:

- *Step 1*: In this step, it should be confirmed if the information available to implement ADD is sufficient; i.e., it is necessary to have the information regarding the prioritization of requirements done by stakeholders.
- *Step 2*: This step focuses on the selection of which elements of the system will be addressed by the ADD method.
- *Step 3*: In this step, the stakeholders' requirements are ranked by priority and

impact on the architecture.

- *Step 4*: The objective of this step is to select the most important elements that will appear in the architecture and the type of relationship between them. This step, in turn, is subdivided into specific activities that allow the architectural components and their respective relationships to be determined. This activities are: (Step 4.1) identify design concerns, (Step 4.2) list alternative patterns for subordinate concerns, (Step 4.3) select patterns, (Step 4.4) determine relationship between patterns and drivers, (Step 4.5) capture preliminary architectural views, and (Step 4.6) evaluate and resolve inconsistencies.

- *Step 5*: In this step, several types of software elements selected in the previous steps are instantiated. The instantiated elements are assigned responsibilities according to their type.

- *Step 6*: In this step, the services and properties required by the design software elements are defined.

- *Step 7*: This step verifies that the decomposition of the elements satisfies the functional, design constrains and quality attributes requirements.

- *Step 8*: Once the previous steps have been completed, return to step 2 to continue breaking down the rest of the elements of the architectural design.

A complete practical example of the ADD method can be consulted in [147].

Implementing the ADD method

In the following sections, we detailed the main results obtained in each step of ADD.

Step 1: Requirement information In this step, we conducted an AS-IS TO-BE analysis in order to obtain an initial diagnosis concerning the current and desired scenario of the laboratory that allows analyzing different points of view and functional

requirements. The essence of this analysis is based on feedback from project stakeholders since they provide relevant information to determine the most appropriate architecture. Table A.1 describes the main NFRs of the platform. Furthermore, we identified the following main design constraints:

- The platform should be designed based on microservices.
- A middleware layer should be used for the integration of software, devices, API management, and event processing.
- The platform must use the publish/subscribe messaging protocol.
- Asynchronous and synchronous events-driven communication among services.
- Independent databases in each microservice.
- It is required to monitor services performance and APIs.

Table 9.5: IoMT platform NFRs

ID	QA	NFR description
NFR1	*Availability*	The services must be highly available in order to be able to access them when real-time information about the patient's movements is needed. Also, they must be available when there are modifications to the sensors. These modifications depend on the needs of the patient and/or geographic environment.
NFR2	*Scalability*	The platform must adapt to the changes that patients need without losing quality. This is the reason why each service must have the ability to grow and handle large volumes of information.
NFR3	*Interoperability*	The platform must have the ability to allow new software and/or tools to be incorporated quickly and efficiently. Furthermore, they must have the ability to exchange information and at the same time, use the information that has been incorporated.
NFR4	*Confidentiality*	Since the platform will use data related to patients, there are security and privacy regulations that the platform must satisfy. Besides, being an IoT platform, security mechanisms must be implemented in order to prevent attacks and threats. Furthermore, security mechanisms produce trust in elderly patients regarding the platform.

Step 2: Selection of system elements To select the system elements to be analyzed, we used the microservices that were identified along with the stakeholders (see

Table A.2). In brainstorming sessions, we identified potential microservices that could provide the architectural and monitoring capabilities to elderly patients demanded by stakeholders and patients. In this regard, we used the guidelines of Domain-Drive Design [51] to identify microservices based on the definition of limited contexts of each functionality and the modeling of the domain.

Table 9.6: IoMT platform microservices

Microservices	Description
Historical events	This service intends to provide information and analysis regarding accidents or incidents in order to discover all the history of an accident so that it can be avoided. Furthermore, the idea of this service is to identify improvements aimed at preventing or mitigating possible accidents in elderly patients. This service also allows the identification of hazards in order to be evaluated and classified.
Alert service	This service aims to generate the corresponding alerts to warn the medical staff, family, or other interested parties. This service should eventually interoperate with other systems.
Devices management	This service processes all the information captured by sensors in the room or house. In addition, this service is not limited to sensors; it can be extended to other devices.
Tracking service	The primary purpose of this service is to provide information to investigate the activities of elderly patients in the rooms.
Patient management	This service manages the different profiles of elderly patients. It uses information from other health institutions that allow it to characterize the patient and his health conditions (for example, patients who, due to health conditions, regularly attend the bathroom).
Middleware services	These services usually provide an abstraction layer for devices. Thus, they are able to ensure data transfer between such services and therefore achieve interoperability.

Step 3: Architectural drivers identification The decision about the software architecture lies in the new drivers that stakeholders require for the platform.

Architectural capabilities Due to the platform will be based on IoMT, this concept points to the collection of medical devices and applications that connect to healthcare systems through networks [88]. Therefore, the main capabilities that the platform must have, from the point of view of IoMT, are the following:

- *Provide real-time data to monitor the patient's health status*: This capacity is relevant since the data that the platform will receive comes from patients. This

means that this real-time communication must ensure that the integrity of the data is accurate. Furthermore, security and privacy aspects must also be taken into account so that the patient can trust on the devices' functionalities.

- *Integration with other devices*: The current situation of the laboratory describes that there is previous work with the use of sensors. The sensors can send data from the environment so that they can be studied and apply innovative techniques to prevent and improve patient care. Yet, IoMT also extends the capacity of platforms to the integration of data from medical devices and portable devices using appropriated middleware services.

- *Alert family members or medical staff when there are abnormalities in the patient's daily behavior*: The platform must have the ability to integrate all the necessary services in order to alert and inform family members or medical staff when in the event of life-threatening circumstances. For this, procedures for checking the status of services in small time windows (time to be defined) must be applied in order to monitor the current status of the services.

Network capabilities The access protocol management module implements an interface for the handling of the different communication protocols (such as IEEE 802.15.x and 802.11).

The connectivity protocol module will use lightweight published/subscribed messaging transport. Publish/subscribe messaging, or pub/sub messaging, is a method of asynchronous service-to-service communication used in serverless and microservices architectures. In a pub/sub model, any message published to a topic is immediately received by all of the subscribers to the topic [60]. Pub/sub messaging can be used to decouple applications in order to increase performance, reliability, and scalability.

Step 4: Choose design concepts This step is the most important in the ADD method because it describes most of the design alternatives, architectural patterns and archiectural tactics selected, the evaluations made to validate the designs, and the

design changes made (detected from deficiencies) that affect the final design.

Step 4.1 requires that design concerns of the desired architecture be identified in order to associate them with architectural tactics and architectural patterns. To do this, we gathered a set of architectural tactics and architectural patterns that have been referenced in the microservices and IoT literature. In this step, we characterized eight design concerns in total, which in turn are associated with a certain amount of architectural tactics and architectural patterns.

In step 4.2, we analyze each detected architectural pattern and complete the description by adding an extension based on discriminating parameters. These parameters allow us to characterize each architectural pattern based on variables that allow measuring responses.

The main objective of step 4.3 is to select the architectural patterns that will be used in the architectural design and to support the selection. For the design of the platform, we selected thirteen architectural patterns.

Later, in step 4.4, we associate the selected architectural patterns to the architectural drivers of the project in order to illustrate the first architectural views of the platform (step 4.5). In this context, stakeholders were only interested in the conceptual view of the architecture. We concluded step 4 with the evaluation of inconsistencies (step 4.6).

Steps 5 and: Architectural elements, responsibilities and interfaces Regarding step 5, we used the selected architectural patterns in step 4 to define the responsibilities between architectural elements.

Concerning step 6, we linked the design components of the microservices architecture following the notation proposed by C. Richardson [119]. We have created a template that details the component name, description and dependencies. We defined dependencies as "invoke" (the operations, which are implemented by other services that this service invokes) and "subscribe" (the messages, which includes events, that this service subscribes to). For example, Table A.5 illustrates the description of the microservice

"Device management".

Table 9.7: Devices management microservice documentation

Name	*Devices management*
Description	See Table A.2
Dependencies	
Invokes	Subscribe to
▪ Middleware service	▪ Historical events ▪ Alert service ▪ Tracking system

Step 7: Refinement In this step, we verified that elements decomposition meets functional requirements, quality attribute requirements, and design constraints. We repeat this step for each microservice defined in the architecture design.

Evaluating implementation alternatives

Once the design of the microservices-based IoMT platform is finalized and approved, we proceed to use μAzimut to generate implementation alternatives for the platform (see Figure 9.3).

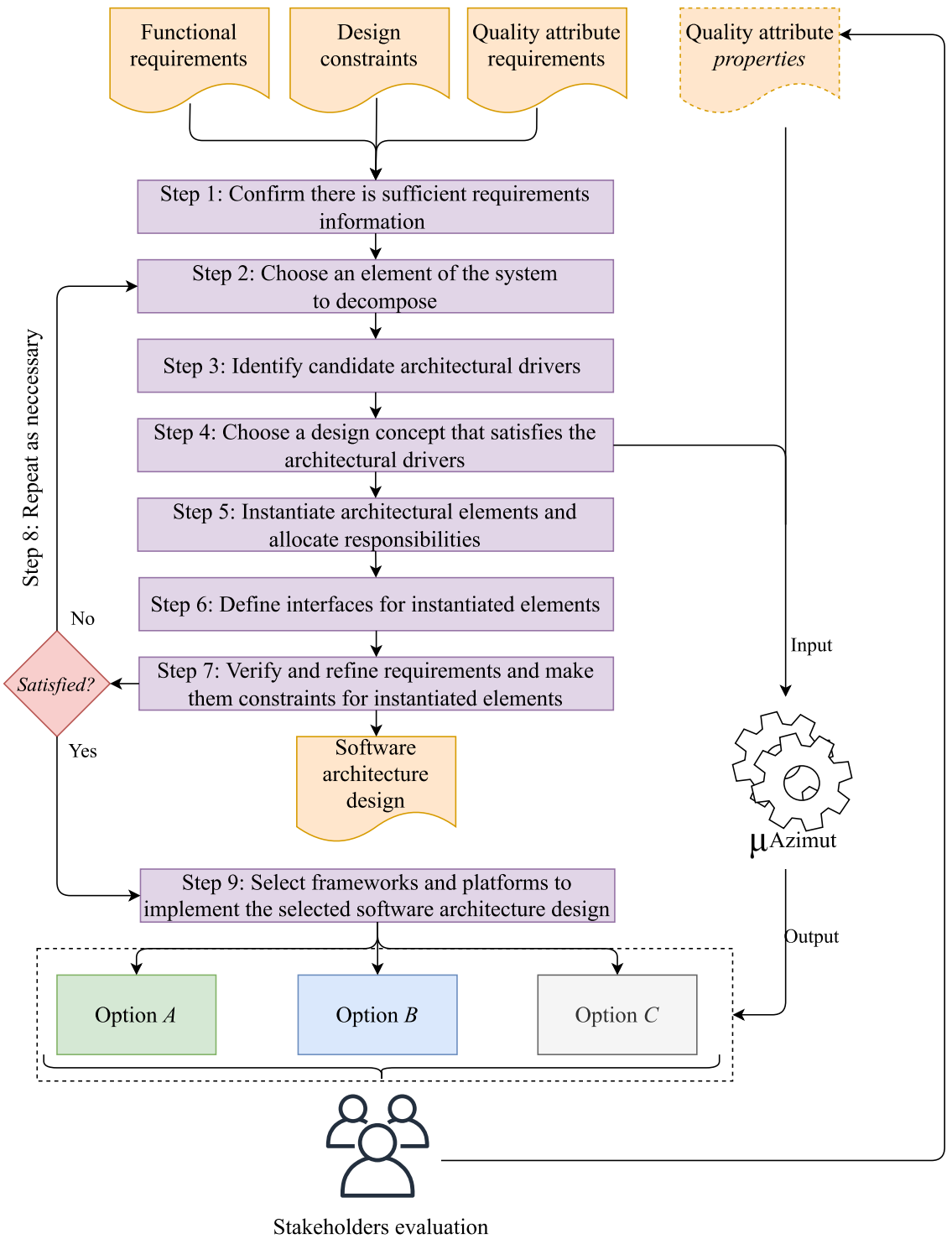


Figure 9.3: Extension of the ADD method to evaluate implementation designs with μAzimut

We presented to the stakeholders all properties that μAzimut has for the NFRs of the study. As the TaSPeR technique suggests, we define time intervals for analyzing and evaluating properties. In 30-minute sessions, stakeholders discussed, prioritized, and selected the properties of each NFR that the AAL system should meet. The project architect assumed the role of the moderator in each session. Accordingly, Table 9.8 summarizes the selected properties.

Once the properties are selected and prioritized, we executed μAzimut. For each NFR,

Table 9.8: Key properties selected by stakeholders

Id	Property	NFR	Summary of stakeholders' rationale
PP1	*Quick broken state recovery*	NFR1	Since the platform depends mainly on the operation of the devices that capture the information of the patient, for the stakeholders, it is of the utmost importance that services related to obtaining data of the devices have a fast recovery in case of failures.
PP2	*High control of failure propagation*	NFR1	The platform manages the care of elderly patients. Any scenario where services fail, compromises the patient's life. This is why this property is critical in order to have contingency plans to recover from failures in critical services.
PP3	*High service monitoring visibility*	NFR1	Property of high interest to stakeholders as they want to know the status of the service at all times. Furthermore, as the platform is intended to use devices and sensors in more homes, stakeholders need to control the status of each service.
PP4	*Periodic heartbeat signal*	NFR1	This property supports the previous property. Heartbeat allows checking the status of each service periodically.
PP5	*Effective technical duplication*	NFR2	Stakeholders are clear in requesting that the platform be scalable in the sense that if the devices increase, the capacity must be maintained. To perform this, load balancing strategies must be applied.
PP6	*High level of individuals, groups, or systems authorization*	NFR4	The platform requires explicit authorization from consumers for the execution of specific tasks. In turn, it is necessary to control users to grant them limited access to the platform resources without having to expose their credentials.
PP7	*High cooperation among components*	NFR3	The platform demands the capability of exchange information among services and devices (such as sensors) to use data that has been exchanged for research and patient monitoring purposes.
PP8	*High coordinated orchestration among components*	NFR3	This property points to a control mechanism to coordinate, manage and sequence the invocation of particular components (which could be ignorant of each other).

μAzimut generates framework assemblies ranked by the support score. The project architect then takes the μAzimut outputs and makes high-level diagrams representing implementation options built on the support score values. For this case study, we decided to create three options (option A, B, and C) because the time provided for implementation analysis is limited, i.e., we cannot analyze all possible options generated by μAzimut. Option A is built with the framework assemblies with the highest support score for each NFR. Subsequently, options B and C are built with the framework assemblies whose support scores correspond to the second and third place in the ranking, respectively.

The following day, we gathered the stakeholders to evaluate the options created by the architect. For each option, the stakeholders will have as an evaluation guideline the priorities they evaluated and prioritized. At the same time, the architect describes each option mentioning the advantages and disadvantages of using each framework in the project. For example, suppose a selected assembly corresponds to [Apache Zookeeper, RabbitMQ]. Subsequently, the following advantages and disadvantages are mentioned

to stakeholders: *Apache Zookeeper has good session timeout support. However, deploying Apache Zookeeper services requires a high consumption of resources (hardware/software). On the other hand, RabbitMQ facilitates the implementation of asynchronous communication between microservices, but if the connection is lost, RabbitMQ can affect the number of long-lived connections.* Therefore, taking into consideration the advantages and disadvantages of each framework, stakeholders can more easily contrast the properties with the objectives they wish to satisfy in the AAL system.

In the following points, we summarize the stakeholders' decisions for each option. We also mentioned the positive and/or negative criteria and relevant frameworks of each decision.

- **Option *A***: There was a unanimous agreement. For example, of all the frameworks obtained, stakeholders liked the idea of having Netflix Zuul (as API Gateway) and Netflix Eureka (to keep track of the services). Furthermore, stakeholders appreciate that in this first draft, there is an interoperability layer that manages both the sensors and the data they transmit.

- **Option *B***: At the beginning, the stakeholders were not agree regarding the acceptance of the implementation design. The combination of frameworks seemed to satisfy, for the most part, the main concerns of stakeholders, especially with regard to Availability. Nevertheless, the draft was rejected because they researched RabbitMQ a bit more thoroughly and realized that the issues, reported by developers mostly in GitHub[3], corresponded to many bugs.

- **Option *C***: Stakeholders thought it was a good idea to use Netflix Eureka and Netflix Hystrix, but they didn't accept Redis as a database. Redis is an in-memory database engine, based on storage in hash tables but which can optionally be used as a durable or persistent database. Although these properties can be a great advantage, stakeholders realized that this framework has many issues in open microservices projects, so they decided to reject the third option.

[3]https://github.com

Results

Table 9.9, as well as Figure A.1, describes the frameworks selected and the overview of the AAL system, respectively.

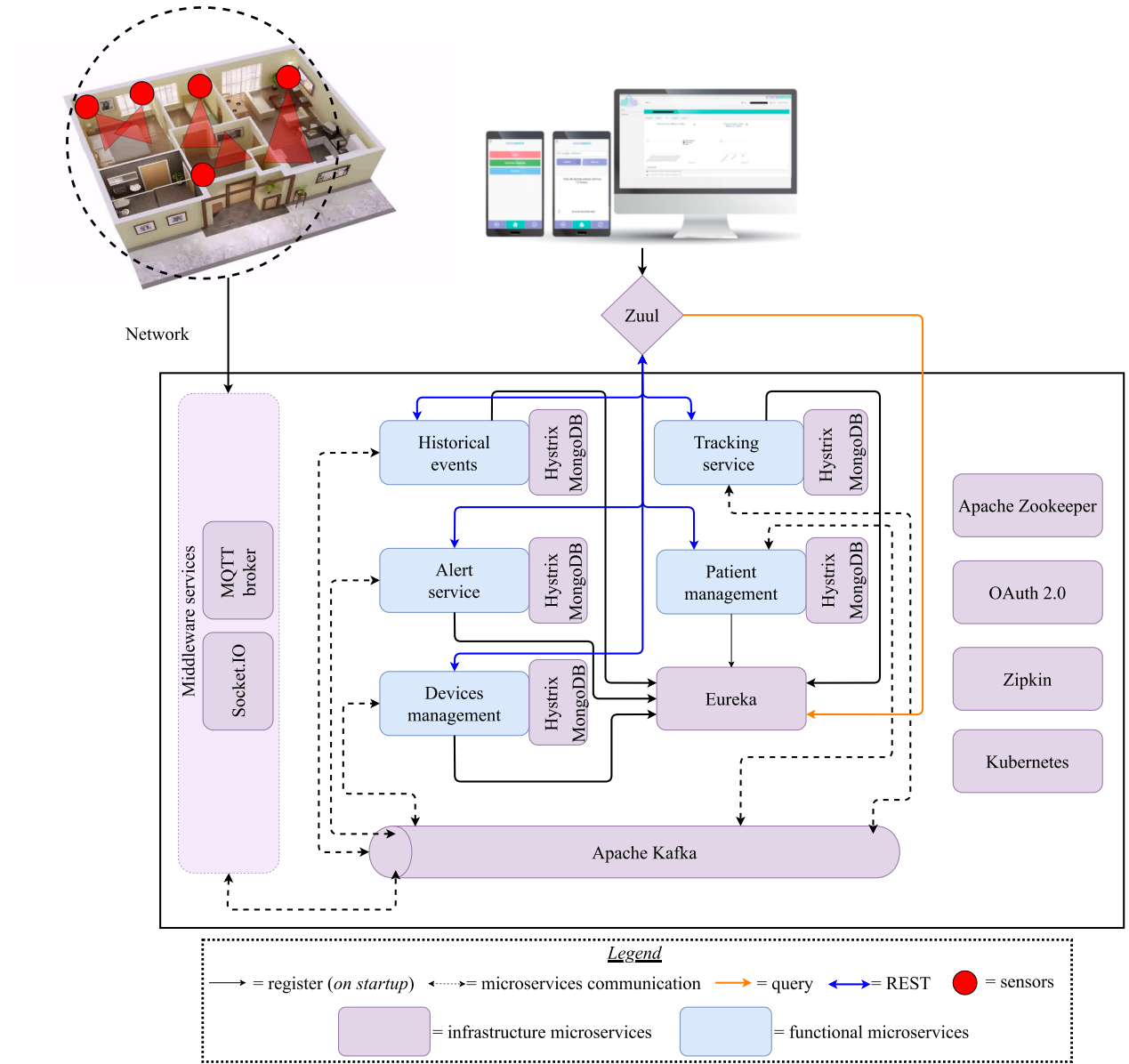


Figure 9.4: The AAL system architecture

The information illustrated in Table 9.9 along with the architectural knowledge generated by μAzimut to obtain frameworks assemblies is significant to document architectural decisions regarding the maintainability of the AAL system. For example, let's consider one of the main functionalities of the AAL system, which is the real-time monitoring of elderly patients (see Figure 9.5).

In order for stakeholders to visualize the output described in Figure 9.5, one of the main concerns is that the data are highly available because it uses external devices (sensors). The output of the Figure 9.5 is produced through the cooperation of two

Table 9.9: Frameworks and platforms selected for the case study

Name	URL	Description	NFR	Property
Netflix Eureka	https://github.com/Netflix/eureka	Service registry for resilient mid-tier load balancing and failover.	NFR1, NFR2	PP2, PP5
Netflix Zuul	https://github.com/Netflix/zuul	Gateway service that provides dynamic routing, monitoring, resiliency, security, and other features.	NFR1, NFR2	PP3, PP5
Netflix Hystrix	https://github.com/Netflix/Hystrix	Latency and fault tolerance library designed to isolate points of access to distributed systems.	NFR1	PP2
Apache Kafka	https://kafka.apache.org	Open-source stream-processing software platform.	NFR1, NFR3	PP4, PP7
Docker	https://www.docker.com	Automates the deployment of applications within software containers.	NFR2	PP5
Apache Zookeeper	https://zookeeper.apache.org	Server which enables highly reliable distributed coordination.	NFR3	PP8
Socket.IO	https://socket.io	Bi-directional communication library between web clients and servers.	NFR2	PP5
OAuth 2.0	https://oauth.net/2/	The industry-standard protocol for authorization.	NFR4	PP6
Zipkin	https://zipkin.io	Distributed tracing system. It helps gather timing data needed to troubleshoot latency problems.	NFR1	PP4
Kubernetes	https://kubernetes.io	System for automating deployment, scaling, and management of containerized application.	NFR2	PP5
MongoDB	https://www.mongodb.com	Database that provides high performance, high availability, and automatic scaling.	NFR1, NFR2	PP4, PP5
Mosquitto	https://mosquitto.org	Message broker that implements the MQTT protocol versions 5.0, 3.1.1 and 3.1	NFR2, NFR3	PP5, PP7

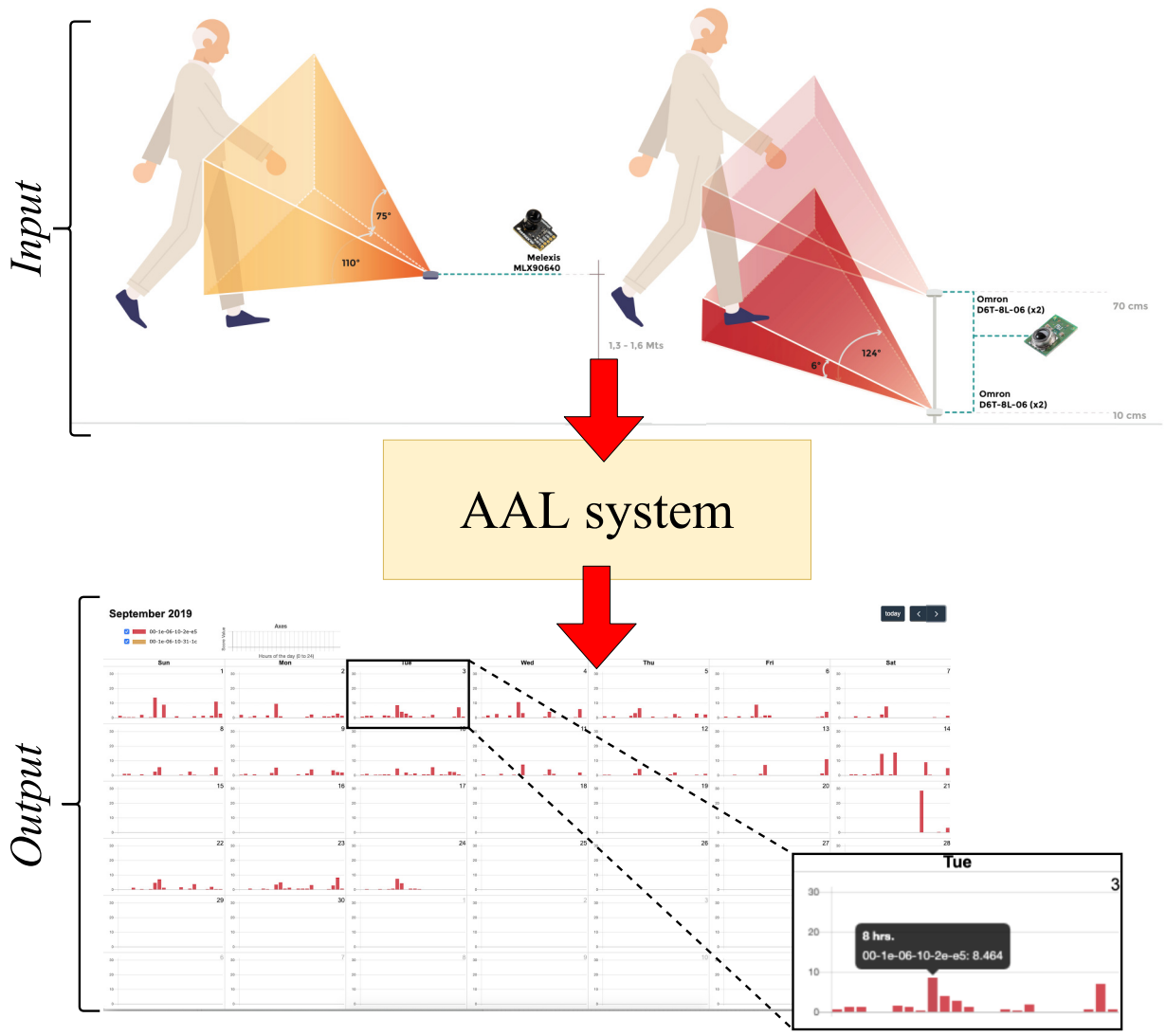


Figure 9.5: Real-time patient monitoring

microservices, “Device management” and “Historical events”.

Mosquitto, along with Socket.IO, captures data from 1D and 2D sensors and sends data to the platform via Apache Kafka. This platform provides periodic heartbeats to examine the status of microservices and, at the same time, promotes high cooperation between them through data pipelines. Regarding the 2D sensor, the data capture is quite clear. On the other hand, for 1D capture, it is necessary to use 4 Omron D6T-8L-06 sensors per room, in order to capture most of a body. The captured data is then stored in structured files as 32×24 matrices (2D sensor) and 1×33 arrays (4 1D sensor measurement + timestamp), and then uploaded to a server.

Because the process of monitoring patients occurs in real-time, it is significant to monitor the status of the microservices. In this regard, Netflix Eureka allows check that “Device Management” and “Historical events” are active and recorded (see Figure 9.6) to be used. In turn, Netflix Hystrix makes it possible to monitor the health status

of the microservices and the number of requests that the API is receiving in real-time.

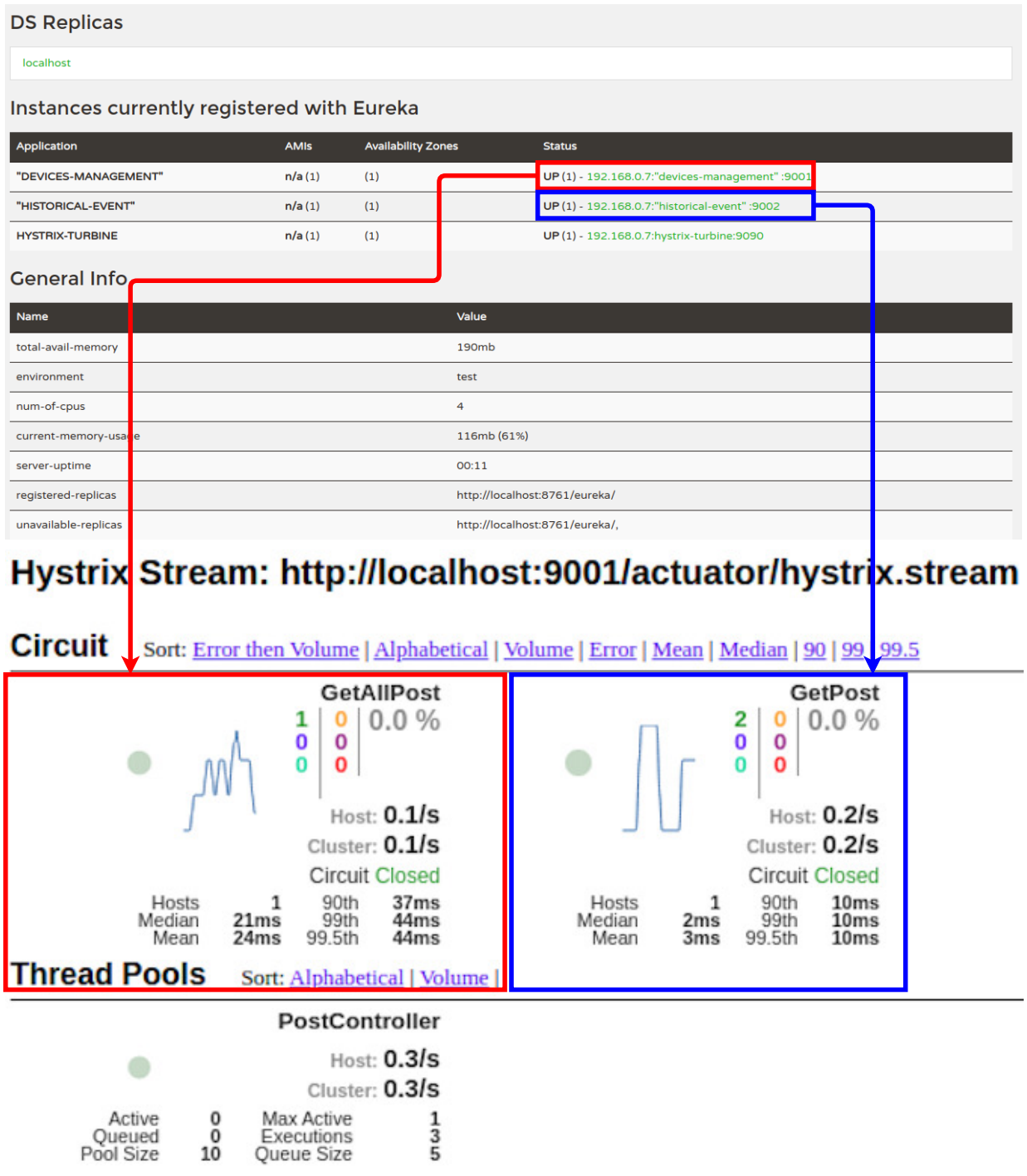


Figure 9.6: Microservices status monitoring. Red line and box correspond to the Device management microservice and blue ones to Historical events microservice

Subsequently, the "Historical events" microservice uses the data captured by the "Devices management" microservice and execute algorithms to identify behavior patterns of people within a room or home through the calculation of *presence scores* using Knowledge Discovery in Databases (KDD) [52].

This example illustrates how the cooperation of different frameworks makes it possible to meet stakeholders' needs. At the same time, this cooperation is supported by different architecture decisions generated by μAzimut that allow the maintainability of the

system to be more straightforward and more documented.

Discussion

Regarding the research question of the case study, results suggest that μAzimut helps reduce the difficulty of evaluating frameworks to stakeholders. Stakeholders agreed that the technique not only helps to reduce the space of solutions that must be analyzed to satisfy NFRs but also allows people, who are not familiar with technologies, frameworks or software architectures, to infer and visualize potential advantages and disadvantages of a certain architecture to satisfy specific requirements. In critical systems, like the AAL system under study, the success or failure of NFRs satisfaction can compromise not only the system itself but also people's lives. In this case study, for example, stakeholders were emphatic in highlighting that Availability is a critical attribute for them because if sensor data is not available at all times, it can compromise a patient's life. Likewise, stakeholders also highlighted that μAzimut is highly useful because, instead of selecting frameworks by "trial and error", μAzimut generates assembled frameworks that are supported by architectural constructs (μAzimut components), which allows trusting the results of the technique.

Our perception suggest that μAzimut is useful for analyzing emerging stakeholder architectures at early stages of software development, as long as the stakeholders have knowledge of software development. In the case study, the stakeholders discussed the advantages and disadvantages of each framework with enough knowledge in software development and programming. They even used GitHub to support their opinions. The final impressions of the case study point out that in order to evaluate architectural designs based on frameworks, the stakeholders must have previous experience in software development.

Another interesting aspect to highlight is the prioritization of properties. What difference would it have made if an architect with expertise in microservices and AAL systems had prioritized properties? A probable answer to that question would be "none".

Nevertheless, the significant difference we detected after analyzing the results and interviewing the stakeholders points to the *understanding* of the stakeholders' views. A collaborative selection among stakeholders helps the architect to understand better how different stakeholders *observe* the system. Therefore, considering stakeholder opinions allows the architect to make architectural decisions by judging and balancing different views of the system.

9.3. Experimental study

This section describes the experimental study that allows evaluating the use of microservices tactics as an intermediate layer to select frameworks assemblies that would enable satisfying one or several NFRs. The activities conducted in the experimental study are described in Figure 9.7.

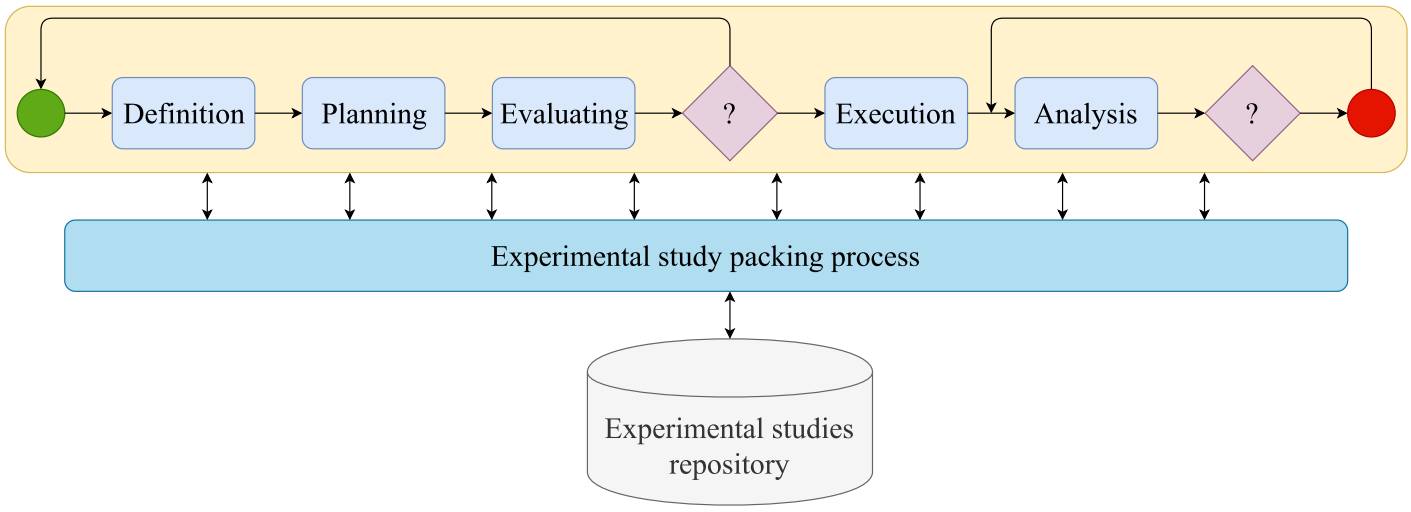


Figure 9.7: Experimental process executed in this study [135]

In this experiment, we focused on the implementation of two availability NFRs in two microservices-based systems. We selected NFRs related to availability because this quality attribute represents one of the properties that characterize microservices architectures [12]. The research of high availability in microservices is related to propose methods and techniques that enable to address fault tolerance through the use of technologies based on containers, APIs, load balancers, and others [120].

9.3.1. Experimental cases

The case studies used in this experimental study are two: an IoT open platform based on a microservices architecture and Salary Payment System of a forestry company.

Sitewhere

SiteWhere is an open-source application enablement platform for building both the infrastructure and applications that make up the Internet of Things. It covers many of the use cases required to manage large IoT deployments and is built with a framework approach that facilitates the addition of new concepts easily. The platform has been designed for reliable, high-throughput, low-latency processing, and dynamic scalability.

SiteWhere separates the many aspects of IoT processing into microservices, each specializing in a specific task. These include functionality such as event ingestion, big-data event persistence, device state management, large-scale command delivery, integration of device data with external systems, and much more. Each microservice is a Spring Boot application wrapped as a Docker container.

Salary Payment system

Currently, small and medium enterprises have a critical process within their organization, which is the payment of salaries. In general, companies have their own information systems that allow them to calculate salaries according to the company's needs and structure. Nevertheless, these systems are expensive, too complex, have too many options that hinder the work, or have a very limited performance that does not cover all the company's needs. For this reason, the ABC company required a Salary Payment System that allows it to facilitate the work and minimize the process times for the calculation of remuneration. Furthermore, this system must communicate with other systems of other companies and government entities.

9.3.2. Ground truth

To compare the answers of the experiment, it is necessary to execute a rigorous process that allows (i) to understand the microservices architectures of each system used in the experiment and (ii) to analyze the objective of each framework in the microservices-based system. This process is necessary because it facilitates the design, development and deployment of the new availability-related NFRs in each system to be used in this experiment. Thus, we can obtain and use a ground truth of frameworks to compare the responses of the subjects.

Therefore, we used the proposal of García *et al.* [59], which is a framework to obtain the ground truths of software architectures. This framework allows recovering software architectures through the application of architecture recovery techniques, system domain analysis, utility component identification, among other procedures (see Table 9.10).

Once we executed the ground truth framework, we validated the recovered microservices architectures as well as the implementation of the availability-related NFRs by contacting directly to the architects and developers of the microservices-based systems, either by email or in personal interviews and working sessions.

Sitewhere Ground Truth

Regarding the Sitewhere project, one of the significant advantages of open-source projects is they have adequate documentation that grants understanding and executing a particular project in a desired machine and environment. Although not all projects have complete documentation, platforms such as Github allow visualizing those projects that do comply with basic documentation regulations.

Sitewhere has a complete description and documentation, which allows not only to understand the system fully but also to use and expand it. However, there are steps in the recovery process that were not trivial. At this point, steps 2 and 4 required more work. Since microservice architectures is an emerging topic, there are proposals that

Table 9.10: Architectural recovery steps

Steps	Description
Gather domain and application information	The first step is for a recoverer to utilize any available documentation to determine domain or application information that can be used to produce domain or application principles.
Select a generic recovery technique	The recoverers can select an existing recovery technique to aid in the production of an authoritative recovery. The use of a recovery technique injects generic principles into the recovery. For example, many recovery techniques will use different metrics or rules to represent generic principles, such as coupling, cohesion, or separation of concerns.
Extract relevant implementation-level information	Architectural recovery often involves automatic extraction of implementation-level information. The information extracted will depend on the recovery technique selected by the recoverer and the available tool support. This extracted information typically includes at least the structural dependencies between the code-level entities.
Apply generic recovery technique	At this point, the recoverers can apply their chosen generic recovery technique to obtain an initial recovery. This step is a straightforward application of the recovery technique since more specific information will be injected through domain, application, and system context principles in later steps.
Utilize domain and application principles	Any mapping principles determined in Step 1 can be used to modify the recovery obtained at the previous step. In particular, any groupings obtained should be modified, if possible, to resemble expected components or connectors.
Identify utility components	The recoverer should identify any utility components (e.g., libraries, middleware packages, application framework classes).
Pass authoritative recovery to certifier	At this point, the recoverer has produced an authoritative recovery that has been enriched and modified with the help of different kinds of mapping principles. Key to the authoritative recovery is the clear relationship between the proposed architectural elements and the code-level elements. The recovered architecture is only now passed to a certifier for verification.
Modify groupings according to certifier' s suggestions	At this point, the recoverer modifies the groupings as suggested by the certifier.

suggest different types of recovery techniques and views of microservices-based systems [96] [48] [7] [18] [80] [64]. However, most of the proposals were made for specific case studies. Very few proposals warrant the replicability of recovery techniques or methods. Moreover, if replicability is possible, it is partial.

Therefore, we used Pekenum (described in Chapter 4) to retrieve and understand the Sitewhere architecture.

Salary Payment System Ground Truth

Concerning the Salary Payment System, the system was developed two years ago using the microservices architecture style. Currently, the system's stakeholders are satisfied with its performance. However, during the last months, the company has performed new businesses, which implies an abrupt growth of new contracts, new forestry business entities, and a salary payment process reform promoted by the local government. Under these new scenarios, the current Payment System must be updated to meet new requirements demanded by stakeholders.

For this system, we used Pekenum as well as the tool *Understand*[4]. This tool is a customizable integrated development environment (IDE) that enables static code analysis through an array of visuals, documentation, and metric tools. It was built to help software developers understand, maintain, and document their source code. It enables code comprehension by providing flow charts of relationships and building a dictionary of variables and procedures from a provided source code.

A differential aspect of this case is some orthogonal variables (such as time, cost, government, among others) greatly influence new requirements. Therefore, the role of the principal architect in the architectural recovery process is relevant.

9.3.3. Definition

This activity intends to define the experimental study's goal. Intuitively, architects select frameworks in order to solve some kind of recurring architectural problem that arises in the software development process. In this regard, architectural patterns address these problems since they provide systematic solutions that seek to resolve these recurring problems.

Nevertheless, even though frameworks fully or partially implement one or more architectural patterns, not all frameworks successfully address NFRs. For example, in the

[4]https://scitools.com

web development context, a recurring problem that developers face is how to separate data from (i) an application, (ii) the user interface, and (iii) the control logic. In this situation, the Model-View-Controller (MVC) pattern appears as a solution; yet, there is a variety of more than 60 frameworks that partially or totally implement the MVC pattern. So, the question at this point is which MVC framework is best suited to satisfy an NFR.

This experiment aims to evaluate architectural tactics as an intermediate layer to reduce the space of solutions that an architect must analyze in order to select the most appropriate framework or framework assembly to satisfy NFRs. In the context of μAzimut, our perception is that microservices tactics reduce the complexity of selecting frameworks to satisfy NFRs in microservices-based systems instead of selecting frameworks based on which pattern they implement. Therefore, this experiment aims to evaluate the precsion, recall and efficiency of frameworks assemblies using and not using microservices tactics as *decision component* with respect to frameworks assemblies selected by the ground truth experts, which they used to implement the two availability-related NFRs.

Subsequently, the research question of this experiment are

RQ4.3
Does the component of microservices tactics produce framework assemblies closer to the solution described in the ground truth?

This RQ attempts to answer whether there is any difference in precision and recall in the frameworks assemblies generated by μAzimut using and not using microservices tactics as an decision component.

RQ4.4
Which are the reasons (regarding the use and not use of the component of microservices tactics) reported by subjects to select frameworks?

This RQ intends to describe the reasoning described by the subjects for selecting the

framework assembly generated with and without microservices tactics as decision component. More precisely, we want to analyse if there are differences in the judgment of the subjects with respect to analysing a set of solutions generated with microservices tactics and another set without microservices tactics.

RQ4.5
Does the component of microservices tactics cause more efficient frameworks assemblies?

To answer this research question, we considered in the subjects' answers two scenarios to determine efficiency. The first one consists of the subject selecting the frameworks that she or he wants to address the NFRs of the experiment. The second scenario considers that there may be combinations of frameworks that can replace another framework. We include this second scenario because in practical situations, the desired framework cannot always be used for several reasons, such as infrastructure restrictions, time restrictions and learning curves, insufficient knowledge of the framework, among others. For example, if the architect wants to use the Zipkin's[5] capabilities to track information from services for performance, monitoring and alerting, but for some reason, she or he cannot use the platform, the architect can eventually combine Netflix Eureka and Netflix Hystrix to supply some of Zipkin's desirable functionalities.

Therefore, we define a variable F (frameworks) where the position of each of its elements corresponds to the selection or not of each of the n eligible frameworks selected by the ground truth expert. We define F as follow:

$$F = \begin{cases} 1 & \text{if } f_i \\ 0 & \text{if } \overline{f_i} \end{cases} \tag{9.1}$$

where $f_i\,(i = 1, \ldots, n)$ is the selection of frameworks generated by the expert, and $\overline{f_i}$ is the non-selection of frameworks generated by the same expert.

[5]https://github.com/openzipkin/zipkin

Subsequently, let F_c a variable where the position of each of its elements corresponds to the possible combinations of frameworks that replace or not the frameworks selected by the expert. We define F_c as follow:

$$F_c = \begin{cases} 1 & \text{if } f_j f_k \\ 0 & \text{if } \overline{f_j f_k} \end{cases} \tag{9.2}$$

where $f_j f_k$ is the combination between the framework f_j and framework f_k that replace the framework f_i defined by the ground truth expert[6]; therefore, $f_j f_k \equiv f_i$. The combinations that do not replace the selected frameworks by the expert are represented by $\overline{f_j f_k}$. The number of q combinations of framework pairs ($q = 2$) from the total of F's n frameworks is given by:

$$n_{f_c}(n, q = 2) = \frac{n!}{2!\,(n-2)!} \tag{9.3}$$

An *efficient* framework assembly is composed of a set of frameworks selected by the ground truth expert. Each framework that structures this assembly will be assigned a weighted value (denoted by $w_i \in W$) between zero and one. Each weighted value w_i is related to the importance of each of the frameworks that represent the ground truth frameworks assembly. Therefore, the framework that has the highest importance is a relevant framework for satisfying the NFR. The sum of all weighted values w_i assigned by the expert is equal to the unit ($\sum_{i=1}^{n} w_i = 1$). The following expression represents the weighted values:

$$W(F) = \begin{cases} w_i > 0 & \text{if } f_i = 1 \\ w_i = 0 & \text{if } f_i = 0 \end{cases} \tag{9.4}$$

where $W(F)$ is a function whose domain (set of input values) are the values of the

[6]For the simplicity of this experiment, we established that the maximum combination of frameworks that can replace another framework is 2.

variable F.

For the assignment of weighted values to the combinations of pairs frameworks, we adopted the following criterion: if there is a combination of frameworks $f_j f_k \in F_c$ similar to $f_i \in F$, it will be assigned the weighted value w_i (associated to f_i in the $W(F)$ function) multiplied by a penalty factor P. Additionally, for the assignment of the weighted values w_c to the possible combinations of $f_j f_k$, the framework f_i cannot be included in the framework's assembly selected in the ground truth framework assembly. The following expression represents the restriction criterion:

$$W(F_c) = \begin{cases} w_c = w_i \cdot P & \text{if } f_c = 1;\ f_j f_k \equiv f_i = 0 \\ w_c = 0 & \text{if } f_c = 0 \end{cases} \tag{9.5}$$

where the penalty factor P is the ratio between the number of criteria n_c^s that achieves the combination of framework pairs $f_j f_k$ and the total number of criteria n_c^t supported by the framework f_i. Therefore

$$P = \frac{n_c^s}{n_c^t} \tag{9.6}$$

Finally, to establish a metric to measure the frameworks selected by the subjects quantitatively, we defined the following efficiency equation:

$$E = \left(\sum_{i=1}^{n} W(F) + \sum_{c=1}^{n_{fc}} W(F_c) \right) \cdot 100\% \tag{9.7}$$

For example, let us suppose two subjects (S_a and S_b) must select frameworks from a set of twelve frameworks (see Table 9.11) to satisfy an NFR. At the same time, the frameworks assembly created by the expert as well as the respective weights (w_i) is the following: $A = [F2\,(0.5)\,, F5(0.3), F8(0.1), F12(0.1)]$. Furthermore, the combination $F1F3$ can replace two of the three main functionalities of $F2$ as well as the combination $F10F11$ can replace two of the three main functionalities of $F12$. The calculation of

the efficiency of both subjects is described in Table 9.11.

Table 9.11: Example of efficiency calculation

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
S_a		x			x							
S_b	x	x								x	x	

↓

	$w(F2)$	$w(F5)$	$w(F8)$	$w(F12)$	$\sum W(F)$
S_a	0.5	0.3	0	0	0.8
S_b	0.5	0	0	0	0.5

	$w_c(F1F3)$	$w_c(F10F11)$	$\sum W(F_c)$	E
S_a	0	0	0	80%
S_b	0	0.0667	0.0667	56.67%

Subject S_a selected F2 and F5; therefore, according to the equation 9.7, the efficiency of the framework assembly is 80%. On the other hand, subject S_b selected four frameworks where F2 is the only one that corresponds to the ideal assembly. Nevertheless, the set of chosen frameworks contains F10 and F11. This combination can replace F12, which implies that it can improve its efficiency. Hence, taking into consideration equations 9.5 and 9.6, the efficiency of the frameworks assembly is 56.67%.

9.3.4. Planning

The purpose of this activity is to define the steps to translate the experimental study's goal towards the formal definition of the hypothesis. To achieve this, the following activities must be performed:

Hypotheses Formulation

This activity involves the definition of the research hypothesis. We defined two kinds of hypotheses: the null and the alternative. The null hypothesis (H_0) indicates an initial statement that is based on previous analysis or specialized knowledge. On the other hand, the alternative hypothesis (H_1) is what might be true or hopes to prove that it is true. Therefore, null hypotheses of this experimental study are:

$H_{0.1}$: *Using microservices tactics as decision component does not produce differences in the space of solutions (frameworks) that the architect must evaluate to satisfy NFRs in microservices-based systems.*

$H_{0.2}$: *The use of the component of microservices tactics does not produce efficient frameworks assemblies.*

Subsequently, alternative hypotheses are:

$H_{1.1}$: *Using microservices tactics as decision component produce differences in the space of solutions (frameworks) that the architect must evaluate to satisfy NFRs in microservices-based systems.*

$H_{1.2}$: *The use of the component of microservices tactics produce efficient frameworks assemblies*

Variables Selections

Independent and dependent variables help to understand how reality works from the analysis of isolated phenomena. They allow us to reduce the complexity of what is being studied and look at simple elements that can reveal scientific knowledge. Therefore, the corresponding independent and dependent variables of the experimental study are:

- Dependent variables
  - *Precision*: Ratio between the correct predictions and the total predictions.
  - *Recall*: Ratio of the correct predictions and the total number of correct items in the set.
  - The efficiency of the framework assemblies.
- Independent variables
  - The μAzimut approach
  - Subecjt's experience
  - NFRs used in the study

In addition, we used the high-availability properties described in Table 5.4.

Subject selection

The selection of subjects is essential when conducting an experiment. The selection is closely connected to the generalization of the results from the experiment. In order to generalize the results to the desired population, the selection must be representative of that population. The selection of subjects is also called a sample from a population. In this experiment, we used simple random sampling (subjects are selected from a list of the population at random) technique to select subjects.

Subjects that participated in the experimental study belongs to the Professional Information Technology Master Program (abbreviated as MTI, Spanish acronym) corresponding to the Federico Santa María Technical University, Chile. MTI is a professional postgraduate program created in 2004 aimed at the formation of high-level human capital that wishes to increase their knowledge and problem-solving capacity by providing advanced training that prepares professionals to assume leadership positions within their organizations specialized industry field related to information technologies. The subject's profile is described below:

- The subject must have at least a bachelor's degree in disciplines related to the MTI, or of an equivalent or higher degree.
- At least three years of job experience in Information Technology.
- Demonstrate sufficient English language proficiency.

9.3.5. Experiment Design

We used the two alternatives on one experimental unit design. This design increase the accuracy of the subsequent analysis that is to be conducted on the experimental results [144]. The alternatives of the experiment correspond to using μAzimut with microservices tactics as an intermediate layer (*technique A*) and μAzimut with only microservices patterns (*technique B*) (see Figure 9.8). Each technique produces a total of 5 framework assemblies. Subjects must select only one assembly for each technique.

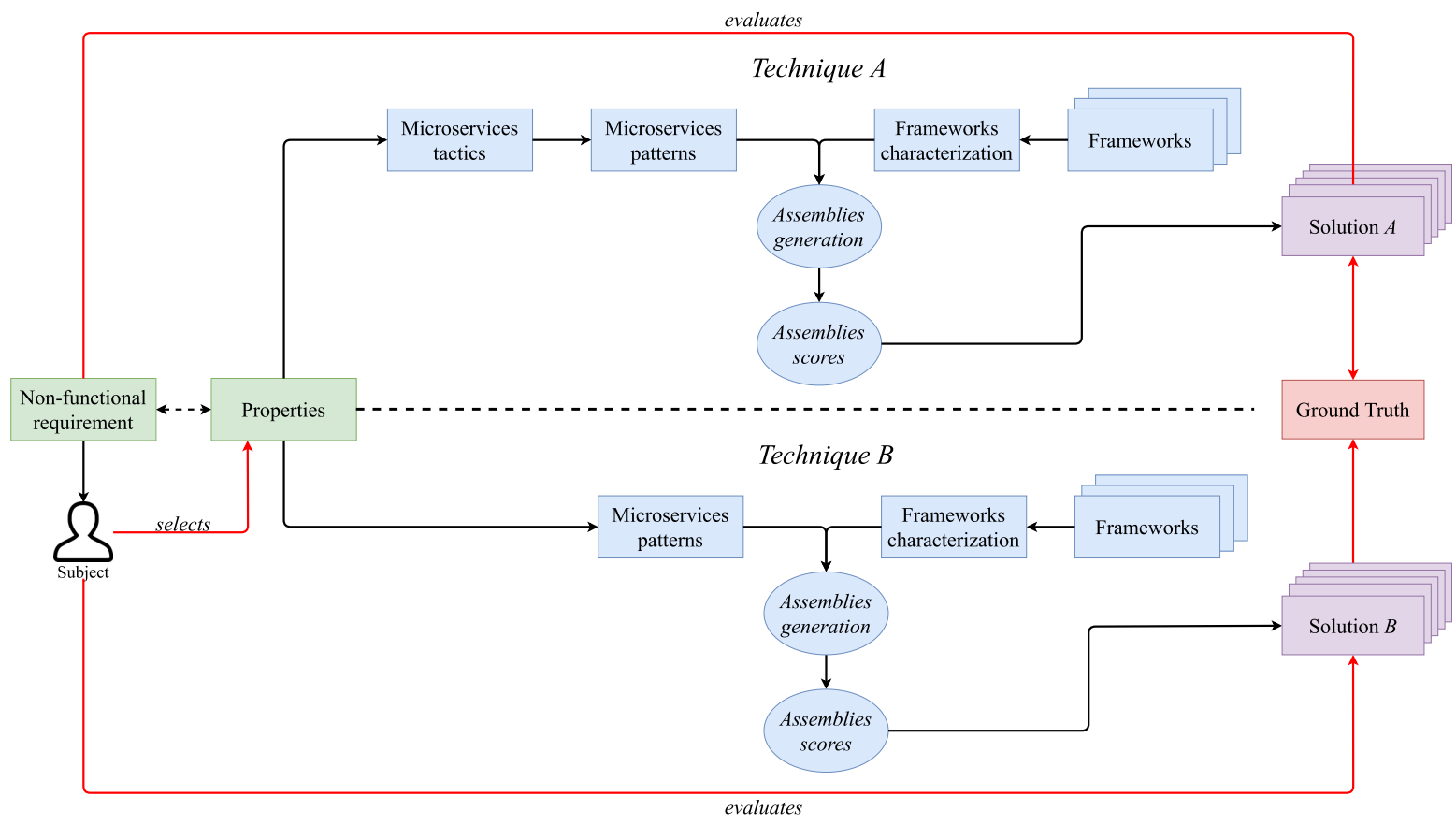


Figure 9.8: Overview of the experimental alternatives

9.3.6. Instrumentation

The resources used in the experiment correspond to documentation describing the case studies of the experiment. In turn, a talk was made in order to explain (i) the context of the experiment, (ii) quality goals, (iii) frameworks used in the experiments, and (iv) the procedure of the experiment.

On the other hand, each subject used a tool in which microAzimut is deployed. The input of this tool are quality goals, and the output are frameworks assemblies.

Finally, a final questionnaire is used to evaluate the experiment and receive feedback from the subjects.

Analysis Mechanisms

This study considers an controlled experiment, which is is an empirical inquiry that manipulates one factor or variable of the studied setting. Based in randomization, different treatments are applied to or by different subjects, while keeping other variables constant, and measuring the effects on outcome variables [144].

The hypothesis test used in this experiment is the Wilcoxon signed-rank test [142]. Because the dependent samples t-test analyzes if the average difference of two repeated measures is zero, it requires metric (interval or ratio) and normally distributed data; the Wilcoxon sign test uses ranked or ordinal data; thus, it is a common alternative to the dependent samples t-test when its assumptions are not met.

Results Validity

Internal validity Threats to internal validity are influences that can affect the independent variable with respect to causality, without the researcher's knowledge. The following threats are identified:

- *Maturation*: This is the effect of that the subjects react differently as time passes.

In order to mitigate this threat, the experiment is conducted after a coffee break. On the other hand, the analysis time of each case is controlled (maximum 20 minutes), and there is a break of 10 minutes after finishing a case.

- *Instrumentation*: This is the effect caused by the artifacts used for experiment execution, such as data collection forms, document to be inspected in an inspection experiment, and others. This threat is mitigated through the prior explanation of each instrument used in the experiment. For this, a practical example of the use of each instrument is shown. Finally, an answer and question session is conducted.

- *Selection*: This is the effect of natural variation in human performance. Depending on how the subjects are selected from a larger group, the selection effects can vary. To mitigate this threat, prior to the experiment, the list of subjects is obtained to know the IT profiles of each subject. For each subject, years of job experience in the IT domain are analyzed in order to identify possible issues (novice versus experts) in the distribution of subjects for the experiment.

- *Mortality*: This effect is due to the different kinds of persons who drop out from the experiment. This threat is mitigated based on the context in which the experiment is conducted. Subjects correspond to the MTI program. Consequently, the experiment was executed in the context of the Software Architecture course as a practical activity. The participation of each subject was rewarded with a grade for the course.

External validity Threats to external validity are conditions that limit our ability to generalize the results of our experiment to industrial practice. In this regard, the "Interaction of selection and treatment" is the most relevant threat. This threat is an effect of having a subject population, not representative of the population we want to generalize to, i.e. the wrong people participate in the experiment. This threat is mitigated based on the different IT profiles that subjects have. Furthermore, the varied professional curriculum of each subject produces no professional bias (for example, only programmers).

Construct validity Construct validity concerns generalizing the result of the experiment to the concept or theory behind the experiment. The following threats are identified:

- *Inadequate preoperational explication of constructs*: This threat means that the constructs are not sufficiently defined, before they are translated into measures or treatments. This threat is mitigated through feedback and results obtained in the pilot study. In this study, each participating subject delivers their feedback aiming at assessing whether the structure of the experimental design is consistent with the experiment's goals.

- *Hypothesis guessing*: When people take part in an experiment they might try to figure out what the purpose and intended result of the experiment is. This threat is mitigated by omitting the subjects that the experimental study considers two types of μAzimut (technique A and B). Each subject will use a μAzimut version (randomly selected) and should only analyze the technique output.

- *Evaluation apprehension*: Some people are afraid of being evaluated. A form of human tendency is to try to look better when being evaluated which is confounded to the outcome of the experiment. To mitigate this threat, it is mentioned that the experimental study is anonymous, and results will only be used for research purposes.

Conclusion validity Threats to the conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment. The following threats are identified:

- *Random irrelevancies in experimental setting*: Elements outside the experimental setting may disturb the results, such as noise outside the room or a sudden interrupt in the experiment. To mitigate this threat, we use an afternoon academic block so that subjects can arrive without problems from work. Also, the University facilities have no environmental noise inconveniences. Finally, we verified

that there were no extra-academic activities that could affect the performance of the experiment.

- *Random heterogeneity of subjects*: This threat is mitigated by analyzing the subject's profile. Subjects participating in the experiment must have job experience in the IT domain. It is considered "IT domain" any field of computer science and informatics.

9.3.7. Evaluating

Aiming at evaluating the experiment, we executed a pilot study. The pilot study's goal is to detect the possible issues or problems of the original experimental study.

In this pilot, two subjects were used for this pilot study. Both subjects are postgraduate students with professional experience in software development and microservices. Subject one used μAzimut with microservices tactics (technique A), and subject two used μAzimut without microservices tactics (technique B). Furthermore, both subjects used the documentation and instruments mentioned in the previous sections. For this pilot study, subjects analysed one case study, which is based on an Availability NFR.

Results

Subject 1 selected the properties P4, P6 and P10. Subsequently as well as the following assembly: [Netflix Eureka-Kubernetes-Apache Zookeeper] with supporting score= 0.7. Subject 1's reasoning focuses primarily on the importance of load balancing with respect to the context of the pilot case. On the other hand, subject 1 also highlights the relevance of replications in different microservices. This is why he selects the assembly containing Apache Zookeeper.

Subject 2 selected the following properties: P4, P6, P7, P8, P10 and P11 as well as the following assembly: [Netflix Eureka-Netflix Hystrix-Netflix Ribbon] with supporting score= 0.77. The rationale described by subject 2 to select this framework assembly

is that Netflix Eureka has functionalities that provide effective load balancing. Furthermore, it provides adequate control of fault propagation and also it has monitoring properties, such as periodic heartbeat. On the other hand, Netflix Hystrix provides efficient timeout control and rapid recovery of failed states. In turn, it also has a high fault propagation control. Finally, Netflix Ribbon complements the properties mentioned previously, but this framework also facilitates integration with other systems.

Finally, the Ground Truth of the case study reveals that subject 1 was closer to the ideal answer than subject 2. Both subjects correctly considered Netflix Eureka as part of the solution. Nevertheless, subject 1 correctly considered Apache Zookeeper as part of the solution. Although subject 2 did have Apache Zookeeper as an option, he did not consider this framework since it was in an assembly with other frameworks unrelated to the case study.

Analysis

The post-pilot survey describes that the pilot study met the desired objective. Both subjects valued the session of the introduction of the frameworks and the technique since it facilitated the analysis to evaluate assemblies of frameworks. Besides, both subjects appreciated the effort to create a technique that allows supporting decision-making regarding how to build and implement microservices-based systems. In summary, the improvements that were applied to the experiment, thanks to the pilot's feedback, are the following:

- *Example of microservices*: Although the subjects already had experience working with microservices-based systems, both recommended showing a simple functional example of a microservices-based system in order to explain the differences between monolithic architecture and microservice architecture.

- *Increase the time to study and analyze frameworks and eliminate extra information*: In the pilot study, subjects were given a document with the description of each framework and a link to their respective documentation and source code.

Furthermore, they were given ten minutes to read the document. However, the pilot's feedback indicates that ten minutes is not enough time to understand and process the information about frameworks. This is because, although subjects have experience in the development of microservice projects, it takes more time for subjects reading the document to infer the advantages and disadvantages of each framework.

9.3.8. Execution

The experiment was executed on August 30, 2019, from 6:00 p.m. to 7:30 p.m. Next, the results will be presented for each case study. For simplicity of the experiment, the α and β values of μAzimut are 1.

Case 1

Case study description For the first case study, the goal is expanding the Sitewhere platform to a particular case. This case is related to a certain customer who needs to add a new smart water meter services to the Sitewhere platform for agricultural purposes. The services receive data from the meter for every minute through a Raspberry Pi board. In this case, subjects don't need to be experts in the Internet of Things (IoT) or know how to program Raspberry Pi boards. Subjects must evaluate, from the point of view of architecture, the following high-availability NFR: *It is demanded that smart water meter services must be available 99.5 % of the day for the Sitewhere platform to generate daily water consumption reports using other services.*

Expectations The expectations for this case suggest that the subjects should obtain the following assembly of frameworks [Netflix Eureka, RabbitMQ, Swagger, Netflix Ribbon]. This assembly represents the group of frameworks that the Ground Truth architect has selected to satisfy the NFR. The main reasons why these frameworks were selected are described below:

- Netflix Eureka provides REST services and also acts as a server. The main objective of this framework is to register and locate existing microservices, inform about their location, status and relevant data of each of them. In addition, it facilitates load balancing and fault tolerance.

- RabbitMQ is a messenger broker or queue manager. In other words, it is a software where queues can be defined; applications can connect to these queues and transfer/read messages on them. This framework facilitates the asynchronous communication required in this NFR.

- Swagger helps manage REST APIs. In addition, the framework can describe, produce, consume and display APIs. The idea of using this framework is to manage the endpoints of the services.

- Netflix Ribbon helps intercomunication processes by providing robust fault tolerance methods.

Results Figure 9.9 illustrates that the property most selected by the subjects for this case study is P10. This property emerges when one service emits a heartbeat message periodically and another service listens for it. If the heartbeat fails, the originating component is assumed to have failed, and a fault correction component is notified. The heartbeat can also carry data. For example, an automated teller machine can periodically send the log of the last transaction to a server. This message not only acts as a heartbeat but also carries data to be processed. On the other hand, the properties that were not considered by the subjects are P3 and P5. Furthermore, the comparison of precision and results of each technique is illustrated in Figure 9.11.

Both figures show the distribution of values achieved in precision and recall. On the precision side, in general, the values obtained are greater than the values obtained in the recall. This means that in case 1, the probability that a randomly selected framework be relevant to solve case 1, is high.

For technique A, the average accuracy is $\approx 60\%$, and for technique B, it is 46%. Regarding the recall, the averages for technique A and B is $\approx 38\%$ and 29%, respectively.

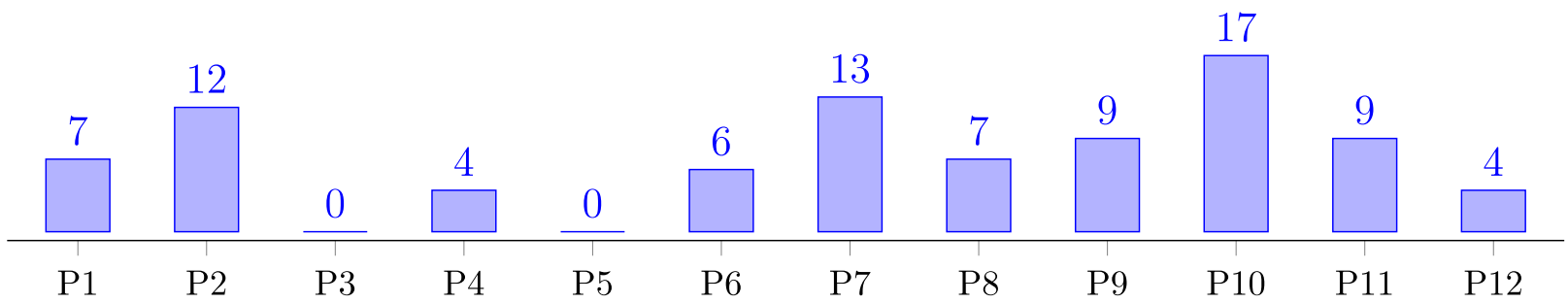


Figure 9.9: Selection frequency for case 1

Everything seems to indicate preliminary that when microservices tactics are used as decision component, the probability of selecting frameworks that an expert architect would select for the same case is increased. Nevertheless, although the average precision and recall values are low, for the purposes of case study 1, the results are encouraging since the ground truth framework set is bounded (in practice, it makes no sense to select all the possible frameworks to satisfy NFRs). This means that the technique with microservices tactics generates few relevant assemblies of frameworks, but precise.

Case 2

Case study description In this case, the analyzed requirement is related to the expansion of the company towards new suppliers. That said, the current microservice architecture has to be adapted to the changes required by stakeholders. Therefore, the following NFR will be analyzed: *It is critical that the last 5 days of each month the remuneration system must provide the workers' payment without any problems (99.9% availability), because another forecast system must quote AFP and FONASA/ISAPRE of each worker. In case the payments are delayed, the ABC company may suffer tax penalties and possible lawsuits.*

Expectations The expectations for this case relies on the fact that the following assembly of frameworks [Netflix Eureka, Netflix Zuul, Apache Zookeeper, Netflix Ribbon] should be selected. This assembly represents the group of frameworks that the project architect has selected to meet the NFR. The main reasons why these frameworks were selected are described below:

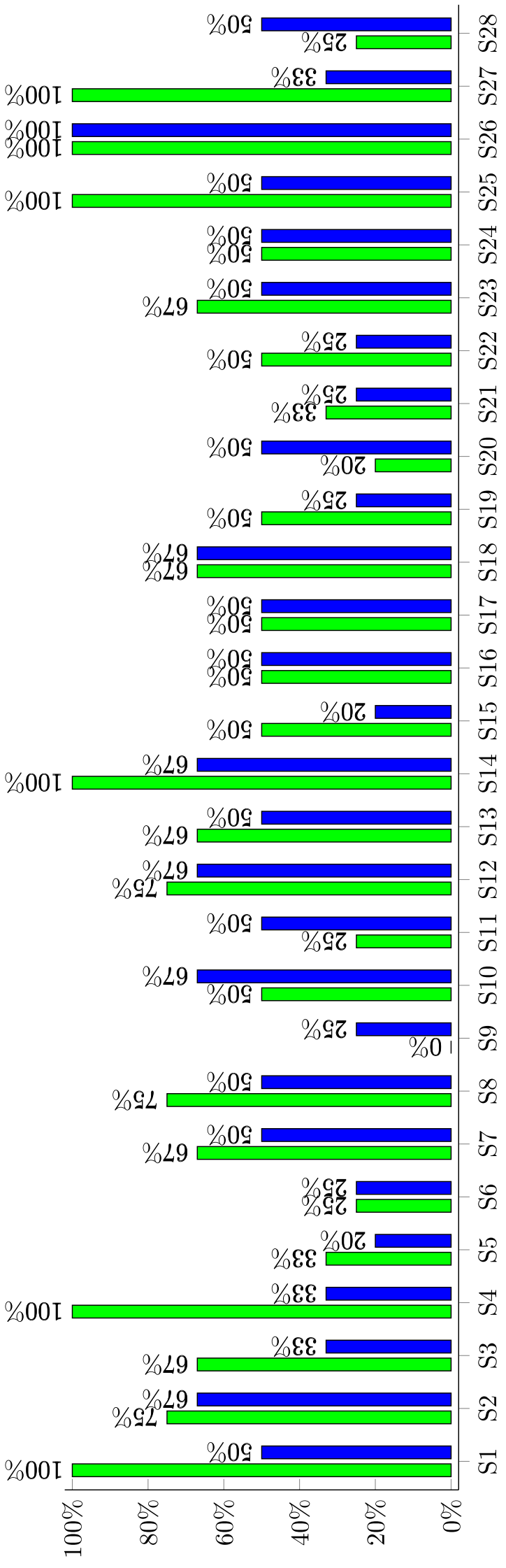


Figure 9.10: Precision results for technique A (TA) and B (TB) on case 1

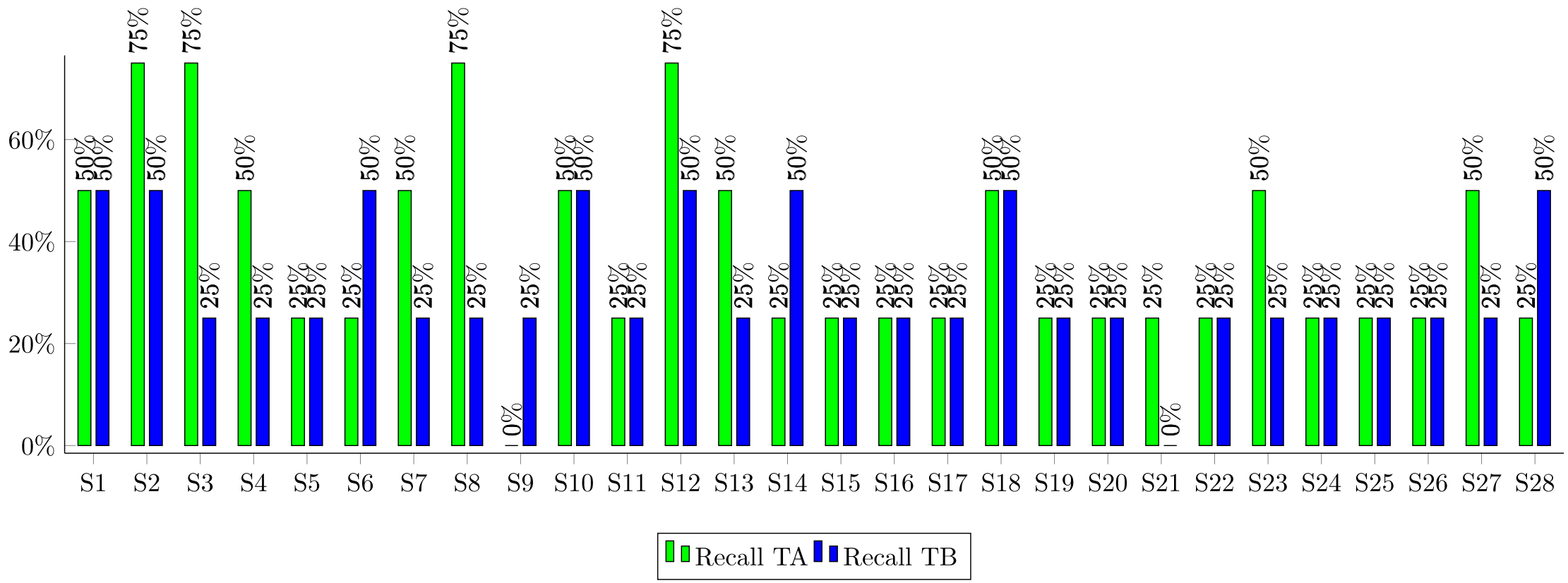


Figure 9.11: Recall results for technique A (TA) and B (TB) on case 1

- Netflix Eureka provides REST services and also acts as a server. The main objective of this framework is to register and locate existing microservices, inform about their location, status and relevant data of each of them. In addition, it facilitates load balancing and fault tolerance.

- Netflix Zuul is necessary to route and filter requests dynamically, monitor, and secure them. This service acts as an entry point to microservices; that is, it is responsible for requesting an instance of a specific microservice from Netflix Eureka and routing it towards the service that wants to consume.

- Apache Zookeeper provides a centralized service for various tasks such as configuration maintenance, distributed synchronization, or grouping services. In order the system does not have failures with PREVIRED/FONSASA or some AFP, this framework must provide correct coordination between distributed processes

- Unlike case 1, Netflix Ribbon is necessary to support Netflix Eureka in load balancing. In addition, support for multiple protocols (HTTP, TCP, UDP) is required as an asynchronous and reactive model.

Results Figure 9.12 describes the frequency of the properties selected for case 2. Unlike case 1, the property most selected by subjects is P7. One of the intrinsic properties of a microservice architecture is resilience. This property is related to the ability to isolate faults through a good definition of the limits of a microservice. For case 2, subjects pointed out that one of the most critical properties that must be satisfied is the rapid recovery of broken states. This means that, for the majority of the subjects, the Payment System must recover from failures. For the period of time described in the NFR, the system does not tolerate failures. Paradoxically, the least selected property is P2. In case 1, P2 is the third most selected property. P2 is related to asynchronous communication, one of the fundamental characteristics of the microservice architecture. However, subjects considered that for case 2, this property is not essential. This is partly because the microservices of the project are well defined and need little communication between services.

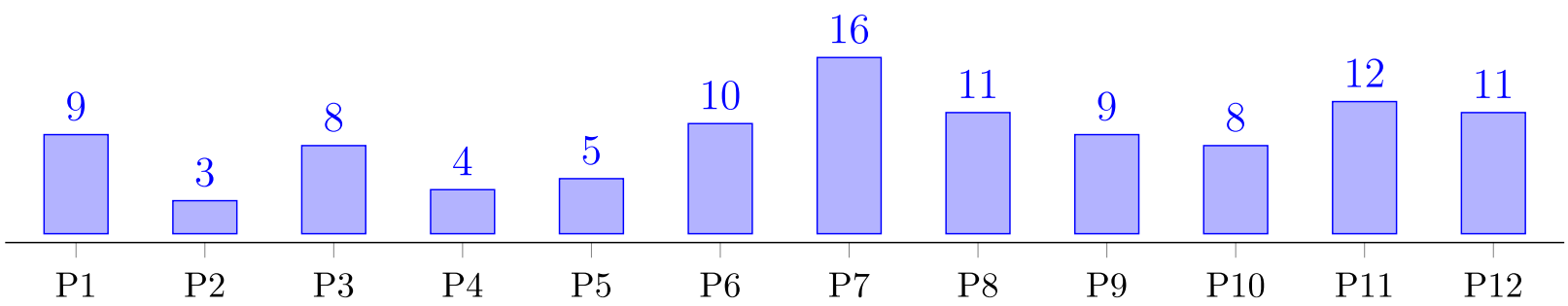


Figure 9.12: Selection frequency for case 2

Figures 9.14 describes the precision and recall results of both techniques as well as the frequency of precision and recall by technique. For case 2, almost the same situation occurs as case 1. The average precision values of each technique are $\approx 60\%$ for technique A and 48% for technique B. Nevertheless, concerning the recall, there is a slight increase in their average values, $\approx 43\%$ and 37%, respectively. Afterward, in this case, the same situation that occurs in case 1 also occurs in case 2; the use of microservices tactics generates a reduced set of assembly of frameworks, but sufficient to satisfy the NFR.

Frameworks assemblies efficiency

Regarding RQ 4.5, The weights (w_i) established for the frameworks assembly in case 1 (A_1 =[Netflix Eureka, RabbitMQ, Swagger, Netflix Ribbon]) are 0.5, 0.3, 0.1, and 0.1, respectively. To determine the weights of case 1, we used the Understand[7] tool to count the number of instances of A_1, which are described in the source code, for each microservice involved in the implementation of case 1 (see Table 9.12). Thus, once we have obtained all the instances of A_1 frameworks, we calculated each framework's percentage of instances with respect to the total of instances of A_1. For example, for the first framework of A_2 (Netflix Eureka), the weight assigned is 0.2, since this framework has 2 instances out of a total of 10 instances (20%). Respectively, for case 2 (A_2 =[Netflix Eureka, Netflix Zuul, Apache Zookeeper, Netflix Ribbon]) the weights are 0.2, 0.5, 0.2, and 0.1. To determine the weights of case 2, we performed the same procedure as case 1.

[7]https://scitools.com

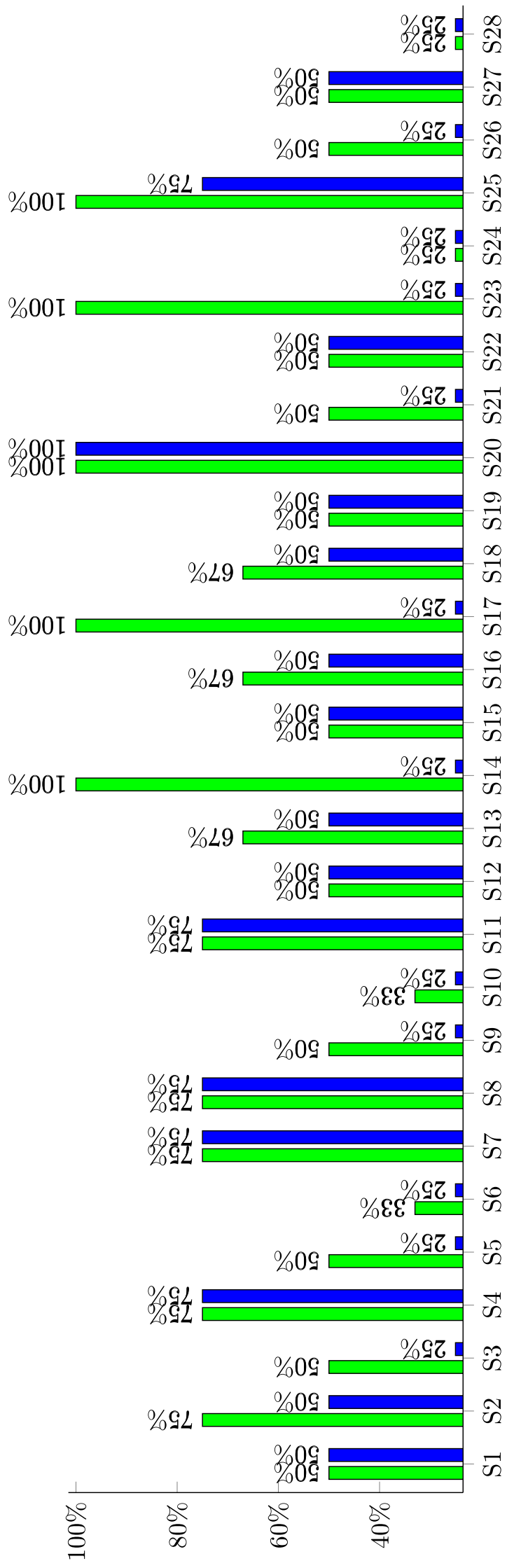


Figure 9.13: Precision results for technique A (TA) and B (TB) on case 2

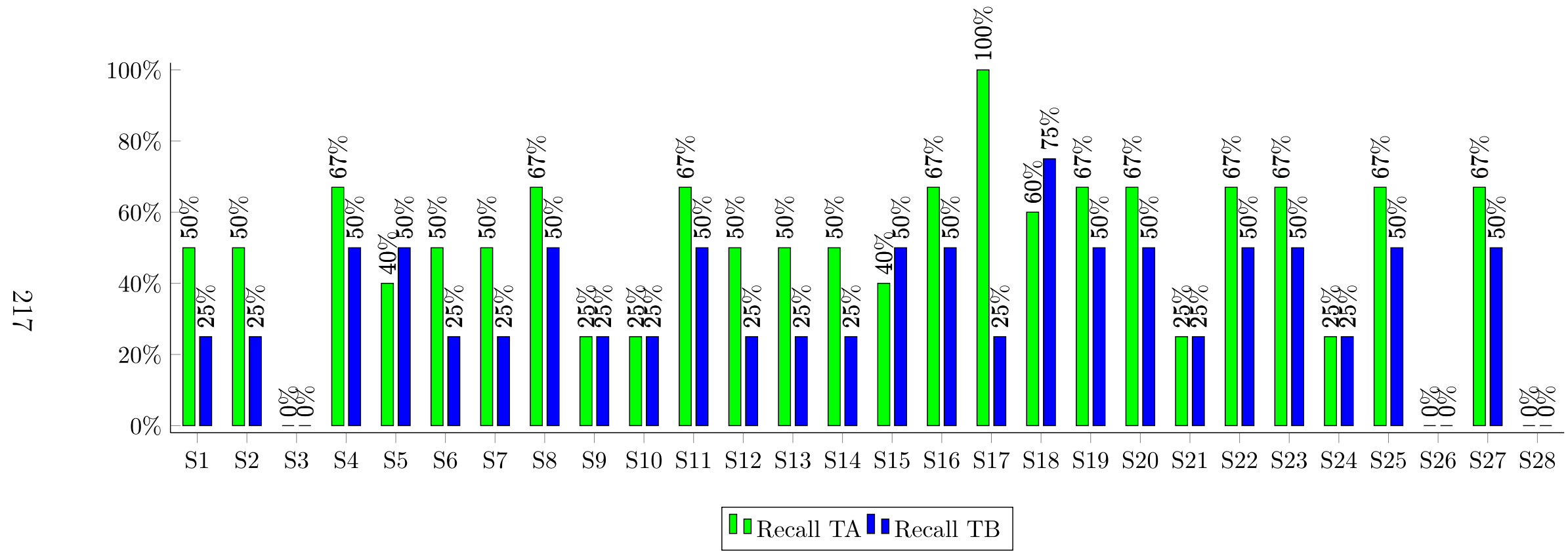


Figure 9.14: Recall results for technique A (TA) and B (TB) on case 2

Table 9.12: Partial analysis of microservices and frameworks instances

	Netflix Eureka	Netflix Zuul	Netflix Hystrix	RabbitMQ	Kubernetes
service-asset-management	1	-	-	-	-
service-batch-operations	-	-	-	-	-
service-command-delivery	-	-	-	1	1
service-device-management	-	-	-	-	-
service-device-registration	-	-	-	-	-
service-event-management	-	-	1	-	-
service-event-search	-	-	1	-	-
service-event-sources	-	-	-	-	-
service-inbound-processing	-	-	1	-	-

...

⋮

On the other hand, Table 9.13 describes the combinations we estimate from the frameworks used in the experiment. For simplicity, we name only the amount of functionalities that a combination of frameworks replaces another framework.

Table 9.13: Frameworks and combination of frameworks

Frameworks	F2	F3	F4	F6	F10	F12
Functionalities	3	3	4	5	3	3
	↓	↓	↓	↓	↓	↓
Combination	F1F3	F4F11	F7F11	F7F12	F4F12	F10F11
Functionalities	2	2	3	3	2	2

Results Figures 9.15 and 9.16 illustrates the results of each subject's efficiency for techniques A and B in cases 1 and 2.

Concerning case 1, the average efficiency value for technique A was 53%, and for technique B 34%. Likewise, in case 2, the average efficiency value was 60% for technique A and 42% for technique B. Additionally, the ground truth experts have established that an acceptance threshold for each subject is above 75% efficiency. If a subject selects an assembly with an efficiency equal to or greater than the threshold, the subject

successfully addresses the availability-related NFR. Otherwise, the subject partially satisfies the NFR.

In case 1, the efficiency of technique A is regular, except for two subjects S2 and S8; both subjects obtained the best result, over 75% efficiency. Since these subjects are concentrated in the higher range of job experience, we believe that their experience allowed a more precise selection of frameworks. Regarding technique B, S2 was the only subject with the most higher result.

Concerning case 2, there are several subjects with an efficiency above 75% using technique A. Yet, the subject that stands out from the rest is subject 20 with an efficiency of 100%. Reviewing the subject's responses, she made a more critical analysis of the outputs of μAzimut, i.e., she made a comparative analysis of the pros and cons of each framework for each assembly. With respect to technique B, the only subject with a performance above 75% was S22.

Regarding Figures 9.15 and 9.16, it is possible to see that for both cases, the efficiency of technique A is over the efficiency of technique B in most subjects. Therefore, taking as reference the results depicted in Figures 9.15 and 9.16 and the average results of efficiency, we can infer using technique A allows obtaining more *efficient* frameworks assemblies to satisfy NFRs.

9.3.9. Analysis

RQ4.3: Microservices tactics and μAzimut

Concerning RQ 4.3, Table 9.14 summarizes the values obtained from the test.

According to the results described in Table 9.14, for case 1, both the precision and recall, p-values are less than 0.05. Therefore, these values indicate that there is a significant difference in the use of μAzimut using microservices tactics as an abstraction layer. The same situation occurs in case 2, p-values are also less than 0.05. Therefore, it can be concluded that $H_{0.1}$ is rejected.

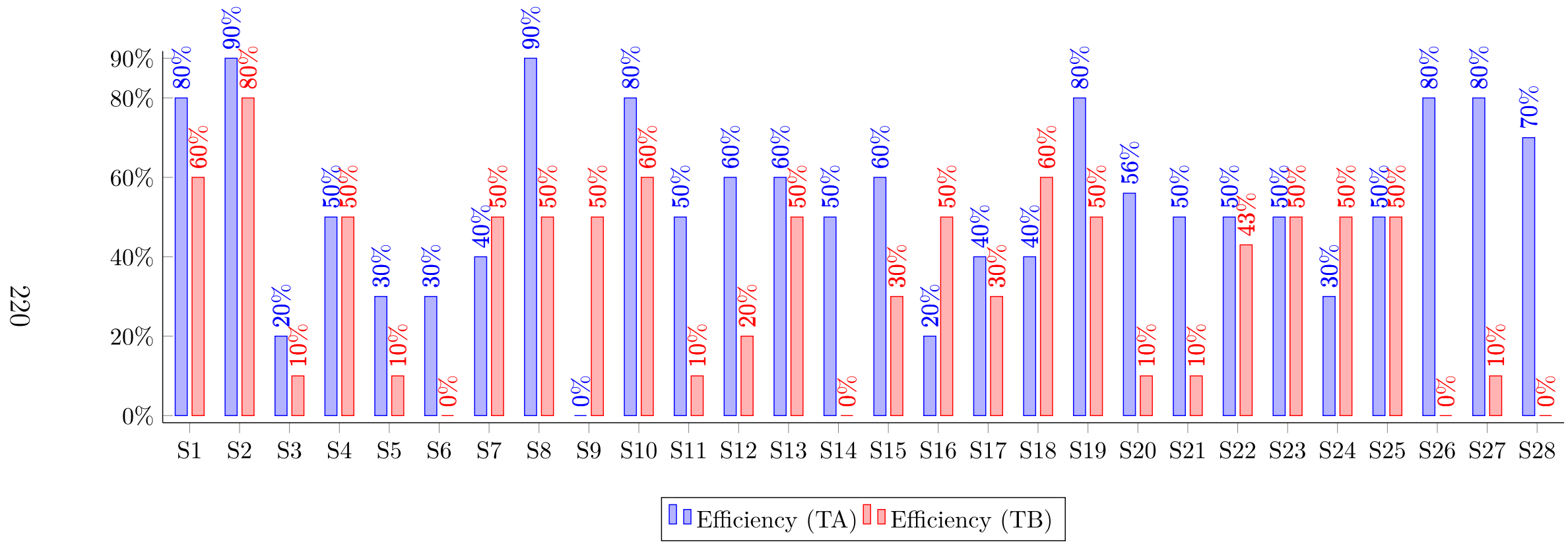


Figure 9.15: Efficiency results of technique A (TA) and technique B (TB) in case 1

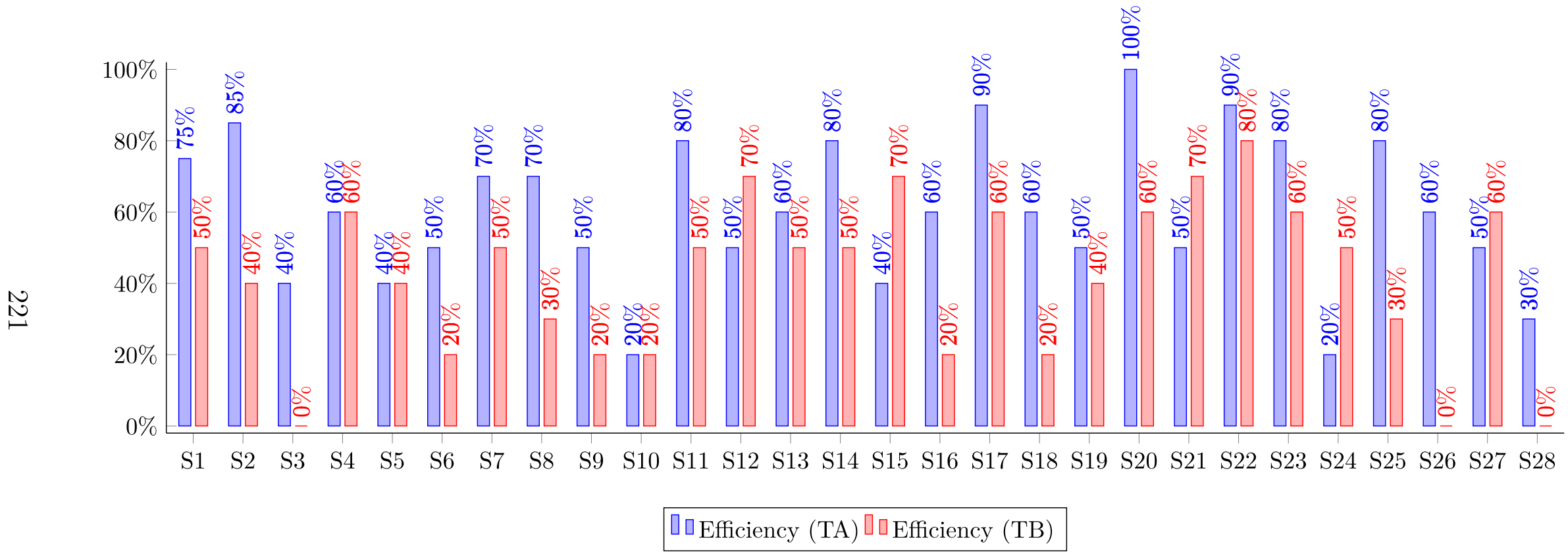


Figure 9.16: Efficiency results of technique A (TA) and technique B (TB) in case 2

Table 9.14: Wilcoxon signed-rank test results. "P" represents *precision* and "R" represents *recall*. Each "P" and "R" compares technique A and B respectively.

		σ	N_r	W^+	W^-	$min\{W^+, W^-\}$	$\alpha_{two-tailed} \leq 0.05$	$z-score$	$p-value$
Case 1	P	0.260681	22	196.5	56.5	56.5	65	-2.272569	0.023051
	R	0.195011	14	85.5	19.5	19.5	21	-2.071624	0.038301
Case 2	P	0.186641	22	213	40	40	65	-2.808279	0.004981
	R	0.196489	15	99	21	21	25	-2.215051	0.026757

Case 1 Analysis Regarding case 1, six subjects obtained 100% precision values (S1, S4, S14, S25, S26, and S17) for technique A. For these subjects, the results that μAzimut showed, based on the properties selected, were the same as the architect chooses in the Ground Truth. However, these results must be counteracted concerning the value given by the recall ratio. For example, for S25 and S26, these subjects selected the assembly where only a single framework emerged. That is why both have a 100% accuracy but a 25% recall. Going deeper into these results, we analyse the years of experience of these subjects. Both subjects have $>$ 10 years of experience. Subsequently, we analysed the feedback of the experiment for both subjects. Their answers indicate that according to experience, they select only one framework because they try to all its capability instead of using a set of framework assemblies. In contrast to these results, the opinions of these two subjects were discussed with the Ground Truth's architect. It was concluded that both subjects prefer to exploit all the capabilities of a framework to reduce working time instead of selecting a framework assembly that may be more optimal. This means that these subjects' decision was reduced to select the assembly with less framework instead of selecting an assembly with more frameworks

These data are related to previous experimental studies concerning decision making between expert and novice professionals. In our previous work [109], we conducted experiments to evaluate the selection of security tactics between expert and novice subjects. In summary, for expert subjects, a technique that suggests what decisions to make regarding security events is not useful, since they will rely on their experience to solve a problem. However, for novice subjects, techniques that suggest what security decisions to make for certain scenarios are useful, since decisions that novice subjects perform with the help of the technique are quite close to the decision that an expert

makes for the same scenario. Therefore, the results of the current experiment suggest, as future work, to explore whether μAzimut is useful for experts or novice professionals in the responsibility of reducing the space of solutions.

In the case of technique B, the average precision and recall values (46% and 29%) indicate that μAzimut without the microservices tactics layer delivers assemblies with more frameworks, but not entirely effective. If we put this interpretation into practice, this usually occurs not only for the development of microservices-based systems but also in software development in general.

Case 2 Analysis Concerning case 2, S20 selected the same frameworks that were determined in the Ground Truth. This result can be attributed to the subject's work experience (see Figure 9.20 in Section 9.3.11). In addition, the subject's feedback indicates that technique A provided better frameworks than technique B.

In general, the results of this case are not much different from the results obtained in case 1. Additionally, the subjects who obtained 100% precision also have the same situation regarding recall that occurred in case 1. Yet, the subjects stated that μAzimut was more useful and effective in addressing the NFRs of case 2. Exploring this observation further, the subjects recognized that unlike case 1, where the system under study was very well documented and accessible, for the system addressed in case 2, μAzimut was critical to decision making. This is explained by the fact that, unlike case 1, case 2 has a high degree of uncertainty because we intentionally selected a complex system. We intended to evaluate μAzimut in a complex NFR to determine if subjects performed well. Finally, based on the results obtained, μAzimut was relevant to obtain good results.

RQ4.4: Subjects rationale

Figure 9.17 shows the distribution of the frameworks selected by subjects in Case 1 using both techniques (A y B). Figure 9.17 shows how Netflix Eureka is the most reported framework. Netflix Eureka aims (mainly) to locate services in order to balance

the load and fail-over of mid-level servers. For this reason, most subjects selected this framework because it features coincides with the property most reported by subjects in the two cases. In the same regard, Netflix Eureka and RabbitMQ facilitate the creation of applications based on fault-tolerant microservices. Subjects chose these frameworks because they allow rapid recovery of failed states in microservices communication. They provide the ability to restart states when they are "broken" for a more extended period, which converges with one of the most reported properties by subjects.

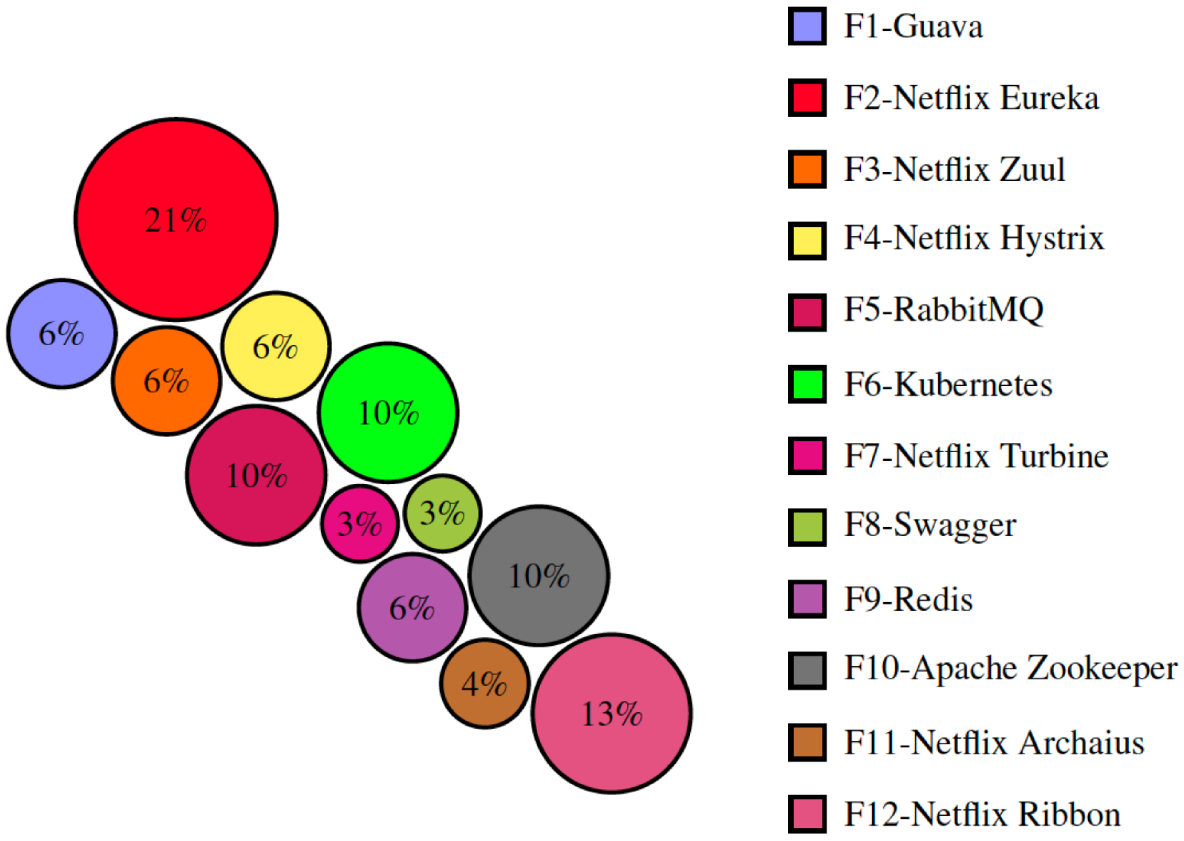


Figure 9.17: Subjects and framework selection using technique A

On the other hand, Figure 9.18 shows the distribution of frameworks most reported by subjects in Case 2 using both techniques (A y B). Just like Case 1, Netflix Eureka is the most selected framework. On the other hand, Netflix Ribbon also provides a good load balance, which is nothing more than the process of efficiently distributing network traffic among multiple back-end services, which is an excellent strategy to maximize scalability and availability. With these frameworks, subjects satisfy the most reported properties (low restart times, recovery from failures, and efficient resource performance).

In addition, Figure 9.19 illustrates frameworks used in the experiment (X-axis) as well

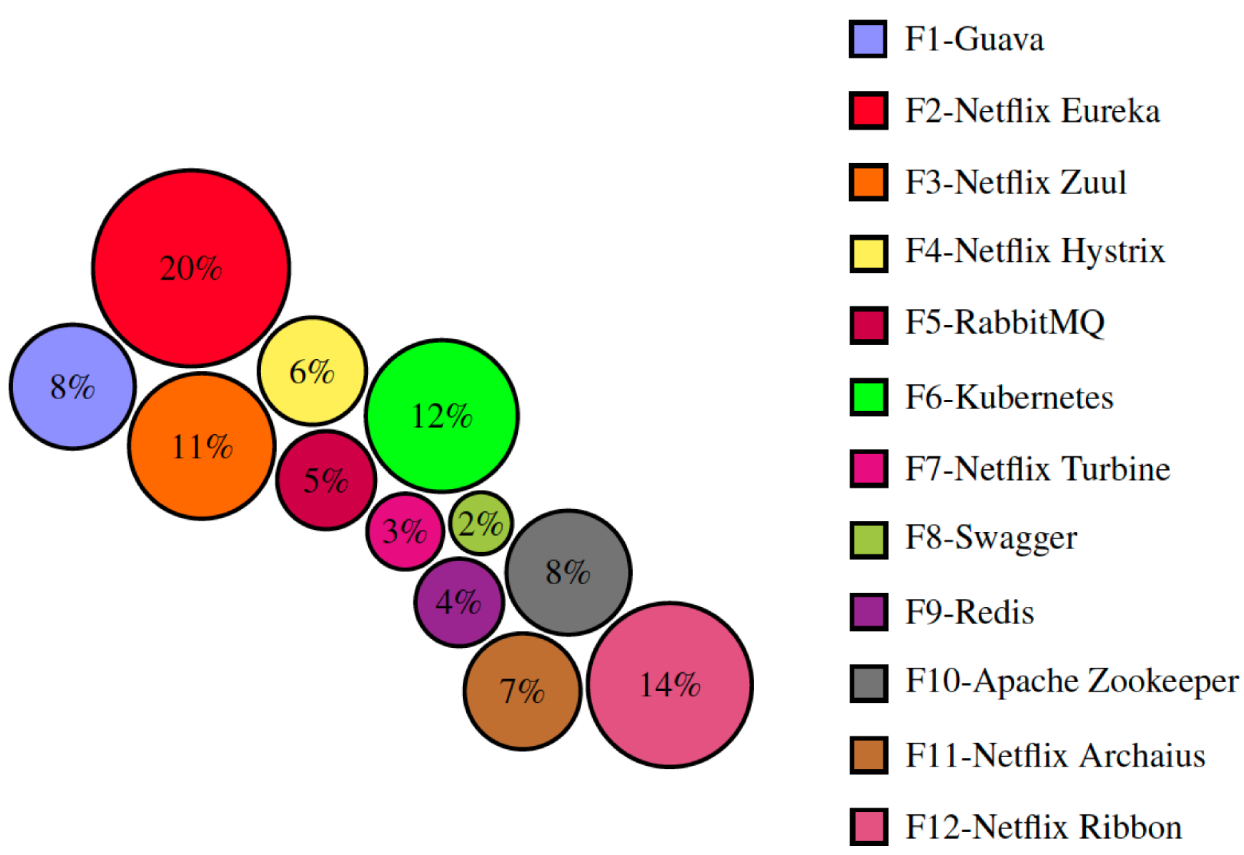


Figure 9.18: Subjects and framework selection using technique B

as the main functionalities (Y-axis) reported by subjects to address NFRs using technique A and B. The interception of both axes (green squares correspond to technique A and blue diamonds correspond to technique B) shows a number that represents the number of subjects who reported that functionality in the framework specified in the X-axis.

Load balance and fault tolerance are the most reported architectural functionalities. According to the subjects' feedback, these two functionalities must be considered to satisfy the NFRs used in the experiment.

Load balance is the answer so that microservices can handle the load, security and remain available. For its part, fault tolerance is the functionality that allows a system to continue functioning correctly in case of failure of some of its components, be they a microservice, a database, a load balance or any other component.

Fault tolerance is essentially based on one concept: redundancy. The best way to ensure the availability of microservice architectures and the data they supply in a highly reliable and uninterrupted manner is to duplicate all their critical elements and to arrange the necessary software and hardware elements so that the redundant

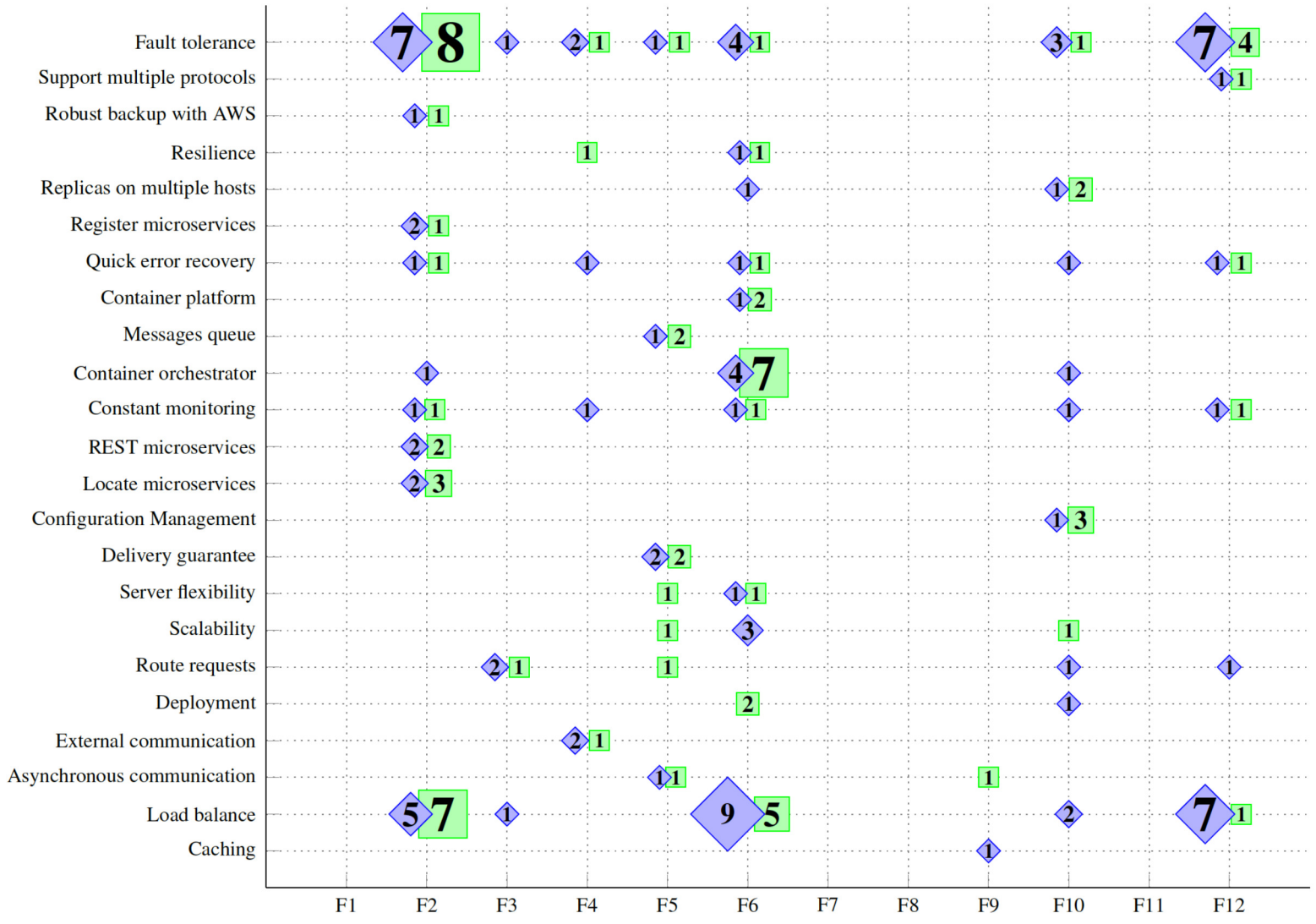


Figure 9.19: Main functionalities and framework selected. Numbers represent the subjects' selection frequency

elements act cooperatively, either actively-actively or actively-passive, but always in a way that is transparent to the user.

On the other hand, the less features reported by the subjects are: asynchronous, external communication, implementation, server flexibility, message queue, container platform, microservice logging, resilience, robust support with AWS and support of multiple protocols.

Concerning frameworks and platforms, Guava, Netflix Turbine, Swagger and Netflix Archaius were less selected by subjects. Although these tools provide functionalities that are relevant for ensuring high availability, the subjects estimated that these frameworks/platforms do not offer value for satisfying NFRs.

RQ4.5: Frameworks efficiency

The results of the execution of the Wilcoxon signed-rank test are illustrated in Table 9.15. The p-values indicate that they are below the threshold 0.05, which implies that hypothesis $H_{0.2}$ is rejected.

Table 9.15: Wilcoxon signed-rank test results of frameworks assemblies efficiency

	σ	N_r	W^+	W^-	$min\{W^+, W^-\}$	$\alpha_{two-tailed} \leq 0.05$	$z-score$	$p-value$
Case 1	0.306320	25	268	57	57	89	-2.8387	.00452
Case 2	0.245650	25	282.5	42.5	42.5	89	-3.2288	0.001243

Architectural tactics are design decisions that influence the control of the response of a quality attribute. Unlike architectural patterns, architectural tactics are more detailed solutions and are focused on specific quality attributes. The experiment results have shown that the specification of architectural tactics allows an architect to evaluate a more precise and efficient set of frameworks rather than evaluating a set of candidate frameworks that implement an architectural pattern.

Some case studies, such as the one conducted by Cervantes *et al.* [31], have shown the importance of using architectural tactics as evaluation criteria to select frameworks. The way that architectural tactics are used is as a kind of analysis: architectural tactics describe the space of possible design objectives with respect to a quality attribute. By determining which architectural tactics realized a framework, a measure of the functional completeness of the framework is obtained.

The results of our controlled experiment complement the results of Cervantes *et al.* in the context of microservices, i.e., we have realized that architectural tactics specifications do indeed generate effective framework assemblies and facilitate the architect's evaluation to satisfy NFRs in the context of microservices-based systems. Therefore, our experimental study results are a starting point for investigating the contribution of architectural tactics to the design and development of microservices-based systems.

9.3.10. Experimental package

We have enabled an experimental package where the artefacts used in our experiment are described in order to replicate the study. This package contains (i) the answer sheets, (ii) the initial talk, (iii) the description of the μAzimut input, and (iv) a description of the frameworks and platforms used in the experiment. Additionally, we include two figures that describe which frameworks were selected by each subject in each experimental case. It is possible to access the experimental package in the following link: `https://doi.org/10.5281/zenodo.3833300`

9.3.11. Post-experiment survey

Figure 9.20 describes the range of years of job experience of the subjects who participated in the experiment. Most concentrate on the range of 5-10 years and 10+ years.

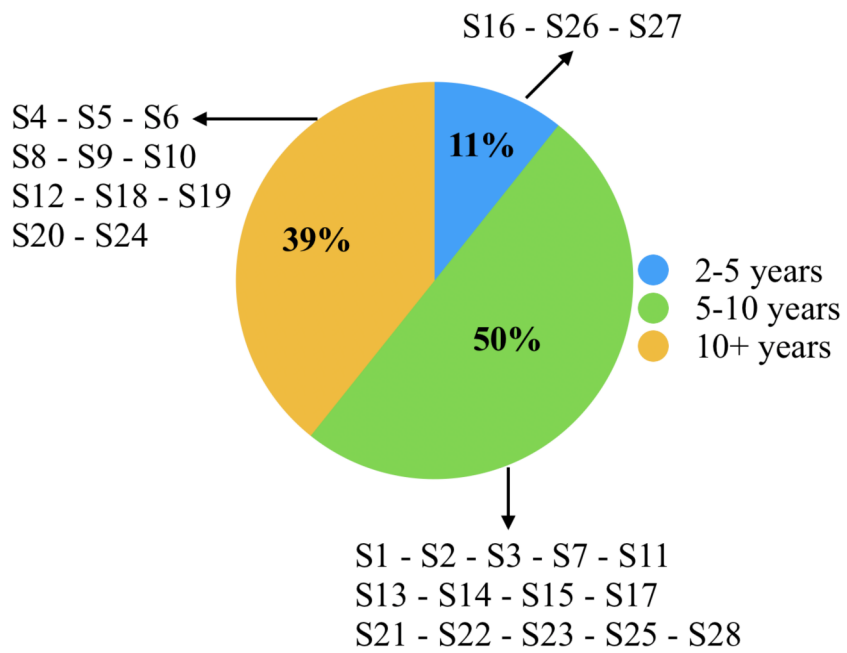


Figure 9.20: Subjects' job experience (years)

Following the experiment, we conducted a survey in which we asked subjects to describe the advantages and disadvantages of microAzimut as well as what improvements they recommended. We then summarize the main advantages, disadvantages and suggestions that are most repeated among the subjects' responses:

Advantages:

- *New alternatives to solve recurring problems*: The subjects valued the effort to gather, describe and illustrate the most recurrent ways of solving typical microservice architecture problems. In addition, for some subjects, it was novel to know patterns of microservices that they did not know.

- *Simplicity*: This was the most mentioned advantage. Given the constant time-to-market that governs software development in any field, the subjects valued the simplicity and clarity of microAzimut, which came as a surprise to us. At the beginning of the experiment, we made an effort to explain the technique as simply as possible, which had an effect on the subjects.

- *Quick identification of frameworks*: Another response that is repeated is the quick identification of frameworks that are relevant to satisfy an NFR. In the final talk we give, apart from the survey, the subjects recognize that, since microservices architecture is still an emerging topic in the development and deployment of applications, most of the companies, where the subjects are inserted, use frameworks without knowing if they are useful or not. This means that developers learn by "trial and error" whether a framework is valuable or not.

- *Focused knowledge*: A large number of subjects valued the fact that the technique structures knowledge that, according to the subjects, "everyone knows but no one orders". This means that the technique makes an effort to gather and systematize the knowledge that the developers of microservices-based systems have in practice.

Disadvantages:

- *More knowledge about architectural tactics*: The subjects mentioned that more knowledge about microservices tactics is needed. Most were unaware of the concept of architectural tactics, but used them in their projects without knowing that they are packaged knowledge.

- *Constant updating of frameworks*: The subjects mentioned that microAzimut, in order to be effective, must be constantly updated regarding the use of frameworks. We have noticed this observation previously.

Suggestions:

- *Extending the technique to DevOps*: The subjects mentioned the possibility of extending microAzimut to other domains, such as DevOps. The subjects in the final discussion mentioned that the use of microservices has become a key component for companies that wish to implement the basic principles of culture, organization, processes and tools of the DevOps philosophy, since it can provide agility in deployments, increase the quality of the software delivered, facilitate the creation of autonomous groups identified by the "value stream" and even eliminate traditional interdepartmental barriers.

- *More collaborative technique*: The experiment was carried out in a personal way by each subject. However, a suggestion that was repeated a lot was to include more members in microAzimut to transform the technique into a collaborative one. Analyzing the options involved in making this change, we have decided to leave this suggestion as future work.

9.4. Summary

This chapter describes three empirical studies that allow the evaluation and testing of μAzimut. The first empirical study corresponds to a case study which evaluates the frameworks assemblies generated by μAzimut with respect to the decision made by an architect for NFRs of a real project. The results of this case study indicate that μAzimut helps in making initial decisions. The feedback from the architect who participated in the case study describes that μAzimut is an excellent ally in evaluating decisions on high uncertainty NFRs.

The second empirical study corresponds to a case study on the design of an IoMT platform based on microservices. In this case study, we used μAzimut to evaluate not only one but several NFRs. We handled μAzimut outputs to generate architecture design options to engage the project stakeholders. The purpose of this case study is to evaluate the importance of μAzimut in the process and design of IoT software architectures using the ADD methodology. The inclusion of the stakeholders allows them to be more involved in the architecture decisions, thus defining priorities and implementation strategies that are in line with the objectives that the stakeholders want to achieve.

The third empirical study corresponds to the evaluation of the impact of using or not using microservices tactics in μAzimut. Globally speaking, the results of the experiment point out that including microservices tactics reduces the space for solutions that the architect must evaluate, thus facilitating decision-making.

Part V

Conclusion and Future Work

Chapter 10

Conclusions

This doctoral thesis presents a novel technique that uses architectural knowledge of architectural patterns and architectural tactics to enable the architect to evaluate framework assemblies for designing microservice architecture. The technique uses multidimensional catalogs of patterns, tactics, and frameworks with imprecise information that support the body of knowledge of the technique. The μAzimut inputs are properties that characterize quality attributes, which are represented by NFRs. These properties guide the architect as to which particular objective he or she wishes to address in the NFR. On the other hand, the output of μAzimut corresponds to a set of assembly frameworks that allow helping the architect in making decisions to design microservices architectures. Framework assemblies are characterized by support scores that would enable evaluating the relevance of the assemblies according to what patterns and tactics they represent. In this way, the architect will be able to trace the decisions that each selected framework represents for the design of microservices architecture.

To deepen in the most important aspects of the technique, in the first part of the thesis, exploratory studies were carried out to characterize and illustrate the state of the art and practice regarding the use of microservices patterns and tactics. The results of these exploratory studies describe that there is a fairly wide range of microservices patterns. Several microservices patterns have been reported in academia as well as in practice, but only a small set of these converge between the two worlds. On the other

hand, regarding the microservices tactics, academic evidence was found, but there is not enough information in practice. Besides exploring the academic and practical evidence of the main constructs of the technique, another relevant aspect that supports μAzimut is the study on the use of frameworks in microservices architectures. To understand the objectives of frameworks in microservices architectures, a technique called Pekenum was created, which allows the recovery of microservices architectures. This technique uses elements and artefacts from microservices projects in order to illustrate a view based on microservices patterns. Pekenum allows understanding how microservices interact with each other as well as the functionality of the frameworks in the project.

In the second part of this doctoral thesis, all the information obtained in the first part is used to build μAzimut. More precisely, Chapter 5 describes a methodology to define the properties of the quality attributes used by μAzimut. This methodology is applied to all quality attributes to detect, define, and characterize properties. Subsequently, Chapter 7 describes a systematic procedure to characterize microservices tactics that emerge in the context of the development and deployment of microservices-based systems. Microservices tactics seek to complement current architectural tactics catalogues by encompassing tactics that are used in both microservice architectures as well as distributed systems. Then, Chapter 3 describes the main microservices patterns that are used in practice and that belong to the μAzimut core. In the first part, a large set of microservices patterns that facilitate the development and deployment of microservices-based systems were detected. Yet, not all these microservices patterns have evidence in practice. Therefore, Chapter describes the microservices patterns that have been reported in order to use them and give a more practical approach to μAzimut. Finally, the second part of the thesis closes with a complete description of the μAzimut technique. In Chapter 8, the main components of the technique are described in detail as well as the procedures to calculate the support score that supports the assembly of frameworks.

Finally, the third part of the thesis concentrates on conducting three empirical strategies related to the technique. Often architects think of patterns to design different types of software architectures. In the context of microservices, this is no exception.

Aiming at designing microservices architectures, architects evaluate a multitude of patterns that are then translated into platforms, frameworks, and tools that shape the architecture of a microservices-based system. Nevertheless, since there are a large number of microservices patterns that satisfy a multitude of architectural problems in microservices-based systems, the selection of the most suitable patterns does not always satisfy the NFRs. This means that if an architect selects a set of microservices patterns without being clear about what architectural objectives he or she wants to address, the set of frameworks that implement those microservices patterns increases, affecting the ability to select the optimal set of frameworks for a microservices architecture. Therefore, the assumption made in Section 9.3 is that microservices tactics allow reducing the solution space and help the architect to select the frameworks that better satisfy NFRs. Having said this, Section 9.3 describes an experiment where μAzimut is used with and without the microservices tactics component. μAzimut without the microservices tactics component represents the regular context that an architect must face to design microservices architecture. On the other hand, μAzimut with the microservices tactics component represents the new context to be experienced. The results of the experiment show encouraging results that point out that μAzimut in its fullness (including the microservices tactics component) has better results than μAzimut without the microservices tactics component.

In the following paragraphs, the RQs that support this doctoral thesis will be fully answered.

RQ1
Which is the current state of the art and practice concerning architectural patterns for microservices?

A list of more than 100 microservices patterns has been identified in the academic and grey literature. These patterns cover different contexts ranging from Design to DevOps. On the other hand, research related to the use of microservices patterns reveals that there are 21 patterns used in practice. We examined 30 open-source microservices projects to identify the use of microservices patterns. On the other

hand, the results obtained indicate that only a small group of microservices patterns found in the academic and grey literature are used in open source projects.

RQ2
Which is the current state of the art and practice concerning architectural tactics for microservices?

The results of this RQ indicate that there is evidence of microservices tactics in microservices architectures, but it is not systematically structured. On the other hand, the sources that provided most evidence of microservices tactics are linked to the grey literature. Although the development and deployment of microservices-based systems use well-known architectural tactics, new microservices tactics have been identified in Availability and Scalability. This new set of microservices tactics complements the current body of knowledge on architectural tactics. Both well-known and new microservices tactics are included in μAzimut in order to generate more optimal and realistic framework assemblies.

RQ3
Which are the emerging properties that characterize quality attributes in the context of microservices architectures?

21 properties of the quality attributes used by μAzimut (availability, scalability, security and interoperability) have been characterized. These properties can drive the architect's decision to better select the framework assemblies. These properties, in turn, describe the main concerns that are reported by developers and architects regarding the development of microservices-based systems.

RQ4
Can a systematic technique, using architectural artifacts (such as properties, architectural patterns, and architectural tactics) as well as imperfect and imprecise information, be used as a method of supporting decision-making in the design of microservice architectures?

The results obtained in the two case studies and in the experiment describe that μAzimut can be used as a support tool for the architect to make decisions regarding the selection of frameworks to design microservices architectures. In the first case study, the architect's feedback describes that μAzimut is useful to initiate a starting point for deciding on the design of the architecture of the system under study. In the second case study, stakeholders were involved in deciding on the best microservice architecture design for an IoMT platform. In this case study, stakeholders evaluated three designs generated from μAzimut-generated framework assemblies. Finally, the experiment evaluated the impact on the generation of frameworks assemblies using or not microservices tactics. The experiment results describe that the microservices tactics reduce the space of solutions that the subjects must analyse. This allows subjects to select frameworks closer to what an architect would select for the same case. Furthermore, the experimental study results are a starting point for investigating the contribution of architectural tactics to the design and development of microservices-based systems.

10.1. Future work

The future work focuses primarily on investigating the relationship between μAzimut and Architectural Technical Debt (ATD) [107]. ATD is a metaphor used to describe consciously decisions taken by software architects to accomplish short-term goals but possibly negatively affecting the long-term health of the system. Moreover, ATD is primarily incurred by architectural decisions with the consequence of immature architectural artefacts. ATD commonly refers to violations of best practices, or consistency and integrity constraints of the architectures, or implementation of immature architecture techniques. To evaluate whether μAzimut positively or negatively affects the ATD of a microservices-based system, we are designing experimental studies to evaluate how μAzimut affects design and deployment decisions in microservices architecture.

Appendix A

IoMT microservices-based platform

A.1. Introduction

In the last few years, the population of adults over 60 years has presented a global expansion. In 2017, it was reported 0.96 billion of elderly people, representing an increase of 252% concerning 1980, which had a population of 0.38 billion aged adults. Furthermore, it is expected that by the year 2050, the world population reaches around 2.1 billion people over the age of 60 [72]. A significant group of older adults is forced to live alone in their homes, implying an increased likelihood of suffering distress situations related to home accidents. In this regard, falls are especially relevant to patients and health systems because approximately one-third of adults older than 65 that live in a community suffer a fall each year.

Aiming at facing current challenges concerning the care of aged patients, sensor-based technologies arises as an alternative. Using modern technologies, such as smart devices, it is possible to collect health-related sensor data from elderly patients, such as physiological, actimetric, and others, which need complex analysis and interpretation. This evaluation helps to predict possible recommendations for patients in order to detect diseases at an early stage and, in turn, enhance the quality of life for persons with chronically diseases (as was done in [132]). Moreover, if the number of persons in a

room is known, these modern technologies can be used to deduce whether the older individual is home alone. This information can be used to identify distress situations and adapt the response accordingly.

One way to address the aforementioned challenges is by using the Internet of Medical Things (IoMT) architectural paradigm. IoMT enables the gathering of different medical devices aiming at supporting health and care purposes through online networks [62]. IoMT has enabled medicine to become more accurate, supported by a deeper understanding of the patient and a constant exchange of information between patients, physicians, and medical devices.

A.2. Problem statement

Currently, the BMBI UMR 7338 laboratory from UTC through eBioMed platform is working in an innovative project related to sensors capable of capturing movement within a room to track and analyze elderly patients' daily activities aiming at preventing falls [72]. The current system that the laboratory has consists of several software components that allow the capture of real-time data from the sensors in order to detect where the patient is in the room, to capture environmental data as well as activity of daily life through sound analysis [121]. Nevertheless, a significant drawback of the current system is the reduced architectural capability for incorporating new emerging research results as well as new requirements demanded by both patients and further research guidelines.

As a solution, the laboratory's research team has decided to build an IoMT platform to satisfy new requirements that the current research demands. More precisely, the IoMT platform should address the following concerns:

- Inform the medical staff of the patient's activities within a room or department to assess risks, perform statistical calculations, among others.
- Inform the patient's relatives about the activities of this one

- Populate a database to record patient activities and movements
- Gather data from sensors and devices in order to research about preventing incidents in patients
- Increase, control, and manage the number of connected sensors.

Although the literature describes novel platforms of IoMT for different medical and healthcare purposes, one of the major drawbacks to adopting IoMT solutions is the lack of systematic processes to create them, which eventually implies a variety of solutions created for *ad-hoc* contexts. This situation directly affects the *maintainability* and *evolution* of IoMT platforms, i.e., it is challenging to perform the traceability between new requirements demanded by stakeholders that play an important role in determining the architecture of the system (hereafter, *architecturally significant requirements* (ARSs) [34]) and the technologies, devices, and networks deployed in the IoMT architecture.

A.3. Proposal

This article outlines a microservices-based IoMT platform for monitoring elderly patients aiming at preventing falls and accidents in home. The novelty of the platform is that we used the Attribute-Driven Design (ADD) [145] method to build it, and further, a microservices-based architecture supports it [104].

ADD is a systematic, step-by-step method that helps in the design of an effective architecture for software-intensive systems. The design process is based on the ASRs, which include functional requirements, quality attribute requirements, and constraints. ADD follows a recursive process that decomposes a system or system element by applying specific solutions that satisfy its driving quality attribute requirements.

On the other hand, the concept of microservices is related to an architectural style

where applications are divided into smaller and independent services. Unlike the traditional approach to applications, in which everything is compiled in one piece, microservices are independent and work together to perform the same tasks. This approach privileges the level of detail, flexibility, simplicity, and the ability to share a similar process in several applications. Furthermore, it is a fundamental component of the optimization of application development towards an IoT-based model.

A.4. Results

Figure A.1 illustrates the microservices-based IoMT platform proposed as solution. For the sake of page limitations, this section describes the main results of the artifacts required by ADD. As ADD dictates, the main step of this method is to identify the ASRs. Therefore, after brainstorming sessions with the principal researchers of the laboratory (hereafter, stakeholders), Table A.1 describes the principal ASRs to be satisfied. On the other hand, Table A.2 describes the main microservices of the platform. We used the concepts and the meta-processes for designing software of Domain-Driven Design [51] to identify and describe the microservices.

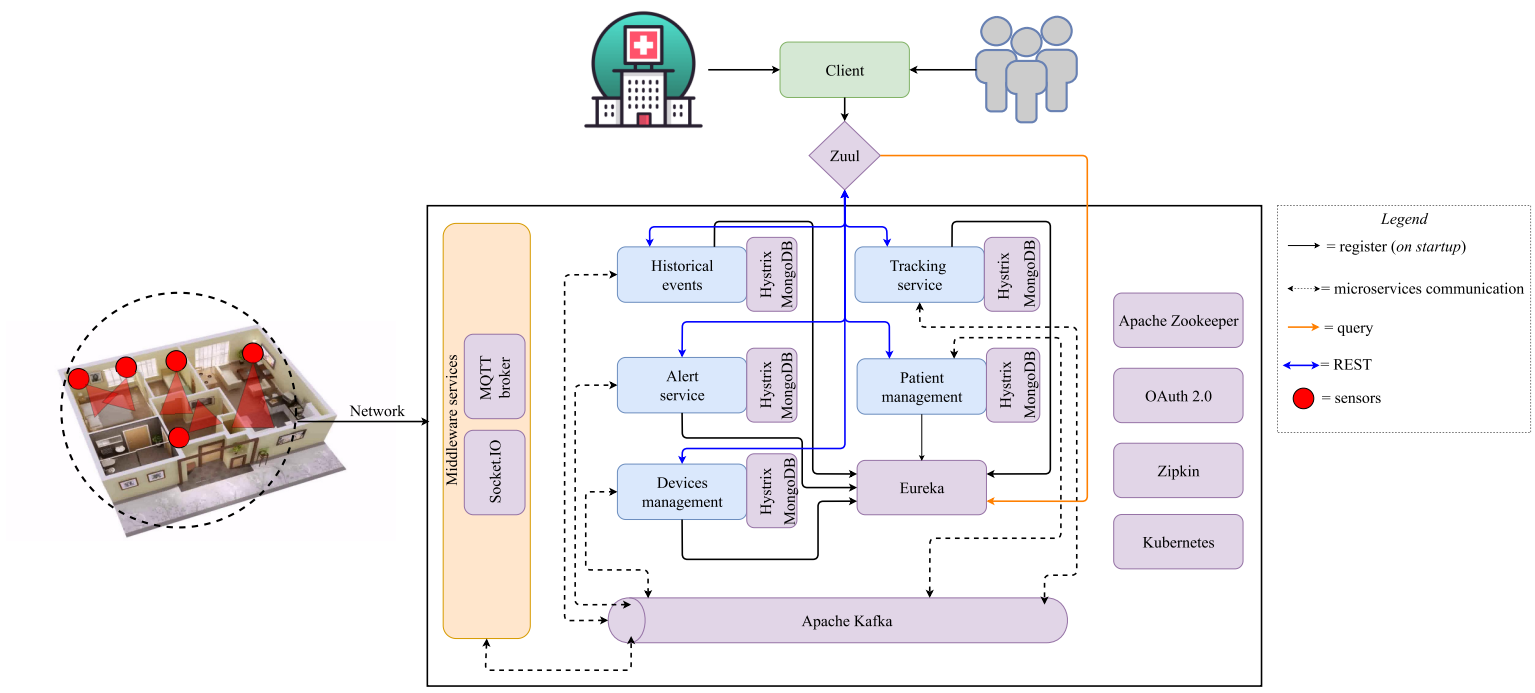


Figure A.1: The microservices-based IoMT platform

Table A.1: Architecturally Significant Requirements

ID	Property	High-level ASR description
ASR1	*Availability*	The services must be highly available in order to be able to access them when real-time information about the patient's movements is needed. Also, they must be available when there are modifications to the sensors. These modifications depend on the needs of the patient and/or geographic environments.
ASR2	*Scalability*	The platform must adapt to the changes that patients need without losing quality. This is the reason why each service must have the ability to grow and handle large volumes of information.
ASR3	*Interoperability*	The platform must have the ability to allow new software and/or devices to be incorporated quickly and efficiently. Furthermore, they must have the ability to exchange information and, at the same time, use the information that has been incorporated.
ASR4	*Security*	Since the platform will use data related to patients, there are security and privacy regulations that the platform must satisfy. In addition, being an IoT platform, security mechanisms must be implemented in order to prevent attacks and threats. Furthermore, security mechanisms produce trust in elderly patients regarding the platform.
ASR5	*Maintainability*	The platform must meet highly changing requirements. That is why the platform must have the ability to be modified effectively and efficiently in order to be improved whenever necessary or adapt to changes in the environment.

A.4.1. Architectural drivers and capabilities

Stakeholders manifested that the network access protocol module must implement an interface for the handling of different communication protocols (such as IEEE 802.15.x and 802.11). Furthermore, the connectivity protocol module will use lightweight published/subscribed messaging transport. Publish/subscribe messaging, or pub/sub messaging, is a method of asynchronous service-to-service communication used in serverless and microservices architectures. In a pub/sub model, any message published to a topic is immediately received by all of the subscribers to the topic. Pub/sub messaging can be used to decouple applications in order to increase performance, reliability, and scalability.

On the other hand, interoperability, subscription, notification, and command execution are operations performed in the middleware layer. This layer offers an abstraction to handle connected objects. This set of services is critical to be able to integrate different devices into the platform.

Table A.2: IoMT platform microservices

Microservice	**Description**
Historical events	This service intends to provide information and analysis regarding accidents or incidents in order to discover all the history of an accident so that it can be avoided. Furthermore, the idea of this service is to identify improvements aimed at preventing or mitigating possible accidents in elderly patients. This service also allows the identification of hazards in order to be evaluated and classified.
Alert service	This service aims to generate the corresponding alerts to warn the medical staff, family, or other interested parties. This service should eventually interoperate with other systems.
Devices management	This service processes all the information captured by sensors in the room or house. In addition, this service is not limited to sensors; it can be extended to other devices.
Tracking service	The primary purpose of this service is to provide information to investigate the activities of elderly patients in the rooms.
Patient management	This service manages the different profiles of elderly patients. It uses information from other health institutions that allow it to characterize the patient and his health conditions (for example, patients who, due to health conditions, regularly attend the bathroom).
Middleware services	These services usually provide an abstraction layer for devices. Thus, they are able to ensure data transfer between such services and therefore achieve interoperability.

The platform has heterogeneous databases; they are spatial, relational, and document databases. Therefore, there is no unifying schema in a central database. Microservices architectures are distributed systems with decentralized data management. This means that every service has its own independent storage subsystem that is isolated from other services. In addition, the platform has an asynchronous event-driven, which is responsible for the web application server for presenting information and notifications in a client's application. It communicates with the rest API by PUT, GET, POST and DELETE commands.

Since the platform is based on IoMT, this concept points to the collection of medical devices and applications that connect healthcare systems through networks. Therefore, the main capabilities of the platform are the following:

Provide real-time data to monitor the patient's health status This capacity is relevant since the data that the platform will receive comes from patients. This means that this real-time communication must ensure that the integrity of the data is accurate. Furthermore, security and privacy aspects must also be taken into account

so that the patient can trust on the devices' functionalities.

Integration with other devices The current situation of the laboratory describes that there is previous work with the use of sensors. The sensors can send data from the environment so that they can be studied and apply novel techniques to prevent and improve patient care. Nevertheless, IoMT also extends the capacity of platforms to the integration of data from medical devices and portable devices using appropriate middleware services.

Alert family members or medical staff when there are abnormalities in the patient's daily behavior The platform must have the ability to integrate all the necessary services in order to alert and inform family members or medical staff when in life-threatening circumstances. For this, procedures for checking the status of services in small time batches must be applied in order to monitor the current status of the services.

A.4.2. Architectural background

The platform uses a set of architectural patterns in order to reuse solutions that have been proposed to recurrent problems that arise in the development and deployment of both IoMT platforms and microservices architectures. In the context of microservices, several patterns help solve microservice architectural problems [90]. The following Table A.3 describes some patterns used to develop the platform.

Table A.3: Partial list of architectural patterns used in the platform

ID	Pattern	Context	Problem	Solution	ASR
P1	Service Registry	The platform has different microservices comprising the application	Decouple the physical address (IP address) of a microservice instance from its clients so that the client code will not have to change if the address of the service changes	Use a Service Registry to map between a unique identifier and the current address of a service instance in order to decouple the physical address of service from the identifier	ASR1
P2	Service Discovery	Each service has one or more instances deployed in the production environment. The number of instances can be changed dynamically, and each of them can be deployed in different places.	Locate services each other dynamically	Setup a Service Discovery which stores service instances addressed. Each service registers itself during initiation.	ASR5
P3	Load Balancer	It is required to scale the client and microservices independently of each other.	A client wants to interact with a microservices-based application without knowing the instances that are serving it	A Load Balancer distributes a workload on a set of equal services. The circuit breaker enables the load balancer to put work only on services that are in a good state of health.	ASR2

An application framework (commonly known as just a *framework*) is a collection of reusable software elements that provide generic functionality addressing recurring domain and quality attribute concerns across a broad range of applications. Furthermore, they are *building-blocks* of both IoT and IoMT platforms. The following Table A.4 shows partial content of frameworks and tools used to develop and deploy the platform.

In turn, the platform is developed and deployed using Spring Boot[1], because its facility for the development of applications and platforms based on microservices.

[1]https://spring.io/projects/spring-boot

Table A.4: Partial list of frameworks (F) and tools (T) used in the platform

ID	Name	Description	Pattern	ASR
F1	Netflix Eureka	Service registry for resilient mid-tier load balancing and failover.	P1	ASR1, ASR2
T2	Kubernetes	System for automating deployment, scaling, and management of containerized application.	P2	ASR5
F3	Netflix Hystrix	Latency and fault tolerance library designed to isolate points of access to distributed systems.	P3	ASR1

A.4.3. Services description

Since the platform is based on a microservices architecture, ADD prompts to document each architectural artifact (in our case, services) in order to highlight the description of each service as well as the dependencies between them. For explanatory purposes, Table A.5 shows the description of the Devices management service. We defined "invoke" as the operations, which are implemented by other services that this service invokes. On the other hand, we defined "subscribe to" as the messages, which includes events, that this service subscribes to.

Table A.5: Services documentation

Name	Devices management
Description	See Table A.2
Dependencies	
Invokes	**Subscribe to**
Middleware services	▪ Historical events ▪ Alert service ▪ Tracking system

A.5. Conclusions and future work

We have presented a microservices-based IoMT platform that aims to monitor elderly patients in order to prevent falls and incidents. The main features of this platform are: (i) it uses a middleware layer for the integration of software, API management, and event processing; (ii) it uses the publish/subscribe messaging protocol; and (iii) it has the ability of monitoring services performance and APIs. Furthermore, the platform is supported by the microservices architectural style, which structures systems as a collection of services that are highly maintainable, loosely coupled, independently deployable, and others.

To further our research we are planning to include new features to the platform in order to encompass more qualities that will prevent falls of elderly patients. More precisely, we are including novel techniques of machine learning and artificial intelligence to specific services aiming at improving the accuracy and sense of data.

Appendix B

Salary Payment System

B.1. Introduction

Today, there is a process that is widely used by companies, which is the process of payment of salaries. Companies have their own information systems that allow them to calculate according to their needs. But, many times, these systems are expensive, too complicated, have too many options that hinder the work instead of automating it or have a minimal performance that does not cover all the needs of the company. For this reason, and in conjunction with the company ABC, it is required to develop a remuneration system that facilitates the work and minimizes the processing times for the calculation of remuneration and allows easy communication between the system and the PreviRed portal. For the calculation of salary, it is necessary to take into account different data such as minimum salary, the value of support units, social security values, and tax ranges.

Due to the payment process is currently too time-consuming to collect (i) the information, (ii) work on assigning concepts per worker, (iii) process the information, and then (iv) upload this data to the PreviRed portal, it can be seen that the current system produces a considerable expense of time to perform this process. For these reasons, the Salary Payment System (SPS) was created to minimize the time of the

tasks in the complete process of calculating salaries. The system includes the collection of social security, tax and legal data from the web, the assignment of concepts by groups of workers and also the delivery of files that allow the entry of information in a consolidated manner to the PreviRed portal.

The main goal of this systems is to provide quality software that allows the calculation of salaries in an automated and dynamic way. The system must be capable of automating the capture of data so that the closing and opening of a new month can be expedited. Furthermore, the system must have a database that can be accessed from any computer that contains the system. Also, it should be capable of entering the various contracts that exist and which can perform a data review process that, in case of error, should not make the entire process again, but should only modify the erroneous data of the employee.

B.2. Architectural foundations

The main NFRs of the system are:

- *Usability*: The system must be easy to use, where the interfaces are intuitive so that it can be used by people who do not have advanced computer knowledge. Besides being able to be used by people who are not so internalized or are new to the system.
- *Security*: It must have the necessary authorization through login in order to enter the system and make changes.
- *Scalability*: The system must be capable of adapting to new requirements and must not directly affect the operation or hinder the calculation of salaries.
- *Availability*: The system must be able to deliver the information and distribute the necessary reports when required.

- *Potability*: The system must be capable of running on various computer operating systems.

The high-level processes, Class Diagram and Data Model are described in Figures B.1, B.2, and B.3.

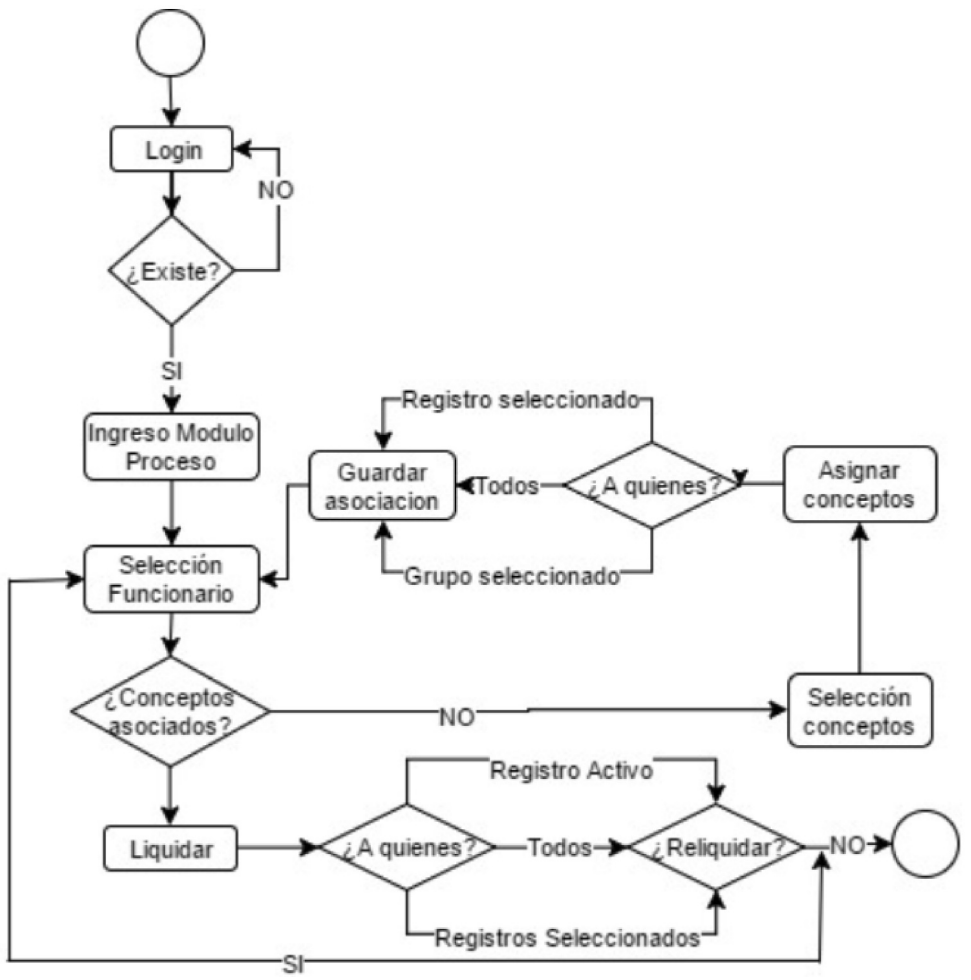


Figure B.1: High-level process of SPS (in Spanish)

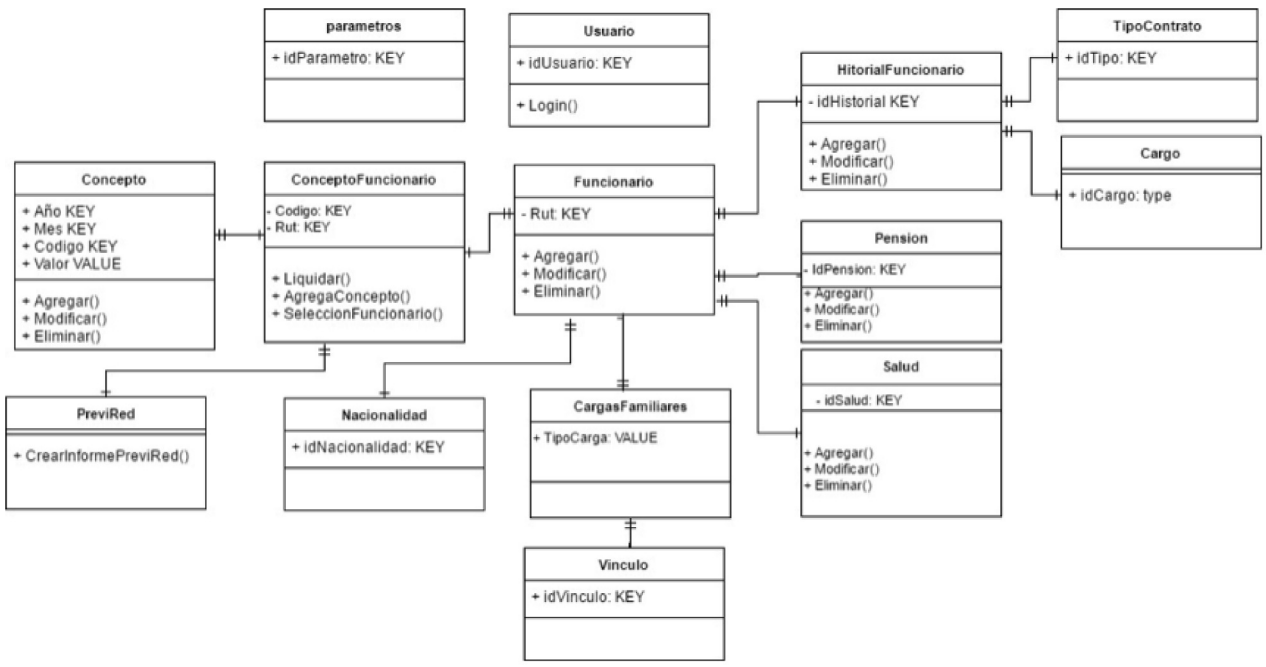


Figure B.2: High-level Class Diagram (in Spanish)

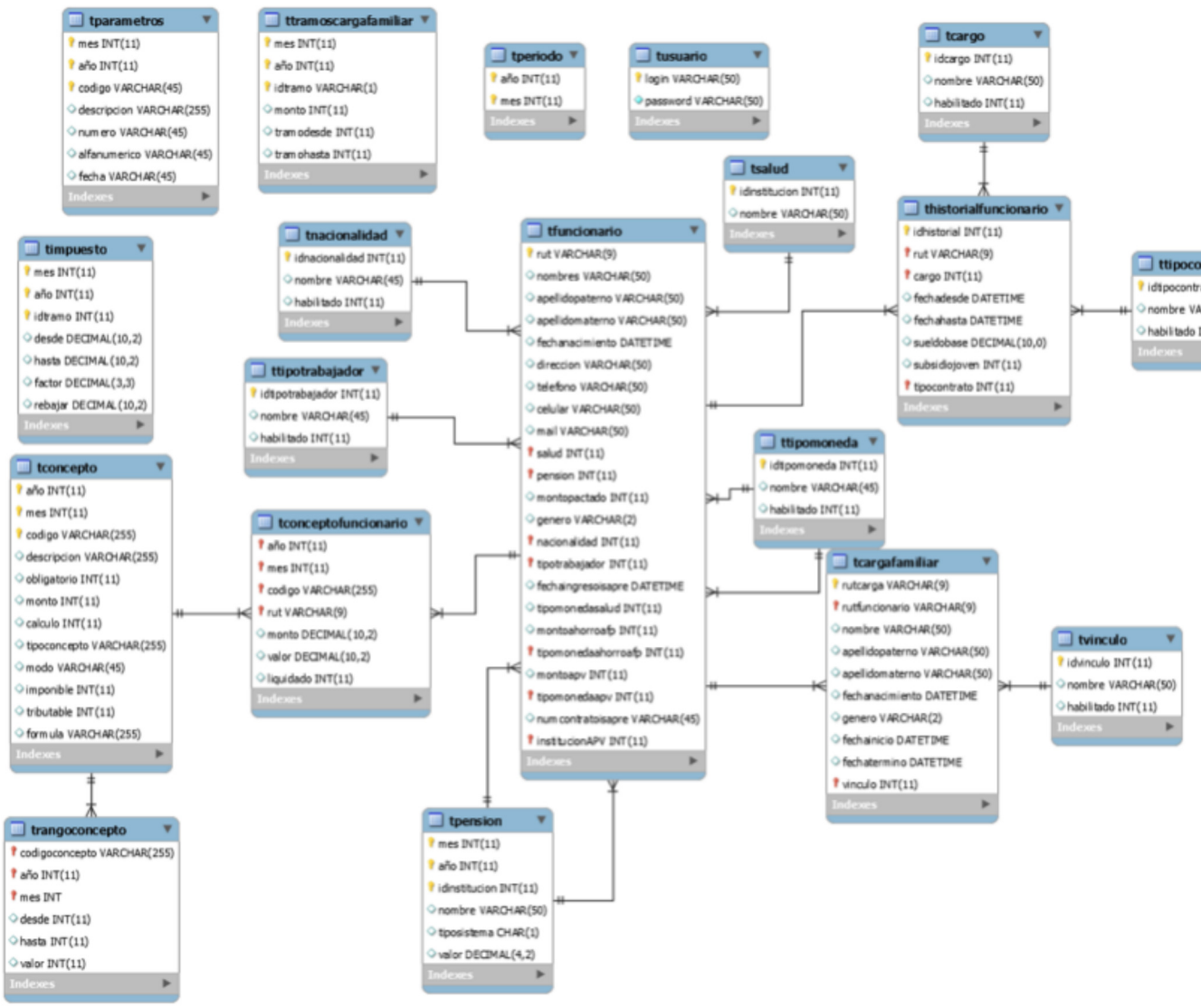


Figure B.3: High-level Data Model (in Spanish)

Appendix C

Architectural patterns for microservices

This appendix describes the architectural patterns mentioned in Chapter 3. The architectural patterns are categorized by name, context, problem and solution.

ID	Name	Context	Problem	Solution
MSP1	*Modern Web Architecture*	Application that needs to provide a rich user interface	Capabilities of the different interface devices?	Combine front-end components with back-end components
MSP	*Single Page Application*	Develop new application web or refactoring a section a web	Take advantage of the capabilities of modern browsers?	Design application as single page
MSP3	*Native Mobile Application*	Building applications that need to support on a variety of platforms	Provide the most optimized user experience?	Native mobile application for each major platforms
MSP4	*Near Cache*	App. must operate efficiently when Internet connectivity is bad	Reduce number of calls ?	Use a near cache located within the client implementation
MSP5	*Microservices Architecture*	App. to be modular, and it want the modules to be independent	Architect an application as a set of independent modules?	Design with modules independent from business function
MSP6	*Business Microservice*	App. using the microservices architecture style or refactoring	Manage business logic?	Develop each business function as a Business Microservice
MSP7	*Backend for Frontend*	No map cleanly to the channel-specific needs of your client	Represent a channel-specific service interface?	Build a backend for frontend that acts as a single API
MSP8	*Adapter Microservice*	Incorporate services but APIs do not use the RESTful approach	Handle translation into good microservices APIs?	Converts API that client microservies will expect
MSP9	*Results Cache*	Making network calls to remote databases or services are expensive	Improve the performance in repeated calls to services?	Shortcuts the need to make repeated calls to the same service
MSP10	*Page Cache*	Backend for frontends to build dispatchers for a application	Return more information easily displayed?	Use a page cache with your backend for frontend
MSP11	*Scalable Store*	Need persistent state to represent user interaction	Represent persistent state in an application?	States in a scalable store
MSP12	*Key Value Store*	You are building an application with a microservices architecture	Store data accessed through a simple key?	Store your data in a scalable key-value store
MSP13	*Document Store*	Representing the contents of HTTP as JSON docs	Efficiently store and retrieve HTTP data responses?	Store your JSON documents in a data store
MSP14	*Microservices DevOps*	Complex application with a microservices architecture	Balance the needs of developers and operations teams?	Isolate each Microservice as much as possible
MSP15	*Log Aggregator*	Debug problems that cross different components of that architecture	Effectively view and search all of the different log?	Pulls all of the files into a single searchable database
MSP16	*Correlation ID*	Application use backends for frontends having complex call graphs	Debug a complex call graph?	Identifier passed in to service request
MSP17	*Service Registry*	Many different business microservices comprising the application	Decouple the physical address of a microservice instance?	Map unique identifier and the current address
MSP18	*Enable Continuous Integration*	Team has decided to migrate the system to microservices	Have available production-ready artifacts?	Set up the continuous integration
MSP19	*Recover the Current Architecture*	Team need to know the current system architecture	What is the big picture of the system?	Keep the artifacts as simple as possible
MSP20	*Decompose the Monolith*	Monolithic software system with a complex domain	Decompose the system into smaller chunks?	Domain-Driven Design (DDD) approach
MSP21	*Decom. based on data ownership*	Monolithic software system with not a complex domain	How big these chunks should be?	Decompose the system based on the ownership of data
MSP22	*Change code depen. to service call*	There is a component in the system that is acting as a dependency	Change the code-level dependency?	Try to keep the services code as separate as possible

Figure C.1: Architectural patterns for microservices (Part I)

ID	Name	Context	Problem	Solution
MSP23	*Intro. service discovery*	The number of instances can be changed dynamically	Locate each other dynamically?	Store service instances addresses
MSP24	*Intro. service discovery client*	The number of instances can be changed dynamically	Knows a new instance has been deployed?	Know the address the service discovery
MSP25	*Intro. internal load balancer*	Each service can be a client of the rest of the services	Balance the load on a service?	Have an internal load balancer
MSP26	*Intro. external load balancer*	Each service can be a client of the rest of the services	Balance the load on a service between its instances?	Retrieves the list of available instances
MSP27	*Intro. circuit breaker*	End-user requests need inter-service communication	Call when calling unavailable service?	Available component would not do anything
MSP28	*Intro. config. server*	List of available instances is available through a service discovery	Modify running instances configuration?	Two separate repositories
MSP29	*Intro. edge server*	Because of service initiation, new services can be introduced easily	Hide the internal service complexity from end-users?	Another layer of indirection in the system
MSP30	*Containerize the Services*	Continuous Integration pipeline is in place and is working	Produce the same results for the same code?	Service in a virtual machine in isolation
MSP31	*Deploy Cluster and Oreh. Cont.*	A production-ready container image is available for the deployment	Deploy a service's instances into a cluster?	Manage a cluster of computing nodes
MSP32	*Monitor the sys. and provide Feed.*	System ran on a cluster of containers with instances in production	Monitor the underlying infrastructure?	Independent monitoring facility
MSP33	*Self-containment of services*	The property of selfcontainment is one of the core aspects	Push software to the limits of maintainability and scalability?	Back-end as part of the service
MSP34	*Circuit Breaker*	Each service provide an interface to hand over monitoring info.	Monitoring and prevention of fault cascading?	Checks health status or remember unsuccessful calls
MSP35	*Load Balancer*	Each service provide an interface to hand over monitoring info.	Monitoring and prevention of fault cascading distributes?	Distributed workload on a set of equal services
MSP36	*Container*	Enclose the microservice itself, including required libraries and data	Simplify deploying applications?	Load balancer workload
MSP37	*Handling diff. service versions*	After an application was tested the artifact is not altered anymore	Not alter specific artifacts?	Include new versions of each services
MSP38	*Blue green deployment*	Replace applications by new versions	Not alter specific artifacts and put into operation?	The old and new versions have to run in parallel
MSP39	*Canary release*	Challenges with automating deployment is the cut-over itself	Not alter specific artifacts?	Route the new services iteratively
MSP40	*Database is the service*	New behaviors at the database level may be reduce complexity	Improvements in terms of speed and scalability?	Addition of new behaviors at the database level
MSP41	*IoT edge provisioning*	Devices scattered geographicall, to reach and large in number	Edge devices as reliable baseline environment?	Container-based virtualization
MSP42	*IoT edge code deployment*	Maintainability in remote IoT devices	Deploy their code to many IoT devices?	Utilize version control systems for deployments
MSP43	*IoT edge orchestration*	Enabling a large number of devices connected via edge layer	Orchestrate IoT devices?	Service discovery mechanisms
MSP44	*IoT edge diameter of things (DoT)*	Mechanisms can vary based on applied business models	Can IoT service provider monitor?	Light-weight protocol for telemetry of IoT applications

Figure C.2: Architectural patterns for microservices (Part II)

ID	Name	Context	Problem	Solution
IMSP1	*Decompose by business capability*	*Not specified*	Decompose an application into services?	Define services corresponding to business capabilities
IMSP2	*Decompose by subdomain*			Define services corresponding t o DDD subdomains
IMSP3	*Multiple service instances per host*	Establish systems as a set of services	How to deploy an application's services?	Deploy multiple service instances on a single host
IMSP4	*Service instance per host*			Deploy each service instance in its own host
IMSP5	*Service instance per VM*			Deploy each service instance in its VM
IMSP6	*Service instance per Container*			Deploy each service instance in its container
IMSP7	*Serverless deployment*			Deploy a service using serverless deployment platform
IMSP8	*Service deployment platform*			Deploy services using a highly automated deployment
IMSP9	*Microservice chassis*	Development spend a significant amount of time	Handle cross cutting concerns?	Framework that handles cross-cutting concerns
IMSP10	*Externalized configuration*	An application typically uses one or more infrastructure		Externalize all configuration such as database location
IMSP11	*Remote Procedure Invocation*	Services must handle requests from the application's clients.	Services use to communicate?	RPI-based protocol for inter-service communication
IMSP12	*Messaging*			Asynchronous messaging for inter-service communication
IMSP13	*Domain-specific protocol*			Use a domain-specific protocol
IMSP14	*API gateway*	Develop multiple versions of the product details about user inteface	External clients communicate with the services?	Service that provides unified interface to services
IMSP15	*Backend for front-end*			A separate API gateway for each kind of client
IMSP16	*Client-side discovery*	Virtualized or containerized environments	Discover the location of a service instance?	Client queries a service registry
IMSP17	*Server-side discovery*			Router queries to discover the locations of service instances
IMSP18	*Service registry*	Determine the location of a service instance to which to send requests	Locate a service instance?	A database of service instance locations
IMSP19	*Self registration*	Register service so that they can be discovered on shutdown	Service instances registered?	Service instance registers itself with the service registry
IMSP20	*Third party registration*			Third party registers a service instance
IMSP21	*Circuit Breaker*	Services somotimes collaborate when handling requests.	Prevent failure from cascading to other services?	Invoke a remote service via a proxy that fails immediately
IMSP22	*Database per Service*	Most services need to persist data in some kind of database	Database architecture in a microservices application?	Each service has its own private database

Figure C.3: Architectural patterns for microservices (Part III)

ID	Name	Context	Problem	Solution
IMSP23	*Shared database*	Most services need to persist data in some kind of database	Database architecture in a microservices application?	Services share a database
IMSP24	*Saga*	Each service has its own database	How to maintain data consistency across services?	Sequences of local transactions
IMSP25	*API Composition*	No longer to implement queries that join data from multiple services	Implement queries in a microservice architecture?	Implement queries by invoking the services
IMSP26	*CQRS*	The data is no longer easily queried		Implement queries by maintaining one or more views
IMSP27	*Event sourcing*	Services must atomically publish events	Atomically publish events whenever state changes?	Persist aggregates as a sequence of events
IMSP28	*Transaction log tailing*			Publish changes captured in the database's transaction
IMSP29	*Database triggers*			Use triggers to capture changes made to data
IMSP30	*Application events*			Application inserts events into a database table
IMSP31	*Access Token*	Verify that a user is authorized to perform an operation	Communicate the identity of the requestor?	A token that securely stores information about user
IMSP32	*Service Component Test*	The application consists of numerous services	How to make testing easier?	A test suite that tests a service in isolation
IMSP33	*Service Integration Contract Test*			A test suite for a service that is written by the developers
IMSP34	*Log aggregation*	Multiple instances that are running on multiple machines	Understand the behavior and troubleshoot problems?	Aggregate application logs
IMSP35	*Application metrics*	Any solution should have minimal runtime overhead		Instrument a service's code to gather statistics
IMSP36	*Audit logging*	Know what actions a user has recently performed		Record user activity in a database
IMSP37	*Distributed tracing*	Requests often span multiple services		Services assigns each external request as unique identifier
IMSP38	*Exception tracking*	Errors sometimes occur when handling requests		Report all exceptions to a centralized exception tracking
IMSP39	*Health check API*	The monitoring system should generate a alert		Service API that returns the health of the service
IMSP40	*Log deployments and changes*	Deployments and other changes occur since issues usually occur		*Not specified*
IMSP41	*Server-side page fragment compo.*	Some UI screens/pages display data from multiple service	Implement a UI screen that displays data from services?	Build a webpage on the server by HTML fragments
IMSP42	*Client-side UI composition*			Build a UI on the client by composing UI fragments
IMSP43	*Cloudify Service-Proxy*	Description of domain model of a particular microservice	How to uses the service input/output as a generic way?	A user can describe an entire service as a single node type
IMSP44	*Fine-Grained SOA*	The point of each service is to provide connectivity to external systems	Coarse-grained services are too difficult to change	Break up services into finer-grained pieces

Figure C.4: Architectural patterns for microservices (Part IV)

ID	Name	Context	Problem	Solution
IMSP45	*Layered APIs*	Manage multiple layers of a large number of fine-grained things	Some structures are difficult to rationalize and reason	Create layers of microservices that are grouped by purpose
IMSP46	*Message-Oriented*	Avoid the side-effects of accessing and mutating state	Need to replicate the state of key business data	Using a message queue allows state to be asynchronously
IMSP47	*Event-driven*	Services receiving it to reconstruct a materialized view	Need to replicate key business events	Use a common event abstraction
IMSP48	*Isolating State*	Coalesce the internal consistency of each microservice	Difficult to achieve data integrity	Microservice that represents the single source of truth
IMSP49	*Replicating State*	Consistency is required	Achieve data integrity when there are multiple sources	Keep a single source of truth of all changes to data
IMSP50	*Aggregator Microservice D. P.*	Some services are expose using a lightweight REST mechanism	Compose them to provide the application's functionality?	Simple web page that invokes multiple services
IMSP51	*Proxy Microservice D. P.*	Microservice may be invoked based upon the business need		The web page can retrieve the data and process
IMSP52	*Chained Microservice D. P.*	Produce a single consolidated response to the request		The request from the client is received by different services
IMSP53	*Branch Microservice D. P.*	Simultaneous response processing from two chains of microservices		Service can invoke two different chains concurrently
IMSP54	*Shared Data Microservice D. P.*	Some applications may benefit from a shared data		May share caching and database stores
IMSP55	*Asyn. Messaging Microservice D. P.*	Microservice architectures may elect to use message queues instead of REST		Shared message queue
IMSP56	*Back-end for Front-end*	Work in differents API, is quicker as it required no coordination	How to improve development?	Have different APIs for mobile and web
IMSP57	*Pseudo-URIs*	Good identifiers across our hundreds of microservices	Implement identities for objects?	Use as identifiers that would contain information of objects
IMSP58	*Service Mesh*	A service talks to another to accomplish some goal for an end-users	Extract the features required by distributed services?	Moving away from a set of independent proxies to a proper
IMSP59	*Database Sharing*	Sharing data across multiple services	Synchronize information between microservices?	Keep the databases in sync
IMSP60	*REST Integration*	Intuitive way of integrating microservices		Dedicated REST endpoint
IMSP61	*Messaging*	Decouple the caller and callee services		Publish an event to the message broker
IMSP62	*Throttling*	Avoid any misbehaving or rouge application	Achieve performance, resilience and stability?	Implement throttling is by providing fixed number of connections
IMSP63	*Timeouts*	Application to take longer time to complete a request		Set timeouts for read requests/ write requests/wait
IMSP64	*Dedicated Thread Pools/Bulkheads*	Have separate dedicated thread pools		Have dedicated pool for individual service
IMSP65	*Circuit Breakers*	Minimize the impact of any of the downstream being not accessible		Check the availability of external systems/services
IMSP66	*Ambassador*	Common client connectivity tasks such as monitoring	*Not specified*	Create helper services that send network requests

Figure C.5: Architectural patterns for microservices (Part V)

ID	Name	Context	Problem	Solution
IMSP67	*Anti-Corruption Layer*	Implement a facade or adapter layer	*Not specified*	Translates requests between modern application and legacy system
IMSP68	*Backends for Frontends*	Avoid customizing a single backend for multiple interfaces		Create separate backend services to be consumed
IMSP69	*Bulkhead*	*Not specified*		Isolate elements of an application into pools so that if one fails
IMSP70	*Gateway Aggregation*	Make multiple calls to different backend systems to perform an operation		Use a gateway to aggregate multiple individual requests
IMSP71	*Gateway Offloading*	Simplify application development by moving shared service functionality		Specialized service functionality to a gateway proxy
IMSP72	*Gateway Routing*	Expose multiple services on a single endpoint		Route requests to multiple services using a single endpoint
IMSP73	*Sidecar*	Services often require related functionality		Deploy components of an application into a separate process
IMSP74	*Strangler*	Services can become increasingly obsolete		Incrementally migrate a legacy system
IMSP75	*Decomposition by Use Case*	Logical continuation of a large-scale decomposition	*Not specified*	Identify use cases
IMSP76	*Decomposition by Resources*	Create a set of microservices which function as channels		Define microservices based on the resources
IMSP77	*Decomposition by Responsibilities/Functions*	Defined set of functions		Define terms of simple functions
IMSP78	*Single Host/Multiple Services*	Deploy multiple instances of a service on a single host		Reduces deployment overhead
IMSP79	*Single Service per Host, VM, or Cont.*	Deploys each service in its own environment		Environment will be a virtual machine or container
IMSP80	*Serverless/Abstracted Platform*	The service runs directly on pre-configured infrastructure		Make use of the microservice is its address

Figure C.6: Architectural patterns for microservices (Part VI)

Bibliography

[1] *Discussing Docker. Pros and Cons.* https://blog.philipphauer.de/discussing-docker-pros-and-cons/. [Online; accessed April 5th, 2019].

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

[3] Aderaldo, M., C. Mendonça, C. Pahl y J. Pooyan: *Benchmark requirements for microservices architecture research*. In Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE'17), páginas 8–13, 2017.

[4] Alexander, C., S. Ishikawa, M. Silverstein, J. R. Ramió, M. Jacobson y I. Fiksdahl-King: *A pattern language*. University of California, Berkeley, 1977.

[5] Ali, S., M. A. Jarwar y I. Chong: *Design methodology of microservices to support predictive analytics for IoT application*. Sensors, 18(12):4226, 2018. doi:https://doi.org/10.3390/s18124226.

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

[7] Alshuqayran, Nuha, Nour Ali y Roger Evans: *Towards Micro Service Architecture Recovery: An Empirical Study*. IEEE International Conference On Software Architecture, páginas 47–4709, 2018.

[8] Anderson, C.: *Docker [software engineering]*. IEEE Software, 32(3):102–c3, 2015.

[9] Apache: *Apache Maven project*, 2018. https://maven.apache.org/.

[10] Astudillo, H., J. Pereira y C. López: *Evaluating alternative COTS assemblies from unreliable information*. QoSA 2006: Third International Workshop on Quality of Software Architecture, 2006.

[11] Aurum, A. y C. Wohlin: *The fundamental nature of requirements engineering activities as a decision-making process.* Information and Software Technology, 45(14):945–954, 2003. doi:https://doi.org/10.1016/S0950-5849(03)00096-X.

[12] B., Christudas: *High Availability and Microservices. In: Practical Microservices Architectural Patterns.* Apress, Berkeley, CA, 2019. doi:https://doi.org/10.1007/978-1-4842-4501-9_9.

[13] Baeza-Yates, R. y B. D. A. N. Ribeiro: *Modern information retrieval.* New York: ACM Press; Harlow, England: Addison-Wesley, 2011.

[14] Baitalmal, Ahmad: *Good Architecture/Bad Architecture*, 2017. https://hackernoon.com/good-architecture-bad-architecture-1a0fc1e9cf01.

[15] Balalaie, A., A. Heydarnoori y P. Jamshidi: *Microservices Migration Patterns.* Technical Report No. 1, TR- SUT-CE-ASE-2015-01, Automated Software Engineering Group, Sharif University of Technology, 2015.

[16] Balalaie, A., A. Heydarnoori y P. Jamshidi: *Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture.* IEEE Software, 33(3):42–52, 2016.

[17] Balalaie, A., A. Heydarnoori, P. Jamshidi, D. A. Tamburri y T. Lynn: *Microservices Migration Patterns.* Software: Practice and Experience, 48(11):2019–2042, 2018.

[18] Baresi, Luciano, Martin Garriga y Alan De Renzis: *Microservices Identification Through Interface Analysis.* Service-Oriented and Cloud Computing, (19–33), 2017.

[19] Basili, V. R.: *Software modeling and measurement: the Goal/Question/Metric paradigm.* 1992.

[20] Bass, L., I. Weber y L. Zhu: *DevOps: A Software Architect's Perspective.* Addison-Wesley Professional, 2015.

[21] Bass, Len, Paul Clements y Rick Kazman: *Software Architecture in Practice (3rd Edition).* SEI Series in Software Engineering, 2013.

[22] Bosch, J.: *Software architecture: The next step.* European Workshop on Software Architecture, 2004. doi:https://doi.org/10.1007/978-3-540-24769-2_14.

[23] Bourque, P. y R. E. Fairley: *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0.* IEEE Computer Society Press, 2014.

[24] Brown, K. y B. Woolf: *Implementation Patterns of Microservices Architectures.* Conference on Pattern Languages of Programs (PLoP), páginas 7:1–7:35, 2016.

[25] Bucchiarone, A., N. Dragoni, S. Dustdar, S. T. Larsen y M. Mazzara: *From monolithic to microservices: an experience report from the banking domain*. IEEE Software, 35(3):50–55, 2018.

[26] Butzin, B., F. Golatowski y D. Timmermann: *Microservices approach for the internet of things*. IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), páginas 1–6, 2016.

[27] Calçado, P.: *Patterns of Microservices Architecture*. `http://philcalcado.com/microservices-patterns.html`. [Online; accessed April 5th, 2019].

[28] Cartaxo, B., G. Pinto y S. Soares: *The Role of Rapid Reviews in Supporting Decision-Making in Software Engineering Practice*. EASE, páginas 24–34, 2018.

[29] Cartaxo, B., Pinto G. y S. Soares: *The Role of Rapid Reviews in Supporting Decision-Making in Software Engineering Practice*. Proceedings of the 22Nd International Conference on Evaluation and Assessment in Software Engineering 2018, páginas 24–34, 2018.

[30] Casalicchio, E. y V. Perciballi: *Measuring docker performance: What a mess!!!* Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, páginas 11–16, 2017.

[31] Cervantes, H., R. Kazman, J. Ryoo, J. Cho, G. Cho, H. Kim y J. Kang: *Data-driven Selection of Security Application Frameworks During Architectural Design*. Proceedings of the 52nd Hawaii International Conference on System Sciences, 2019. url:`http://hdl.handle.net/10125/60170`.

[32] Cervantes, Humberto: *Conceptos de Diseño Patrones, Tácticas y Frameworks*, 2012. `https://sg.com.mx/revista/38/conceptos-dise~no-patrones-t\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{a\global\mathchardef\accent@spacefactor\spacefactor}\accent19a\egroup\spacefactor\accent@spacefactor\futurelet\@let@token\protect\penalty\@M\hskip\z@skipcticas-y-frameworks`.

[33] Chen, H. M., R. Kazman, S. Haziyev, V. Kropov y D. Chtchourov: *Architectural Support for DevOps in a Neo-Metropolis BDaaS Platform*. IEEE 34th Symposium on Reliable Distributed Systems Workshop (SRDSW), páginas 25–30, 2015.

[34] Chen, L., M. A. Babar y B. Nuseibeh: *Characterizing architecturally significant requirements*. IEEE software, 30(2):38–45, 2012.

[35] Cherradi, G., A. E. Bouziri, A. Boulmakoul y K. Zeitouni: *Real-time micro-services based environmental sensors system for Hazmat transportation networks monitoring*. Transportation Research Procedia, 27, 873-880. doi:`https://doi.org/10.1016/j.trpro.2017.12.087`.

[36] Chung, L., B. A. Nixon, E. Yu y J Mylopoulos: *Non-functional requirements in software engineering*, volumen 5. Springer Science and Business Media., 2000.

[37] Churchman, M.: *Top Patterns for Building a Successful Microservices Architecture.* https://www.sumologic.com/blog/devops/top-patterns-building-successful-microservices-architecture/. [Online; accessed April 5th, 2019].

[38] Cysneiros, L. M. y E. Yu: *Non-functional requirements elicitation.* Perspectives on software requirements, páginas 115–138, 2004.

[39] Dabbagh, M. y S. P. Lee: *A consistent approach for prioritizing system quality attributes.* 4th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, páginas 317–322, 2013. doi:10.1109/SNPD.2013.9.

[40] Dabbagh, M., S. P. Lee y R. M. Parizi: *Functional and non-functional requirements prioritization: empirical evaluation of IPA, AHP-based, and HAM-based approaches.* Soft Computing, 20(11):4497–4520, 2016. doi:https://doi.org/10.1007/s00500-015-1760-z.

[41] Dhall, R.: *Performance Patterns in Microservices based Integrations.* https://www.computer.org/web/the-clear-cloud/content?g=7477973&type=blogpost&urlTitle=performance-patterns-in-microservices-based-integrations. [Online; accessed April 5th, 2019].

[42] Di Francesco, P., P. Lago y I. Malavolta: *Architecting with microservices: A systematic mapping study.* Journal of Systems and Software, 150:77–97, 2019.

[43] Di Francesco, P., I. Malavolta y P. Lago: *Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption.* IEEE International Conference on Software Architecture (ICSA), páginas 21–30, 2017.

[44] Di Noia, T., M. Mongiello, F. Nocera y U. Straccia: *A fuzzy ontology-based approach for tool-supported decision making in architectural design.* Knowledge and Information Systems, 58(1):83–112, 2019. doi:https://doi.org/10.1007/s10115-018-1182-1.

[45] Dong, M., K. Ota y A. Liu: *Preserving Source-Location Privacy through Redundant Fog Loop for Wireless Sensor Networks.* 2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing, 2015.

[46] Dougherty, K. L. y J. Edward: *The properties of simple vs. absolute majority rule: cases where absences and abstentions are important.* Journal of Theoretical Politics, 22(1):85–122, 2010.

[47] Dragoni, N., S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, M. Montesi, R. Mustafin y L. Safina: *Microservices: yesterday, today, and tomorrow.* CoRR, 2016.

[48] Engel, Thomas, Melanie Langermeier, Bernhard Bauer y Alexander Hofmann: *Evaluation of Microservice Architectures: A Metric and Tool-Based Approach.* Information Systems in the Big Data Era, páginas 74–89, 2018.

[49] Ernst, N., R. Kazman y P. Bianco: *Component Comparison, Evaluation, and Selection: A Continuous Approach.* IEEE International Conference on Software Architecture Companion (ICSA-C), páginas 87–90, 2019. doi:10.1109/ICSA-C.2019.00023.

[50] Esposito, C., A. Castiglione, C. Tudorica y F. Pop: *Security and privacy for cloud-based data management in the health network service chain: a microservice approach.* IEEE Communications Magazine, 55(9):102–108, Sep. 2017. https://ieeexplore.ieee.org/document/8030494.

[51] Evans, E.: *Domain-driven design: Tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[52] Fayyad, U., G. Piatetsky-Shapiro y P. Smyth: *The KDD process for extracting useful knowledge from volumes of data.* Communications of the ACM, 39(11):27–34, 1996. doi:https://doi.org/10.1145/240455.240464.

[53] Fernandez, Eduardo B., Hernán Astudillo y Gilberto Pedraza-García: *Revisting architectural tactics for security.* Software Architecture. Springer International Publishing, páginas 55–69, 2015.

[54] Fowler, Martin: *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[55] Frye, Alan: *Architecture Cubed.* https://www.benefitfocus.com/blogs/design-engineering/architecture-cubed, 2016.

[56] Gadea, C., M. Trifan, D. Ionescu, M. Cordea y B. Ionescu: *A microservices architecture for collaborative document editing enhanced with face recognition.* EEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI), (441-446):441–446, 2016.

[57] Gamma, Erich, Richard Helm, Ralph Johnson y John M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley professional computing series, 1994.

[58] Gan, Y. y C. Delimitrou: *The Architectural Implications of Cloud Microservices.* IEEE Computer Architecture Letters, 17(2):155–158, 2018.

[59] Garcia, J., I. Krka, N. Medvidovic y C. Douglas: *A Framework for Obtaining the Ground-Truth in Architectural Recovery.* WICSA/ECSA, páginas 292–296, 2012.

[60] Garlan, D., S. Khersonsky y J. S. Kim: *Model checking publish-subscribe systems.* International spin workshop on model checking of software, páginas 166–180, 2003. doi:https://doi.org/10.1007/3-540-44829-2_11.

[61] Garousi, V., M. Felderer y M. V. Mäntylä: *Guidelines for including grey literature and conducting multivocal literature reviews in software engineering.* Information and Software Technology, 106:101–121, 2019.

[62] Gatouillat, A., Y. Badr, B. Massot y E. Sejdić: *Internet of medical things: A review of recent contributions dealing with cyber-physical systems in medicine.* IEEE internet of things journal, 5(5):3810–3822, 2018.

[63] Gradle: *Gradle User Manual*, 2018. https://docs.gradle.org/current/userguide/userguide.html.

[64] Granchelli, G., M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino y A. D. Salle: *MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems.* 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), páginas 298–302, 2017.

[65] Groot, Bas: *OS Framework Selection: How To Examine An API*, 2019. https://dzone.com/articles/article-1-framework-selection-how-to-examine-an-ap.

[66] Groot, Bas: *OS Framework Selection: How to Spot Immature Frameworks During Selection*, 2019. https://dzone.com/articles/article-2-how-to-spot-immature-frameworks-during-s.

[67] Gupta, A.: *Microservice Design Patterns.* http://blog.arungupta.me/microservice-design-patterns/. [Online; accessed April 5th, 2019].

[68] Gupta, A. y C. Gupta: *Towards Dependency Based Collaborative Method for Requirement Prioritizatio.* Eleventh International Conference on Contemporary Computing (IC3), páginas 1–3, 2018. doi:10.1109/IC3.2018.8530542.

[69] Harrison, N. B., P. Avgeriou y U. Zdun: *On the impact of fault tolerance tactics on architecture patterns.* Proceedings of the 2nd International Workshop on Software Engineering for Resilient Systems, páginas 12–21, 2010.

[70] Harrison, Neil B. y Paris Avgeriou: *How do architecture patterns and tactics interact? A model and annotation.* Journal of Systems and Software, 83(10):1735–1758, 2010, ISSN 01641212. http://dx.doi.org/10.1016/j.jss.2010.04.067.

[71] Haselböck, S., R. Weinreich y G. Buchgeher: *Decision guidance models for microservices: service discovery and fault tolerance.* Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems, página 4, 2017. doi:10.1145/3123779.3123804.

[72] Istrate, D., C. Taramasco, A. Fleury, N. Houmani, M. Hariz y J. Boudy: *Well-being and -ageing with chronical disease: the BV2 project.* JETSAN 2019: Journées d'Etude sur la TéléSanté, Sorbonne Universités, 2019.

[73] Johnson, Ralph E.: *Frameworks = (Components + Patterns).* Commun. ACM, 40(10):39–42, 1997.

[74] Kazman, Rick: *Rapid Software Composition by Assessing Untrusted Components.* https://insights.sei.cmu.edu/sei_blog/2018/11/rapid-software-composition-by-assessing-untrusted-components.html Accessed: 2018-11-26.

[75] Kircher, M. y J. Prashant: *Pattern-Oriented Software Architecture. Patterns for Resource Management.* Wiley series in software design patterns, West Sussex, England, 2004.

[76] Kitchenham, B.: *Procedures for Performing Systematic Reviews.* Informe técnico, Keele University, 2004.

[77] Kitchenham, B. y S. Charters: *Guidelines for performing Systematic Literature Reviews in Software Engineering.* Engineering, 2:1051, 2007, ISSN 00010782.

[78] Kitchenham, B. y S. L. Pfleeger: *Principles of survey research part 4: questionnaire evaluation.* ACM SIGSOFT Software Engineering Notes, 27(3):20–23, 2002.

[79] Kitchenham, B. y S. L. Pfleeger: *Principles of survey research: part 5: populations and samples.* ACM SIGSOFT Software Engineering Notes, 27(5):17–20, 2002.

[80] Kleehaus, Martin, Ömer Uludağ, Patrick Schäfer y Florian Matthes: *MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-Based Environments.* Information Systems in the Big Data Era, páginas 148–162, 2018.

[81] Kruchten, P.: *The rational unified process: an introduction.* Addison-Wesley Professional, 2004.

[82] Kwong, C. K., L. F. Mu, J. F. Tang y X. G. Luo: *Optimization of software components selection for component-based software system development.* Computers & Industrial Engineering, 58(4):618–624, 2010. doi:https://doi.org/10.1016/j.cie.2010.01.003.

[83] Lawton, G.: *How microservices patterns made Uber's architecture perform better.* http://www.theserverside.com/feature/How-microservices-patterns-helped-Uber-systems-perform-better, 2017.

[84] Lenarduzzi, V. y O. Sievi-Korte: *Software Components Selection in Microservices-based Systems.* Proceedings of ACM Conference (XP'18), página 3, 2018.

[85] Lewis, J. y M. Fowler: *Microservices. A definition of this new architectural term.* https://martinfowler.com/articles/microservices.html. [Online; accessed April 5th, 2020].

[86] López, C. y H. Astudillo: *Explicit architectural policies to satisfy NFRs using COTS.* International Conference on Model Driven Engineering Languages and Systems, páginas 227–236, 2005. doi:https://doi.org/10.1007/11663430_24.

[87] Luz, W., E. Agilar, M. C. de Oliveira, C. E. R. de Melo, G. Pinto y R. Bonifácio: *An experience report on the adoption of microservices in three Brazilian government institutions.* Proceedings of the XXXII Brazilian Symposium on Software Engineering, páginas 32–41, 2018. doi:https://doi.org/10.1145/3266237.3266262.

[88] Magsi, H., A. H. Sodhro, F. A. Chachar, S. A. K. Abro, G. H. Sodhro y S. Pirbhulal: *Evolution of 5G in Internet of medical things.* international conference on computing, mathematics and engineering technologies (iCoMET), páginas 1–7, 2018. doi:10.1109/ICOMET.2018.8346428.

[89] Majerowicz, L.: *Integration Patterns for Microservices Architectures: The good, the bad and the ugly.* http://hecodes.com/2017/04/integration-patterns-microservices-architectures-good-bad-ugly/. [Online; accessed April 5th, 2019].

[90] Márquez, G. y H. Astudillo: *Actual Use of Architecture Patterns in Microservices-based Open Source Projects.* 25th Asia-Pacific Software Engineering Conference (APSEC), páginas 31–40, 2018.

[91] Márquez, G., Y. Lazo y H. Astudillo: *Evaluating Frameworks Assemblies In Microservices-based Systems Using Imperfect Information.* IEEE International Conference on Software Architecture Companion (ICSA-C), páginas 250–257, 2020. doi:10.1109/ICSA-C50368.2020.00049.

[92] Márquez, G., F. Osses y H. Astudillo: *Review of Architectural Patterns and Tactics for Microservices in Academic and Industrial Literature.* IEEE Latin America Transactions, 16(9):2321–2327, 2018.

[93] Márquez, G., F. Osses y H. Astudillo: *Review of Architectural Patterns and Tactics for Microservices in Academic and Industrial Literature.* Avances en Ingenieria de Software a Nivel Iberoamericano, CIbSE 2018, páginas 71–84, 2018.

[94] Márquez, G., J. Soldani, F. Ponce y H. Astudillo: *Frameworks and High-Availability in Microservices: An Industrial Survey.* XXIII Ibero-American Conference on Software Engineering (CIbSE), *In press.*

[95] Márquez, G., M. Villegas y H. Astudillo: *A pattern language for scalable microservices-based systems.* 12th European Conference on Software Architecture: Companion Proceedings (ECSA '18), páginas 24:1–24:7, 2018.

[96] Mayer, B. y R. Weinreich: *An Approach to Extract the Architecture of Microservice-Based Software Systems.* 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), páginas 21–30, 2018.

[97] McGee, R. A., U. Eklund y M. Lundin: *Stakeholder identification and quality attribute prioritization for a global Vehicle Control System.* Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, páginas 43–48, 2010. doi:https://doi.org/10.1145/1842752.1842765.

[98] Messina, A., R. Rizzo, P. Storniolo, M. Tripiciano y A. Urso: *The Database-is-the-Service Pattern for Microservice Architectures.* International Conference on Information Technology in Bio-and Medical Informatics, páginas 223–233, 2016.

[99] Messina, A., R. Rizzo, P. Storniolo y A Urso: *A Simplified Database Pattern for the Microservice Architecture.* The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA), 2016.

[100] Mirakhorli, Mehdi y Jane Cleland-Huang: *Detecting, Tracing, and Monitoring Architectural Tactics in Code.* IEEE Transactions on Software Engineering, 42(3):205–220, 2015, ISSN 0098-5589.

[101] Montesi, Fabrizio y Janine Weber: *Circuit Breakers, Discovery, and API Gateways in Microservices*, 2016. http://arxiv.org/abs/1609.05830.

[102] Mujhid, I. J., J. C. Santos, R. Gopalakrishnan y M. Mirakhorli: *A search engine for finding and reusing architecturally significant code.* Journal of Systems and Software, 130:81–93, 20‘7. doi:https://doi.org/10.1016/j.jss.2016.11.034.

[103] Nagothu, Deeraj, Ronghua Xu, Seyed Yahya Nikouei y Yu Chen: *A Microservice-enabled Architecture for Smart Surveillance using Blockchain Technology.* CoRR, abs/1807.07487, 2018. http://arxiv.org/abs/1807.07487.

[104] Newman, S.: *Building microservices: designing fine-grained systems.* O'Reilly Media, Inc., 2015.

[105] Ngankam, H. K., H. Pigot, M. Parenteau, M. Lussier, Laliberté C. Aboujaoudé, A., M. Couture, N. Bier y S. Giroux: *An IoT Architecture of Microservices for Ambient Assisted Living Environments to Promote Aging in Smart Cities.* International Conference on Smart Homes and Health Telematics, páginas 154–167, 2019. doi:https://doi.org/10.1007/978-3-030-32785-9_14.

[106] Noppen, J., P. van den Broek y M. Akşit: *Software development with imperfect information.* Soft computing, 12(1):3, 2008.

[107] Nord, R. L., I. Ozkaya, P. Kruchten y M. Gonzalez-Rojas: *In search of a metric for managing architectural technical debt.* Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, páginas 91–100, 2012. doi:10.1109/WICSA-ECSA.212.17.

[108] Osses, Felipe, Gastón Márquez y Hernán Astudillo: *Poster: Exploration of Academic and Industrial Evidence about Architectural Tactics and Patterns in Microservices.* ICSE'18 Companion: 40th International Conference on Software Engineering Companion., 2018.

[109] Osses, Felipe, Gastón Márquez, Mónica M Villegas, Cristian Orellana, Marcello Visconti y Hernán Astudillo: *Security tactics selection poker (TaSPeR): a card game to select security tactics to satisfy security requirements.* En *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, página 54. ACM, 2018.

[110] Pahl, M., F. Aubet y S. Liebald: *Graph-based IoT microservice security.* En *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, páginas 1–3, April 2018.

[111] Pereira, A., G. Márquez, H. Astudillo y E. B. Fernández: *Security Mechanisms Used in Microservices-Based Systems: A Systematic Mapping.* XLV Latin American Computing Conference, *In press*, 2019.

[112] Perry, D. E. y A. L. Wolf: *Foundations for the study of software architecture.* ACM SIGSOFT Software engineering notes, 17(4):40–52, 1992. doi:https://doi.org/10.1145/141874.141884.

[113] Petersen, K., D. Badampudi, S. M. A. Shah, K. Wnuk, T. Gorschek, E. Papatheocharous, J. Axelsson, S. Sentilles, I. Crnkovic y A. Cicchetti: *Choosing component origins for software intensive systems: In-house, COTS, OSS or outsourcing?—A case survey.* IEEE Transactions on Software Engineering, 44(3):237–261, 2017. doi:10.1109/TSE.2017.2677909.

[114] Petersen, K., D. Badampudi, S. M. A. Shah, K. Wnuk, T. Gorschek, E. Papatheocharous, J. Axelsson, S. Sentilles, I. Crnkovic y A. Cicchetti: *Choosing component origins for software intensive systems: In-house, COTS, OSS or outsourcing?—A case survey.* IEEE Transactions on Software Engineering, 44(3):237–261, 2017. doi:10.1109/TSE.2017.2677909.

[115] Punter, T., M. Ciolkowski, B. Freimut y I. John: *Conducting on-line surveys in software engineering.* International Symposium on Empirical Software Engineering, (80-88), 2003.

[116] Qanbari, Soheil, Samim Pezeshki, Rozita Raisi, Samira Mahdizadeh, Rabee Rahimzadeh, Negar Behinaein, Fada Mahmoudi, Shiva Ayoubzadeh, Parham Fazlali, Keyvan Roshani, Azalia Yaghini, Mozhdeh Amiri, Ashkan Farivarmoheb, Arash Zamani y Schahram Dustdar: *IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications.* IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI), páginas 277–282, 2016.

[117] Rademacher, F., P. Sorgalla, J.and Wizenty, S. Sachweh y A. Zündorf: *Graphical and Textual Model-Driven Microservice Development.* Microservices, páginas 147–179, 2020. doi:https://doi.org/10.1007/978-3-030-31646-4_7.

[118] Richards, M.: *Microservices AntiPatterns and Pitfalls.* O'Reilly Media, 2016.

[119] Richardson, Chris: *Microservices Patterns.* Manning Publications, 2018.

[120] Richter, D., M. Konrad, K. Utecht y A. Polze: *Highly-available applications on unreliable infrastructure: Microservice architectures in practice.* IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), páginas 130–137, 2017. doi:10.1109/QRS-C.2017.28.

[121] Robin, M., D. Istrate y J. Boudy: *Remote monitoring, distress detection by slightest invasive systems: Sound recognition based on hierarchical i-vectors.* 39st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), páginas 2744–2748, 2017.

[122] Rose, Stuart, Dave Engel, Nick Cramer y Wendy Cowley: *Automatic keyword extraction from individual documents.* Text Mining: Applications and Theory, páginas 1–20, 2010.

[123] Shalom, N.: *Building Large Scale Services with Microservices.* http://cloudify.co/2017/07/20/building-large-scale-services-micro-services-tosca.html. [Online; accessed April 5th, 2019].

[124] Software, SmartBear: *Why You Can't Talk About Microservices Without Mentioning Netflix.* https://blog.smartbear.com/microservices/why-you-cant-talk-about-microservices-without-mentioning-netflix/, 2015.

[125] Stangarone, Joe: *5 big problems caused by bad application architecture*, 2012. https://www.mrc-productivity.com/blog/2012/02/5-big-problems-caused-by-bad-application-architecture/.

[126] Stubbs, J., W. Moreira y R. Dooley: *Distributed systems of microservices using docker and serfnode*. 7th International Workshop on Science Gateways, páginas 34–39, 2015.

[127] Svahnberg, M.: *An industrial study on building consensus around software architectures and quality attributes*. Information and Software Technology, 46(12):805–818, 2004. doi:https://doi.org/10.1016/j.infsof.2004.02.001.

[128] Taibi, D., V. Lenarduzzi y C. Pahl: *Architectural patterns for microservices: a systematic mapping study*. Proc. 8th Int. Conf. Cloud Computing and Services Science, 2018.

[129] Taibi, Davide: *An Empirical Investigation on the Motivations for the Adoption of Open Source Software*. Tenth International Conference on Software Engineering Advances (ICSEA), 2015.

[130] Taneja, M., N. Jalodia, J. Byabazaire, A. Davy y C. Olariu: *SmartHerd management: A microservices-based fog computing–assisted IoT platform towards data-driven smart dairy farming*. Software: Practice and Experience, 49(7), 2019. doi:https://doi.org/10.1002/spe.2704.

[131] Tang, J. F., L. F. Mu, C. K. Kwong y X. G. Luo: *An optimization model for software component selection under multiple applications development*. European Journal of Operational Research, 212(2):301–311, 2011. doi:https://doi.org/10.1016/j.ejor.2011.01.045.

[132] Taramasco, C., T. Rodenas, F. Martinez, P. Fuentes, R. Munoz, R. Olivares, V. H. De Albuquerque y J. Demongeot: *A novel monitoring system for fall detection in older people*. IEEE Access, 6(43563-43574), 2018.

[133] Thakurta, R.: *A framework for prioritization of quality requirements for inclusion in a software project*. Software Quality Journal, 21(4):573–597, 2013. doi:https://doi.org/10.1007/s11219-012-9188-5.

[134] Toffetti, G., S. Brunner, M. Blöchlinger y Edmonds A. Dudouet, F.: *An architecture for self-managing microservices*. Proceedings of the 1st International Workshop on Automated Incident Management in Cloud, páginas 19–24, 2015. doi:10.1145/2747470.2747474.

[135] Travassos, G. H., S. L. Peeger y V. R. Basili: *Experimental software engineering: an introduction*. st Experimental Software Engineering Latin American Workshop-ESELAW, 2004.

[136] Ueda, T., T. Nakaike y M. Ohara: *Workload characterization for microservices.* IEEE international symposium on workload characterization (IISWC), (1-10), 2016.

[137] Vale, A. P., G. Márquez, H. Astudillo y E. B. Fernandez: *Security Mechanisms Used in Microservices-Based Systems: A Systematic Mapping.* XLV Latin American Computing conference (CLEI), *In press*, 2019.

[138] Viennot, N., M. Lecuyer, J. Bell, R. Geambasu y J. Nieh: *Synapse: a microservices architecture for heterogeneous-database web applications.* Proceedings of the Tenth European Conference on Computer Systems, página 21, 2015.

[139] Vural, H., M. Koyuncu y S. Guney: *A Systematic Literature Review on Microservices.* International Conference on Computational Science and Its Applications, páginas 203–217, 2017.

[140] Wasson, M.: *Design patterns for microservices.* `https://azure.microsoft.com/en-us/blog/design-patterns-for-microservices/`. [Online; accessed April 5th, 2019].

[141] Watada, J., A. Roy, R. Kadikar, H. Pham y B. Xu: *Emerging Trends, Techniques and Open Issues of Containerization: A Review.* IEEE Access, 7:152443–152472, 2019.

[142] Wilcoxon, F., S. K. Katti y R. A. Wilcox: *Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test.* Selected tables in mathematical statistics, 1:171–259, 1970.

[143] Wohlin, C.: *Guidelines for snowballing in systematic literature studies and a replication in software engineering.* Proceedings of the 18th international conference on evaluation and assessment in software engineering, página 38, 2014.

[144] Wohlin, C., P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell y A. Wesslén: *Experimentation in software engineering.* Springer Science and Business Media, 2012.

[145] Wojcik, R., F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord y B. Wood: *Attribute-driven design (ADD), version 2.0.* No. CMU/SEI-2006-TR-023. Carnegie-Mellon University of Pittsburgh, Software Engineering Institute, 2006.

[146] Wolff, E.: *Microservices: Flexible Software Architecture.* Addison-Wesley Professional, 2016.

[147] Wood, W.G.: *A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0.* Software Engineering Institute, Technical Report CMU/SEI-2007-TR-005, 2007.

[148] Xiao, Z., I. Wijegunaratne y X. Qiang: *Reflections on SOA and Microservices*. 4th International Conference on Enterprise Systems (ES), páginas 60–67, 2016.