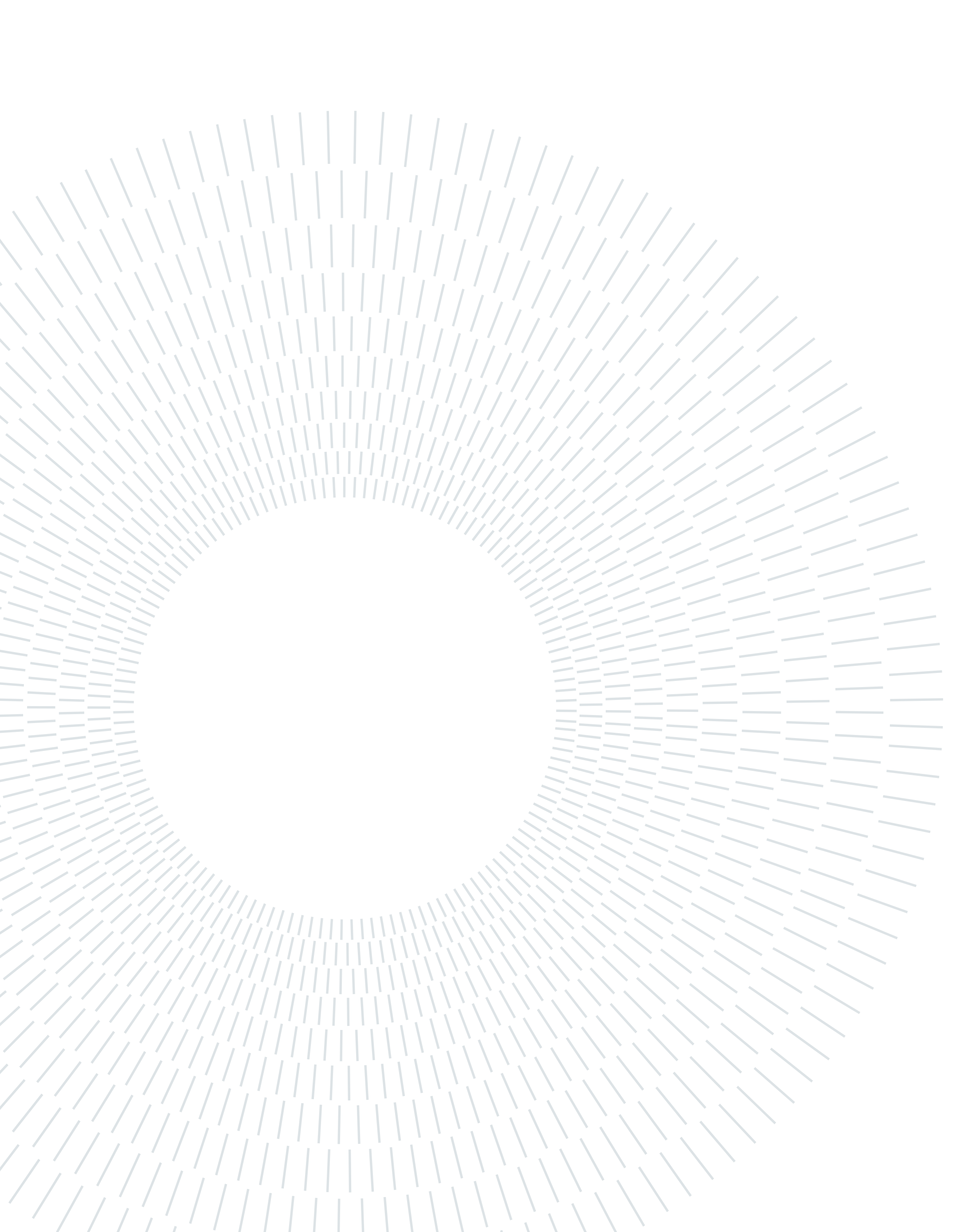
Engineering microservice-based Self-Adaptive Systems: the case of RAMSES

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Authors: **Giancarlo Sorrentino, Vincenzo Riccio**

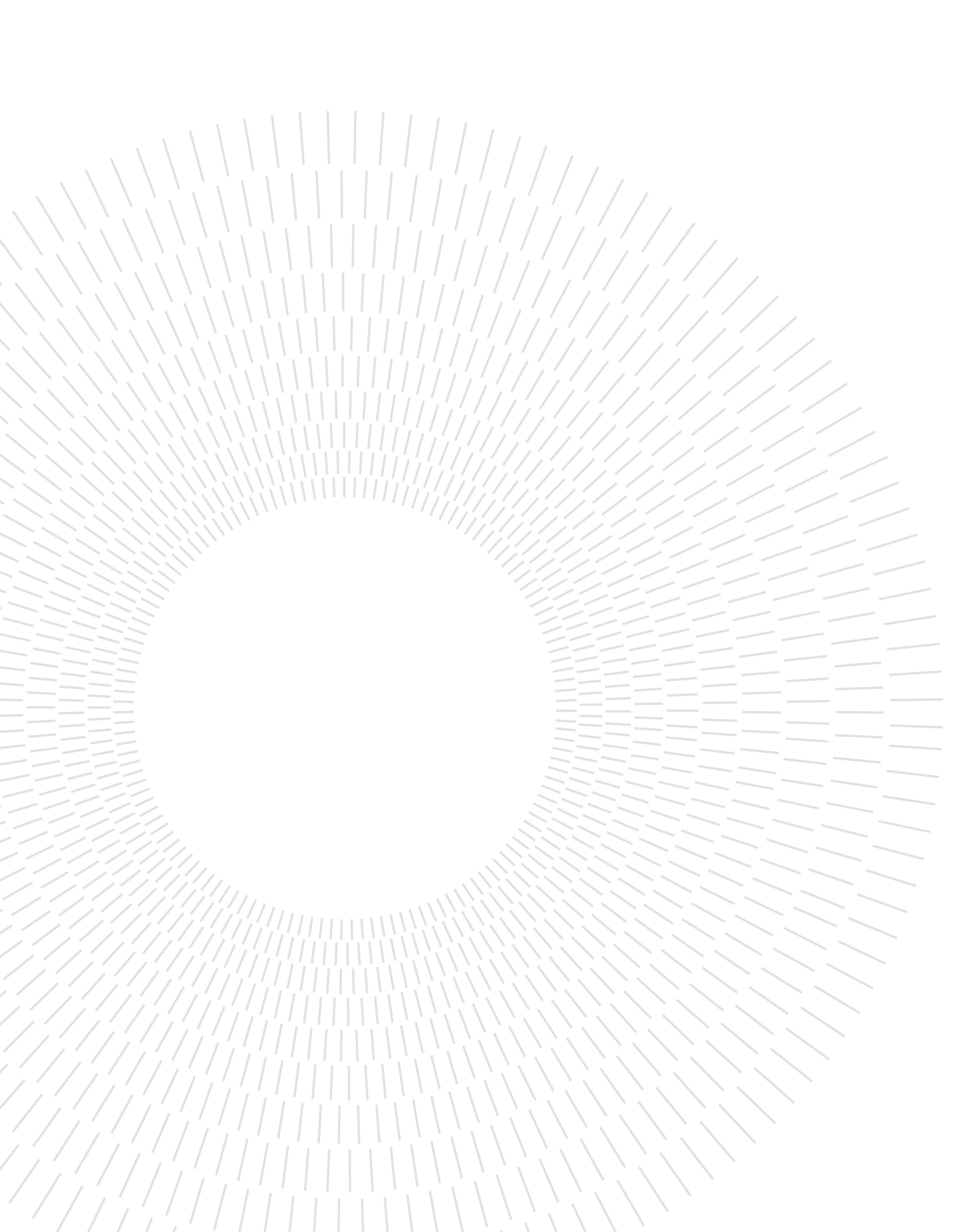
Student IDs: 968003, 967517
Advisor: Prof. Raffaela Mirandola
Co-advisor: Prof. Matteo Camilli
Academic Year: 2021-22



Abstract

The increasing complexity of modern software systems leads to the problem of finding an effective approach to automate their management process. Self-Adaptive Systems (SASs) solve this problem by autonomously adapting themselves in order to achieve high-level user-defined goals. However, SASs are usually challenging and expensive to develop, both in terms of time and resources. The goal of our work is to find a better trade-off between development costs and effectiveness of a SAS, by providing a managing subsystem to be reused with different service-based systems (SBSs). To address this problem, we developed an extensible software framework called RAMSES, a Reusable Autonomic Manager for microServicES. The goal of RAMSES is to enforce the satisfaction of user-defined QoS specifications for the target SBS at runtime, while improving its overall performance. Our work also includes a standalone microservice-based application – SEFA – that serves as a managed subsystem, providing the scientific community with a fully-implemented SAS exemplar, which comprises two reusable and independent subsystems.

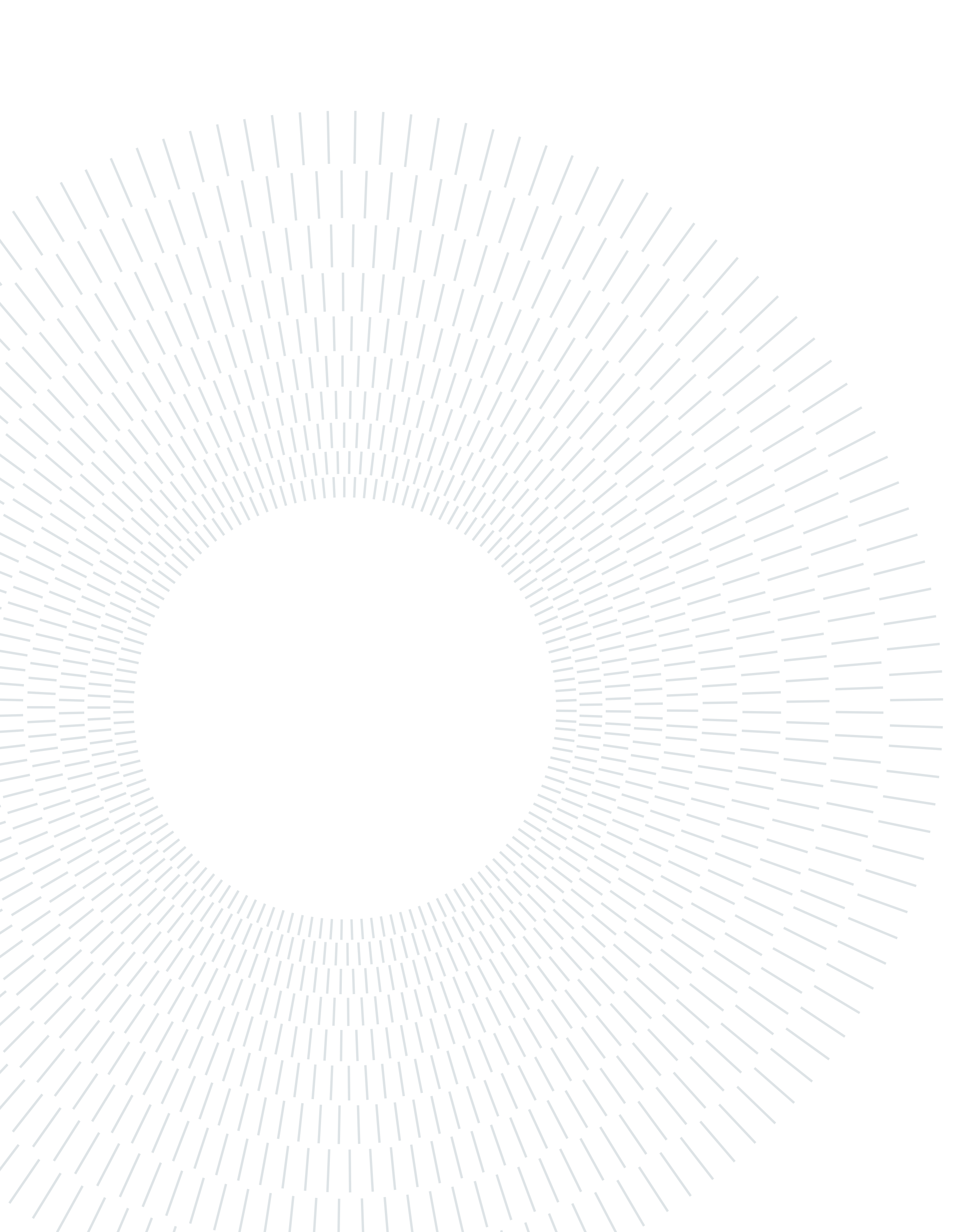
Keywords: self-adaptive, systems, reusable, microservices, adaptation



Abstract in lingua italiana

La crescente complessità dei moderni sistemi software solleva il problema di trovare un approccio efficace per automatizzare la gestione di tali sistemi. I sistemi *Self-Adaptive* (SAS) risolvono questo problema adattandosi autonomamente per raggiungere obiettivi di alto livello definiti dall'utente. Tuttavia, i SAS sono solitamente impegnativi e costosi da sviluppare, sia in termini di tempo che di risorse. Il nostro lavoro mira a trovare un miglior compromesso tra costi di sviluppo ed efficacia dei SAS, fornendo un *managing system* riutilizzabile con diversi sistemi *service-based* (SBS). Per affrontare il problema, abbiamo sviluppato RAMSES (*Reusable Autonomic Manager for microServicES*), un framework software estendibile e riutilizzabile. Il suo obiettivo è garantire che il SBS da adattare soddisfi le specifiche di *Quality-of-Service* (QoS) richieste dall'utente, e allo stesso tempo migliorare le performance di tale sistema. Includiamo inoltre un'applicazione a microservizi – SEFA – usata come *managed subsystem*, fornendo alla comunità scientifica un *exemplar* di SAS completamente implementato, che include due sottosistemi indipendenti e riutilizzabili per vari scopi.

Parole chiave: self-adaptive, sistemi, microservizi, riutilizzabile, adattamento



Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**

2 State Of The Art **3**
2.1 An Introduction To Self-Adaptive Systems 3
2.2 Existing Exemplars 7

3 Our Proposal **11**
3.1 Problem statement and proposed Solution 11
3.2 The Managed System: SEFA 13
3.2.1 Introduction to the application 14
3.2.2 Relevant design choices and system architecture 15
3.2.3 API overview 20
3.2.4 Probe and Actuator requirements and implementation 22
3.3 The Managing System: RAMSES 25
3.3.1 Prerequisites and assumptions 26
3.3.2 System Model 27
3.3.3 Knowledge component 34
3.3.4 Monitor 35
3.3.5 Analyse 38
3.3.6 Plan 46
3.3.7 Execute 53
3.3.8 Dashboard 57

4 Evaluation **59**

4.1 Research questions 59
4.2 Design of the evaluation 60
4.3 Results 64
4.3.1 Scenario S1 - QoS not satisfied 64
4.3.2 Scenario S2 - Service unavailable 80
4.3.3 Scenario S3 - Better implementation available 85
4.3.4 Reusability of the Managing System 88

5 Conclusions And Future Work 93
5.1 Conclusions 93
5.2 Future directions 94

Bibliography 97

A Appendix A - Experimental results 103
A.1 Scenario S1 - QoS not satisfied 104
A.1.1 E1 - Choice of the experiment duration 104
A.1.2 E2 - Benefits of the adaptation 108
A.1.3 E3 - Impact of the Metrics Window Size 116
A.1.4 E4 - Impact of the Analysis Window Size 120
A.1.5 E5 - Impact of the number of users 124
A.2 Scenario S2 - Service unavailable 128
A.2.1 E1 - Analysis of the self-healing capabilities 128
A.2.2 E2 - Analysis of the failure tendency detection 128
A.2.3 E3 - Impact of the Monitor Period 129
A.3 Scenario S3 - Better implementation available 130
A.3.1 E1 - Analysis of the self-optimization capabilities 130
A.4 Reusability of the Managing System 134
A.4.1 E1 - Analysis of the Managing System reusability 134

B Appendix B - Components interfaces 137
B.1 Managed System 137
B.1.1 Restaurant Service 138
B.1.2 Ordering Service 139
B.1.3 Payment Proxy Service 139
B.1.4 Delivery Proxy Service 140

B.2 Probe . . . 140
B.3 Actuator . . . 141
B.4 Managing System . . . 141
B.4.1 Monitor Component . . . 141
B.4.2 Analyse Component . . . 142
B.4.3 Plan Component . . . 142

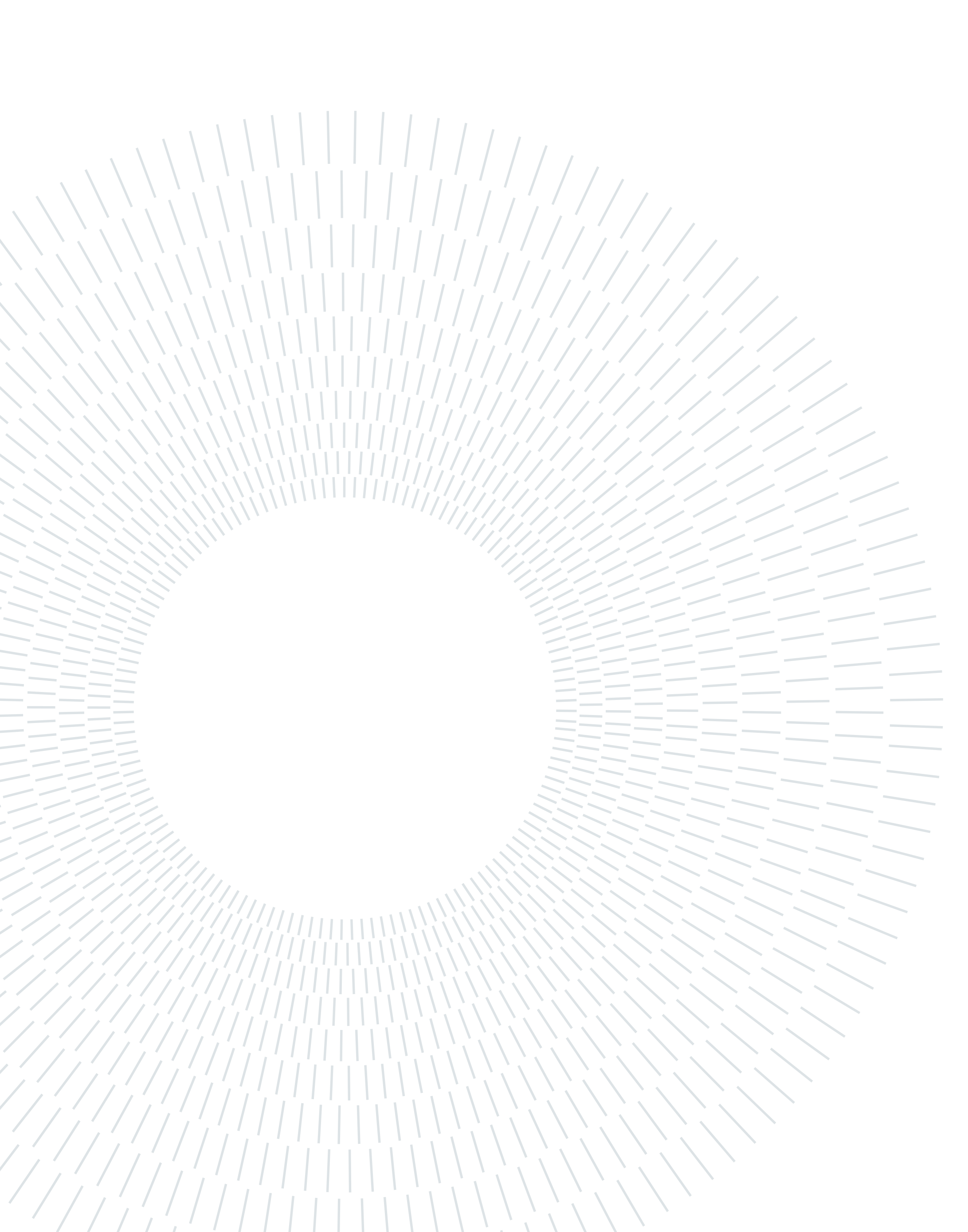
C Appendix C - Structure of JSON configuration files 143
C.1 System Architecture . . . 143
C.2 System Benchmarks . . . 144
C.3 QoS Specification . . . 145

List of Figures 147

List of Tables 151

List of Algorithms 153

Acknowledgements 155



1 | Introduction

Nowadays, as the size and complexity of software systems keep increasing, a microservice architecture is often chosen as the architectural style for service-based systems (SBSs). These systems integrate several (micro)services, which are commonly located in multiple computing infrastructures, spread across the world. Moreover, by applying the microservice architectural style it is possible to build dynamic systems where the set of involved services and their instances may change at runtime. In this context, the problem of managing such big and complex systems is noteworthy, since direct human intervention is challenging, time-consuming and error-prone.
Autonomic managers [19] try to mitigate this problem by proposing an automatic mechanism to (re)configure, optimise and repair a *managed subsystem*, which implements the business logic. As a whole, these systems are referred to as *autonomic systems*. Among them, Self-Adaptive Systems (SASs) are able to autonomously adapt themselves, to achieve user-defined goals in response to changes in the underlying environment or in the system itself, without human intervention [41]. These systems are usually built using a feedback control loop made of four stages – Monitor-Analyse-Plan-Execute – over a Knowledge base (i.e., a MAPE-K loop).

In general, the engineering of Self-Adaptive Systems is a hard challenge to tackle, drawing the attention of the scientific community. Indeed, their realisation is often demanding, in terms of time, resources and costs. When trying to adapt an existing application, or building an entire SAS from scratch, one of the common approaches is to realise a managing system tailored on the managed one. However, when dealing with service-based systems, different Self-Adaptive Systems usually share part of their adaptation goals. More specifically, they usually address the problem of dynamically ensuring some Quality-of-Service (QoS) specifications while improving the overall system's performance. The goal of this work is the development of RAMSES, a Reusable Autonomic Manager for microServicES. RAMSES is a software framework made by configurable and extendable components, whose behaviour does not depend on the specific Managed System to be adapted, and is thus reusable with different service-based Managed Systems. Indeed, RAMSES aims at finding a better trade-off between development costs – in terms of time,

resources and design – and effectiveness of the self-adaptation process thanks to the reuse of the managing subsystem.
To evaluate RAMSES and to build a system that could be easily reused and adapted for research purposes, we also developed a Java-based microservice application to be managed by RAMSES, SEFA: a SErvice-based eFood Application.
As a result, we provide the scientific community with a complete self-adaptive system, made of two non-simulated and fully-implemented subsystems, which can be reused for different goals.

The following chapters describe in detail the problem we faced and the proposed solution, together with its experimental evaluation. In particular, Chapter 2 introduces the reader to the Self-Adaptive Systems, providing an overview of their state of the art.
Chapter 3 provides the reader with an in-depth analysis of the problem and our solution, describing in detail RAMSES and the managed system designed for experimentation purposes, SEFA.
Chapter 4 focuses on the conducted experimental evaluation, which aims at answering 10 research questions.
Chapter 5 summarises the overall project, highlighting the most relevant conclusions and proposing future improvements to RAMSES.
Finally, three appendices conclude this work, that include all the graphs generated during the experiments (Appendix A), all the APIs offered by the implemented components (Appendix B) and the structure of the configuration files used during the initialisation of RAMSES (Appendix C).

The source code of the all the components implemented for this research are publicly available on GitHub[1].

[1]GitHub repository available at `https://github.com/ramses-sas/ramses-sefa-SAS`

2 | State Of The Art

This chapter aims to provide the reader with an overview of the state of the art concerning the field of self-adaptive systems.
This chapter is structured as follows:

- Section 2.1 introduces the concept of Self-Adaptive Systems, and describes their properties;
- Section 2.2 illustrates some relevant exemplars of Self-Adaptive Systems.

2.1. An Introduction To Self-Adaptive Systems

During the last years, the complexity of software systems started increasing fast. They became no longer restricted to few components located in small and easily controllable areas, but made of a huge number of interconnected and distributed devices, as in the Internet of Things field. Hence, the need for automatic (re)configuration and optimization mechanisms for those systems arose, aiming at satisfying the admin's goals without human intervention.

In 2001, the proposal of *autonomic computing* [19] by IBM's senior vice president of research, Paul Horn, put the attention of the scientific community on these computing systems, that can manage themselves when provided with high-level goals from their administrators. Hence, the main concern of these systems is *self-management*, which can be specialised in four main aspects (or *adaptation goals*):

- **Self-configuration**. Configuring and integrating complex systems is challenging, time-consuming, and error-prone. Given a set of *declarative* high-level policies (i.e., describing what is the desired state, not how to reach it), autonomic systems configure themselves automatically;
- **Self-optimization**. From an engineering point of view, one of the main goals is to maximize the performance and minimize the cost of a system. Autonomic systems can improve their operation, identifying and seizing opportunities to make

themselves more efficient in performance or cost;

- **Self-healing**. As stated by Werner Vogels, Amazon's VP and CTO, "everything fails, all the time": any computing system is continuously exposed to failures. Autonomic systems can detect, diagnose, and repair localized problems resulting from bugs or failures in either software or hardware;
- **Self-protection**. One of the main concerns of system administrators and engineers is how to protect the system from malicious attacks or harmful problems, resulting in one of the goals of an autonomic system.

Between all the autonomic systems, we focus on *Self-Adaptive Systems* (SAS), systems that are able to adjust their behaviour in response to their perception of the environment and of the overall system [7].
Even if the publication of IBM's vision was a game changer in the field of autonomic computing, the first reference to self-adaptive systems in literature was in 1990. After then, self-adaptive systems gained more and more interest, and self-adaptation is now considered an effective approach to deal with the problems they address [41, 44].

In *Software Engineering Processes for Self-Adaptive Systems* [1], Andersson et al. propose a conceptual architecture for self-adaptive systems (Figure 2.1), promoting separation of concerns between a *Managing (Sub)system* and a *Managed (Sub)system*. The Managing Subsystem implements the adaptation logic that manages the Managed Subsystem, which encloses the domain logic. The self-adaptive system operates in an observable environment, which might affect the adaptation logic. To fulfil its goals, the Managing Subsystem monitors the environment and the Managed Subsystem and adapts the latter when necessary. For this reason, a self-adaptive system bases its operation on two awareness properties: self-awareness and context-awareness [31]. The former describes the ability of the system to be aware of itself and of its behaviours. The latter means that the system is aware of the context in which it operates.

During the last 30 years, the scientific community deeply analysed different aspects of self-adaptation, from multiple points of view. This increasing interest in the SASs field led to the presentation of numerous works, and also to the birth of specialised conference series, as SEAMS [18] and ACSOS [17].
To summarise these and other works, a valuable effort was made by Krupitzer et al. in [21], where the authors proposed a comprehensive taxonomy for self-adaptation. It was the result of an extensive literature review and of the integration of existing taxonomies and works on self-adaptation. The taxonomy is structured to answer the 5W + 1H questions for eliciting adaptation requirements already introduced by Salehie and Tahvildari in [31],

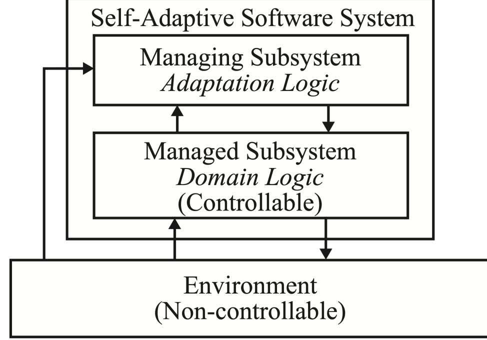


Figure 2.1: A Conceptual Architecture for Self-Adaptive Software Systems [1]

summarised in Table 2.1.

Question	Dimension of the taxonomy
When?	Time (reactive vs. proactive)
Why?	Reason (context, technical resources, user)
Where?	Level (application, system software, communication, technical resources, context)
What?	Technique (parameter, structure, context)
Who?	N/A (nature of a SAS leads to an automatic type of adaptation)
How?	Adaptation control (approach, adaptation decision criteria, degree of decentralization)

Table 2.1: Relation of the taxonomy dimensions and the questions [21]

A first aspect to consider when designing self-adaptive systems is related to *when* the system should be adapted. Even if proactive adaptation is preferable, since it anticipates the need for adaptation, the prediction algorithms are complex to develop and faulty results can cause suboptimal or malicious adaptations.

Concerning *where* the adaptation should be applied, it can happen to different levels of the technology stack, from the underlying hardware to the application software. Even if we analyse software systems, the first architectures proposed for self–management correspond nearly exactly with the early sense-plan-act SPA architectures used in robots [20].

Between the reasons *why* a system should be subject to adaptation, the scientific community put a lot of attention on achieving and maintaining well-defined Quality of Service (QoS) properties in a changing environment, which is still a key challenge for self-adapting systems [6].

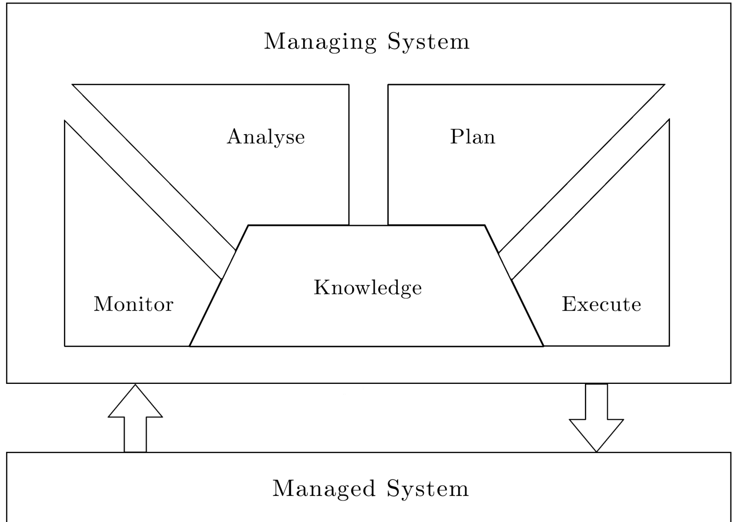


Figure 2.2: Structure of a SAS implementing a MAPE-K loop (based on [41])

Concerning the *how* aspect, the most common approach is the *external approach* [21], as it splits the SAS into adaptation logic (i.e., the managing subsystem) and managed resources (or managed subsystem), increasing the maintainability through modularization. A common engineering approach of this type consists in using a feedback control loop made of four stages – Monitor-Analyse-Plan-Execute – over a Knowledge base. This kind of loop is referred to as MAPE-K loop [4, 19] and it is represented in Figure 2.2. The Knowledge component is the source of truth for the other components of the MAPE loop. It stores data concerning the Managed System, the environment, the adaptation logic, and other relevant data for the adaptation. The Monitor component collects data from the Managed System and the environment through probes (or sensors) and stores them in the Knowledge. The Analyse component performs analysis on the data collected so far in order to check whether the Managed System requires adaptation. When adaptation is required, the Plan component chooses the adaptation actions needed to fulfil the system's goals. Finally, the Execute component is in charge of executing the actions identified by the Plan through the effectors (or actuators) of the Managed System.

A further challenge in engineering self-adaptive systems is to decentralise the adaptation logic by distributing the MAPE components. In a centralised adaptation logic, one sub-system implements the adaptation logic. This solution is cost-effective and easy to maintain, but is not suitable for large systems, due to the high amount of information and the computational power. As a result, patterns on fully decentralised and hybrid approaches (Figure 2.3) were proposed [43].

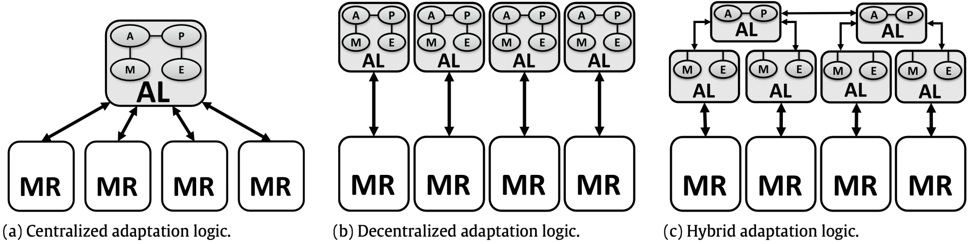


Figure 2.3: Comparison of adaptation logic structures [21]

2.2. Existing Exemplars

Whereas some members of the scientific community propose analytical and methodological approaches to QoS management and optimisation, others point their attention to technological research, which is also the focus of the proposed work.
To analyse the technological works proposed so far, the systematic review of SASs proposed in [44] by Wong et al. was used as a reference.
To restrict our study to our domain of interest, among all the works presented in the review, we selected the ones concerning service-based systems and web services applications and using a closed-loop approach. In addition, only the ones not relying on simulated environments were selected.

Therefore, the most relevant works we focused on are the [23, 26, 40] and [39].
The first work, proposed in 2011 by Patikirikorala et al., focuses on multi-model feedback control systems. In particular, it analyses the application of Multi-Model Switching and Tuning (MMST) to QoS performance management of software systems [26].
The second work, proposed in 2011 by Lulli et al., presents a modular software framework for resource management, based on the integration of dynamic allocation, trading, and self-adaptation mechanisms [23].
The third work, proposed in 2013 by Tamura et al., provides an implementation of the DYNAMICO (Dynamic Adaptation, Monitoring and Control Objectives) reference model for self-adaptive systems, which provides guidelines for designing and implementing SASs where both the adaptation goals and the context monitoring requirements may change at runtime [40].
The last work, proposed in 2014 by Swanson et al., extended the well-known Rainbow framework with new components and algorithms targeting failure avoidance, proposing REFRACT [39]. The Rainbow framework uses software architectures and a reusable infrastructure to support the engineering of self-adaptive software systems [15]. The reusable infrastructure comes together with mechanisms for specializing such infrastructure to the user's needs. It uses an abstract architectural model to monitor an executing system's

runtime properties, then evaluates the model for constraint violation and performs adaptations on the running system, if needed. At the time of writing, the Rainbow framework is widely used as a test bed and extensible tool in self-adaptive systems research. However, it requires some effort to fully understand how to set it up and combine it with a preexisting web or service-based application. It requires both a formal definition of the model – using the ad-hoc *Stitch* self-adaptation language – and a translation infrastructure for communications from and to the managed system, which implies both a high design and development complexity.

In addition, some exemplars published on the website *Software Engineering for Self-Adaptive Systems*[1] were compared to the previous ones. After filtering on the domain of interest (Web / Cloud / Service), we ended up with the following relevant exemplars:

- *Hogna* is a platform for deploying self-managing web applications on the cloud and automating a set of operations, such as the booting of the instances and their setup. Moreover, it enables the continuous monitoring of the health status of the applications [3];
- *TAS* is a service-based system (SBS), whose objective is to provide researchers with an exemplar of service-based application to be adapted [42];
- *SEAByTE* is an exemplar that proposes an experimental framework for testing novel self-adaptation solutions to enhance the automation of continuous A/B testing of a micro-service-based system [28].

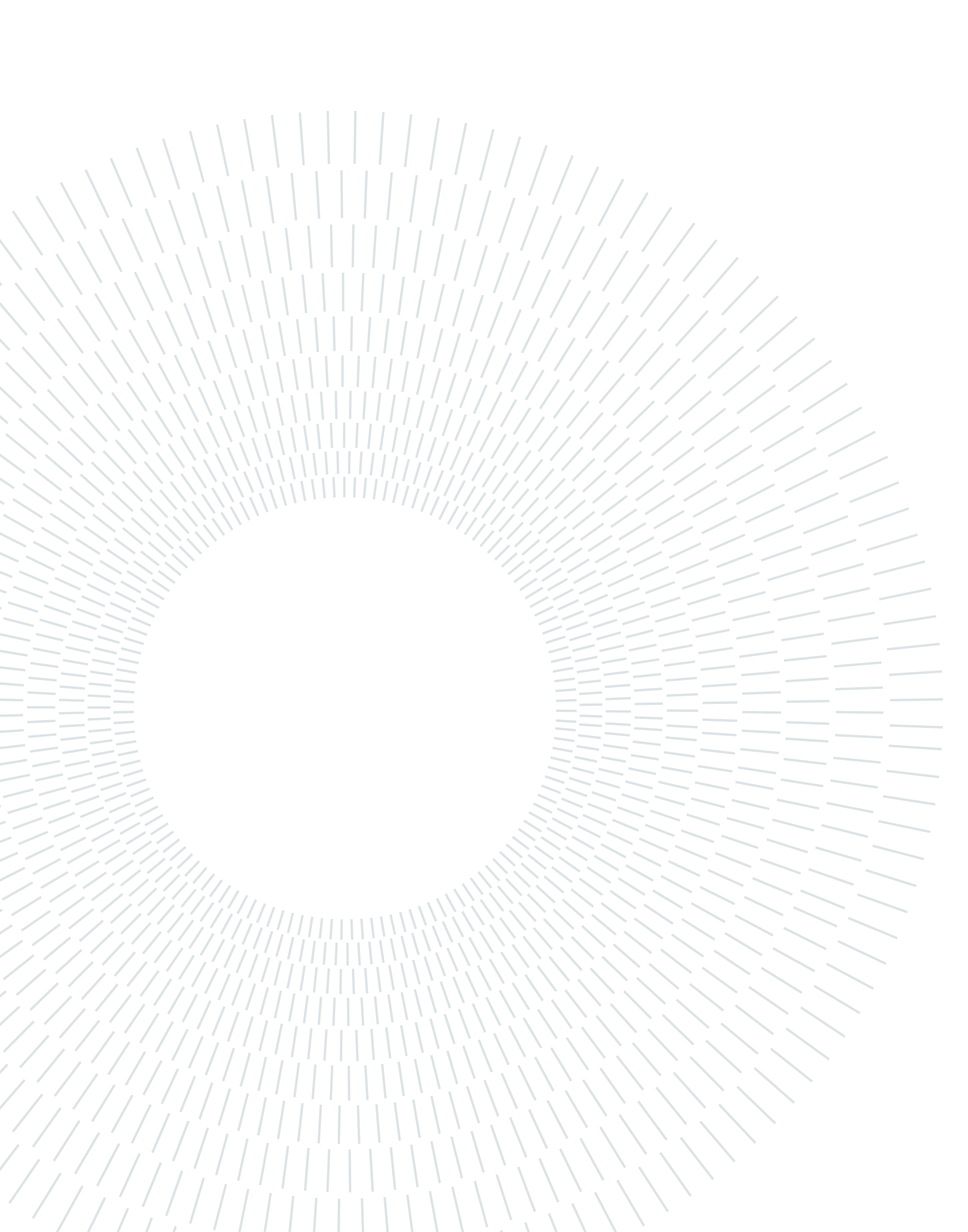
Other exemplars (e.g., *SWIM* [25], *RDMSim* [32] and *EWS* [12]) implement specific managed subsystems ready to be adapted, that can be reused by researchers to evaluate and compare different adaptation logics. For instance, in the case of *SWIM*, an external adaptation logic can interact with it through a TCP-based interface [25].

Among the previously mentioned exemplars, SEAByTE represents a ready-to-use framework, implemented using a well-known technology stack (microservices, Spring, Docker, REST/JSON). However, its application is limited to the specific domain of A/B testing. Conversely, Hogna provides a modular Managing System, which is however specifically focused on the deployment and configuration of cloud applications, especially on the *Amazon EC2* deployment platform.
Finally, the TAS exemplar provides a useful SBS to be used as the managed subsystem of a SAS. Moreover, it provides the definition of a set of generic adaptation scenarios applicable to SBS, to deal with uncertainties of the system itself and of the environment

[1]http://www.self-adaptive.org/exemplars

it is set in, which inspired the one proposed in Chapter 3.

With respect to the presented exemplars, our solution is not limited to domain-specific adaptations or to a specific technology stack, but aims at providing a flexible and reusable Managing System, focused on QoS and performance-related adaptation scenarios, to support different SBSs.



3 | Our Proposal

In this chapter we describe the problem we addressed and the proposed solution.
The chapter is structured as follows:

- Section 3.1 describes the problem statement, and introduces our solution;
- Section 3.2 describes *SEFA*, the Managed System to be adapted by our solution. In particular, this section focuses on its functionalities, on its architecture, and on its relevant design choices;
- Section 3.3 describes *RAMSES*, the Managing System proposed as a solution to the presented problem. In particular, this section provides a deep overview of the system, describing its components, their respective algorithms, and the overall interactions between the system itself and the Managed System it aims to adapt.

3.1. Problem statement and proposed Solution

Nowadays, the majority of service-based systems have specific quality of service (QoS) requirements and constraints to satisfy. However, ensuring their satisfaction is a hard challenge, since the environment in which these systems are set is often subject to frequent changes, that may impact the overall Quality of Service.
Self-Adaptive systems try to mitigate this problem, by proposing a paradigm that, when applied, eases the management of a given system. Indeed, as highlighted in Chapter 2, their goal is to help admins in ensuring that their system satisfies a set of user-defined constraints.
This is done by providing a subsystem, called Managing System, in charge of managing the base application, called Managed System, based on its admins' needs.

When developing a Self-Adaptive System, designing both the Managing and the Managed System together introduces many advantages: the Managing subsystem would be tailored to the Managed one, that in turn would be designed to efficiently and effectively interact with the former. Indeed, when designing a Self-Adaptive System, it is fundamental to provide the Managing Subsystem with the possibility to retrieve data from and apply changes

to the Managed one: without an adequate monitoring process, the need for an adaptation cannot be identified and, therefore, the adaptation process cannot be started [21]; also, without the possibility to actually modify the Managed System behaviour, the identified adaptation options would be useless.
However, this is not often the case: in general, a common situation is the one in which an already existing application needs to be supported by a not yet implemented Managing System. Again, in this case, a solution would be to design an ad-hoc Managing System, that suits the specific domain and needs of the Managed one.

Despite their benefits, both these practices have downsides: indeed, designing a Managing System tailored to a given application introduces tight coupling in the overall SAS. This binding could lead to maintainability issues and increasing costs.
In fact, given the nowadays pervasiveness of software systems, companies holding different applications would need to design several different ad-hoc Managing Systems, which may be infeasible due to their high development time and cost. Moreover, since most of the service-based applications share some high-level QoS requirements (e.g., availability, response time), it is very likely that the different Managing Systems to be developed would share their non-domain-specific logic.
Our solution to the problem of finding a better trade-off between costs – in terms of time, resources, design and development – and effectiveness of the SAS is to design a modular MAPE-K-based Managing System, RAMSES. RAMSES is a Reusable Autonomic Manager for microServicES, made by configurable and extendable components, whose behaviour does not depend on the specific Managed System to be adapted, and is thus reusable with different Managed Systems.

In order to design RAMSES, it was necessary to identify the adaptation scenarios common to the majority of SBSs. When designing (micro)service-based systems, it is common to include in their architecture load-balancing components and resiliency patterns (e.g., Circuit Breaker), described in Section 3.2.2. This, together with the common interest of nowadays systems to satisfy specific QoS requirements, made us define the adaptation scenarios summarized in Table 3.1, which guided the overall design process of the proposed solution.

With respect to the current Self-Adaptive Systems taxonomy, mentioned in Section 2.1, RAMSES offers reactive adaptation (*when?*) at the application level (*where?*). To reach its goals, it follows an external approach, implementing a MAPE-K loop through separate components (*how?*), easing the implementation of centralized, decentralized and hybrid adaptation logic. Hence, parameter-level and structure-level adaptation (*what?*) is performed in reaction to the changes that make the managed subsystem violate specific QoS

Scenario	Observable properties	Examples of adaptations
S1: Violation of QoS specifications	Values of the QoS indicators of the service over time (e.g., availability, average response time)	– Change the current service implementation – Add (load balanced) instances in parallel – Shutdown of an instance with low performance – Change configuration properties (e.g., load balancer weights, circuit breaker parameters)
S2: Service unavailable	Success or failure of each service invocation	– Change the current service implementation – Add (load balanced) instances in parallel
S3: Better service implementation available	Properties of the service implementations (e.g., average response time, preference)	– Change the current service implementation

Table 3.1: Adaptation scenarios

requirements and constraints (*why?*). Also, concerning the properties of autonomic systems, described in Section 2.1, RAMSES is meant to provide the Managed System with Self-optimizing and Self-healing capabilities.

Finally, in order to provide a complete Self-Adaptive System, a managed subsystem, SEFA, close to a real-world application is provided, in order to test the functionalities and effectiveness of RAMSES in a non-simulated environment.
An in depth description of both SEFA and RAMSES is provided, respectively, in Section 3.2 and Section 3.3.

3.2. The Managed System: SEFA

In order to test the proposed solution, described in Section 3.3, and to build a system that could be easily reused and adapted for research purposes, our project includes a Java-based microservice application that will be object of the adaptation, SEFA: a SErvice-based eFood Application. Since it was not the aim of this work, its design is open to future improvements and refinements. Despite this, the system is complete and close to a real-world application, which satisfies our need of testing RAMSES in a non-simulated environment.

3.2.1. Introduction to the application

SEFA is an eFood application[1] with a microservice-based architecture, implemented using the Spring Boot [33] and Spring Cloud [35] frameworks.
The goal of this application is to allow customers to browse the list of restaurants handled by SEFA and their respective menus, choose some dishes, and finally place orders, paying them by credit card and getting them delivered to a specific address.

As the architecture is designed according to the *microservice pattern* [30], the server-side logic is made up of multiple components – the *core* services. The services expose REST APIs using the JSON format. The core services are the following:

- the **Restaurant Service**, in charge of managing the restaurants available on the application and their respective menus;
- the **Ordering Service**, in charge of managing all the customers' carts, and of allowing them to place their orders; it interacts with the Restaurant Service and with the two proxy services to reach its goal;
- the **Payment Proxy Service**, in charge of mediating with a third-party payment service provider (i.e., who processes the payment) during the elaboration of the order;
- the **Delivery Proxy Service**, in charge of mediating with a third-party delivery service provider (i.e., who delivers the order to the customer) during the elaboration of the order.

Since multiple third-party providers can be used for the payment and for the delivery of the orders, the system is designed according to the *adapter pattern* [14]. This allows the Ordering Service to interact with different third-party APIs, without the need of handling separately the web interface of each third-party provider. Indeed, each of them is contacted by the Ordering Service through its own proxy, which must offer to the Ordering Service a predefined interface, common to all the proxies of the same service (i.e., Payment and Delivery).
Together with SEFA, three different third-party providers for the payment service – resulting in three different implementations of the Payment Proxy Service – and three possible third-party providers for the delivery service – resulting in three implementations of the Delivery Proxy Service – have been realised. However, the Managed System is assumed to be using no more than one specific implementation per service at the same time, as

[1]Inspired by the educational projects proposed by professor Luca Cabibbo [5].

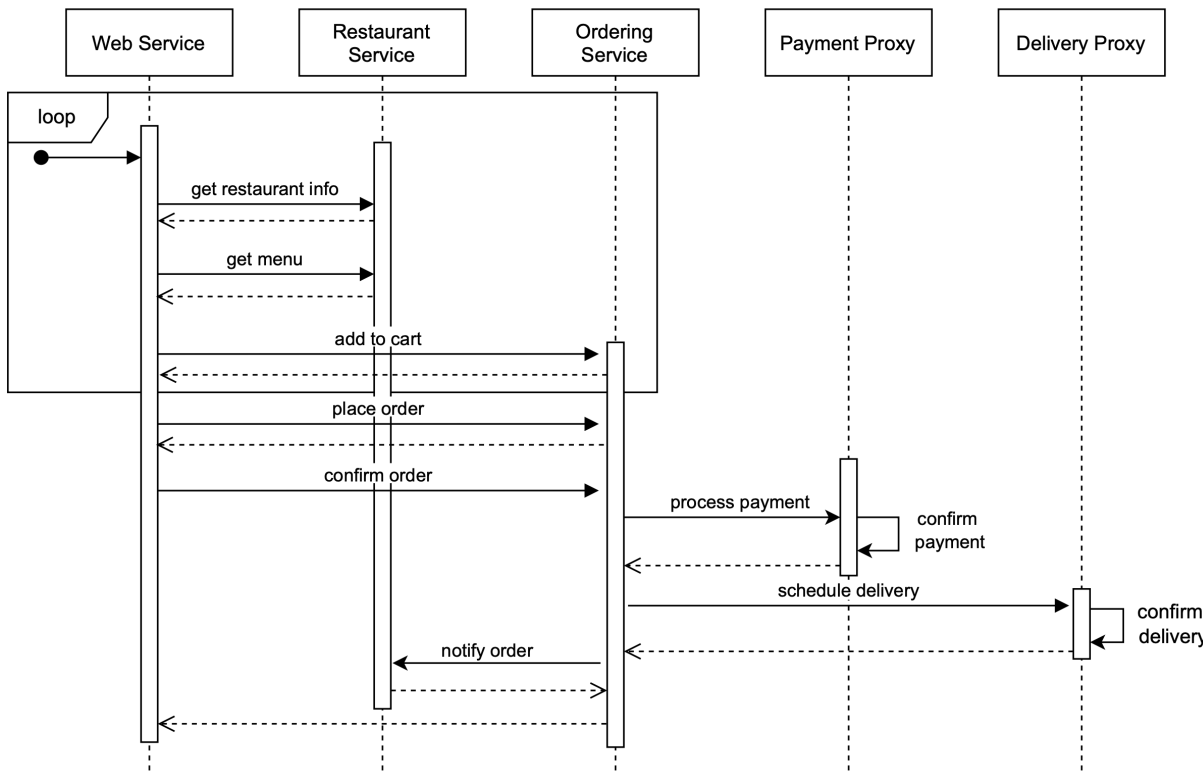


Figure 3.1: Use case of a complete interaction with the application – Sequence diagram

described in Section 3.3.1. Hence, for both services, RAMSES might propose adaptation on the third-party provider to choose.

The UML sequence diagram in Figure 3.1 illustrates the workflow originating from the interaction of the user with the application.

3.2.2. Relevant design choices and system architecture

This section motivates the main choices concerning the design of the application and illustrates the final system architecture.

To begin with, since SEFA highly depends on third-party services, the circuit breaker pattern [8] is used, in order to improve the resiliency of the application and to mitigate the impact that potential failures or abnormal response times of the third-party services may have on the end-user.
The circuit breaker does not interfere with the normal operations when it is in the CLOSED state, but its state evolves from CLOSED to OPEN when either the failure rate or the slow-calls rate are greater than the respective thresholds. In this case, the circuit breaker temporarily disallows further calls to that third-party service, and performs

user-defined fallback operations in place of the requested ones.
The failure rate and the slow call rate thresholds, as well as the other properties of the circuit breaker, are configurable by the user and they might be subject to adaptation if handled by the Managing System.
In our system, this resiliency pattern is applied to the Ordering Service, implemented using the Spring Resilience4j circuit breaker [29]. In particular:

- for the delivery service, the fallback method provides the user with the possibility to pick up the order at the restaurant;
- for the payment service, the fallback method allows the users to complete the order by paying cash upon delivery or pick-up.

In order to reproduce a real-world application, and to take advantage of the microservices architecture, multiple instances of each service can run in parallel, where the number of instances and their location might change dynamically.
Hence, one of the problems to address when making a request is how to discover the location of a service instance, and how to select one instance among all the available ones. The solution to the problem is, in our application, to apply client-side service discovery and load-balancing [9].
In order to make a request, the client of a service first asks the location of the available instances by querying a *service registry*. In our case, the service registry is implemented using Spring Cloud Netflix Eureka Server [11], and its location (i.e., its public address) must be known to all the microservices. Finally, the instance is chosen by applying the following load-balancing rule. A *fitness* is assigned to each instance, which is chosen using *fitness proportionate selection*, also known as *roulette wheel selection* [24]. In this paper, the fitness of an instance is also referred to as weight, and the load balancing rule as *Weighted Random*.
In order to ease future works on the proposed solution, the load balancers are defined and implemented in a separate project library, which at the time of writing includes:

- a Round-Robin algorithm;
- a Weighted Round-Robin algorithm;
- a pure Random selection rule;
- the already mentioned Weighted Random selection rule, which is the only one currently handled by RAMSES, as described in Section 3.3.

The introduction of client-side service discovery and load-balancing functionalities brings the need of simplifying the interaction between the end users and the application. Indeed,

in contrast with direct client-to-microservice communication, a single entry point in charge of processing and routing the end users' requests to the internal microservices eases the communication.
This is achieved by applying the API gateway pattern [2], implemented using Spring Cloud Gateway [37]. As a result, the gateway is a standalone microservice acting as a front door to the core services: any REST client can interact with SEFA only by making requests to the gateway.

Having multiple instances of the same service running in a distributed setting introduces another challenge, which consists in how to let them share the same configuration. In our case, we want all the instances of the same service to look the same to an external client, from a behavioural point of view. As a result, the solution is to have a centralized configuration server, which, for each service, holds the configuration properties to use at run-time, and updates all the involved instances when a configuration change is needed.

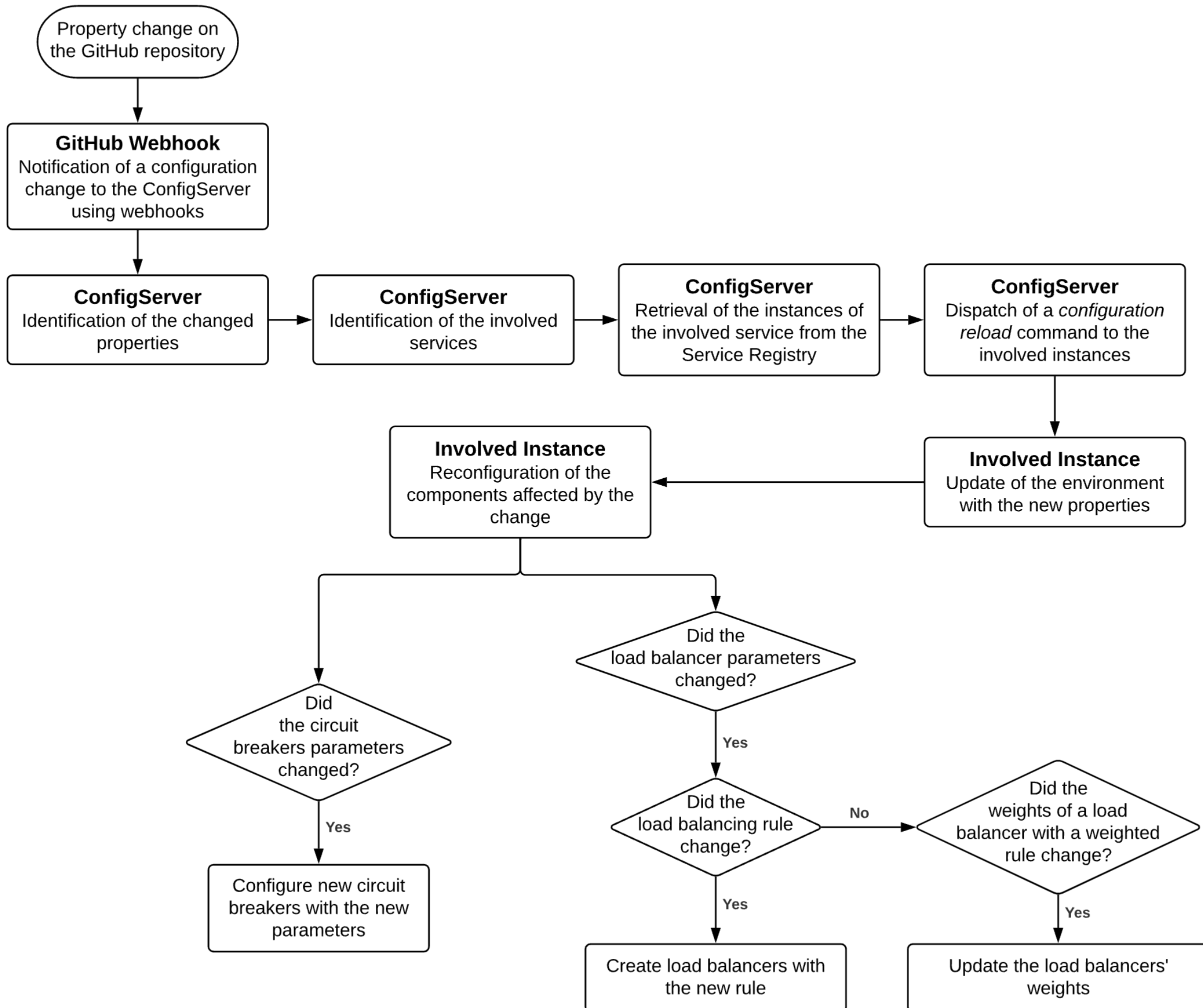
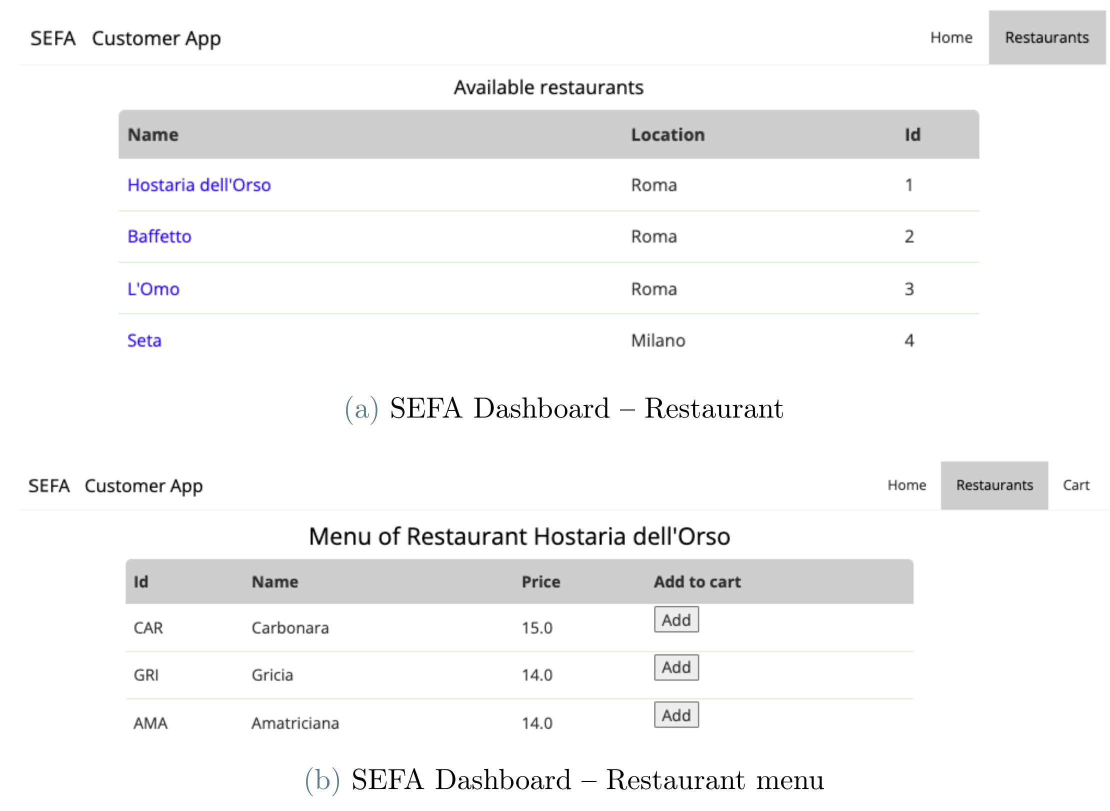


Figure 3.2: Event Diagram of a property change

This mechanism is implemented using Spring Cloud Config [36]. Indeed, a configuration server runs as a standalone Spring Boot application, while the property sources are stored on a remote *git* repository: in the case of our application, the repository is hosted on GitHub[2].
All the core services use the repository as a property source: during the boot process each instance fetches the configuration from the repository to initialize its environment.
When a property is changed on the repository, the configuration server is notified via a dedicated webhook [16]: when this happens, the configuration server analyses the new configuration and notifies the instances involved by the changes. Hence, all the core services implement a handler to react to such events.
The flow of the property change event is summarized by the event diagram shown in Figure 3.2.

Due to the nature of the application, in order to offer their functionalities, the Restaurant Service and the Ordering Service require some data to be persisted (e.g., the restaurants and their menus, the orders made by the customers). As a consequence, the database-per-service pattern is applied [10]. Indeed, these services persist their data in independent MySQL databases through the use of Spring Data JPA [38].



(a) SEFA Dashboard – Restaurant

(b) SEFA Dashboard – Restaurant menu

Figure 3.3: SEFA Dashboard

[2]GitHub, Inc. – `https://www.github.com`

Finally, in order to let a human interact with SEFA through a graphical user interface (GUI), a web application is also provided. It serves as the front-end of the application, acting as a REST client that communicates with the API gateway.
The web server is a Spring Boot application, that uses Thymeleaf[3] as template engine. An example of the GUI is shown in Figure 3.3.

As a result, the final software architecture is illustrated by the microservices architecture diagram shown in Figure 3.4.

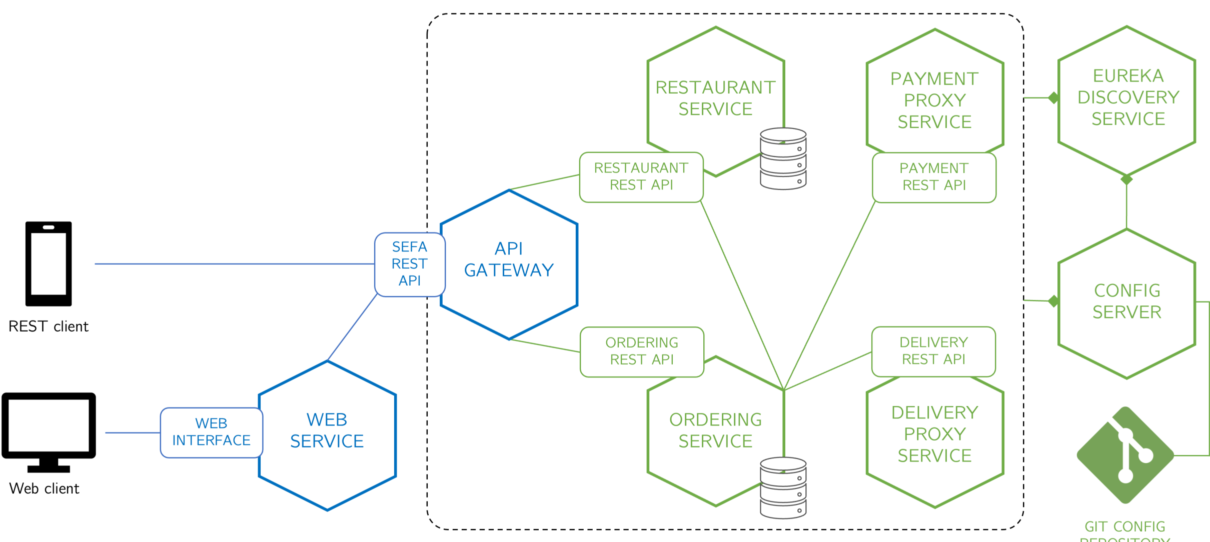


Figure 3.4: Software architecture - Microservices diagram

In particular:

- The Microservices in blue are the ones exposed to the end user: they represent the front-end of the architecture.
- The Microservices in green constitute the back-end of the architecture. Obviously, the components implementing the adapter pattern (namely the Payment Proxy Service and the Delivery Proxy Service) interact with the respective third-party service providers, which are not included in the diagram as they are not part of the developed Managed System. However, we provide mocked third-party services for testing.
- The Microservices in the dashed rectangle communicate with the Eureka Discovery Service – for client-side service discovery – and with the Config Server – for centralized configuration management. All of them, with the exception of the API

[3]The Thymeleaf Team – `https://www.thymeleaf.org/`

Gateway, are the ones implementing the server-side logic and are subject to adaptation.

- A REST client simulating a complete and automated interaction with SEFA is also provided for testing.

3.2.3. API overview

As shown in Figure 3.4, both the core services and the API Gateway offer different APIs, in order to allow clients to perform their requests.

The Restaurant Service offers two different sets of REST APIs, one for the customers' operations and one for the system admin's operations.
The former API includes the following endpoints:

- GET `/rest/customer/restaurants/`, which provides the client with the list of restaurants managed by SEFA;
- GET `/rest/customer/restaurants/{restaurantId}`, which provides the client with the details of the specified restaurant;
- GET `/rest/customer/restaurants/{restaurantId}/menu`, which provides the client with the menu of the specified restaurant;
- GET `/rest/customer/restaurants/restaurantId/item/{itemId}`, which provides the client with the details of the specified item present in the menu of the specified restaurant;
- POST `/rest/customer/restaurants/{restaurantId}/notify`, which allows the client to notify the specified restaurant when a new order for that restaurant is confirmed.

Conversely, the latter API includes the following endpoints:

- GET `/rest/admin/restaurants/`, which provides the client with the list of restaurants managed by SEFA;
- POST `/rest/admin/restaurants/`, which provides the client with the possibility of adding a new restaurant to the list of restaurants managed by SEFA;
- GET `/rest/admin/restaurants/{restaurantId}`, which provides the client with the details of the specified restaurant;
- GET `/rest/admin/restaurants/{restaurantId}/menu`, which provides the client

with the menu of the specified restaurant;

- PUT `/rest/admin/restaurants/{restaurantId}/menu`, which provides the client with the possibility of creating or updating the menu of the specified restaurant.

A more specific representation of the described APIs, which includes details concerning the requests and responses bodies, is provided in Appendix B.1.1.

The Ordering Service provides a REST API that allows the customers to create and manage their cart, and to place orders. In particular, the API includes the following endpoints:

- POST `/rest/`, which allows the client to create a new cart associated with the restaurant specified in the request;
- GET `/rest/{cartId}`, which provides the client with details about the specified cart;
- POST `/rest/{cartId}/addItem`, which allows the client to add the item specified in the request to the selected cart;
- POST `/rest/{cartId}/removeItem`, which allows the client to remove the item specified in the request from the selected cart;
- POST `/rest/{cartId}/confirmOrder`, which allows the client to place a new order based on the selected cart;
- POST `/rest/{cartId}/confirmCashPayment`, which allows the client to confirm the customer's possibility to pay the order associated with the specified cart in cash, upon delivery or takeaway, when online payment is not available;
- POST `/rest/{cartId}/confirmTakeAway`, which allows the client to confirm the customer's possibility to personally pick up the order associated with the specified cart at the restaurant when delivery is not available;
- POST `/rest/{cartId}/rejectTakeAway`, which allows the client to cancel the order associated with the specified cart at the restaurant when delivery is not available;

A more specific representation of the described API, which includes details concerning the requests and responses bodies, is provided in Appendix B.1.2.

The Payment Proxy Service offers a REST API composed of a single endpoint, that allows its clients to contact the third-party provider that its implementation is associated with. In particular, the API is the following:

- POST `/rest/processPayment`, which allows the client to provide all the required pieces of information for making a payment.

A more specific representation of the described API, which includes details concerning its request and response body, is provided in Appendix B.1.3.

Analogously to the Payment Proxy Service, the Delivery Proxy Service offers a REST API composed of a single endpoint, that allows its clients to contact the third-party provider that its implementation is associated with.
In particular, the API is the following:

- POST `/rest/deliverOrder`, which allows the client to provide all the required pieces of information for requesting the delivery of an order.

A more specific representation of the described API, which includes details concerning its request and response body, is provided in Appendix B.1.4.

To conclude, the API Gateway offers a set of APIs serving as a single entry point to the application. Its APIs are a small variation of the ones offered by the Restaurant Service and by the Ordering Service. In particular:

- the `/rest` prefix is removed from all the endpoints;
- the prefix `/customer/cart` is added to all the URLs of the Ordering Service API.

3.2.4. Probe and Actuator requirements and implementation

Prerequisites

In order to allow RAMSES to correctly monitor its Managed System and to effectively perform operations on it, a *Probe* component and an *Actuator* component must be provided together with the Managed System itself. These two components are an abstraction layer to enforce the reusability of RAMSES, decoupling it from the actual technology stack used by the Managed System.

The Probe component is considered by RAMSES as the source of truth when determining which are all the running managed instances at any moment. Thus, it must be able to provide RAMSES with an always up-to-date representation of the Managed System runtime architecture, which is the list of running instances for each service. Moreover, it is assumed that, if the Probe includes an instance in the runtime architecture, that instance is currently able to process requests.
As stated in Section 3.3.1, since a service might have multiple implementations but only

one currently active, the information about the current implementation of a service must be known by an external observer (in our case, RAMSES itself), and thus it must be included in the run-time architecture representation returned by the Probe Component. Finally, the Probe component must be able to provide RAMSES with the up-to-date configuration of any managed service.
The interface that the Probe component must provide is defined in Appendix B.2.

The Actuator component is the one in charge of effectively applying the instructions imposed by RAMSES. Indeed, it must be able to boot new instances of a specific service implementation, shut down specific instances and change the configuration of a given service. Moreover, it is assumed that all the requested actions are eventually performed correctly. However, service configuration changes should be applied in a reasonably short amount of time, which indicatively should be shorter than a threshold, defined by the Equation 3.17 in Section 3.3.7.
The interface that the Actuator component must provide is defined in Appendix B.3.

Our implementation

Using Spring Boot as a framework to develop the microservices, it is easy to instrument them, thanks to Spring Boot Actuator [34]. Spring Boot Actuator is a Web API made by multiple endpoints, that can be used for monitoring and managing each application (i.e., in our case, each microservice).
The most relevant endpoints used in our solution include:

- `GET /env`, in order to retrieve the runtime environment of an instance;
- `GET /health`, in order to retrieve the status of an instance, of its circuit breakers and of its database connections, if any;
- `GET /info`, in order to retrieve information about the application;
- `GET /prometheus`, in order to get data from the Prometheus monitoring tool, better described later in this section;
- `POST /refresh`, in order to make the application reload the configuration properties after a property change. In particular, this endpoint is used by the Config Server.

A noteworthy mention is needed for the Prometheus endpoint. The Prometheus tool eases the retrieval of many metrics of an application, such as statistics on the HTTP requests made to the exposed endpoints and on the resource usage (e.g., CPU usage, memory usage). These metrics are collected by the Probe component of the Managed System, which is required by RAMSES as described in Section 3.3.1.

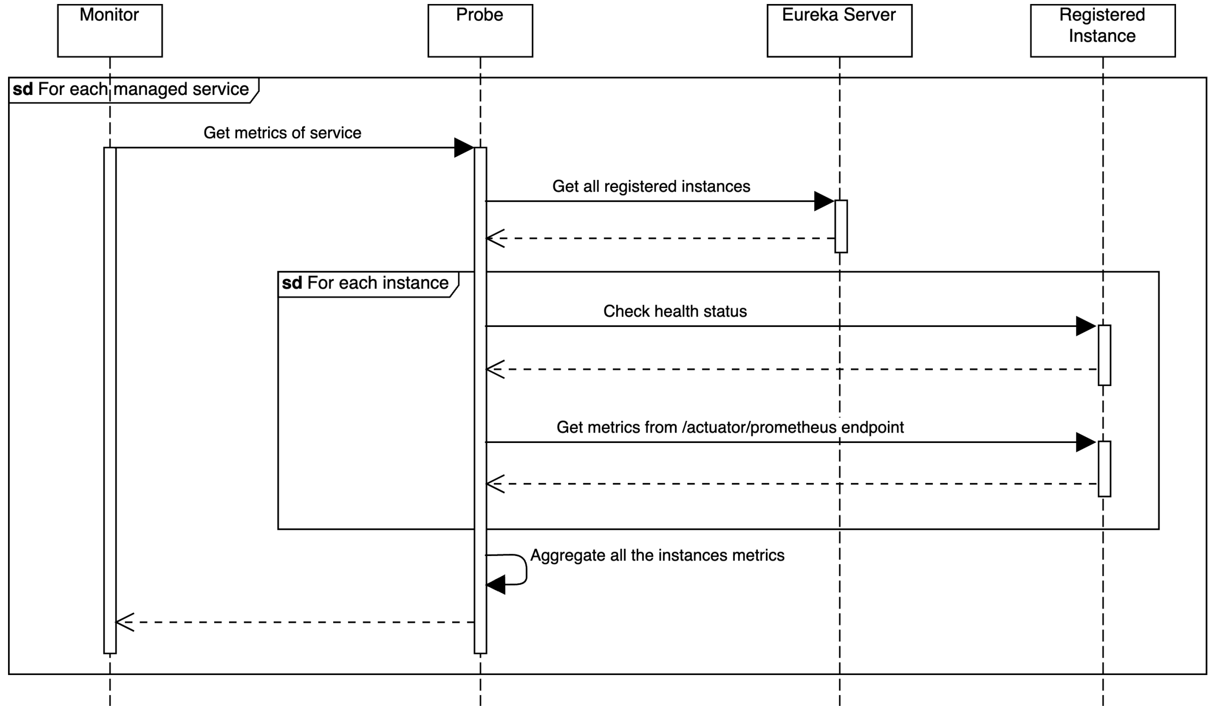


Figure 3.5: Workflow of the Probe main task – Sequence diagram

RAMSES – and in particular its *Monitor* component, described in 3.3.4 – interacts with a Probe component to maintain an up-to-date runtime architectural model of the Managed System, as proposed in [41]. The logic of the Probe of SEFA is based on the interaction with the Eureka Server – to retrieve the runtime architecture of each service as seen from the point of view of a client – and with the Prometheus endpoint exposed by the Spring Boot Actuator framework – to retrieve the latest metrics of each instance. The result of the interaction with Prometheus is then parsed using a data scraper, imported as external library[4].
The workflow of our implementation of the Probe is illustrated by the sequence diagram shown in Figure 3.5.

Finally, RAMSES – and in particular its *Execute* component, described in 3.3.7 – interacts with an Actuator component in order to make the adaptation options effective [41].
In our case, the Actuator component is actually made of two subcomponents:

- the *Instances Manager*, in charge of allocating and deallocating instances upon request. More specifically, it starts or stops service instances allocated in Docker containers, hosted on a machine exposing the Docker daemon;
- the *Config Manager*, in charge of updating the configuration repository when a

[4] An adapted version of the open-source Prometheus Metrics Scraper [27].

property needs to be changed, added or removed. More specifically, it commits and pushes the changes to the GitHub repository used by the configuration server.

The machine target of the Instances Manager and the repository target of the Config Manager are properties configurable by the user through dedicated environmental variables.

The workflow of our implementation of the Actuator can be illustrated with the sequence diagram in Figure 3.6.

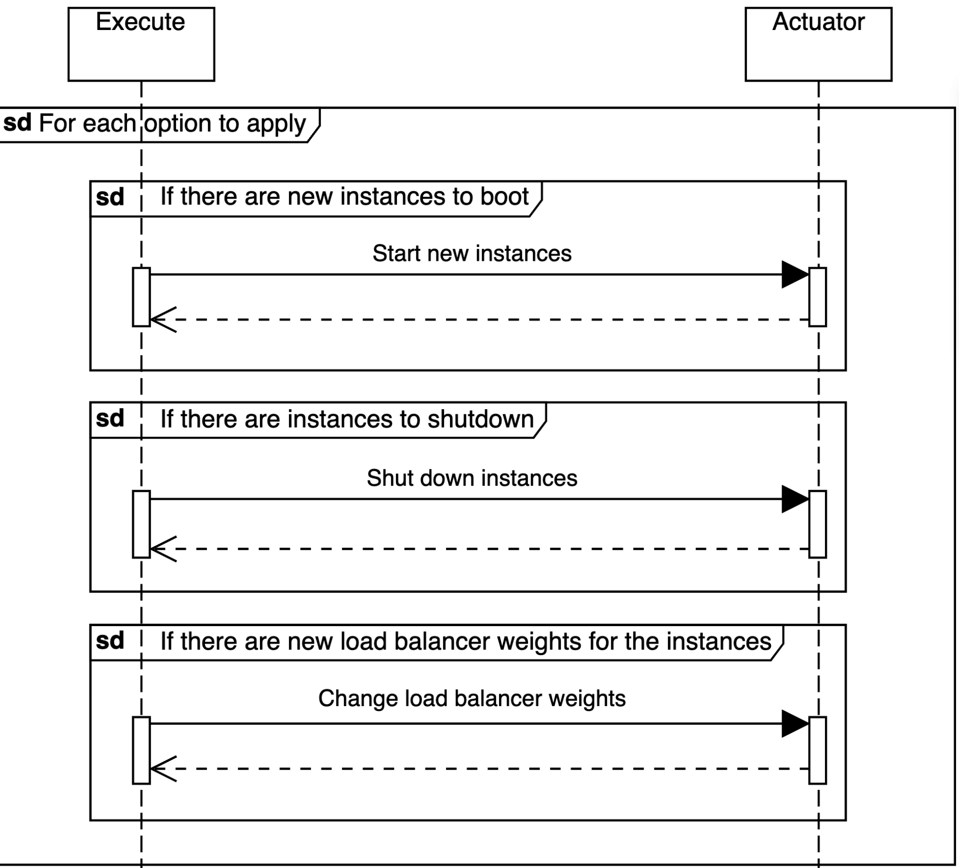


Figure 3.6: Workflow of the Actuator main task – Sequence diagram

3.3. The Managing System: RAMSES

Our proposed solution to the problem is RAMSES, a Reusable Autonomic Manager for microServicES. Indeed, RAMSES is a Managing System designed as a distributed MAPE-K loop, implemented in Java using the Spring Boot framework [33]. In particular, each component of the MAPE-K loop has been designed as a standalone microservice, to bring new advantages to the classical monolithic implementation of the loop. In this way, we allow the users to deploy and replace the components of the loop independently. Moreover,

the design of RAMSES is agnostic with respect to the system it is in charge of adapting, enforcing its reusability with different Managed Systems.

In the following section, a more detailed explanation of the different stages of the loop and their respective components is provided, together with a complete description of the System Model of our solution and how it is represented in the Managing System, and all its prerequisites and assumptions. Figure 3.7 summarizes the workflow of a single iteration of the MAPE-K loop.

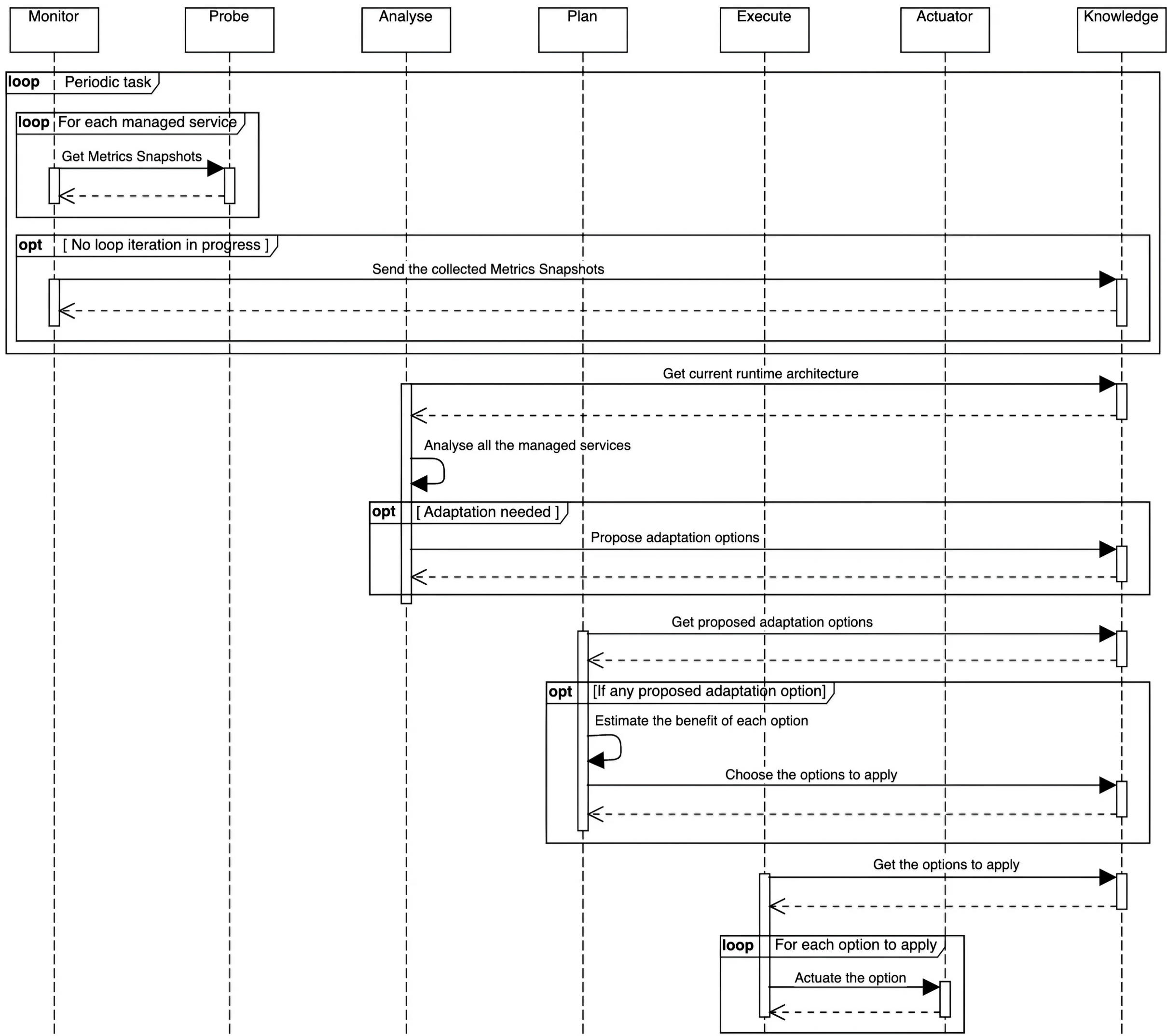


Figure 3.7: Workflow of a MAPE-K loop iteration – Sequence diagram

3.3.1. Prerequisites and assumptions

Our Managing System is designed to interact with a SBS, called Managed System. There are different prerequisites that the Managed System must satisfy in order to allow the

Managing one to correctly perform its operations.

To begin with, in order to let RAMSES operate, the user must describe the Managed System and its architecture in the dedicated configuration files. More details on their content and structure are provided in Section 3.3.3.

Secondly, even if each service may be implemented by different service implementations, it is assumed that, at any moment, all the running instances of the same service are instances of the same service implementation, also known as the current implementation of that service.

Moreover, as anticipated in Section 3.2.4, a Probe component and an Actuator component must be provided together with the Managed System itself, in order to allow RAMSES to correctly monitor it and to effectively perform operations on it.

Concerning the operations performed via the Actuator, RAMSES also assumes that no external actor is allowed to change the Managed System configuration elements which are subject to adaptations (i.e., the addition and shutdown of instances, and the change of load balancer weights).

Finally, when monitoring each service and proposing adaptation options, RAMSES always assumes that all the instances of a given service that are present in the runtime architecture provided by the Probe component actually contribute to elaborating and processing the clients' requests directed to their service. This should happen independently from the load-balancing technique adopted by the Managed System.

3.3.2. System Model

In order to let RAMSES interact with its Managed System and propose adaptations, it is fundamental to precisely define a System Model, shown in Figure 3.8, that represents all the entities that the system needs to correctly model the environment and perform adaptation.

The *Service* is the principal entity of the model. It is the model counterpart of each service of the Managed System, and it includes all the relevant pieces of information about a managed service. Each Service is identified by a unique ID, may be associated with the other services it depends on (also called service dependencies), and holds a *Service Configuration*.

A Service Configuration encapsulates information about the configuration of each managed service, that include the type of load-balancer used by the service, and the weights

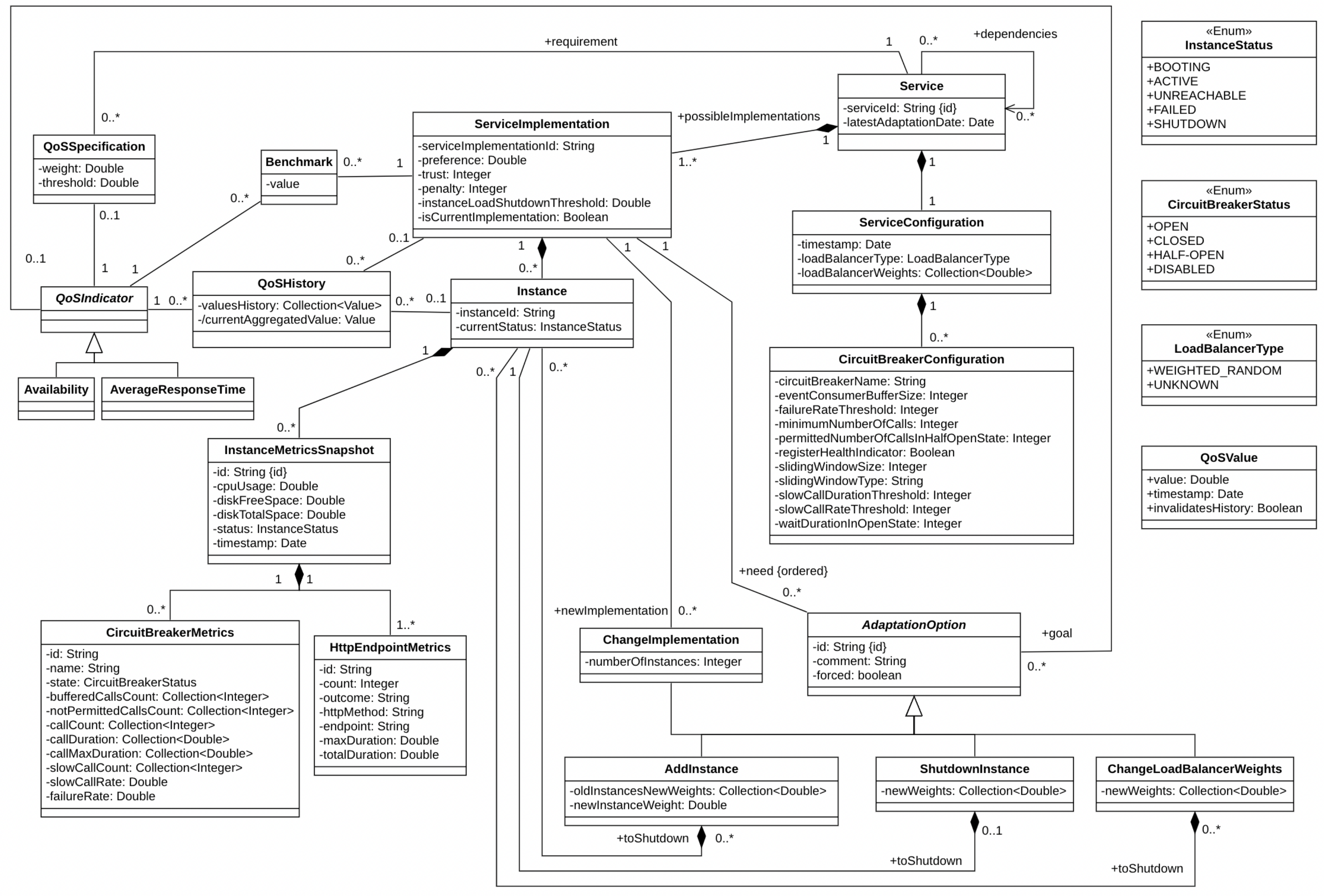


Figure 3.8: UML class diagram describing the System Model

of each service instance when the load-balancer type is *WEIGHTED RANDOM*. Also, if the service presents one or more circuit breakers, their configuration is represented in different *Circuit Breaker Configuration*s, held by the Service Configuration itself.

Since a managed service may have different implementations, each Service keeps track of what are all its possible *Service Implementation*s. A Service Implementation models its homonym counterpart of a given managed service. It is identified by its *Service Implementation ID* and holds different values: a *penalty* indicator, a *trust* parameter, a *preference* parameter and an *instance-load shutdown threshold* parameter.
The first parameter – *trust* – indicates how much an implementation is trusted by the user and represents a threshold for the *penalty*. Indeed, the *penalty* indicator represents the tendency of a Service Implementation to require adaptation. It is a value initialized to 0 that is incremented whenever that implementation requires adaptation; when it exceeds the threshold, the Managing System considers changing the current implementation for that service.
The second parameter – *preference* – represents a preference of the user towards a specific implementation of a service with respect to the others, and it is involved in the decision-making process of determining if and how to change the current implementation of the service.
Finally, if the service requests are distributed among its instances through a weighted load-balancer, the *instance-load shutdown threshold* parameter, also referred to simply as *shutdown threshold*, represents the minimum amount of requests that each instance should process in order to not being shut down. It is expressed as a percentage of the number of requests the instance would process if the request load was equally distributed between all the instances of that service. This means that each running instance weight must be greater or equal to the *shutdown threshold* divided by the number of instances of that service. The explanation of this result follows.

Call s the *shutdown threshold*, i the considered instance, r_i the number of requests to be processed by i, I_r the set of instances to load balance, n its cardinality and R the total number of requests (i.e., $R = \sum_{i \in I_r} r_i$).
By definition, s is computed as

$$s = \frac{r_i}{R/n}, \tag{3.1}$$

where the denominator represents the number of requests that an instance would process if the requests load was equally distributed between all the instances.

Given s, we want the number of requests r_i to assign to i to satisfy the following relation

$$r_i \geq s \cdot {}^{R}/_{n}, \tag{3.2}$$

obtained from 3.1.
Since statistically the weight w_i of the instance i can be expressed as

$$w_i = {}^{r_i}/_{R}, \tag{3.3}$$

combining Equation 3.2 with Equation 3.3, it results that

$$w_i \geq \frac{s \cdot {}^{R}/_{n}}{R}, \tag{3.4}$$

$$w_i \geq \frac{s}{n}. \tag{3.5}$$

As stated before, Equation 3.5 shows that the weight of an instance must be greater than or equal to the *shutdown threshold* divided by the number of instances of that service.

This threshold provides the user with another instrument to customize the behaviour of RAMSES, and add constraints to the Managed one.
Moreover, since the weights belong to the field of real numbers, this additional threshold helps to prevent the case in which an instance is assigned with an infinitesimal weight due to its poor performance, without being shut down.

The three parameters are specified by the user when defining the architecture of the Managed System, as described in Section 3.3.3.

In addition to those values, we keep track of each implementation whether it is the currently active one or not: only one of the possible implementations of a service can be the currently active one, as specified in Section 3.3.1. For this reason, from this moment on, when we refer to properties of a service that actually belong to the Service Implementation entity (e.g., the service instances, the service QoS History, etc.), we actually refer to the properties of its currently active implementation.

Together with each Service Implementation, it is fundamental to have in the System Model an entity to model the running instances of such implementations. In our model, an *Instance* of a given service is identified by its instance ID, provided by the Probe component, and it is bound to its specific Service Implementation. Moreover, the model keeps track of the current status of the instance, which can be:

- *BOOTING*, when the instance is started by the Actuator upon the request of the

Managing System;

- *ACTIVE*, when the instance is running (i.e., in the run-time architecture provided by the Probe) and responding to new requests;
- *UNREACHABLE*, when the instance is running but not responding to new requests;
- *FAILED*, when the instance is not running anymore without any shutdown request performed by the Managing System;
- *SHUTDOWN*, when the instance is shut down by the Actuator upon the request of the Managing System.

From this moment on, we identify as *running instances* all the instances whose status is either *BOOTING*, *ACTIVE* or *UNREACHABLE*. Also, we say that a service is in *TRANSITION* state if it has at least one instance with *BOOTING* or *SHUTDOWN* status.

Every Instance can hold different *Instance Metrics Snapshots*. The Instance Metrics Snapshot is used to represent and store the different metrics collected by the Managing System from the instance it refers to. It includes the date and time of creation of the snapshot, the status of the instance when the metrics were collected (either *ACTIVE* or *UNREACHABLE*) and hardware-related metrics of the instance as the CPU usage and the disk free and total space. In addition, an Instance Metrics Snapshot also comprehends metrics related to the HTTP requests elaborated by the instance and to the circuit breakers of that instance, if any. The former are stored as *HTTP Endpoint Metrics*, while the latter as *Circuit Breaker Metrics*.
The Circuit Breaker Metrics entity comprehends the name of the circuit breaker it refers to, its status and all its statistics, which include the failure rate, the slow call rate and different counters relative to the number of requests and their duration, grouped by request outcome, as shown in Figure 3.8.

Instead, the HTTP Endpoint Metrics entity models the different metrics related to the HTTP request processed by the instance it refers to, divided by endpoint, HTTP method and request outcome.

The different metrics collected by RAMSES are used to extrapolate values of different *Quality of Service Indicators*. In our case, these indicators are the *Availability*, and the Average Response Time per successful request.
Among the three ways availability can be empirically measured, proposed by the AWS whitepaper on availability of distributed systems [22], our Managing System considers server-side request success rate.

In this case, the availability A of a service can be expressed as

$$A = \frac{SuccessfullyProcessedUnitsOfWork}{TotalValidUnitsOfWorkReceived}, \tag{3.6}$$

where the unit of work is the HTTP request, as ours are request-based services. Nonetheless, the by-design modularity of RAMSES easily allows to define and handle additional indicators.
These indicators can be computed at different granularity levels, going from the instance to the service implementation. Thus, in our model, the Service Implementation and the Instance hold a history of values of those indicators, the *QoS History*, one for each QoS indicator (i.e. the availability and the average response time).
The values held by the QoS Histories can be either valid or not valid: this difference determines whether or not a value is meaningful with respect to the current state of the Managed System.
Additionally, each QoS History also stores what it is considered by RAMSES to be the current value of the QoS indicator it refers to. More details on how the different QoS values are derived are provided in Section 3.3.5.

For each service implementation of any service, the user is required to provide a value of all the different QoS indicators, called *QoS Benchmark*s (i.e., reference values). These represent a sort of Service Level Agreement between the Managing System and different service implementations. These benchmarks are specified by the user at the startup and are supposed to be updated when needed, in order to provide RAMSES with an always reliable and up-to-date estimation of each QoS value. A crucial aspect of RAMSES decision-making process is the reliability of the Benchmarks' values. For example, when allocating new instances, or when a new instance still lacks of values in its QoS histories, these Benchmarks' values are the only metrics which RAMSES can rely on for its estimations.

The primary goal of our Managing System is to allow the user to specify constraints for each Quality of Service indicator, and then perform adaptation in order to satisfy those constraints. Therefore, it is fundamental to represent those specifications in our model, which is done through the *QoS Specification* entity.
Indeed, a QoS Specification is a requirement of a specific Service, is related to a specific QoS and holds a *threshold* to satisfy (lower bound for the availability, upper bound for the average response time). Moreover, for each service, the user can specify how much a QoS Specification is relevant with respect to the others of the same service: this preference is represented by a *weight*.

To ensure that the different managed services are compliant with the different QoS Specifications, and also to provide recovery capabilities after detecting failures, the Managing System must propose and then execute adaptation options. These are represented in our model by the *Adaptation Option* entity. An Adaptation Option may have a specific goal referred to a specific QoS Indicator, and always refers to a specific Service Implementation. Adaptation options may have different priority levels. The current version of RAMSES is able to handle two levels of priority: the *forced* and the *non-forced* ones. A forced adaptation option is always applied after its proposal, without considering its potential benefits. Conversely, the non-forced options related to the same service are compared to select the one bringing the highest estimated benefit to their service. More details on this behaviour are provided in Sections 3.3.5 and 3.3.6.
Our system includes four different types of adaptation options:

- the *Add Instance* option, which represents the action of adding a new instance of the (current) service implementation it refers to. In case a *Weighted Load Balancer* is used by the service, it includes the new weights of its instances, and may also indicate some instances to shut down if their new weights are below the *shutdown threshold* specified in their service implementation;
- the *Shutdown Instance* option, which represents the action of shutting down the specific instance it refers to. In case a *Weighted Load Balancer* is used by the service, it includes the new weights of the remaining instances;
- the *Change Implementation* option, which represents the action of shutting down all the instances of the (current) service implementation, and then of starting the same number of instances of the new selected service implementation specified by the option;
- the *Change Load Balancer Weights* option, which, for a service balanced through a Weighted Random load balancer, represents the action of redistributing the weights associated with all the running instances of the (current) service implementation it refers to. It may also indicate some instances to shut down if their new weights are below the *shutdown threshold* specified in their service implementation.

As for the QoS indicators, this is the set of default adaptation options, which can be expanded as long as the logic for handling the new ones is implemented.

Finally, an ordered history of the Adaptation Options required while being the current implementation of a service is held by the Service Implementation. Additionally, the date and time of the latest Adaptation Option applied on a given service are stored by the

Service entity, in order to better appreciate the variation of the different QoS indicators after each adaptation.

3.3.3. Knowledge component

The Knowledge component of the Managing System is the source of truth for the entire MAPE-K loop. Indeed, the System Model is defined and managed by the Knowledge itself, which mediates the interaction between the other managing components and the system knowledge-base. For this reason, all the other components of the Managing System require that the Knowledge component is active.
The Knowledge is also in charge of storing the relevant entities of our model in a dedicated database: in particular, the knowledge stores all the collected Instance Metrics Snapshots, all the QoS Histories of each managed instance, service implementation and service, all the Adaptation Options performed on each service implementation and all the different Service Configurations adopted by each managed service.
Finally, another task of the Knowledge component is keeping track of which stage of the MAPE-K loop is running at any moment.

A fundamental step for the consistency of the System Model stored by the Knowledge with respect to the reality it represents is its initialization process. Indeed, the user of our system is required to fill three different configuration files, in JSON format:

- the *System Architecture* configuration file, that lists the services to manage, their IDs, the managed services they depend on and their different implementations. For each implementation, the configuration file lists its ID and the *trust*, *penalty* and *instance-load shutdown threshold* parameters;
- the QoS Specification configuration file, that lists, for each service, what are its QoS indicators to consider and what are its *threshold* and *weight* parameters;
- the *System Benchmarks* configuration files, that lists, for each service and service implementation, their initial QoS Benchmarks.

More details about the configuration files' structure are provided in Appendix C.

At the startup, after processing those configuration files, the Knowledge component contacts the Probe component, whose IP address must be specified by an environmental variable, and retrieves the list of all the instances already running for all the managed services. This is done in order to build a complete representation of the Managed System's current state, which will be the starting point for performing the first MAPE-K

loop iteration[5].

Finally, the configuration of each managed service is retrieved by the Probe and a Service Configuration object is constructed for each of them.

3.3.4. Monitor

The first phase of the MAPE-K loop of our Managing System is the Monitor one. It involves its homonym component and the Knowledge component. During this stage, the Monitor is in charge of interacting with the Managed System to determine its overall status and collect metrics from all its components. Those metrics are then sent to the Knowledge component, which will store and process them to perform failure detection. After that, the loop continues towards the *Analyse* phase.

Monitor component

The Monitor component represents the first building block of the MAPE-K loop, since it is in charge of periodically collecting data from the Managed System. These data will then be analysed by the Analyse component.
To collect data, at each iteration the Monitor component queries the Probe component provided with the Managed System, asking to perform a snapshot of all the instances of a specific service; this is done for all the services in the Managing System knowledge-base. During a monitor iteration, the outcome of the entire interaction between the Monitor and the Probe is a collection of Instance Metrics Snapshot objects, also named *Metrics Snapshot* or simply *snapshot* for brevity. Each snapshot aggregates all the metrics collected from a specific instance of a specific service.

In order to improve the performance and the failure detection capabilities of RAMSES, we introduce the following optimization to the standard MAPE-K loop: the Monitor routine runs asynchronously with respect to the rest of the loop, periodically. Indeed, while the other components are busy in a specific adaptation loop iteration, the Monitor component continues collecting the snapshots from the instances, accumulating them in a temporally ordered buffer.
In this way, since it is not possible to know a priori how long a single MAPE-K loop iteration will last, we do not stop the Monitor component from collecting metrics snapshots of the instances, helping also in providing a more reliable failure detection mechanism.

When the Monitor routine ends, the different snapshots collected in the just finished

[5]In order to let the Knowledge component initialize properly, it is assumed that there is at least an instance already running for each managed service.

iteration are aggregated in a collection of snapshots and then added to the buffer. This operation allows distinguishing between snapshots of the same instance collected during different iterations. Then, if there is no MAPE-K loop iteration in progress, the snapshot buffer is sent to the Knowledge component and emptied, and the Analyse component is notified to let it start its analysis. Otherwise, if a MAPE-K loop iteration is in progress, the Monitor routine simply restarts.

The scheduling period of the Monitor routine – also known as *Monitor Period* or *MP* – can be tuned according to the users' needs: the component offers a REST API to change the scheduling period through an HTTP PUT request, defined in Appendix B.4.1. While there is no upper bound to its value, the lower bound comes from the Spring scheduling mechanism and it's equal to 1 ms.

Being in a distributed setting, communication failures may happen during the interactions between different hosts. When this happens between the Probe component and one of the managed instances due to some Probe connection issues, or when this happens between the Probe and the Monitor component itself, the currently running Monitor iteration is invalidated and interrupted. Thus, all the metrics snapshots collected during that iteration are discarded and not added to the buffer. This is done to avoid collecting a partial snapshot of the Managed System.

The entire Monitor logic is summarized by Algorithm 1.

Interaction with the Knowledge component

The Monitor component interacts with the Knowledge component in order to determine which are the managed services, and to store the different Metrics Snapshots in the knowledge-base.

However, differently from the interactions between the other managing components and the Knowledge, when the Knowledge itself is about to store the snapshots received by the Monitor, some additional processing is performed.
In fact, the Instance Metrics Snapshots are used to detect potential failures among the Managed System components. When a new buffer is sent from the Monitor to the Knowledge, it is inspected to determine how the status of all the instances of each service evolved while monitoring them. As a reminder, the buffer is a set of collections of Instance Metrics Snapshots, where each collection aggregates all the snapshots collected during a single iteration of the Monitor routine.
For each collection C of Instance Metrics Snapshots in the Monitor buffer, before saving the snapshots in the knowledge-base, it is fundamental to determine what was the set of

Algorithm 1 Monitor

Input:
MP: scheduling period of the Monitor routine.
$Probe$: the interface of the Probe component.
$Knowledge$: the interface of the Knowledge component.
Output:
B: buffer containing collections of Instance Metrics Snapshots, stored in the Knowledge.

```
1:  S ← Knowledge.getManagedServices( )  // Set of managed services
2:  B ← {}  // Buffer of collections of Instance Metrics Snapshots
3:  loop // Every MP milliseconds
4:      C ← {}
5:      iv ← true
6:      for s ∈ S do
7:          IMS_s ← Probe.takeSnapshot(s)
8:          if IMS_s is NULL then  // Invalidate iteration
9:              iv ← false
10:             break
11:         C ← C ∪ {IMS_s}
12:     if iv = true then  // If iteration is valid
13:         B ← B ∪ {C}
14:     if no loop iteration in progress then
15:         Knowledge.processIMSBuffer(B)
16:         B ← {}
```

active instances when C was originally created.
Hence, for each Metrics Snapshot ms in the collection C, different processing operations must be performed. First of all, if the service implementation has changed from when ms was computed, ms is discarded, because the collected metrics refer to a past configuration of the system.
Otherwise, let us define i as the instance which ms refers to. If the status of i in the Knowledge is *SHUTDOWN*, it means that the Monitor component has collected a snapshot of an instance whose shutdown has been requested by the Managing System. Being in this particular (distributed) setting, this scenario can arise due to a shutdown process not yet completed. When this happens, ms is simply ignored.
Otherwise, if the status of i is not *SHUTDOWN* and ms contains new pieces of information with respect to the latest Metrics Snapshot of i saved in the Knowledge, we save the new ms and set the status of i to the status of ms (either *ACTIVE* or *UNREACHABLE*, as described in Section 3.3.2).

Once all the snapshots in the single snapshot collection C have been correctly processed, the failure detection logic identifies which were the failed instances during the Monitor iteration that generated C, if any. These are the instances still present in the knowledge-base with status either *ACTIVE* or *UNREACHABLE*, that however were not present

anymore in the Probe run-time architecture during the Monitor iteration that generated C (thus, there is no Metrics Snapshots for those instances in C).
Then, for each of those Failed Instances, the Knowledge component sets their status to *FAILED* and creates a new empty Metrics Snapshot reporting that status.

Finally, the Knowledge deletes from its knowledge-base all the instances that have been previously shut down by the Managing System (i.e. with status *SHUTDOWN*) for which no new Metrics Snapshots were received in any collection, which means that they have been correctly removed from the Managed System.

The logic explained so far is summarized by Algorithm 2.

Algorithm 2 Knowledge – Add Metrics

Input:
PAI: set of ACTIVE or UNREACHABLE instances during the previous loop iteration.
S: set of managed services.
B: buffer of collections of Instance Metrics Snapshots.
Output:
The modified system runtime architecture and the latest collected Metrics Snapshots.

```
1:  SISM ← {}   // Set of shut down instances with metrics in B
2:  for C ∈ B do   // C is a collection of snapshots belonging to the same Monitor iteration
3:      // Set of currently ACTIVE or UNREACHABLE instances
4:      CAI ← {ms.instance | ms ∈ C ∧ (ms.status = ACTIVE ∨ ms.status = UNREACHABLE)}
5:      for ms ∈ C do
6:          i ← ms.instance
7:          s ← {s ∈ S | s = i.service}
8:          if i.serviceImplementation = s.currentImplementation then
9:              if i.status ≠ SHUTDOWN then
10:                 i.status ← ms.status
11:                 i.latestMS ← ms
12:             else
13:                 SISM ← SISM ∪ {i}
14:     FI ← {i | i ∈ (PAI \ CAI) ∧ i.status ≠ SHUTDOWN}   // Set of FAILED instances
15:     for i ∈ FI do
16:         i.status ← FAILED
17:         i.latestMS ← createFailedMS(i)
18: for s ∈ S do
19:     CSI ← {s.instance | s.instance.status = SHUTDOWN ∧ s.instance ∉ SISM}
20:     s.instances ← s.instances \ CSI
```

3.3.5. Analyse

The Analyse phase is the second stage of the MAPE-K loop implemented by our Managing System. During this stage, both the Analyse component and the Knowledge component are involved.

The former is in charge of retrieving the latest metrics of all the managed instances from the Knowledge, of processing them and of computing the latest and current values for each QoS indicator in our model, for both the instances and the services. Furthermore, it is in charge of determining the adaptation options to force and/or to propose for all the managed services.
The latter, instead, is in charge of providing the Analyse component with the representation of the current Managed System's architecture and with the latest metrics not yet analysed for a given instance, and also of storing in the knowledge-base the new QoS indicators values and the new adaptation options computed by the Analyse component.

Analyse component

The Analyse component is where all the logic to extract relevant information from the Metrics Snapshots, and to propose Adaptation Options, lies.
The analysis starts as soon as this component is notified by the Monitor. To begin with, the Analyse retrieves the up-to-date representation of the entire Managed System from the Knowledge. Then, the actual analysis process can begin.
The entire analysis is divided into two subphases, the QoS values computation phase and the adaptation proposal phase.

In the first subphase, each managed service s and each one of its instances i that are in the knowledge-base are considered. While analysing i, different scenarios can arise.
The first distinction is made based on the current status of i. Indeed, in the following cases no further analysis is performed on i in this subphase:

- if the status is *SHUTDOWN*;
- if the status is *BOOTING*. Moreover, if the boot time exceeds a given threshold, a forced Shutdown Instance adaptation option is created to shutdown i. This threshold is specified via an environmental variable. This is done in order to avoid services being stuck in *TRANSITION* state;
- if the status is *FAILED*. In this case, a forced Shutdown Instance adaptation option is created to shutdown i. Also, from this moment on, s will be considered in *TRANSITION* state in this and the next analysis subphase, since one of its instances will soon have status *SHUTDOWN*.

Otherwise, if the status of i is either *ACTIVE* or *UNREACHABLE*, its analysis in this subphase proceeds as follows.
The first step to analyse i is to retrieve its latest *Metrics Window*. Given an instance, its

latest Metrics Window is a collection of the latest n subsequent Metrics Snapshots of that instance not yet analysed by the Analyse component, if any; n is the *Metrics Window Size*, a parameter specified by the user via an environmental variable. If the size of latest Metrics Window of i is less than n (i.e., not enough Metrics Snapshots were collected by the Monitor component between the current and the previous loop iterations), no further analysis is performed on i in this subphase. Otherwise, three different rates are computed for i:

- the unreachable rate, which is the rate of snapshots with status *UNREACHABLE* over the latest Metrics Window of i. Its maximum admissible value, called *Unreachable Rate Threshold*, is specified by the user via an environmental variable;
- the failure rate, which is the rate of snapshots with status *FAILED* over the latest Metrics Window of i. Its maximum admissible value, called *Failure Rate Threshold*, is specified by the user via an environmental variable;
- the inactive rate, which is the sum of the unreachable and failure rates. Its maximum admissible value is 1.

If one of those rates exceeds its maximum admissible value, no further analysis is performed on i in this subphase; then, a forced Shutdown Instance adaptation option is created to shutdown i. Also, as before, from this moment on s will be considered as in *TRANSITION* state in this and the next analysis subphase, since one of its instances will soon have status *SHUTDOWN*. Otherwise, the new latest QoS indicators values of i are computed.

The latest Availability value of i is determined by considering all the requests processed by i, whose data is stored in each Metrics Snapshot. In fact, the following ratio is computed:

$$\frac{\sum_{e \in E_l} successfulRequestsCount_e - \sum_{e \in E_o} successfulRequestsCount_e}{\sum_{e \in E_l} requestsCount_e - \sum_{e \in E_o} requestsCount_e}, \tag{3.7}$$

where E_l and E_o are the sets of HTTP Endpoint Metrics for each of the instance's endpoints contained respectively in the latest and oldest snapshot of the Metrics Window.

If the total number of new requests (i.e., the denominator of the ratio) is equal to 0, the latest value is set to the instance's current availability value.

A similar approach is used to compute the latest value of the Average Response Time of

i. Indeed, it is computed as

$$\frac{\sum_{e \in E_l} successfulRequestsDuration_e - \sum_{e \in E_o} successfulRequestsDuration_e}{\sum_{e \in E_l} successfulRequestsCount_e - \sum_{e \in E_o} successfulRequestsCount_e}, \tag{3.8}$$

where E_l and E_o are defined as before.
Again, if the total number of new successful requests is equal to 0, the latest value is set to the instance's current average response time value.

Once all the instances of s have been analysed, three different scenarios can arise:

- all of its instances are either in *SHUTDOWN* status or will be shut down at the end of this MAPE-K iteration upon the execution of forced Shutdown Instance adaptation options. In this case, a new forced Add Instance adaptation option is created, since otherwise s would have no running instances;
- none of its instances have a new QoS latest value for any of its QoS indicators. In this case, no further analysis is performed on s in this subphase;
- none of the previous apply. In this case, for each instance i of the service s, their new QoS latest values are added to their respective QoS History.

Two additional steps are then performed when in the third scenario if s is not in *TRANSITION* state.

To begin with, a new latest value is computed for each QoS indicator of s. In particular, both the new Availability and Average Response Time values are computed by averaging the respective latest values of all the instances of s. Moreover, if s uses Weighted Random Load Balancer, each value is weighted with the load balancer weight of the instance it refers to; otherwise, a simple arithmetic mean is performed.
The process of computing the latest value of a QoS indicator for the services and their instances, when this is possible, is summarized in Figure 3.9.

The second step is to consider a new collection of values, called *Analysis Window*. Given a service or an instance and a QoS Indicator, the Analysis Window of that service or instance is a collection of the latest n valid values in the service or instance QoS History for the specified indicator. n is the *Analysis Window Size*, a parameter specified by the user via an environmental variable.
Indeed, for each service s, if a full Analysis Window exists for all the QoS Indicators of s, we determine the new current value of its QoS Indicators, to be stored in the respective QoS Histories. These current values are computed as the arithmetic mean of all the values

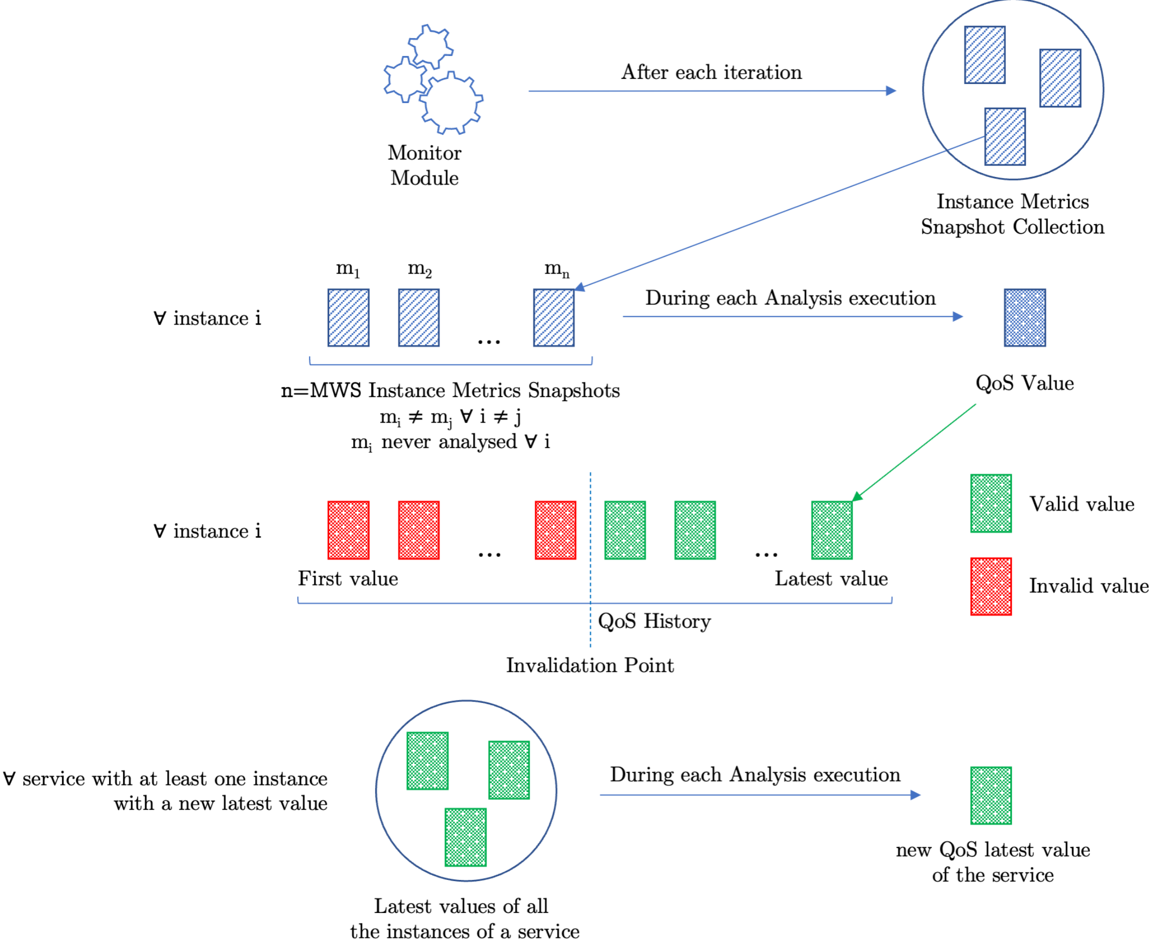


Figure 3.9: Computation process of a QoS latest value

in their respective Analysis Window, since, for each QoS indicator, they aim to represent an estimation that is more reliable than one that considers only its latest value.
After correctly computing the new current values for s, we perform the same operations on all its instances. However, it can happen that a specific instance i does not have a full Analysis Window. In this case, we fill the Analysis Window of i with replicas of its current value for the QoS indicator which the window refers to. This is done because we want an up-to-date current value for i, but at the same time, since in a not full Analysis Window there could be potential outliers, we try to mitigate their impact on the new current value to be computed.
The process of computing the current value of a QoS indicator for the services and their instances, when this is possible, is summarized in Figure 3.10.

Finally, all the modified QoS Histories of each service and instance are sent to the Knowledge component in order to correctly store them in the knowledge-base.
The first analysis subphase logic is summarised by Algorithm 3.

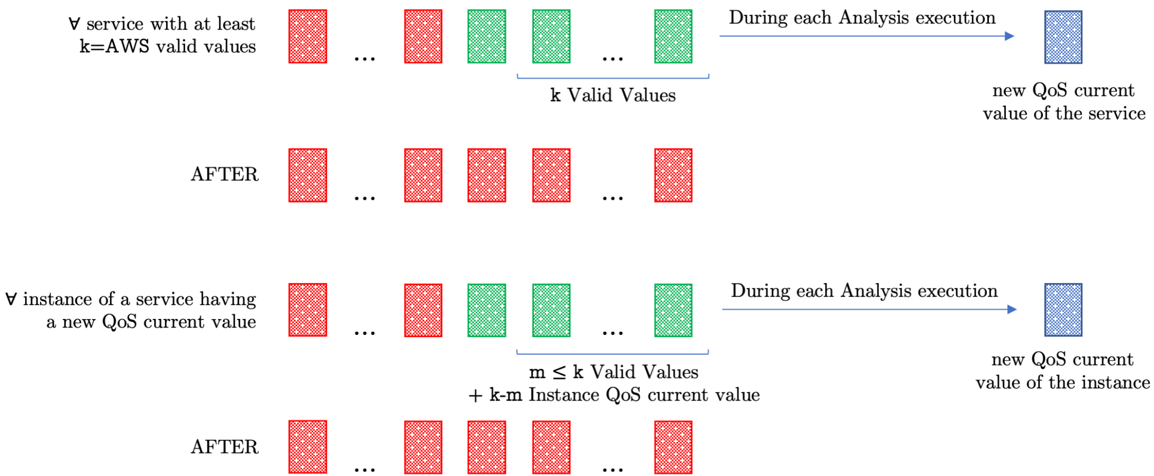


Figure 3.10: Computation process of a QoS current value

The second subphase of the Analyse component is the one in which, for each service, one or more adaptation options are proposed, if needed.
For each service s, if s is in *TRANSITION* state or does not have an Analysis Window for each QoS Indicator, it is simply skipped, and no adaptation option is proposed for it. Otherwise, we can proceed with the following steps.

To begin with, if s has more than one possible implementation, and if the *penalty* value of the current one is greater than or equal to its *trust* parameter, multiple Change Implementation options are created, one for each QoS Indicator, that becomes the goal of their respective adaptation option.

Then, for each QoS Indicator q (i.e., availability and average response time) we check if the rate of values satisfying the specification of q in the Analysis Window of s is lower than the *QoS Satisfaction Rate*, a parameter that must be specified by the user via an environmental variable. When this happens, the Analyse component creates a new Add Instance adaptation option for s having q as its goal.
Furthermore, if s uses a Weighted Random load balancer, and at least one of its instances' current value for q satisfies its specification, the Analyse component creates a new Change Load Balancer Weights adaptation option for s, having q as its goal.

Once all the QoS Indicators have been considered, if no adaptation options were proposed for s in this subphase, its analysis terminates. Otherwise, we consider the dependencies of s. If s has one or more service dependencies, the same analysis described so far (including this dependency analysis) is recursively performed on each of its dependencies. Then, if at least one dependency is either in *TRANSITION* state or has new proposed adaptation options, all the non-forced adaptation options proposed for s during the current loop

Algorithm 3 Analysis – First Subphase

Input:
Knowledge: the interface of the Knowledge component.
mws: Metrics Window size.
aws: Analysis Window size.
maxBootTime: maximum allowed time for an instance to correctly boot.
failureRateThreshold: maximum admissible value for the failure rate of an instance.
unreachableRateThreshold: maximum admissible value for the unreachable rate of an instance.
Output:
O: set of (forced) proposed Adaptation Options.

```
1: S ← Knowledge.getManagedServices( )  // Set of managed services
2: O ← {}
3: for s ∈ S do
4:     I_s ← s.instancesSet
5:     Q_s ← s.QoSSpecificationSet
6:     for i ∈ I_s do
7:         st_i ← i.status
8:         if st_i = ACTIVE or st_i = UNREACHABLE then
9:             mw_i ← Knowledge.getLatestMetricsWindow(instance : i, maxSize : mws)
10:            if mw_i.size = mws then
11:                fr_i ← rate of snapshots with status = FAILED in mw_i
12:                ur_i ← rate of snapshots with status = UNREACHABLE in mw_i
13:                if fr_i ≥ failureRateThreshold or ur_i ≥ unreachableRateThreshold or fr_i + ur_i ≥ 1 then
14:                    O ← O ⋃ {createForcedShutdownInstanceOption(i)}
15:                    s.status ← TRANSITION
16:            else
17:                for q_s ∈ Q_s do
18:                    i.computeLatestValue(q_s.qosIndicator, mw_i) // see Eq. 3.7 and Eq. 3.8
19:        else if st_i = FAILED or st_i = BOOTING and bootingTime_i ≥ maxBootTime then
20:            O ← O ⋃ {createForcedShutdownInstanceOption(i)}
21:            s.status ← TRANSITION
22:    SI_s ← {i | i ∈ I ∧ (i.status = SHUTDOWN ∨ O.containsShutdownOptionFor(i))}
23:    if |SI_s| = |I| then
24:        O ← O ⋃ {createForcedAddInstanceOption(s)}
25:        s.status ← TRANSITION
26:    INLV_s ← {i|i ∈ I ∧ i has new latest QoS values} // set of Instances with a new Latest Value
27:    if |INLV_s| > 0 then
28:        if s.status ≠ TRANSITION then
29:            for q_s ∈ Q_s do
30:                aw_qs ← Knowledge.getAnalysisWindow(service : s, qos : q_s.qosIndicator)
31:                s.computeCurrentValue(aw_{q_s})
32:                for i ∈ I do
33:                    aw_qi ← Knowledge.getAnalysisWindow(instance : i, qos : q_s.qosIndicator)
34:                    i.computeCurrentValue(aw_{q_i})
35:    Knowledge.updateInstances(I)
36:    Knowledge.updateService(s)
```

iteration are discarded, s itself as marked as in *TRANSITION* state and no further analysis is performed on it. In fact, even if one or more adaptation options were proposed

for s in this subphase, the causes behind the dissatisfaction of a QoS Specification may not lay in s itself, but in one of the its dependencies that require adaptation or that are in a *TRANSITION* state, awaiting for previous adaptations to be over. Indeed, the guilty dependency may be propagating its issues to s itself, and it would be incorrect to assume a priori that also s has some issues. Thus, the idea is to perform adaptation on the dependencies at first and to wait for them to be over, in order to correctly determine if s actually needs adaptation or not. Otherwise, if none of its dependencies neither need adaptation nor are in *TRANSITION* state, we simply end the analysis of s, keeping its proposed adaptation options.

Once all the managed services have been correctly analysed, all the proposed adaptation options of all the services, including the forced ones proposed during the first analysis subphase, are sent to the Knowledge component to be stored in the knowledge-base, where they can be retrieved by the *Plan* component.

Finally, the Plan component is notified to start its iteration, while the current Analyse iteration terminates.
The second analysis subphase logic is summarised by Algorithm 4.

As a final note, the Metrics Window Size, the Analysis Window Size, the Failure Rate Threshold, the Unreachable Rate Threshold and the QoS Satisfaction Rate can be modified by the user at runtime[6] via a dedicated REST API, defined in Appendix B.4.2.

Interaction with the Knowledge component

The Analyse component interacts with the Knowledge component to retrieve all the elements of the System Model needed to perform the analysis, such as the system architecture, the Metrics and Analysis Windows and the QoS Histories of the desired service or instance.
In particular, the logic behind the construction of a Metrics Window for a given instance i not yet analysed is entrusted to the Knowledge component. Indeed, the snapshots of i are filtered by considering only the ones collected after the computation of the latest values of i for each QoS Indicator.

During each loop, the Analyse component also interacts with the Knowledge to store the results of its execution in the knowledge-base. In particular, whenever a new adaptation option is proposed, the Knowledge component increments the *penalty* indicator of the Service Implementation to be adapted, since it represents the tendency of that implementation to require adaptation, as stated in Section 3.3.2.

[6]If the Analyse iteration is in progress, all the changes are applied at the next iteration.

Algorithm 4 Analysis – Second Subphase

Input:
Knowledge: the interface of the Knowledge component.
aws: Analysis Window size.
QoSSatisfactionRate: the minimum rate of values in the analysis window of a QoS indicator that must satisfy its specification.
O: set of already proposed Adaptation Options.
Output:
O: set of proposed Adaptation Options.

```
1: S ← Knowledge.getManagedServices( )  // Set of managed services
2: AS ← {}  // Set of already analysed services
3: for s ∈ S do
4:     O ← O ∪ computeOptions(s, AS, O)
1: function computeOptions(service : s, analysedservices : AS, proposedOptions : O)
2:     if s ∈ AS then
3:         return {o | o ∈ O ∧ o.service = s}
4:     O_s ← {}
5:     Q_s ← s.QoSSpecificationSet
6:     s.isAdaptable ← s.status ≠ TRANSITION
7:     for q_s ∈ Q_s do
8:         aw_{q_s} ← Knowledge.getAnalysisWindow(service : s, qos : q_s.qosIndicator)
9:         if not (s.isAdaptable and aw_{q_s}.size = aws) then
10:            s.isAdaptable = false
11:            break
12:    if s.isAdaptable then
13:        for q_s ∈ Q_s do
14:            if |s.possibleImplementations| > 1 and
                   s.currentImplementation.penalty > s.currentImplementation.trust then
15:                O_s ← O_s ∪ {createChangeImplementationOption(service : s, goal : q_s.qosIndicator)}
16:            sr_{q_s} ← |{v | v ∈ aw_q ∧ q.isSatisfied(v)}| / aws  // Rate of values in aw_q satisfying their specification
17:            if sr_{q_s} < QoSSatisfactionRate then
18:                O_s ← O_s ∪ {createAddInstanceOption(service : s, goal : q_s.qosIndicator)}
19:                if s.loadbalancer is Weighted Random then
20:                    O_s ← O_s ∪ {createChangeLoadBalancerWeights(service : s, goal : q_s.qosIndicator)}
21:        if |O_s| > 0 then
22:            for d_s ∈ s.dependencies do
23:                O_d ← computeOptions(s, AS, O)
24:                if d_s.status = TRANSITION or |O_d| > 0 then
25:                    s.status ← TRANSITION
26:                    AS ← AS ∪ {s}
27:                    return O_d
28:    AS ← AS ∪ {s}
29:    return O_s
```

3.3.6. Plan

The Plan is the third stage of the MAPE-K loop implemented by our Managing System. During this phase, the Plan component is in charge of analysing all the adaptation options

proposed by the Analyse component, and of determining which of them should be actually applied or not.

Plan component

The goal of the Plan component is to determine which are the best adaptation options among all the ones proposed during the current loop iteration. The chosen adaptation options will then be stored in the system knowledge-base, in order to allow the Execute component to apply them.

To begin with, for each managed service s, the Plan retrieves from the Knowledge all the adaptation options proposed by the Analyse component during the current loop iteration. Subsequently, if there is at least one forced option among the ones proposed for s, the non-forced ones are discarded, while all the forced ones are processed, and finally directly chosen. Conversely, if no forced adaptation option was proposed for s, all the options are processed, and then compared to extract the one which is estimated to bring more benefits to the service it refers to. The extracted one, which is said to be *chosen*, is the one that the Managing System should apply at the end of the current loop iteration.
The logic behind the processing of the options follows.

The Add Instance options are processed by considering the type of load balancer used by the services they refer to. In fact, if the service does not use a Weighted Random load balancer, no processing is needed. On the contrary, it is fundamental to determine which will be the weight of the new instance that will be added to the service when a Weighted Random load balancer is involved: as a design choice, we decided to grant to the new instance a weight equal to $\frac{1}{NumberOfRunningInstances+1}$. However, since we must guarantee that, for each service, the sum of all instances' weights is equal to 1, it is necessary to reduce the weights of the already running instances such that the sum of their weights will be equal to $1 - \frac{1}{NumberOfRunningInstances+1}$. As a design choice, we decided to evenly reduce all the weights, multiplying them by a factor of $(1 - \frac{1}{NumberOfRunningInstances+1})$.
After reducing the weights, it may happen that some of those weights become smaller than the ratio $\frac{shutdownThreshold}{NumberOfRunningInstances+1}$ (described in Section 3.3.2). Hence, those instances are selected to be shut down, and their residual weights are evenly split among the remaining ones, including the new instances. Once the weights are updated, this same process is repeated after correctly reducing the *NumberOfRunningInstances* value (i.e., subtracting from the original one the number of instances that will be shut down), until no instance weight is below the ratio.

The Shutdown Instance options are processed once again considering the type of load

balancer used by the services they refer to. In fact, if the service uses a Weighted Random load balancer, it is fundamental to redistribute the weight of the instance to shutdown to the remaining instances, in order to guarantee that the sum of their weights is equal to 1. This is done by evenly splitting and redistributing the weight of the instance to shutdown among the remaining ones.

The Change Implementation options are processed by considering the QoS Indicator q specified as the goal of the option, the Benchmark value of each possible implementation of the service to be adapted, and the *preference* of the user towards a specific implementation. As a reminder, the benchmark represents an estimation of the value that q will have after changing implementation.
When q is the Availability, to determine the new implementation we multiply the availability benchmark value of a possible implementation with its *preference*, and then select the implementation that maximizes this product.
Conversely, if q is the Average Response Time, we divide the average response time benchmark value of a possible implementation with its *preference*, and select as the new implementation the one that minimizes this quantity.
As mentioned in Section 3.3.2, the number of instances of the new implementation to allocate is equal to the number of running instances for the current implementation. This is done in order to allow the new instances to correctly handle a request load which required a specific number of instances (i.e., the number of currently active ones) to be correctly handled.

Differently from the previous ones, the Change Load Balancer Weights options require more complex processing. Indeed, a Mixed-Integer Linear Programming (M-ILP) problem [13] is solved to obtain the new weights of the existing instances and to determine the ones that should be shut down, if any.
The parameters of the problem are the following:

- I_s, the set of running instances of the current service s;
- n, the cardinality of I_s;
- lt_s, the *instance-load shutdown threshold* of the current implementation of s;
- k_s, the performance indicator of service s, defined by the ratio $^{AvailabilityCurrentValue_s}/_{AverageResponseTimeCurrentValue_s}$;
- k_i, the performance indicator of the instance $i \in I_s$, defined by the ratio $^{AvailabilityCurrentValue_i}/_{AverageResponseTimeCurrentValue_i}$;
- z_i, the performance ratio of the instance $i \in I_s$, defined by the ratio $^{k_i}/_{k_s}$;

- p_i, the current weight of the instance $i \in I_s$.

Among these parameters, the most important are the performance-related ones, which require a more detailed elaboration.
To begin with, it is crucial to underline the fact that, in a real-world scenario, the performance of a given instance i depends on multiple factors (e.g., available resources and their usage, response time, network bandwidth, availability, etc.). However, since the aim of RAMSES is not to provide an in-depth and accurate computation of the performances of the instances, the decision we took is to define the performance indicators k_i and k_s as reported before. Indeed, the problem of finding a better mathematical model representing the instance's performance could be addressed by further works on RAMSES, as proposed in Chapter 5.
In addition to k_i and k_s, given an instance and its performance indicator, we measure how good its performance is with respect to the average performance of the service, which takes into account the performances of all its instances. This indicator, computed as the previously described ratio z_i, contributes to determining whether to increase or decrease the weight of an instance.

The variables of the problem are the following:

- $w_i \in [0, 1]$, a continuous variable representing the new weight of the instance $i \in I_s$;
- a_i, an integer binary variable representing whether the instance $i \in I_s$ should keep running ($a_i = 1$) after changing the weights, or whether it should be shut down ($a_i = 0$).

The objective function to minimize is the following:

$$obj : \sum_{i \in I_s} (\frac{1}{z_i} w_i - z_i a_i) \tag{3.9}$$

The first term in the summation aims to make the solver reduce the weights of the instances with a low performance ratio (i.e., instances that, on average, have worse performances with respect to the average of the entire service). On the contrary, the weights of the instances with a high (i.e. > 1) performance ratio are enforced to grow.
The second term aims to induce the solver to shut down the instances with poor performances, while rewarding the decision to keep active the good ones.

Finally, the constraints of the problem are the following:

$$w_i \leq a_i \qquad \forall i \in I_s \tag{3.10}$$

$$w_i \geq \frac{lt_s}{n} a_i \qquad \forall i \in I_s \tag{3.11}$$

$$\sum_{i \in I_s} w_i = 1 \qquad \forall i \in I_s \tag{3.12}$$

$$w_i \leq \frac{k_i}{k_j} w_j + (1 - a_j) \qquad \forall i, j \in I_s | k_i \geq k_j \tag{3.13}$$

$$w_i \leq z_i p_i + z_i \sum_{\substack{j \in I_s, \\ i \neq j}} p_j (1 - a_j) \qquad \forall i \in I_s \tag{3.14}$$

Equation. 3.10 is an activation constraint: indeed, it links the activation variable a_i with w_i, imposing that, for the instances to shut down, $w_i = 0$.
Equation 3.11 determines the lower bound of the weight of a given instance. In fact, if the instance i is active (i.e., a_i is equal to 1), then its weight must be greater or equal to the ratio described in Section 3.3.2.
Equation 3.12 is the one in charge of guaranteeing that the sum of the weights of all instances of a given service is equal to 1.
Equation 3.13 introduces a relationship between the weights of different instances. Indeed, this constraint sets an upper bound to the growth of w_i, which is proportional to the performance ratio between instance i and every other worst-performing instance j. In this way, instances with similar performances are enforced to have similar weights. It must be noted that when w_j is zero also a_j is zero, and vice versa, making this constraint to be always satisfied, since it becomes $w_i \leq 1$.
Equation 3.14 introduces a new upper bound to w_i, which takes into consideration the performance ratio of instance i. Based on the value of z_i we can interpret this constraint in two different ways: if $z_i > 1$, we are interested in having $w_i > p_i$. Furthermore, it is also important to prevent an uncontrolled growth of w_i with respect to p_i (and consequently, an uncontrolled decrease of the other weights), which could lead to unwanted oscillatory behaviours. Thus, we multiply p_i by z_i, which is reasonable since z_i indicates how better the instance i is with respect to the service average. At the same time, the second term in the sum is introduced in order to redistribute the weights of the instances to be shut down among the active ones. This quantity is once again scaled by i's performance ratio, for the same reasoning explained before. The fifth constraint should be interpreted differently when $z_i \leq 1$. In this case, we are interested in a decrease of w_i with respect to its previous value, to allow better-performing instances to handle more requests. Thus, this constraint poses a lower bound on the minimum decrease of w_i with respect to p_i.

The result obtained from the M-ILP problem, solved using the Google OR-Tools suite[7], is the set of new weights and the set of instances to shut down (i.e., the ones with $w_i = 0$), and are associated to the adaptation option itself.

Once all the options are processed, the Plan component needs to decide which proposed options must be applied. As anticipated before, all the forced adaptation options are automatically chosen. On the contrary, in their absence, for each service only one proposed option is chosen, which is the one with the highest estimated benefit.
Indeed, for each service s and each one of its proposed options o, the plan determines the *benefit* of option o based on its type and goal. The benefit is a quantity that represents the relative estimated improvement (or the worsening) of the QoS Indicator selected as a goal for o, compared to the current situation: indeed, when its value is greater than 1, o is expected to have a positive impact on the service it refers to.
When the goal is the Availability, the following ratio is computed:

$$benefit_O = \frac{EstimatedValue_A}{CurrentValue_A}, \tag{3.15}$$

where $CurrentValue_A$ is the current availability value held by s, while $EstimatedValue_A$ is an estimation of what the current availability value held by s will be, after applying the option o.
The $EstimatedValue_A$ is computed differently based on O's type:

- for the Shutdown Instance options, it is computed as the average of the remaining instances' current availability values, weighted by their new weights, if any;
- for the Add Instance options, it is computed as the average of the remaining instances' current availability values, weighted by their new weights (if any), and of the new instance availability benchmark value, also weighted by the new instance weight (if any);
- for the Change Implementation options, it is the value of the availability benchmark provided for the implementation selected after processing the option;
- for the Change Load Balancer Weights options, it is computed as the average of the remaining instances' current availability values, weighted by their new weights.

On the contrary, when the goal is the Average Response Time, the following ratio is computed:

$$benefit_O = \frac{CurrentValue_T}{EstimatedValue_T}, \tag{3.16}$$

[7]Google Inc., `https://developers.google.com/optimization`

where $CurrentValue_T$ is the current average response time value held by s, while $EstimatedValue_T$ is an estimation of what the current average response time value held by s will be, after applying the option o.
The $EstimatedValue_T$ is computed differently based on O's type, analogously to how it is computed for the Availability.

Once all the benefits of all the options proposed for s are computed, the chosen option is determined by selecting the one with $benefit_O > 1$ that maximizes the product $benefit_O \cdot weight_Q$. This product represents, given the benefit of an option, how much it is preferable over the others, since $weight_Q$ is the weight assigned to q (the goal of o) in its Specification for s (i.e., how much options with q as their goal should be prioritized over the others).

Finally, the chosen options are sent to the Knowledge in order to let the Execute component apply them. Additionally, before notifying the Execute to start its iteration, the Plan requests to the Knowledge component to invalidate all the values in all the QoS Histories of the services and their respective instances, when needed. Indeed, this is done for all the services that are in *TRANSITION* state, that have a new chosen adaptation option or that depend on a service that meets these three conditions, recursively. The reason behind this operation is to prevent the Analysis component to consider QoS indicator values that refer to an old configuration of the Managed System when analysing the different services during the next loop iterations.
The logic implemented by the Plan component is summarized by Algorithm 5.

To conclude, it must be noted that the reasoning made when designing the decision-making process of the Plan component (i.e., what is the option to choose among the non-forced ones) comes from the assumption that all the QoS Indicators' current values (i.e., availability and average response time) of the various instances to be adapted will not significantly change after applying the chosen adaptation options. Hence, it is assumed that all the instances to be adapted will be able to handle a different request load without a drastic reduction in their performance. Indeed, trying to model a relationship between the instances parameters and indicators and their performances would require a more in-depth analysis, which is not the aim of this project.

Interaction with the Knowledge component

The interaction between the Plan and the Knowledge components is finalized to retrieve the Managed System architecture from the knowledge-base, to retrieve all the adaptation options proposed by the Analyse component during the current loop iteration, to correctly store the chosen adaptation options, and to invalidate the QoS Histories of the services

Algorithm 5 Plan

Input:
Knowledge: the interface of the Knowledge component.
Output:
CAO: set of Chosen Adaptation Options, stored in the Knowledge.

1: $S \leftarrow Knowledge.getManagedServices()$ *// Set of Managed Services*
2: $PAO \leftarrow Knowledge.getProposedOptions()$ *// Set of Proposed Adaptation Options*
3: $CAO \leftarrow \{\}$ *// Set of Chosen Adaptation Options*
4: **for** $s \in S$ **do**
5: $PAO_s \leftarrow \{p \mid p \in PAO \ \wedge \ p.service = s\}$ *// Set of Proposed Adaptation Options for s*
6: $FPAO_s \leftarrow \{p \mid p \in PAO_s \ \wedge \ p \text{ is } forced\}$ *// Set of Forced Adaptation Options for s*
7: $NFPAO_s \leftarrow PAO_s \setminus FPAO_s$ *// Set of Non-Forced Adaptation Options for s*
8: $CAO_s \leftarrow \{\}$ *// Set of Chosen Adaptation Options for s*
9: **if** $FPAO_s \neq \emptyset$ **then**
10: **for** $o \in FPAO_s$ **do**
11: $o \leftarrow processOption(o)$
12: $CAO_s \leftarrow CAO_s \cup FPAO_s$
13: **else**
14: **for** $o \in NFPAO_s$ **do**
15: $o \leftarrow processOption(o)$
16: $o.benefit \leftarrow computeBenefit(o)$
17: $bo \leftarrow \text{argmax}_{o \in NFPAO_s}\{o.benefit \cdot o.goal.preference \mid o.benefit > 1\}$ *// Extract best option*
18: $CAO_s \leftarrow CAO_s \cup \{bo\}$
19: **if** $CAO_s \neq \emptyset$ **or** $s.state = TRANSITION$ **then**
20: $Knowledge.invalidateQoSHistory(s)$
21: $D_s \leftarrow \{d \mid d \in S \ \wedge \ s \in d.dependenciesSet\}$ *// Set of services depending on s*
22: **for** $d \in D_s$ **do**
23: $Knowledge.invalidateQoSHistory(d)$
24: $CAO \leftarrow CAO \cup CAO_s$
25: $Knowledge.chooseOptions(CAO)$

(and of all their instances) that require adaptation.

3.3.7. Execute

The Execute is the fourth and last stage of the MAPE-K loop implemented by our Managing System. During this phase, the Execute component is in charge of retrieving from the Knowledge the adaptation options chosen during the current loop iteration, and of interacting with the Actuator component to effectively perform adaptation, if needed.

Execute component

The Execute component is the last building block of the proposed Managing System, capable of interacting with the Managed System via the provided Actuator component. During each loop iteration, its goal consists in retrieving from the Knowledge all the

adaptation options chosen during the current iteration, if any, and in processing and applying them, by contacting the Actuator component and by consequently updating the knowledge-base.
The logic behind the processing of the chosen options follows.

When an Add Instance option for the service s is among the chosen ones, the Execute component asks the Actuator to allocate a new instance for s via its dedicated API. The Actuator boots the new instance and communicates to the Execute component its IP address and port, used to construct the new instance ID. Moreover, if with the Add Instance Option a new set of weights for the running instances of s is provided, the Actuator API in charge of changing the service configuration is contacted to correctly update them. In this case, the Execute component also checks if the Add Instance option includes instances to shut down, and if so, it asks the Actuator to gracefully terminate their execution.
Finally, the Knowledge component is notified of all the changes to the service configuration, by including the ID of the new allocated instance and, if s uses a Weighted Random load balancer, the new instances' weights and the list of the just shutdown instances, if any.

Similarly to the case in which instances must be shut down when executing an Add Instance option, when a Shutdown Instance option is among the chosen ones, the Execute component contacts the Actuator via its dedicated API to shut down the selected instance. Moreover, if the service involved by the option is load balanced using the Weighted Random algorithm, the Execute component asks the Actuator to update the service configuration with the new instances weights specified by the option itself.
Finally, the Knowledge component is notified of the execution, including the ID of the just shutdown instance and the new weights of the remaining ones.

The Change Implementation options are handled by initially contacting the Actuator to allocate the instances of the new selected implementation, whose number is specified by the option itself. Then, all the running instances of the old service implementation are shut down via the dedicated Actuator API. After that, if the service under adaptation is load balanced by a Weighted Random load balanced, the Execute component also asks the Actuator to correctly update the service configuration by removing all the old weights and by assigning to all the new instances the same weight, equal to $1/_{NumberOfInstances}$.
Finally, the Knowledge component is notified of the change of implementation, including the IP addresses and ports of the new allocated instances, which are provided by the Actuator as a response to its request and will be used to generate the new instances' IDs.

Finally, the Change Load Balancer Weights options are processed by first asking the Actu-

ator to shut down all the instances specified by the option, if any. Then, the configuration of the service to adapt is updated via the dedicated Actuator API, assigning to all the remaining instances their respective new weights, which are provided with the option. Eventually, the Knowledge component is notified, including the IDs of the instances that have been shut down, and the new configuration of the adapted service, holding the new weights of the instances of the service under adaptation.

After processing all the chosen options, the Execute component terminates the current MAPE-K loop iteration and contacts the Monitor component in order to let it start a new iteration of the loop.
The logic implemented by the Execute component is summarized by Algorithm 6.

Algorithm 6 Execute

Input:
Actuator: the interface of the Actuator component.
Knowledge: the interface of the Knowledge component.
Output:
The modified system runtime architecture and configuration, stored in the Knowledge.

```
1: CAO ← Knowledge.getChosenOptions()  // Set of Chosen Adaptation Options
2: for o ∈ CAO do
3:     if o is Add Instance then
4:         Actuator.bootInstance()
5:         if o.service.loadbalancer is Weighted Random then
6:             Actuator.updateWeights(o.newWeights)
7:         for i in o.instancesToShutdown do
8:             Actuator.shutdownInstance(i)
9:     else if o is Shutdown Instance then
10:        if o.service.loadbalancer is Weighted Random then
11:            Actuator.updateWeights(o.newWeights)
12:        Actuator.shutdownInstance(o.instanceToShutdown)
13:    else if o is Change Load Balancer Weights then
14:        Actuator.updateWeights(o.newWeights)
15:        for i in o.instancesToShutdown do
16:            Actuator.shutdownInstance(i)
17:    else if o is Change Implementation then
18:        for k ← 1 to |o.service.instances| do
19:            Actuator.bootInstance()
20:        for i in o.service.instances do
21:            Actuator.shutdownInstance(i)
22:    Knowledge.update(o.service)
```

As anticipated in Section 3.2.4, the Execute component, and consequently RAMSES itself, assumes that all the operations requested to the Actuator are eventually executed, and that all the changes of service configurations (i.e., change of load balancer weights) are performed within a reasonably short amount of time.

This last assumption is made in order to avoid unexpected behaviours: indeed, if the configuration changes are not applied promptly, the Managing System could propose adaptation on a given service considering its new configuration, while the actual running configuration could be (still) inconsistent with its counterpart stored in the knowledge-base. Indeed, being in a distributed setting, this could happen because of a variety of reasons.
To limit the consequences of this situation, we advise the user of RAMSES to try to guarantee that these changes do not require more time than the following threshold:

$$MP \cdot MWS \cdot AWS \cdot \frac{1 - QoSSR}{n}, \tag{3.17}$$

where MP is the Monitor Period of the Managing System, MWS and AWS are respectively the Metrics Window Size and the Analysis Windows Size of the Analyse component, $QoSSR$ is the QoS Satisfaction Rate, and n is a parameter chosen by the user.
Indeed in the worst case, if this constraint is satisfied, given a QoS indicator, at most n of the values in the Analysis Window of a service that do not satisfy the respective QoS Specification refer to a transient configuration of the considered service. This happens since $MP \cdot MWS$ is the minimum amount of time needed to fill the Metrics Window of a given instance, and thus compute one new QoS value in the Analysis Window of its service.

Interaction with the Knowledge component

In addition to retrieving the adaptation option chosen during the current loop iteration, the interaction between the Execute and the Knowledge components is fundamental to correctly update the knowledge-base after performing the required operations.

The Knowledge performs specific operations when the Execute component notifies the changes in the runtime architecture and configuration of a given service.
When one or more instances of a service are shut down, the Knowledge sets their status to *SHUTDOWN* and creates a final empty metric for each of those instances, with the same status.
When a new instance of a service is booted, the Knowledge is in charge of creating its model counterpart, setting its status to *BOOTING*, and associating it to the current implementation of its service.
When the Knowledge is notified about the change of current implementation of a given service, the current implementation of the service is set to the new one and the *penalty* indicator of the old one is initialized to 0. Then, the instances of the old service implemen-

tation are treated as common shutdown instances, and the operations performed on them are the same already described when there is an instance to shut down. Moreover, the same operations for adding a new instance to the current implementation are performed, for all the instances of the new service implementation started by the Execute component. Finally, when a change in the configuration of a service is communicated (i.e., a change of load balancer weights), the service configuration stored in the System Model is updated accordingly.

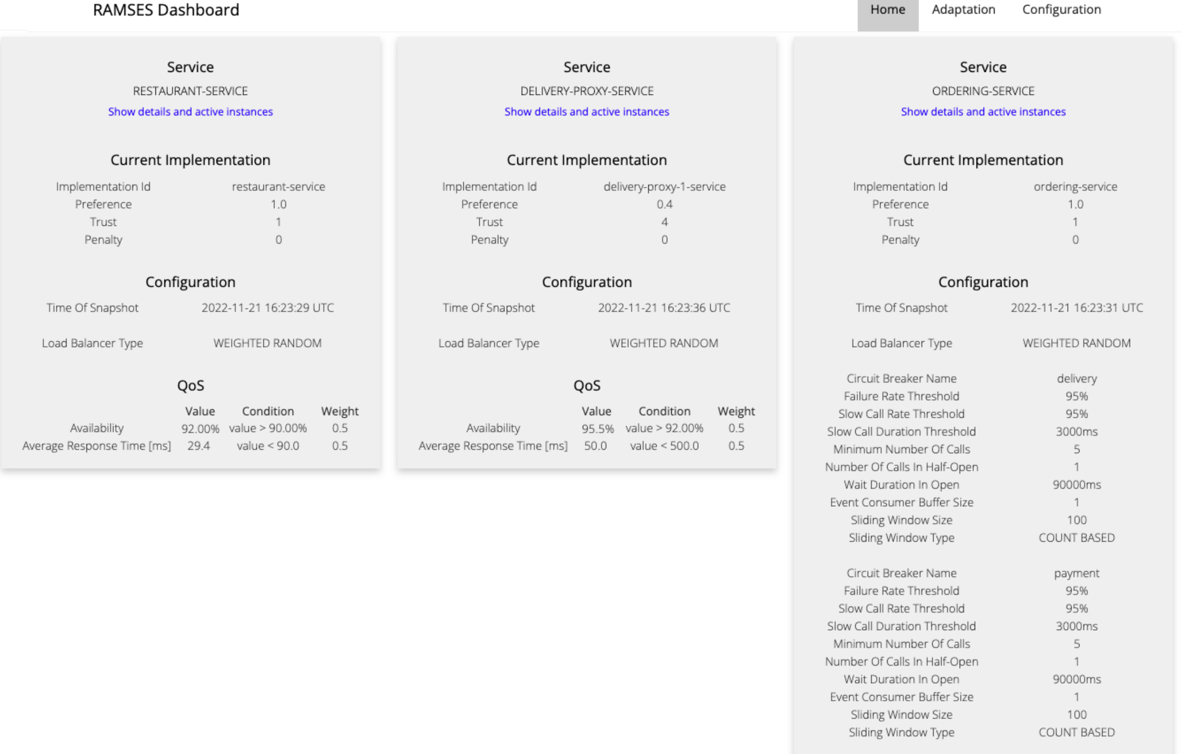
3.3.8. Dashboard

In order to allow the user to better interact with the system, modifying parameters and monitoring the status of the services, of their instances, and of the overall MAPE-K loop itself, RAMSES is provided with an interactive dashboard. In particular, it is a web application serving as the front-end to the knowledge-base, developed using Spring Boot, with Thymeleaf as template engine. It is structured in three macro sections, one of which is dedicated to the configuration of the Managing System components.

The first section of the dashboard, shown in Figure 3.11, focuses on the Managed System, on its runtime architecture, and in particular on the data that RAMSES collected or computed about it.
For both the service and its instances the current value and the history of values of each QoS indicator are shown. Moreover, in the case of the instances, also the latest Metrics Snapshot is displayed.
Finally, it also displays the configuration of each managed service.

The second section of the dashboard shows the latest three adaptation options applied for each managed service, if any. Additionally, it shows the current phase of the loop.

The third section allows the admin of the system to change the configuration parameters of the Monitor Component (i.e., the Monitor Scheduling Period) and of the Analyse Component (i.e., the Metrics Window Size, the Analysis Window Size, the QoS Satisfaction Rate, the failure rate and the unreachable rate). It also allows to stop the MAPE-K loop, starting from the next iteration, or to simply stop RAMSES to perform adaptation, using it to simply monitor the Managed System.



(a) RAMSES Dashboard – Homepage

(b) RAMSES Dashboard – Service Detail

Figure 3.11: RAMSES Dashboard

4 | Evaluation

In this chapter we describe the experimental campaign conducted to analyse the benefits of the proposed solution.
The chapter is structured as follows:

- Section 4.1 introduces our research questions;
- Section 4.2 illustrates the setup of the environment in which the experiments were run and the way the experimental campaign was designed and conducted.
- Section 4.3 describes all the experiments, grouped by adaptation scenario. In particular, we describe the analysis carried out for each experiment, answering the proposed research questions.

4.1. Research questions

The conducted experimental campaign, described in Section 4.2, aims to answer 10 research questions, to better understand how RAMSES behaves in specific scenarios that synthetically reproduce relevant operating conditions.
The research questions are the following:

- **RQ1**: How long should we run an experiment to capture the phenomena of interest during its execution?
- **RQ2**: Does the proposed Managing System eventually improve the performance of the Managed System under degraded QoS?
- **RQ3**: What is the impact of the Metrics Window Size on the behaviour of the Managing System?
- **RQ4**: What is the impact of the Analysis Window Size on the behaviour of the Managing System?
- **RQ5**: How does the Managing System perform when the Managed System is under heavy load?

- **RQ6**: Are the Managing System self-healing capabilities effective when a service is unavailable?
- **RQ7**: Are the Managing System self-healing capabilities able to detect and replace instances prone to be unreachable or to fail?
- **RQ8**: What is the impact of the Monitor Period on the behaviour of the Managing System?
- **RQ9**: Are the Managing System self-optimizing capabilities effective?
- **RQ10**: Is the Managing System agnostic and reusable with respect to the managed system?

4.2. Design of the evaluation

We executed all the experiments in the same environment, made of two physical machines on the same Local-Area Network (LAN):

- a 2020 Apple MacBook Air running the Managed System and the Knowledge component of the MAPE-K loop, and hosting the single instance of the MySQL DBMS used by both the managed and the managing systems. The technical specifications follow.
  - SoC: Apple M1 - 8-core CPU, 7-core GPU
  - RAM: 16GB LPDDR4
  - Storage: 256GB on NVMe SSD
  - OS: macOS Monterey 12.6
  - Software: JRE v16.0.2, Docker v20.10.17 (allocating 6 CPUs, 10GB Memory, 1.5GB Swap), MySQL Server v8.0.25
- a 2020 Apple Mac Mini running all the components of the Managing System but the Knowledge one, plus the Probe and the Actuator components and the Managing System's dashboard. The technical specifications follow.
  - SoC: Apple M1 - 8-core CPU, 8-core GPU
  - RAM: 16GB LPDDR4
  - Storage: 512GB on NVMe SSD
  - OS: macOS Monterey 12.4

  - Software: JRE v16.0.2

The purpose of this setup is dual: on one hand it highlights the advantages of having a microservice-based Managing System, since its architecture allows the different components to be distributed over a network; on the other hand, it allows to lighten the resource usage of a single machine, and to generate a higher load for the Managed System without altering the functioning of the hosting machine due to lack of resources.

The pipeline adopted for each experiment involves the following steps:

- The Managed System was deployed from scratch before each experiment. Details about the deployment of each service are provided in each experiment description;
- The database was cleared before each experiment;
- Some load (i.e., artificial requests) to the Managed System was generated during the whole experiment;
- The Monitor component was allowed to start collecting metrics after 10 seconds from the start of the load generation (ramp-up phase), and it was stopped at the end of the experiment.

The load generation is delegated to an ad-hoc Java application, deployed as an independent Java Archive (JAR) application. Every 10 milliseconds the *Load Generator* simulates a new user starting a complete interaction with the Managed System. The maximum number of concurrently active users is a configurable property of this application. If an error occurs during the interaction of a user, that user immediately terminates its interaction.

The components of the Managed System were deployed individually in Docker containers, with all the instances of the same service have the same load balancer weight, resulting in an equally-distributed load among the instances of the same service.
Concerning the Managing System, the MAPE-K components were deployed as independent Java Archive (JAR) applications, with a configuration depending on the running experiment.

In order to create an experimental environment in which services exhibited unwanted behaviours (e.g., QoS constraints not satisfied, high failure rate, etc.), we synthetically injected issues in the managed system by manipulating the services instances. The instances of the managed services referred to as *manipulated* are instances whose performances were altered, by artificially slowing their execution or causing failures. We used the aspect-oriented paradigm to implement the issue injection mechanism. In particular, we created an *aspect* capable of reacting to events such as the invocation of service meth-

ods. In particular, the computation time of each service method is artificially increased by a value sampled from a Gaussian distribution, whose mean and variance are referred to as *lag mean* and *lag variance*, respectively. Failures are injected by generating an artificial exception, whose occurrence probability is referred to as *failure rate*.
Conversely, the instances which were not manipulated are defined as *nominal*.
The way the instances were manipulated depends on the specific experimental campaign, and it is described in the corresponding section.

In order to understand the benefits of the proposed solution and to analyse how it behaves in specific conditions, ten experiments were conducted, each of them answering a different research question.
Our experimental campaign is composed of 10 experiments, that we analysed to answer the research questions reported in Section 4.1.

The first nine experiments are grouped by adaptation scenario – as proposed in Table 3.1. Moreover, a final experiment was conducted, which aims at showing the Managing System reusability.

The experiments are grouped by scenarios, as follows:

- Scenario S1 - Some QoS indicators are not satisfied
  - **E1 - Choice of the experiment duration**. Different experiments of different lengths were executed, to choose the duration of the following ones. In particular, the first experiment lasted 1 hour, allowing us to motivate the chosen duration, which was set to 20 minutes.
  - **E2 - Benefits of adaptation**. The second experiment compares the performance of the Managed System when no adaptation option is applied with the performance obtained by performing adaptation under the same circumstances. This experiment is considered as the *reference experiment* for all the others in this experimental evaluation.
  - **E3 - Impact of the Metrics Window Size**. The third experiment aims at illustrating the impact of the size of the Metrics Window on the behaviour of the Managing System. In particular, we reduced the Metrics Window Size of the reference experiment from 6 to 3, and then we compared their results.
  - **E4 - Impact of the Analysis Window Size**. The fourth experiment aims at illustrating the impact of the size of the Analysis Window on the behaviour of the Managing System. In particular, as reported in its detailed description, with respect to the reference experiment, the Analysis Window Size was

decreased from 5 to 1.

  - **E5 - Impact of the number of users**. The fifth experiment aims at illustrating the impact of the number of users that make requests to the Managed System on the decisions of the Managing System. In particular, as reported in its detailed description, with respect to the reference experiment, the number of parallel users was increased from 50 to 500.

- Scenario S2 - Service unavailable
  - **E1 - Analysis of the self-healing capabilities**. This experiment aims at evaluating the self-healing properties of the system, showing its behaviour when one of the managed services is not available at all.
  - **E2 - Analysis of the failure tendency detection**. This experiment aims at highlighting the self-healing capabilities of the Managing System, showing its behaviour when one of the instances of the managed services exhibits the tendency of being unreachable or failing too often.
  - **E3 - Impact of the Monitor Period**. Compared to E1 and E2, we changed the Monitor Period from 5 seconds to 15 seconds, to analyse its impact on the failure detection capability of the Managing System.
- Scenario S3 - Better service implementation available
  - **E1 - Analysis of the self-optimization capabilities**. This experiment aims at evaluating the self-optimization properties of the system, showing its behaviour when one of the managed services has a better implementation available with respect to its current one.
- Reusability of the Managing System
  - **E1 - Analysis of the Managing System reusability**. This experiment aims at showing the reusability of the proposed Managing System, which is designed to be agnostic with respect to the Managed System.

Finally, as mentioned in Section 3.2, both the Payment Proxy Service and the Delivery Proxy Service are provided with three different implementations, which are referred to respectively as *I1*, *I2* and *I3* for each proxy service.
In all the experiments, each proxy is initially deployed with the respective *I1* implementation.

4.3. Results

This section describes the results of the experiments. All the experiments are introduced by a brief description, followed by comments on the applied adaptation options, and on the final outcome of the experiments themselves.

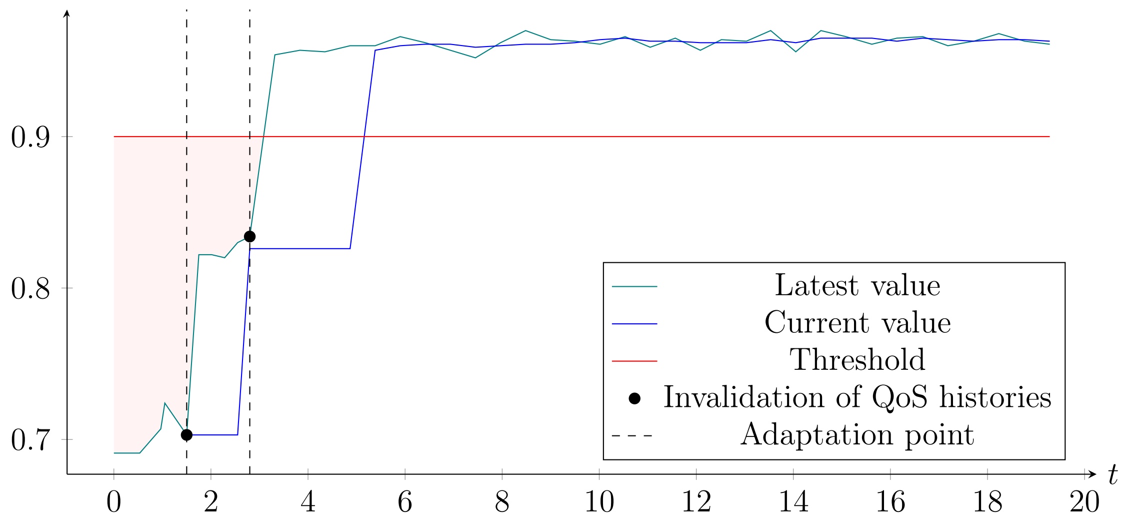


Figure 4.1: Example of graph used in the experiments

For each experiment, we report a summary of the results and we refer the reader to Appendix A for the complete set of plots built out of the experiments, such as the plot in Figure 4.1.
The graphs points are a discrete representation of the trend of the value, the current value and the threshold of the QoS indicator they refer to. Each graph spans the whole experiment duration, hence each point of the graph refers to a specific moment in the experiment duration.
Furthermore, each graph shows the instants when the histories were invalidated or an adaptation option was chosen.
Finally, each graph highlights the moments when the specification of the QoS it refers to is not satisfied by filling the area between the QoS trend and its threshold. This area also helps to quantify the results of each experiment and to compare the different configurations proposed.

4.3.1. Scenario S1 - QoS not satisfied

The scenario S1 concerns how the Managing System behaves when a service does not satisfy the specifications of one or more QoS indicators.

As a reminder, we say that a service does not satisfy its specification of a QoS indicator when, in its latest Analysis Window, the rate of QoS values not satisfying the specifications is at least $1 - QoSSatisfactionRate$.

Before each experiment, the Managed System is deployed into the execution environment with the following initial configuration:

- Restaurant Service: 3 instances.
  All the instances run with the following manipulation:

Instance	Failure Rate	Lag Mean [ms]	Lag Variance [ms]
A	0.8	100	80
B	0.15	150	70
C	0.05	20	10

Table 4.1: Restaurant Service manipulation table

- Ordering Service: 3 instances.
  The other two instances run on random ports with the following manipulation:

Instance	Failure Rate	Lag Mean [ms]	Lag Variance [ms]
A	0.05	500	50
B	0.07	400	100

Table 4.2: Ordering Service manipulation table

- Payment Proxy Service: 1 instance.
  The only instance runs with the following manipulation:

Instance	Failure Rate	Lag Mean [ms]	Lag Variance [ms]
A	0.01	250	100

Table 4.3: Payment Proxy Service manipulation table

- Delivery Proxy Service: 1 instance.
  The only instance is nominal.

During each experiment execution, some abnormal events are injected to trigger relevant behaviours of the Managing System, whose occurrence is summarised in Figure 4.2.
In particular, for each experiment two types of issues are injected:

- **Lag injection**. The instances of the Ordering Service are slowed down twice during the experiments. The first time after 11 minutes from the start of the experiment, lasting 15 seconds. The second time after about 13 minutes from the start, lasting approximately 2 minutes.
- **Network failure**. After 10 minutes from the start, two *UNREACHABLE* Metrics Snapshots referring to one instance of the Restaurant Service are injected in the Probe component, simulating a network failure (e.g., a packet loss or connection timeout).

To analyse the results in a quantitative way, we adopt the so-called *QoS Degradation Area* (QoSSDA) indicator. We define the QoSDA as the total area between the considered QoS threshold and its corresponding Latest Value trend, in the portion of graph where the QoS specification is not satisfied.

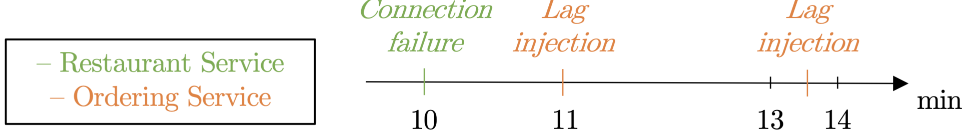


Figure 4.2: Timeline of injected issues

E1 - Choice of the experiments duration

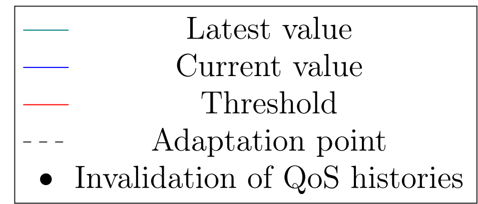
The first experiment aims at identifying the amount of time required by all the experiments. Therefore, we executed a long run (1 hour), and we monitored the system to determine the relevant time window.

The configuration parameters chosen for this experiment are summarized in Table 4.4.

Failure rate threshold	10%	Monitor Scheduling Period	5 sec
Unreachable rate threshold	35%	Instances Shutdown Threshold	40%
Metrics Window Size	6	Experiment duration	60 min
Analysis Window Size	5	User-generation delay	10 ms
Max instance booting time	120 sec	Number of parallel users	50

Table 4.4: S1 – Configuration parameters of experiment E1

The following legend holds for all the plots of this experiment.



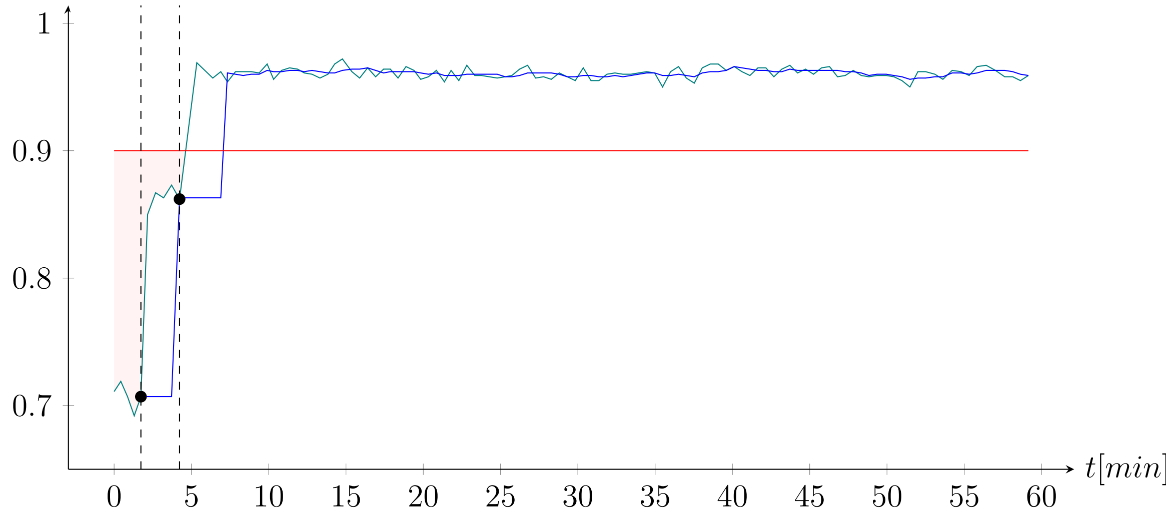


Figure 4.3: S1E1 – Restaurant Service availability

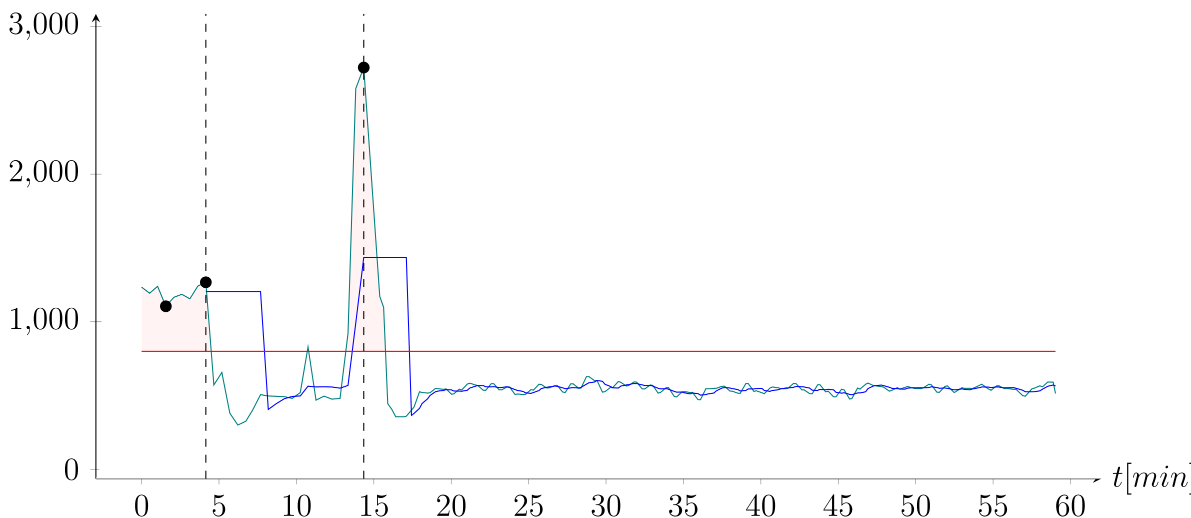


Figure 4.4: S1E1 – Ordering Service average response time

As shown in Figure 4.3 and Figure 4.4, the state of the system becomes steady and stable before the first third of the experiment duration, thanks to the adaptation options applied while the system was not satisfying its QoS specifications. Thus, a duration on

20 minutes was decided for the following experiments, since it is enough time to observe all the relevant events in the system.

RQ1 Summary: An experiment duration of 20 minutes is enough to observe all the events that bring the managed services into a steady QoS state.

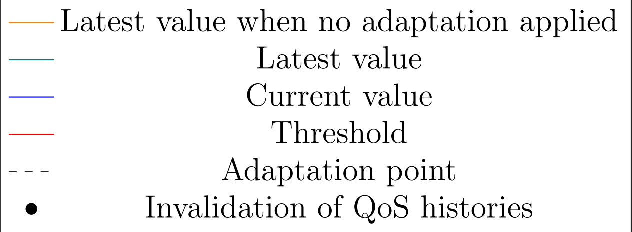
E2 - Adaptation benefit

This experiment aims at analysing the benefit of using the proposed Managing System, compared to when no adaptation is performed. Moreover, it is used as the reference experiment for the next ones of this experimental campaign: indeed, the configuration parameters used in this experiment are used as a starting point for the following experiments, modified accordingly to their needs.
Table 4.5 shows the configuration parameters adopted in E2. According to results of E1, the duration is 20 minutes.

Failure rate threshold	10%	Monitor Scheduling Period	5 sec
Unreachable rate threshold	35%	Instances Shutdown Threshold	40%
Metrics Window Size	6	Experiment duration	20 min
Analysis Window Size	5	User-generation delay	10 ms
Max instance booting time	120 sec	Number of parallel users	50

Table 4.5: S1 – Configuration parameters of experiment E2

The following legend holds for all the plots of this experiment.



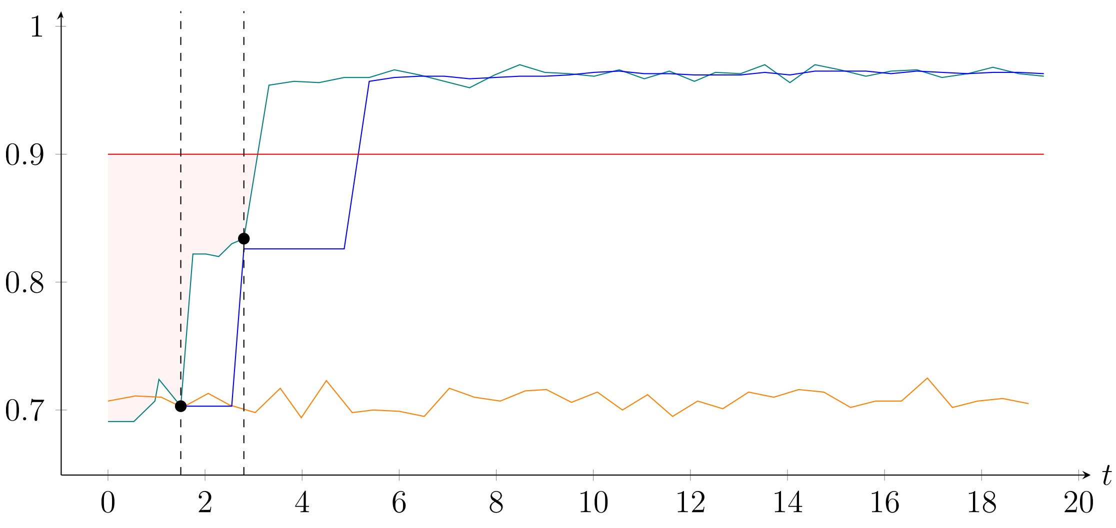


Figure 4.5: S1E2 – Restaurant Service availability

Concerning the Restaurant Service, as shown in Figure 4.5, at the beginning of the experiment (i.e., when $t < 2$) its availability is very low, far from the threshold. Hence, the Managing System changes the load balancer weights of the instances, in order to penalise the bad-performing one. Indeed, one of the instance of the Restaurant Service was manipulated to have a very low availability (see instance A of Table 4.1).
This operation improves the service availability, that however is still not enough. At this point (when $t \approx 3$), another Change Load Balancer option is applied: two instances out of three are shut down, due to their poor performances.
It results in the ultimate choice for the service, which finally reaches a stable and desired value for its availability.
Figure 4.5 shows that the Restaurant Service availability QoSDA is equal to $4.28 \cdot 10^2$ when adaptation is performed (red area), while it is equal to $3.65 \cdot 10^3$ when no adaptation is performed. The adaptation options executed by the Managing System reduce the QoSDA by 88%, compared to the case in which no adaptation is performed.

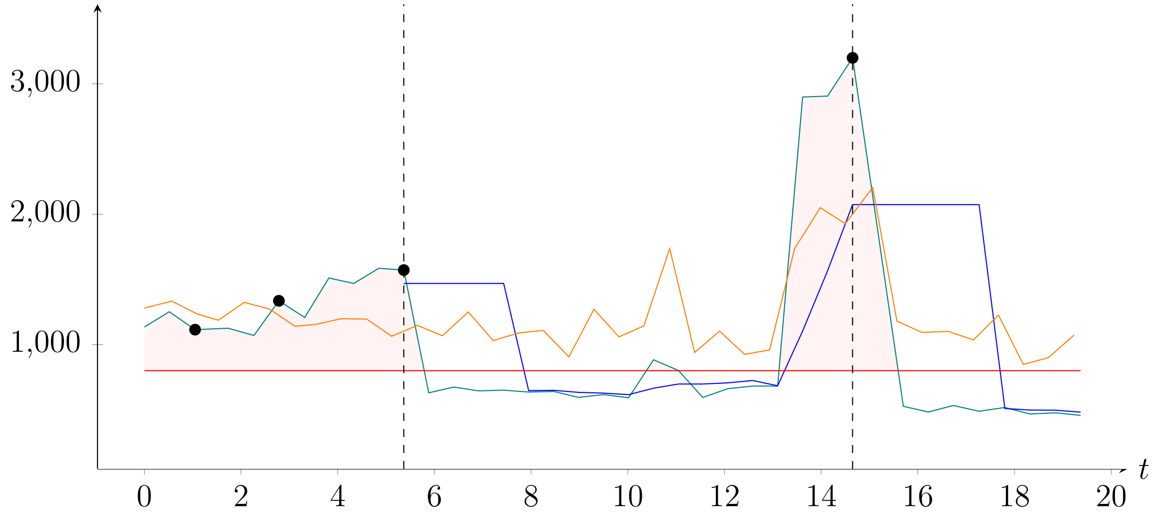


Figure 4.6: S1E2 – Ordering Service average response time

Concerning the Ordering Service, as shown in Figure 4.6, at the beginning of the experiment, its average response time is above the threshold defined by the QoS specification. However, when analysing the service, the Managing System prioritizes the adaptation of the service dependencies, as described in Section 3.3.5. In this specific case, the Ordering Service depends on the Restaurant Service, which does not satisfy all its QoS specifications before $t \approx 3$ (see Figure 4.5).

Thus, all the QoS histories of the Ordering Service are invalidated when an adaptation option is applied for the Restaurant Service. These events are marked by two black points on the plot shown in Figure 4.6: the first at $t \approx 1$ and the last at $t \approx 3$.

When its dependencies do not require adaptation anymore, the Ordering Service is finally adapted, by applying a Change Load Balancer Option when $t \approx 5$.

The Ordering Service is adapted again at $t \approx 15$, during the second *lag injection* event. According to the injections in Figure 4.2, during this experiment the Ordering Service instances are all slowed down for approximately 2 minutes from $t \approx 13$.

This time a new instance is added, improving the QoS of the the Ordering Service, that satisfies all the QoS specifications starting from $t \approx 16$.

It total, the Ordering Service average response time QoSDA when adaptation is performed is equal to $6.39 \cdot 10^3$, while it is equal to $8.22 \cdot 10^3$ when no adaptation is performed. The adaptation options performed by the Managing System reduce the QoSDA by 22% compared to the case in which no adaptation is performed.

RQ2 Summary: The adaptation options proposed and applied by the Managing System improve the QoS of the Managed System.

E3 - Impact of the Metrics Window Size

This experiment aims to illustrate the impact of the size of the Metrics Window on the behavior of the Managing System.

With respect to the reference experiment E2, we reduced the Metrics Window Size (MWS) from 6 to 3. This means that the Analyse component requires 3 Metrics Snapshots to compute a new QoS value, and that 2 Metrics Snapshots with status *UNREACHABLE* in the same Metrics Window make the Managing System considering the instance they refer to as *faulty* (i.e., to shut down).
The configuration parameters are summarized in Table 4.6.

Parameter	Value	Parameter	Value
Failure rate threshold	10%	Monitor Scheduling Period	5 sec
Unreachable rate threshold	35%	Instances Shutdown Threshold	40%
Metrics Window Size	3	Experiment duration	20 min
Analysis Window Size	5	User-generation delay	10 ms
Max instance booting time	120 sec	Number of parallel users	50

Table 4.6: S1 – Configuration parameters of experiment E3

The following legend holds for all the plots of this experiment.

——	Latest value
——	Current value
——	Threshold
- - -	Adaptation point
•	Invalidation of QoS histories

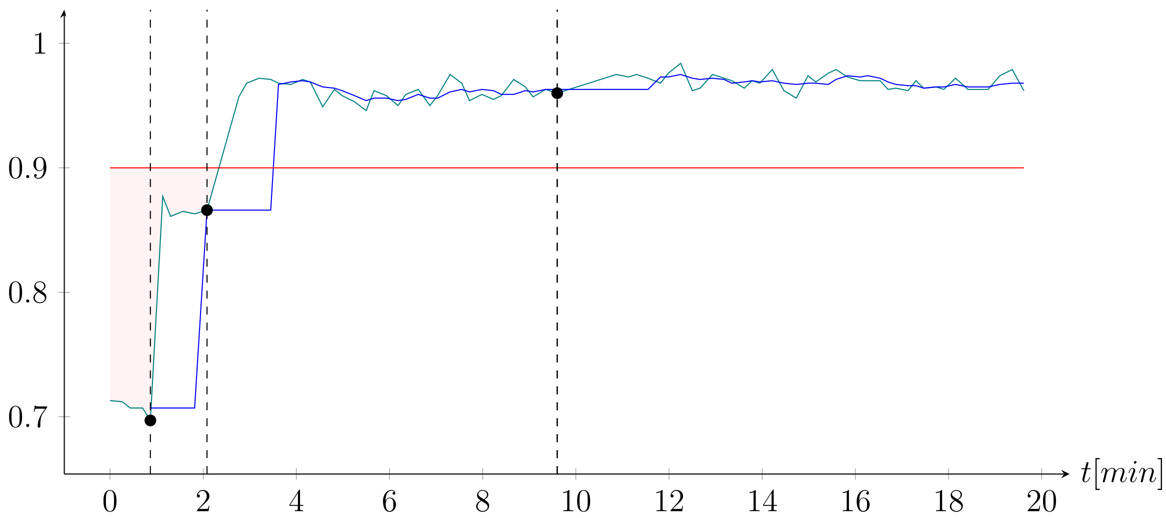


Figure 4.7: S1E3 – Restaurant Service availability

Concerning the Restaurant Service, as shown in Figure 4.7, at the beginning of this experiment the Restaurant Service availability is below the specified threshold, resulting in the execution of two Change Load Balancer Weights options, respectively at $t \approx 1$ and $t \approx 2$.

As in the reference experiment, the first time the option is applied, all the three instances are kept alive, distributing most of the load (specifically, about the 75% of the total load) to the only nominal instance. Instead, the second time the load is redirected entirely to the nominal instance, shutting down the other two.

From that moment on, no more adaptation is performed, since the Restaurant Service availability becomes steady and satisfies the corresponding QoS specifications.

Compared to the reference experiment, in this experiment the Managing System requires less time to propose an adaptation option. Indeed, the time required to determine whether a service requires adaptation is strictly related to the size of its Metrics Window: the smaller the Metrics Window, the less Metric Snapshots are required by the Analyse component to generate a new latest value for each QoS indicator.

This results in a QoSDA availability value of $2.42 \cdot 10^2$, which is 50% smaller than the availability QoSDA of the Restaurant Service in the reference experiment, equal to $4.28 \cdot 10^2$. However, this positive result comes along with a higher cost in terms of number of adaptations. Indeed, at $t \approx 10$ the Restaurant Service is adapted again, after the *network failure injection* event. As a reminder, during this and all the other experiments of this scenario, two Metrics Snapshots of the Restaurant Service are manipulated by the Probe component, simulating a network failure by considering the remaining instance *UNREACHABLE* (see Figure 4.2 for reference). As a consequence, since the rate of *UNREACHABLE* Met-

rics Snapshot in the Metrics Window is above the corresponding threshold, the running instance is shut down and a new instance is started.
By comparing this event with the results obtained in E2, we can see how the last adaptation option was actually not necessary, as shown in Figure 4.5.

The size of the Metrics Window impacts on the decision of shutting down the instances that are *UNREACHABLE* or *FAILED*: while a smaller size could lead to unnecessary adaptations, a larger size could lead to postpone adaptation or ignore such events if they happen rarely.

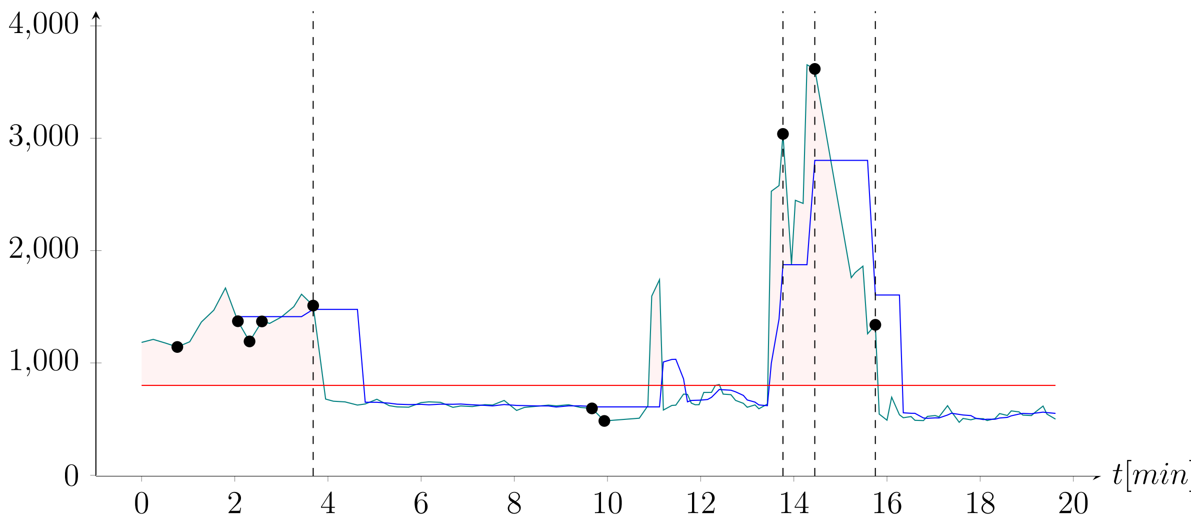


Figure 4.8: S1E3 – Ordering Service average response time

Concerning the Ordering Service, as shown in Figure 4.8, at the beginning of the experiment the average response time of the Ordering Service is above the threshold defined by the QoS specification. However, as in the reference experiment, the Managing System does not adapt the Ordering Service until its dependencies satisfy their constraints. Indeed, the Ordering Service depends on the Restaurant Service, which, according to Figure 4.7, does not satisfy all of its QoS specifications before $t \approx 2$. Thus, all the QoS Histories of the Ordering Service are invalidated when an adaptation option is applied to the Restaurant Service, and while the adaptation process is still in progress. These events are highlighted by the black marks on the graph that are not in correspondence of an adaptation point.
When the dependencies do not require adaptation anymore, the Ordering Service is adapted, by applying a Change Load Balancer Weights option when $t \approx 4$.
The Ordering Service is adapted again at $t \approx 14$, $t \approx 15$ and $t \approx 16$, during the second

lag injection event (see Figure 4.2 for reference).
In chronological order, the Managing System first changed the load balancer weights, then added a new instance and finally changed the load balancer weights again, making the Ordering Service satisfy all the QoS specifications starting from $t \approx 16$.

As in the case of the Restaurant Service, compared to the reference experiment, it takes less time before an adaptation option can be proposed by the Analyse Component. Thus, the QoSDA of this experiment is equal to $6.04 \cdot 10^3$, versus $6.39 \cdot 10^3$ in the reference experiment.

RQ3 Summary: The size of the Metrics Window affects the number of *UNREACHABLE* or *FAILED* Metrics Snapshots required to consider an instance as faulty, in addition to the time needed to compute a new QoS value.
A high Metrics Window size leads to a slower computation of a new QoS latest value. On the other hand, this results in a more reliable computation of this value, since it is based on more snapshots. Moreover, the more snapshots are collected, the more the Managing System waits before considering an instance as faulty.
A small size leads to a faster computation of a new QoS value, and it makes the Managing System considering an instance faulty after a shorter time, possibly leading to false positives and higher number of performed adaptation options.

E4 - Impact of the Analysis Window Size

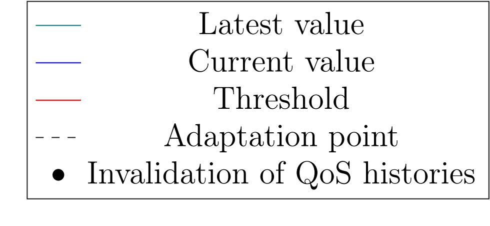
This experiment aims to illustrate the impact of the size of the Analysis Window on the behavior of the Managing System.

With respect to the reference experiment E2, we changed the Analysis Window Size (AWS) from 5 to 1. This means that a single QoS value is taken into consideration when determining whether to perform adaptation or not. This reduces to immediate adaptation, as soon as a QoS value does not satisfy its specifications.
Table 4.7 summarizes the configuration parameters chosen for this experiment.

Failure rate threshold	10%	Monitor Scheduling Period	5 sec
Unreachable rate threshold	35%	Instances Shutdown Threshold	40%
Metrics Window Size	6	Experiment duration	20 min
Analysis Window Size	1	User-generation delay	10 ms
Max instance booting time	120 sec	Number of parallel users	50

Table 4.7: S1 – Configuration parameters of experiment E4

The following legend holds for all the plots of this experiment.
It is worth noting that, since AWS is equal to 1, the latest value is equal to the current value for all time instant t of the experiment.



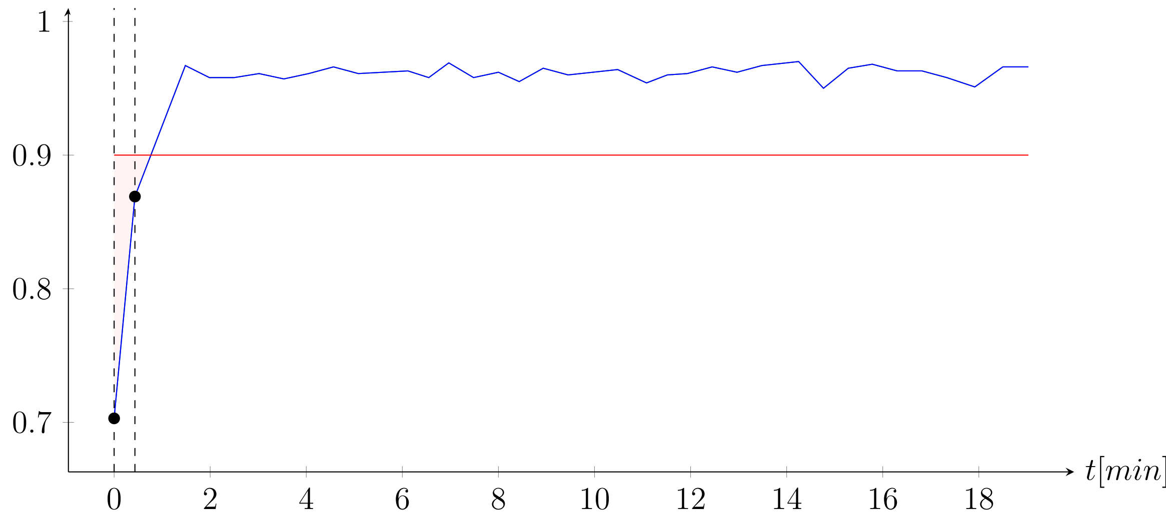


Figure 4.9: S1E4 – Restaurant Service availability

Concerning the Restaurant Service, as shown in Figure 4.9, at the beginning of this experiment the Restaurant Service availability is very low, resulting in two Change Load Balancer Weights options performed by the Managing System: one as soon as the experiment starts (i.e., at $t = 0$), and one at $t \approx 1$, resulting in the shut down of the manipulated instances, as in the previous experiments.
From that point on, the Restaurant Service availability QoS is steady and satisfies its

specifications.
Compared to the reference experiment, it takes less time before an adaptation option could be proposed by the Analyse Component, resulting in a QoSDA equal to $6.54 \cdot 10^1$ for the Restaurant Service availability, which is $\approx$ 85% smaller compared the reference experiment (i.e., $4.28 \cdot 10^2$).

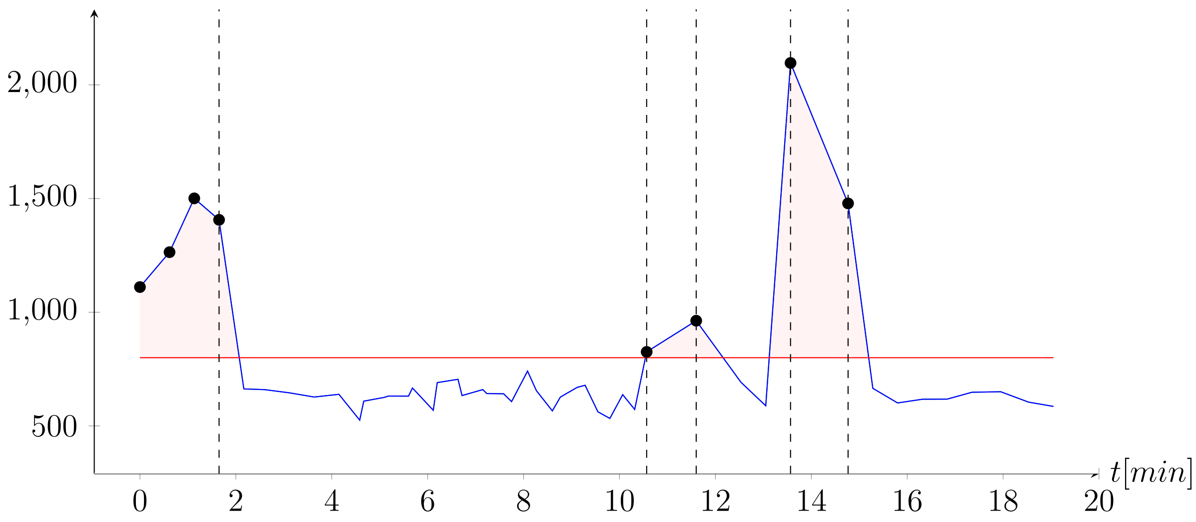


Figure 4.10: S1E4 – Ordering Service average response time

Concerning the Ordering Service, as shown in Figure 4.10, the behaviour of both the Ordering Service and the Managing System in the first half of the experiment is similar to the one exhibited in the previous experiments. Thus, its description is omitted for the sake of brevity.
The first significant event occurs at $t \approx 11$, during the first *lag injection* event (see Figure 4.2 for reference). Compared to the other experiments, where this event is not cause of adaptation due to its limited duration, in this case adaptation is performed twice, with an interval of 1 minute between each other. This is a direct consequence of an Analysis Window Size equal to 1, since the adaptation is performed as soon as there is a QoS value not satisfying its respective specifications.
The same happens during the second *lag injection* event.
In total, for the Ordering Service, five adaptation options were applied:

- 3 Change Load Balancer Weights options (at $t \approx 2$, at $t \approx 10$ and at $t \approx 15$) and
- 2 Add Instance Options (at $t \approx 12$ and at $t \approx 14$)

Again, an instant adaptation results in a smaller QoSDA for the average response time, equal to $2.57 \cdot 10^3$, versus $6.39 \cdot 10^3$ in the reference experiment.

This improvement increases the cost in terms of adaptation options applied, which are performed even when the cause is a transitory event having a small duration (i.e, a false positive).

RQ4 Summary: The size of the Analysis Window has a strong impact on the reaction time of the Managing System.
A high size delays the proposal of adaptation options, incrementing the response time of the Managing System to the issues of the Managed System.
On the other hand, a low size results in a quicker adaptation, at the cost of possibly having more adaptation options applied, even when the cause of adaptation is a transitory event having a small duration, leading to a higher cost.

E5 - Impact of the number of users

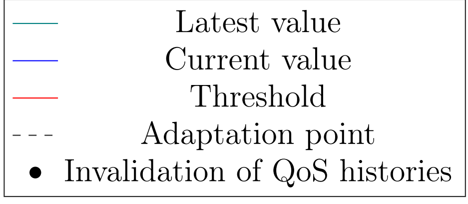
This experiment aims to illustrate the functioning of the Managing System when the Managed System performs under heavy load.

With respect to the reference experiment E2, the number of parallel users that generate requests increases of a 10x factor, from 50 to 500.
Table 4.8 summarizes the configuration parameters chosen for this experiment.

Parameter	Value	Parameter	Value
Failure rate threshold	10%	Monitor Scheduling Period	5 sec
Unreachable rate threshold	35%	Instances Shutdown Threshold	40%
Metrics Window Size	6	Experiment duration	20 min
Analysis Window Size	5	User-generation delay	10 ms
Max instance booting time	120 sec	Number of parallel users	500

Table 4.8: S1 – Configuration parameters of experiment E5

The following legend holds for all the plots of this experiment.



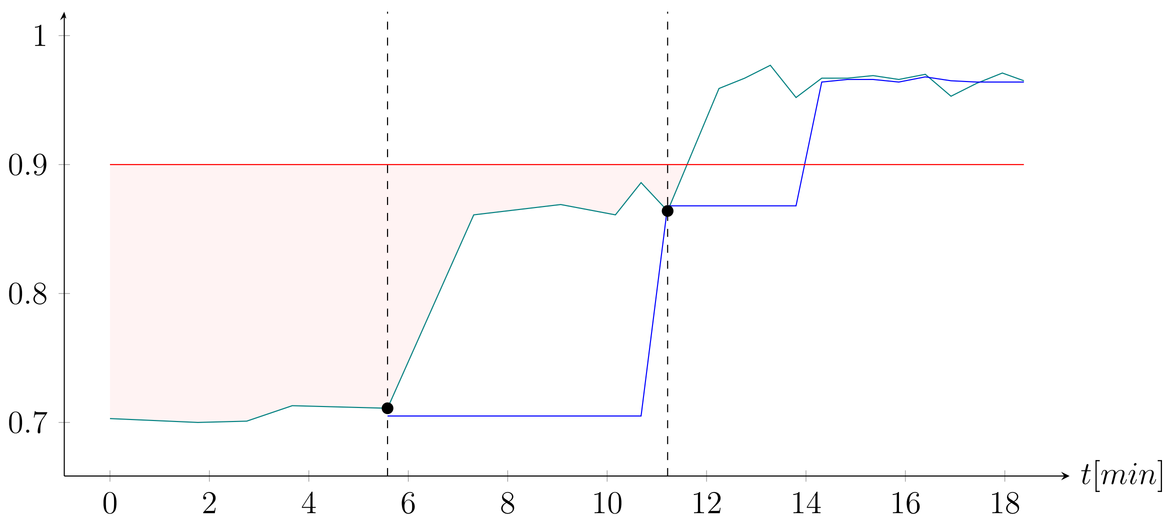


Figure 4.11: S1E5 – Restaurant Service availability

Concerning the Restaurant Service, as shown in Figure 4.11, at the beginning of this experiment the service availability is below its threshold. However, with respect to the reference experiment (see Figure 4.5), it now takes more time to the Monitor component to collect the Metrics Snapshots from all the instances. Indeed, the high load of the instances slows down the overall interaction between RAMSES and the Managed System, which negatively impacts on the duration of each loop iteration.
As a result, the first adaptation option – a Change Load Balancer Weights option – is applied at $t \approx 6$.
For the same reason, since the specifications on the service availability are still not satisfied, a new adaptation is performed at $t \approx 11$. Indeed, another Change Load Balancer Weights option is applied, this time shutting down the two instances with the highest failure rate.

As a result, the QoSDA is noticeably higher, due to the higher time elapsed from the discovery of the problem to the application of the proposed adaptation option. Indeed, the availability QoSDA is equal to $1.43 \cdot 10^3$, versus $4.28 \cdot 10^2$ in the reference experiment.

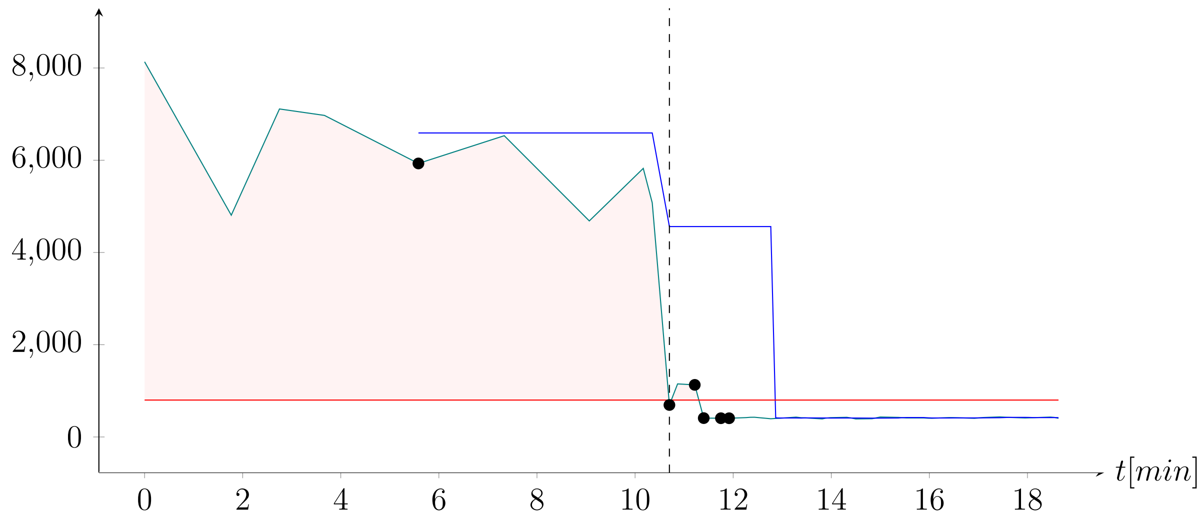


Figure 4.12: S1E5 – Ordering Service average response time

As for the Restaurant Service, it takes more time for the Managing System to collect the Metrics Snapshots from all the Ordering Service instances, delaying the first adaptation option, as shown in Figure 4.12.
Indeed, the first (and only) adaptation option, which is a Change Load Balancer Weights option, is applied at $t \approx 11$.

For the same reason explained before, the average response time QoSDA is equal to $5.61 \cdot 10^4$, versus $6.39 \cdot 10^3$ in the reference experiment.

RQ5 Summary: When the Managed System is handling a huge number of requests simultaneously, the retrieval of the Metrics Snapshots takes more time, due to an increased response time of each managed service. This slows down the adaptation process, but the Managing System still behaves as expected.

4.3.2. Scenario S2 - Service unavailable

The scenario S2 is the second adaptation scenario under analysis. In particular, it focuses on how the Managing System behaves when a service is unavailable.
A service is defined as unavailable when, from the point of view of a client of that service, there is no instance capable of processing the client's requests.

In this scenario, before each experiment is executed, the Managed System is freshly deployed with one nominal instance per service (i.e., without any manipulation).

During each experiment execution, issues are periodically injected to trigger relevant behaviours of the Managing System.
In particular, the only instance of the Payment Proxy Service running at the start of the experiments is made artificially fail after 40 seconds, and then restarted after 15 seconds from the inducted fail. This procedure is repeated, until that instance is not shut down by the Managing System, if this happens.

All the proposed experiments for this scenario have a fixed duration of 5 minutes.

Each experiment is described individually in the next sections, which include the graphs showing the number of instances of the Payment Proxy Service both from the point-of-view of a client (i.e., a user of the Managed System), and from the point-of-view of the Managing System. It must be noticed that, for the sake of clarity of the provided graphs, all of them only show the first 2 minutes of each experiment, since all the relevant events happen before them.
The proposed graph was chosen in order to highlight the behaviour of the Managing System in this scenario, which is not appreciable from the kind of graphs generated for the experiments belonging to S1.

E1 - Analysis of the self-healing properties

This experiment aims at highlighting the self-healing capabilities of the Managing System, that allow to effectively react when one of the managed services is completely unavailable.

Table 4.9 summarizes the configuration parameters chosen for this experiment.

Failure rate threshold	10%	Monitor Scheduling Period	5 sec
Unreachable rate threshold	35%	Instances Shutdown Threshold	40%
Metrics Window Size	6	Experiment duration	5 min
Analysis Window Size	5	User-generation delay	10 ms
Max instance booting time	120 sec	Number of parallel users	50

Table 4.9: S2 – Configuration parameters of experiment E1

The evolution of the number of the Payment Proxy Service running instances during the experiment is shown in Figure 4.13, both from the point of view of an external client and from the point of view of the Managing System.

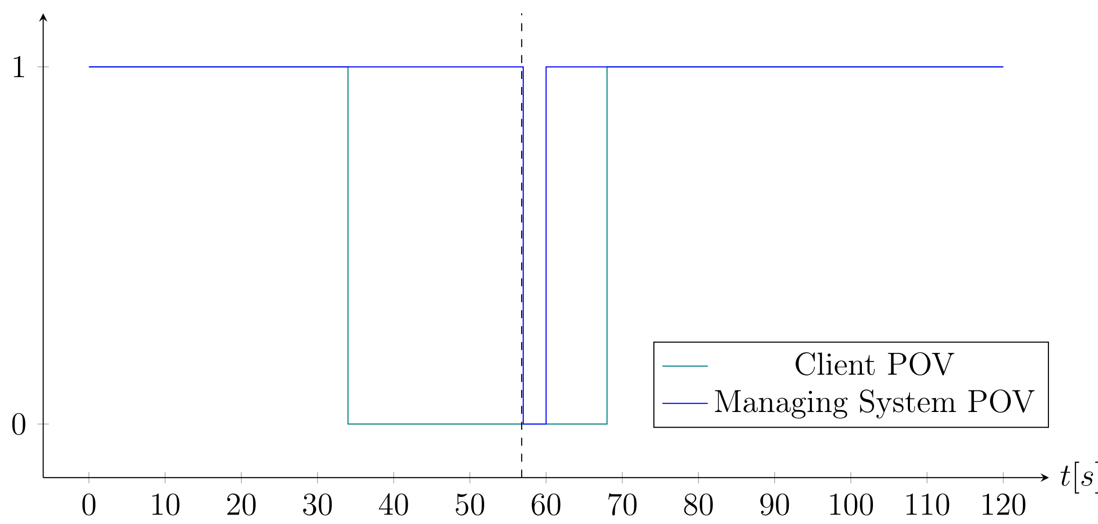


Figure 4.13: S2E1 – Payment Proxy Service – Number of instances

As shown in Figure 4.13, in the case of this experiment, the instance of the Payment Proxy Service behaves correctly before $t = 34$. At that point, a failure occurs, and it takes the Managing System approximately 25 seconds to react to the failure. Indeed, the Metrics Window of that instance needs to be filled with *UNREACHABLE* or *FAILED* Metrics Snapshots, with a rate above their respective thresholds, as described in Section 3.3.5, in order to let the Managing System consider it as unavailable. Considering the configuration parameters of the experiments, and in particular the Metrics Window Size and the Monitor Period, this justifies the 25 seconds required to trigger the adaptation.
Approximately at $t \approx 57$, the Managing System proposes to shut down the faulty instance and to start a new one, which becomes ready to accept requests at $t \approx 68$.

As a result, the Payment Proxy Service becomes available again to a potential end user.

RQ6 Summary: When a service is completely unavailable due to the lack of active instances, the Managing System correctly restore the health of the service by starting a new instance (self-healing property).

E2 - Analysis of the failure tendency detection

This experiment was designed to analyse the self-healing capabilities of the proposed Managing System, that shall take into account the tendency of an instance to be *UNREACHABLE* or *FAILED*.

Table 4.10 summarizes the configuration parameters chosen for this experiment.

Failure rate threshold	10%	Monitor Scheduling Period	5 sec
Unreachable rate threshold	35%	Instances Shutdown Threshold	40%
Metrics Window Size	6	Experiment duration	5 min
Analysis Window Size	5	User-generation delay	10 ms
Max instance booting time	120 sec	Number of parallel users	50

Table 4.10: S2 – Configuration parameters of experiment E2

The evolution of the number of the Payment Proxy Service running instances during the experiment is shown in Figure 4.14, both from the point-of-view of an external client and from the point-of-view of the Managing System itself.

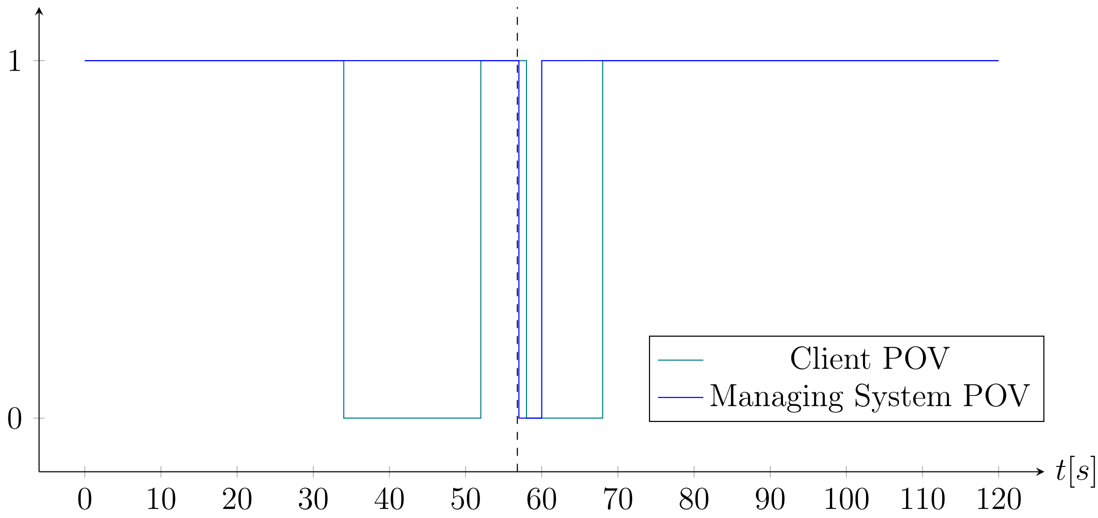


Figure 4.14: S2E2 – Payment Proxy Service – Number of instances

During this experiment, the instance of the Payment Proxy Service behaves correctly before $t = 34$. At that point, a failure occurs. After a little time, the failed instance restarts spontaneously, becoming available to elaborate requests again when $t = 52$. However, at $t = 57$ the Managing System proposes to shut down the instance, and to start a new one, which becomes available to a potential user at $t = 68$. This happens because the Metrics Window of the original instance is filled with *UNREACHABLE* or *FAILED* Metrics Snapshots, whose rate is above the respective thresholds, as described in Section 3.3.5.

As a result, the Managing System decides to replace the faulty instance, in order to reduce the risk of users experiencing repeated downtimes when interacting with the service.

RQ7 Summary: When an instance is unreachable or failed for a substantial amount of time, such that the rate of *UNREACHABLE* or *FAILED* Metrics Snapshot in the Metrics Window is higher than their respective thresholds, the Managing System is able to replace the instance with a new one.

E3 - Impact of the Monitor Period

This experiment aims to analyse the impact of the Monitor Scheduling Period on the behavior of the Managing System. With respect to the other experiments of this scenario, the Monitor Period changes from 5 seconds to 15 seconds.

Table 4.11 summarizes the configuration parameters chosen for this experiment.

Parameter	Value	Parameter	Value
Failure rate threshold	10%	Monitor Scheduling Period	15 sec
Unreachable rate threshold	35%	Instances Shutdown Threshold	40%
Metrics Window Size	6	Experiment duration	5 min
Analysis Window Size	5	User-generation delay	10 ms
Max instance booting time	120 sec	Number of parallel users	50

Table 4.11: S2 – Configuration parameters of experiment E3

The evolution of the number the Payment Proxy Service running instances during the experiment is shown in Figure 4.15, both from the point-of-view of a client and from the point-of-view of the Managing System. Note that even if only the first two minutes are shown in the graph, the oscillatory behaviour of the system should be considered periodic. This is a direct consequence of the fault injection performed on the single service instance,

as anticipated in the description of this scenario.

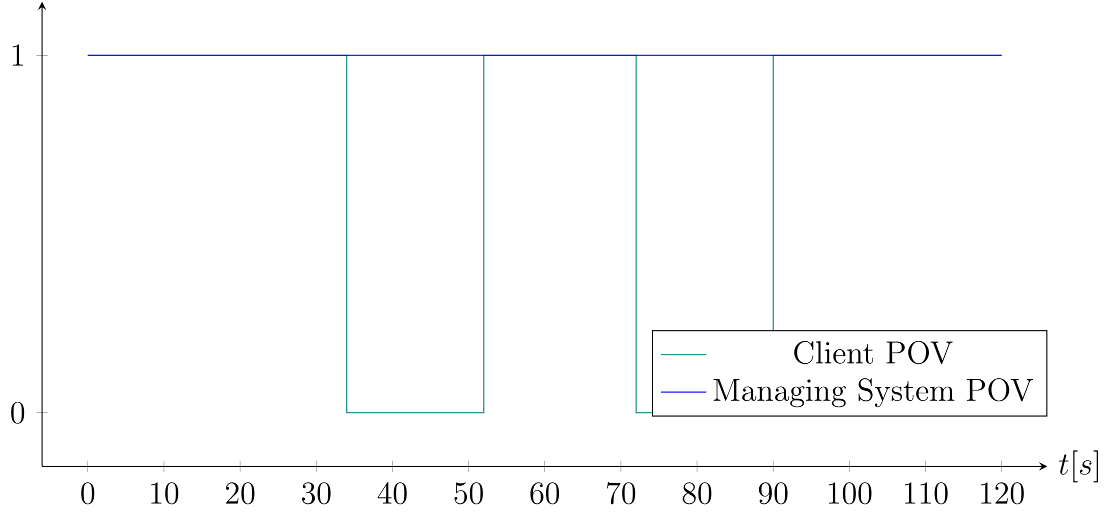


Figure 4.15: S2E3 – Payment Proxy Service – Number of instances

As in the previous experiments of this scenario, the instance of the Payment Proxy Service fails approximately at $t = 34$. After a short time, the failed instance restarts spontaneously, becoming available to elaborate requests again when $t = 52$.
However, due to a lower frequency of the Monitor component routine (i.e., a higher Monitor Period) with respect to the previous experiments, the failure is not detected and no adaptation is performed.
In addition, when this behavior is recurrent – as in the case of the initial instance of the Payment Proxy Service – the impact is even higher on the client of the service, that experiences a higher overall downtime.

RQ8 Summary: The Monitor Period has a strong impact on the failure detection capabilities of the Managing System. A high period reduces the ability of detecting occurring failures. A low period yields a higher overhead for both the Managed and the Managing System. Thus, this quantity must be chosen by the users of the Managing System according to their requirements with respect to the failure detection of the instances, but also according to the performance of their instances, since the overhead generated by the Managing System when collecting the snapshot may negatively affect the performance of the managed services.

4.3.3. Scenario S3 - Better implementation available

S3 is the third adaptation scenario considered for the experimental campaign. It concerns how the Managing System behaves when there is a better implementation available for a managed service.
For this scenario, we conducted a single experiment with the following setting:

- Restaurant Service: 2 instances.
  The instances are both nominal and run on random ports.
- Ordering Service: 2 instances.
  The instances are both nominal and run on random ports.
- Payment Proxy Service: 1 instance.
  The single instance runs on a random port with the following manipulation:

Instance	Failure Rate	Lag Mean [ms]	Lag Variance [ms]
A	0.14	19	1

Table 4.12: Payment Proxy Service manipulation table

- Delivery Proxy Service: 1 instance.
  The only instance is nominal and it runs on port 58095.

E1 - Analysis of the self-optimization capabilities

This experiment aims to illustrate the self-optimization capabilities of the proposed Managing System.

Table 4.13 shows the configuration parameters used by this experiment, which are the same of the reference experiment E2 in S1.

Parameter	Value	Parameter	Value
Failure rate threshold	10%	Monitor Scheduling Period	5 sec
Unreachable rate threshold	35%	Instances Shutdown Threshold	40%
Metrics Window Size	6	Experiment duration	20 min
Analysis Window Size	5	User-generation delay	10 ms
Max instance booting time	120 sec	Number of parallel users	50

Table 4.13: S3 – Configuration parameters of experiment E1

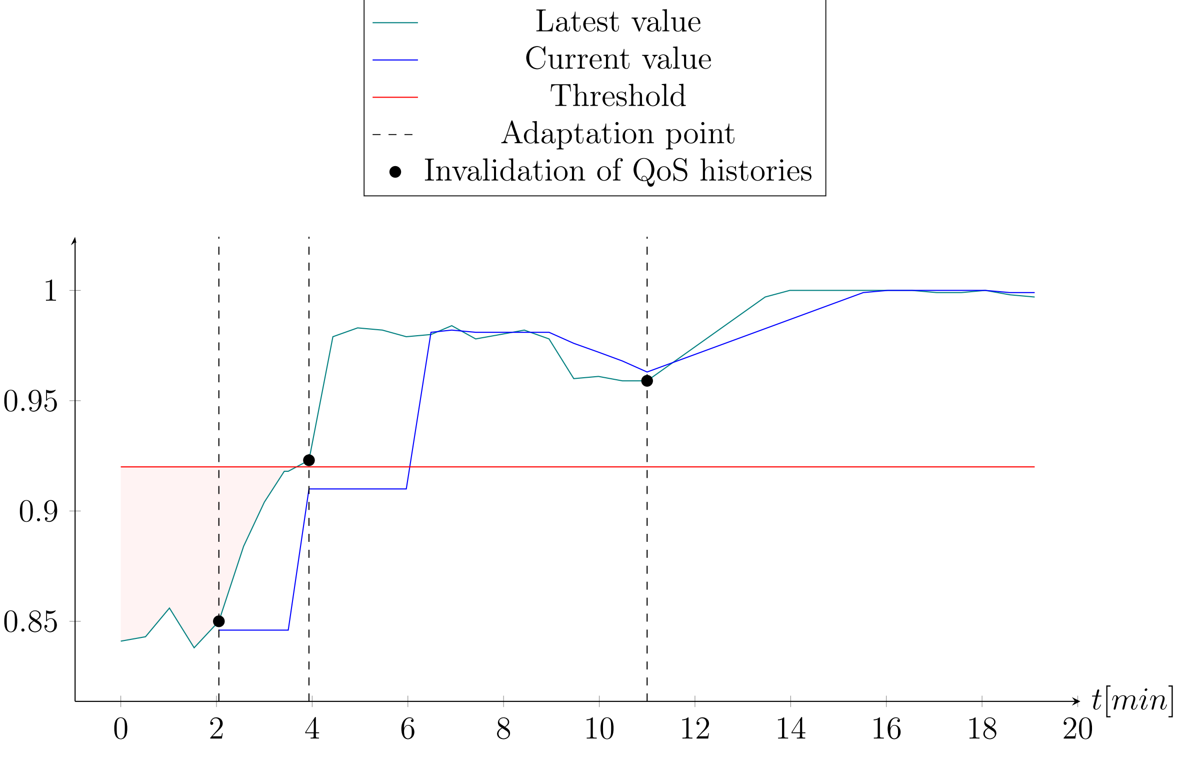


Figure 4.16: S3E1 – Payment Proxy Service availability

As stated in Section 3.3.5, the Change Implementation option is considered by the Managing System when the *penalty* of a service having multiple possible implementations is less than or equal to the *trust* given to the current implementation of that service.
The current implementation at the beginning of the experiment has a *trust* equal to 2.
Due to the manipulation of the instance of the Payment Proxy Service, its failure rate makes the service not satisfy the QoS specifications for availability. Hence, the Managing System first applies an Add Instance option – at $t \approx 2$ – and then applies a Change Load Balancer Weights option – at $t \approx 4$ – which results in the shutdown of the manipulated instance.

At this point, the value of the penalty indicator is equal to the trust of the current implementation of the service, since adaptation was required twice. However, considering the availability benchmarks of the other two possible implementation of the Payment Proxy Service, the change of implementation is not beneficial to the service performance before $t \approx 11$. Indeed, at $t \approx 11$, we simulate a change performed by the system admin on the availability benchmark of one of the other possible implementations: the new availability benchmark is set to 1.0.
The Change Implementation option – which was always considered from $t \approx 4$ – is applied at $t \approx 11$, since now that option is estimated to bring a benefit to the availability of the

Payment Proxy Service.

As a result, the service availability with the new implementation is actually better than the previous, coherent with the new provided benchmark.

RQ9 Summary: According to our experience, it results that the proposed Managing System is able to change the implementation of a managed service in order to improve the service's performance. This realizes the self-optimizing capability of RAMSES, since the system recognizes better service implementations and moves itself towards higher QoS levels, even if the corresponding specifications are satisfied.

4.3.4. Reusability of the Managing System

To show the reusability of the proposed Managing System, and answer RQ10, we built a simple microservice-based system as an alternative to SEFA.

This additional application has been developed from scratch using Spring Boot and Spring Cloud frameworks. It has the following two services:

- the **Randint Producer Service**, which generates random integer numbers upon request. It exposes a REST-API with a single endpoint, which returns the new generated integer;
- the **Randint Vendor Service**, which, upon request, contacts the Randint Producer Service via its API to obtain a new random integer. It exposes a REST-API with a single endpoint, which returns the integer computed by the Randint Producer Service.

This simple system adopts the following design choices.

- Client-side service discovery and load balancing is performed, implemented respectively by Spring Cloud Netflix Eureka and by the Weighted Random Load Balancer (i.e., load balancing with proportionate fitness selection rule), implemented in the provided load balancing library;
- An API gateway acts as single entry point for the application and it is implemented using Spring Cloud Gateway;
- The Probe and Actuator used are the same provided with SEFA (see Section 3.2.4);
- Concerning the deployment of the system, all the services are run in Docker containers.

Before the experiment is executed, the Simple Managed System is freshly deployed as follows:

- Randint Producer Service: 2 instances.
  The instances run on random ports with the following manipulation:

Instance	Failure Rate	Lag Mean [ms]	Lag Variance [ms]
A	0.001	40	10
B	0.001	180	2

Table 4.14: Randint Producer Service manipulation table

- Randint Vendor Service: 1 instance.
  The single instance runs on a random port with the following manipulation:

Instance	Failure Rate	Lag Mean [ms]	Lag Variance [ms]
A	0.001	120	1

Table 4.15: Randint Vendor Service manipulation table

The experiment is described in details in the next section, that includes the plots of the average response time trend for both the Randint Producer Service and the Randint Vendor Service. The plots of their availability trend is not included in this chapter for the sake of brevity.

E1 - Analysis of the Managing System reusability

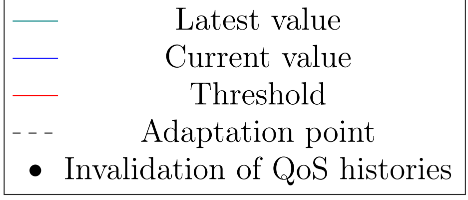
This experiment aims at highlighting the reusability capabilities of the Managing System, which is designed to be agnostic with respect to the Managed System.

Table 4.16 summarizes the configuration parameters chosen for this experiment.

Parameter	Value	Parameter	Value
Failure rate threshold	10%	Monitor Scheduling Period	5 sec
Unreachable rate threshold	35%	Instances Shutdown Threshold	40%
Metrics Window Size	6	Experiment duration	20 min
Analysis Window Size	5	User-generation delay	10 ms
Max instance booting time	120 sec	Number of parallel users	50

Table 4.16: Reusability experiment – Configuration parameters

The following legend holds for all the plots of this experiment.



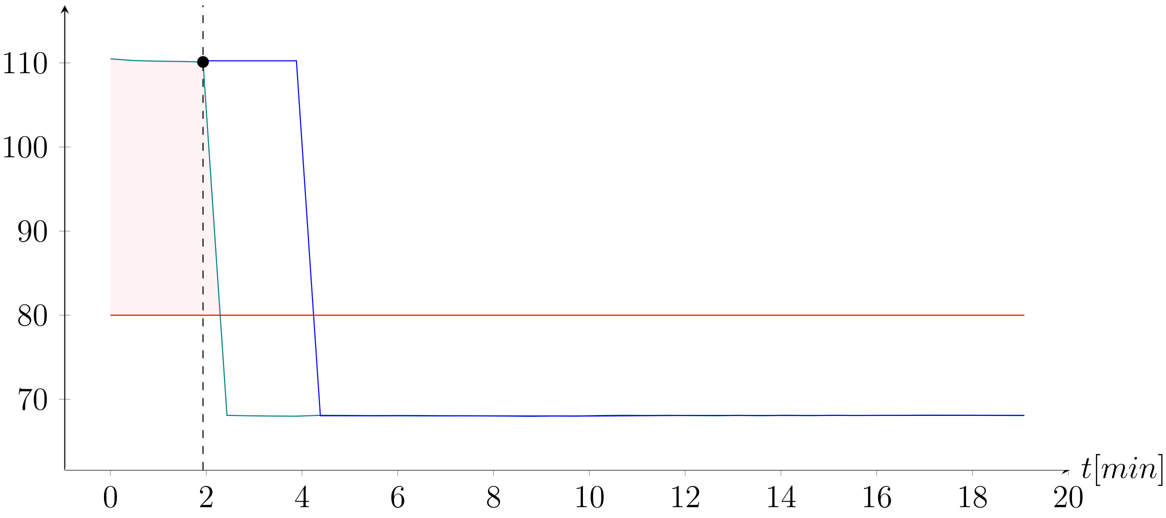


Figure 4.17: S4E1 – Randint Producer Service average response time

As shown in Figure 4.17, at the beginning of the experiment, the Randint Producer Service does not satisfy its constraint on the service average response time. Indeed, of the two running instances, one is manipulated to heavily delay its responses (see instance B of Table 4.14).
Hence, Change Load Balancer option is applied: this results to be a good decision, since after performing this adaptation the service reaches a stable and desired value for its average response time.

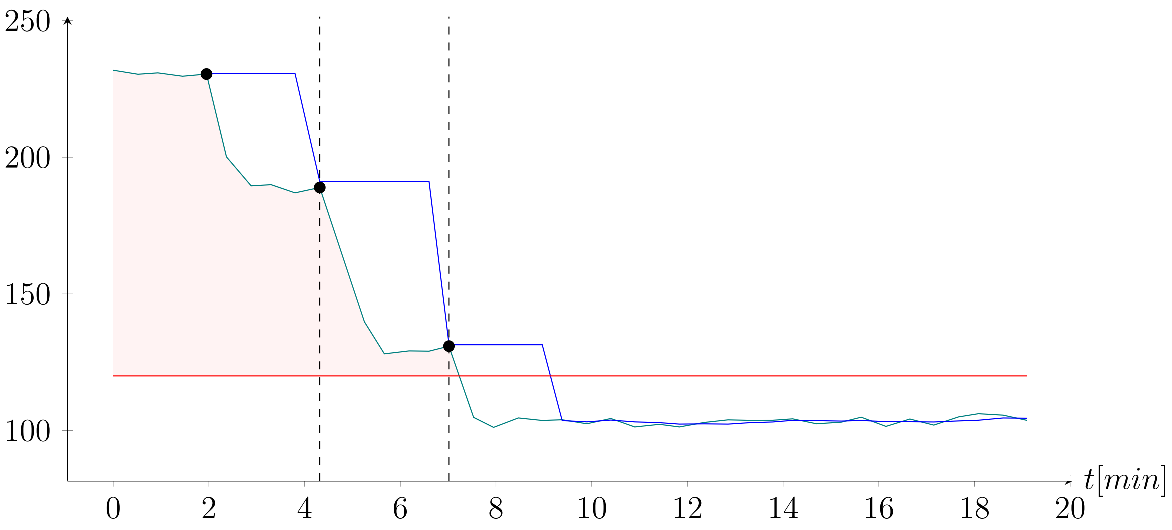


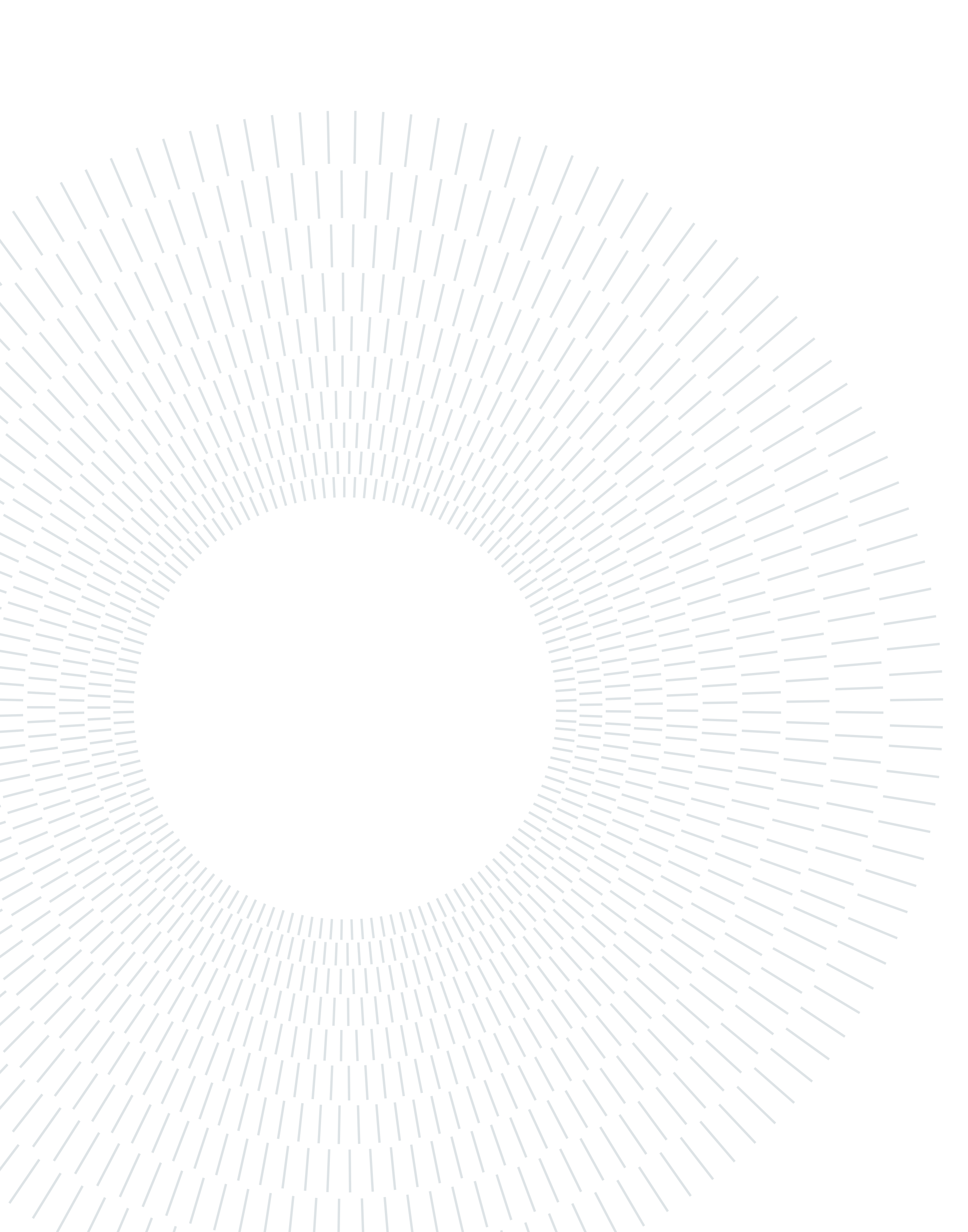
Figure 4.18: S4E1 – Randint Vendor Service average response time

As shown in Figure 4.18, at the beginning of the experiment, also the Randint Vendor Service does not satisfy its constraint on the service average response time. In partic-

ular, this is a direct consequence of the performance of the Randint Producer Service, which is a dependency of the Randint Vendor Service. Thus, as shown in S1-E2 (see Section 4.3.1), the Managing System prioritizes the adaptation of the Randint Producer Service. However, even if from $t \approx 2$ the Randint Producer Service satisfies the constraint on the service average response time, the average response time of the Randint Vendor Service is still above the threshold.

Hence, a new instance is added at $t \approx 4$. However, this single adaptation is not enough to make the Randint Vendor Service satisfy the constraint on the average response time. This leads the Managing System to apply another adaptation option, at $t \approx 7$. This time a Change Load Balancer option is applied, and from $t \approx 7$ the service reaches a stable and desired value for its average response time.

RQ10 Summary: It results that the proposed Managing System is agnostic with respect to the system to adapt, and it can be easily reused without changing its internal logic.



5 | Conclusions And Future Work

This chapter focuses on summarising the whole project and on highlighting the most relevant conclusions of our work. Furthermore, suggestions on possible future improvements to our work are provided.

5.1. Conclusions

Developing Self-Adaptive Systems is a challenging task. Their development process is cost and time consuming, whether developing both the Managing and Managed subsystems from scratch or starting from a preexisting application to be adapted.
Even if some works have already addressed this problem by proposing a reusable infrastructure to ease the engineering of SASs (e.g., the RAINBOW framework [15]), due to their abstraction and generality they are not ready-to-use, and a significant amount of time and effort is required to set them up.

Our solution, RAMSES, aims at providing a Managing System that is easily reusable for service-based applications. Since these applications usually have the common need of satisfying constraints on some QoS properties, RAMSES tackles this need by abstracting its logic from the managed system's one. This makes RAMSES able to adapt different service-based applications without changing its managing logic: it only requires the managed system to provide a probe and an actuator component offering specific interfaces and to satisfy a set of prerequisites. Moreover, since it was designed according to the microservice architectural pattern, it also allows to exploit the advantages brought by the use of this pattern, such as modularity, decoupling and easy maintainability.
In addition, to test the proposed solution and to build a microservice-based system that could be easily reused and adapted for research purposes, we implemented a real-world microservice-based eFood application – SEFA – using a well-known technology stack (e.g., Java, Spring, Docker) and following design patterns that are common to this kind of applications (e.g., service discovery, load-balancing, circuit breakers). Thus, we provide the scientific community with a complete exemplar of a self-adaptive system, made of two non-simulated and fully-implemented subsystems, which can be reused for different goals.

Finally, we conducted an experimental campaign to question the effectiveness of the proposed solution by answering 10 research questions. As a result, the experiments highlighted the benefits brought by RAMSES to a real-world application and allowed us to better understand the consequences of tuning the different configuration parameters of the Managing System.

5.2. Future directions

The proposed solution is a first attempt at realising a modular and reusable Managing System, which is open to further improvements.

Future versions of RAMSES could propose a deeper analysis process or extend its set of adaptation options. An enhanced analysis routine could take into account other metrics in order to build more reliable and accurate indicators of the managed services' performances and to perform new specific adaptation options. For instance, a future version of RAMSES could exploit metrics on the resources usage and on the circuit breakers – which are already collected by RAMSES – either to include them when considering the existing adaptation options, or to propose new types of adaptation options (e.g., reallocation of resources, change of the circuit breakers' configuration).

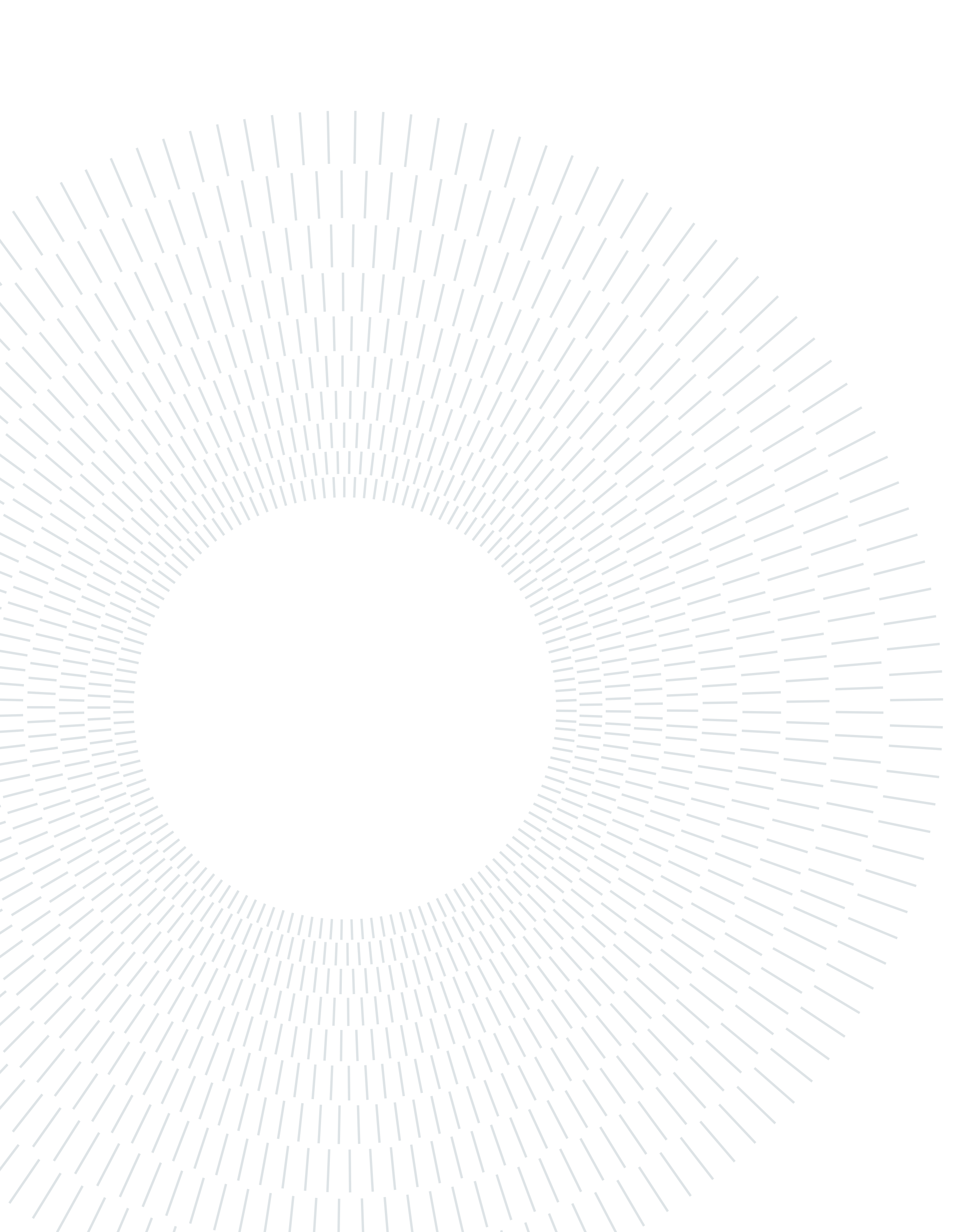
Concerning the metrics monitored by the system, a more advanced version of RAMSES could also extend its scope, monitoring not only the managed systems but also the properties of the environment (e.g., temperature, network bandwidth, the performance of the physical device hosting an instance).

In addition, to enhance the separation of concerns between loop components, the performance indicators of both the services and the instances could be computed by the Analyse component, allowing the Plan component to be agnostic with respect to the way the performance indicators are defined. In this way, the Analyse and the Plan components would be less coupled, and the microservice architecture could be better exploited to separately maintain these components. Moreover, the performance indicators could be defined by more complex relations, taking into account all the metrics considered by the Analyse component.

At the time of writing, the adaptation options to apply are chosen according to the benefit they are estimated to bring to the system, as stated in Section 3.3.6. Further versions of the Plan Component could encompass a more complex decision-making process, enriching the existing one by taking into account the cost of an adaptation option and the risks that can arise after applying it. Moreover, the benefit estimation could also consider

analysing the history of performed adaptation options, using machine-learning techniques to quantify the actual benefits they brought to the managed system.

Finally, RAMSES might be improved by performing operation-level analysis, rather than the current service-level one. This, together with the definition of new adaptation options, could help in improving even more the performance of the managed services. Moreover, a deeper dependency analysis could be performed, allowing the user to define dependencies between services at the operation-level rather than at the service-level, allowing RAMSES to better determine if the issues of a given service are actually caused by one of its dependencies.



Bibliography

[1] J. Andersson, L. Baresi, N. Bencomo, R. d. Lemos, A. Gorla, P. Inverardi, and T. Vogel. Software engineering processes for self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*, pages 51–75. Springer, 2013.

[2] API Gateway Pattern. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern, 2022. Accessed: 2022-09-01.

[3] C. Barna, H. Ghanbari, M. Litoiu, and M. Shtern. Hogna: A platform for self-adaptive applications in cloud environments. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 83–87, 2015. doi: 10.1109/SEAMS.2015.26. URL https://doi.org/10.1109/SEAMS.2015.26.

[4] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. *Engineering Self-Adaptive Systems through Feedback Loops*, pages 48–70. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02161-9. doi: 10.1007/978-3-642-02161-9_3. URL https://doi.org/10.1007/978-3-642-02161-9_3.

[5] L. Cabibbo. https://github.com/aswroma3/asw/tree/master/projects/asw-825-spring-boot, 2022. Accessed: 2022-06-10.

[6] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, 2011. doi: 10.1109/TSE.2010.92. URL https://doi.org/10.1109/TSE.2010.92.

[7] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw,

M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, pages 1–26. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02161-9. doi: 10.1007/978-3-642-02161-9_1. URL https://doi.org/10.1007/978-3-642-02161-9_1.

[8] Circuit Breaker Pattern. https://martinfowler.com/bliki/CircuitBreaker.html, 2022. Accessed: 2022-09-11.

[9] Client-Side Service Discovery and Load-balancing. https://microservices.io/patterns/client-side-discovery.html, 2022. Accessed: 2022-10-24.

[10] Database-per-Service Pattern. https://microservices.io/patterns/data/database-per-service.html, 2022. Accessed: 2022-09-12.

[11] Eureka Discovery Service. https://cloud.spring.io/spring-cloud-netflix/reference/html/, 2022. Accessed: 2022-10-20.

[12] R. R. Filho, E. Alberts, I. Gerostathopoulos, B. Porter, and F. M. Costa. Emergent web server: An exemplar to explore online learning in compositional self-adaptive systems. In *2022 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 36–42, 2022. doi: 10.1145/3524844.3528079. URL https://doi.org/10.1145/3524844.3528079.

[13] C. A. Floudas. Mixed-Integer Linear Optimization. In *Nonlinear and Mixed-Integer Optimization: Fundamentals and Applications*. Oxford University Press, 11 1995. ISBN 9780195100563. doi: 10.1093/oso/9780195100563.003.00010. URL https://doi.org/10.1093/oso/9780195100563.003.00010.

[14] E. Freeman, E. Robson, B. Bates, and K. Sierra. *Head First Design Patterns*, chapter 7. O'Reilly, 10 2004.

[15] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10): 46–54, 2004. doi: 10.1109/MC.2004.175. URL https://doi.org/10.1109/MC.2004.175.

[16] GitHub webhooks. https://docs.github.com/en/developers/webhooks-and-events/webhooks/about-webhooks, 2022. Accessed: 2022-11-02.

[17] International Conference on Autonomic Computing and Self-Organizing Systems – ACSOS series, 2020-2022. Accessible at https://dblp.org/db/conf/acsos/index.html.

[18] International Symposium on Software Engineering for Adaptive and

Self-Managing Systems – SEAMS series, 2006-2022. Accessible at https://dblp.org/db/conf/seams/index.html.

[19] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1): 41–50, 2003. doi: 10.1109/MC.2003.1160055. URL `https://doi.org/10.1109/MC.2003.1160055`.

[20] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering (FOSE '07)*, pages 259–268, 2007. doi: 10.1109/FOSE.2007.19. URL `https://doi.org/10.1109/FOSE.2007.19`.

[21] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17:184–206, 2015. ISSN 1574-1192. doi: 10.1016/j.pmcj.2014.09.009. URL `https://doi.org/10.1016/j.pmcj.2014.09.009`.

[22] P. Lambert. Availability and beyond: Understanding and improving the resilience of distributed systems on aws. Technical report, Amazon AWS, 2021. Chapter *Measuring Availability*. Accessible at: `https://docs.aws.amazon.com/pdfs/whitepapers/latest/availability-and-beyond-improving-resilience/availability-and-beyond-improving-resilience.pdf#measuring-availability`.

[23] G. Lulli, P. Potena, and C. Raibulet. Resource allocation, trading and adaptation in self-managing systems. In C. Salinesi and O. Pastor, editors, *Advanced Information Systems Engineering Workshops*, pages 385–396, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22056-2. doi: 10.1007/978-3-642-22056-2_41. URL `https://doi.org/10.1007/978-3-642-22056-2_41`.

[24] M. Mitchell. *An Introduction to Genetic Algorithms*, pages 124–125. The MIT Press, 1999.

[25] G. A. Moreno, B. Schmerl, and D. Garlan. Swim: An exemplar for evaluation and comparison of self-adaptation approaches for web applications. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '18, page 137–143, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357159. doi: 10.1145/3194133.3194163. URL `https://doi.org/10.1145/3194133.3194163`.

[26] T. Patikirikorala, A. Colman, J. Han, and L. Wang. A multi-model framework to implement self-managing control systems for qos management. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing*

Systems, SEAMS '11, page 218–227, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305754. doi: 10.1145/1988008.1988040. URL https://doi.org/10.1145/1988008.1988040.

[27] Prometheus Metrics Scraper. https://github.com/jmazzitelli/prometheus-scraper, 2018. Accessed: 2022-08-14.

[28] F. Quin and D. Weyns. Seabyte: A self-adaptive micro-service system artifact for automating a/b testing. In *2022 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 77–83, 2022. doi: 10.1145/3524844.3528081. URL https://doi.org/10.1145/3524844.3528081.

[29] Resilience4j Circuit Breaker. https://resilience4j.readme.io/docs/circuitbreaker, 2022. Accessed: 2022-09-11.

[30] C. Richardson. Pattern: Microservice architecture, 2022. URL https://microservices.io/patterns/microservices.html.

[31] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM transactions on autonomous and adaptive systems (TAAS)*, 4(2):1–42, 2009.

[32] H. Samin, L. H. G. Paucar, N. Bencomo, C. M. C. Hurtado, and E. M. Fredericks. Rdmsim: An exemplar for evaluation and comparison of decision-making techniques for self-adaptation. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 238–244, 2021. doi: 10.1109/SEAMS51251.2021.00039. URL https://doi.org/10.1109/SEAMS51251.2021.00039.

[33] Spring Boot. https://spring.io/projects/spring-boot, 2022. Accessed: 2022-10-20.

[34] Spring Boot Actuator. https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/, 2022. Accessed: 2022-08-03.

[35] Spring Cloud. https://spring.io/projects/spring-cloud, 2022. Accessed: 2022-10-20.

[36] Spring Cloud Config. https://docs.spring.io/spring-cloud-config/docs/current/reference/html, 2022. Accessed: 2022-10-24.

[37] Spring Cloud Gateway. https://spring.io/projects/spring-cloud-gateway, 2022. Accessed: 2022-09-01.

[38] Spring Data JPA. https://spring.io/projects/spring-data-jpa, 2022. Accessed: 2022-09-02.

[39] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone. Beyond the rainbow: Self-adaptive failure avoidance in configurable systems. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 377–388, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635915. URL https://doi.org/10.1145/2635868.2635915.

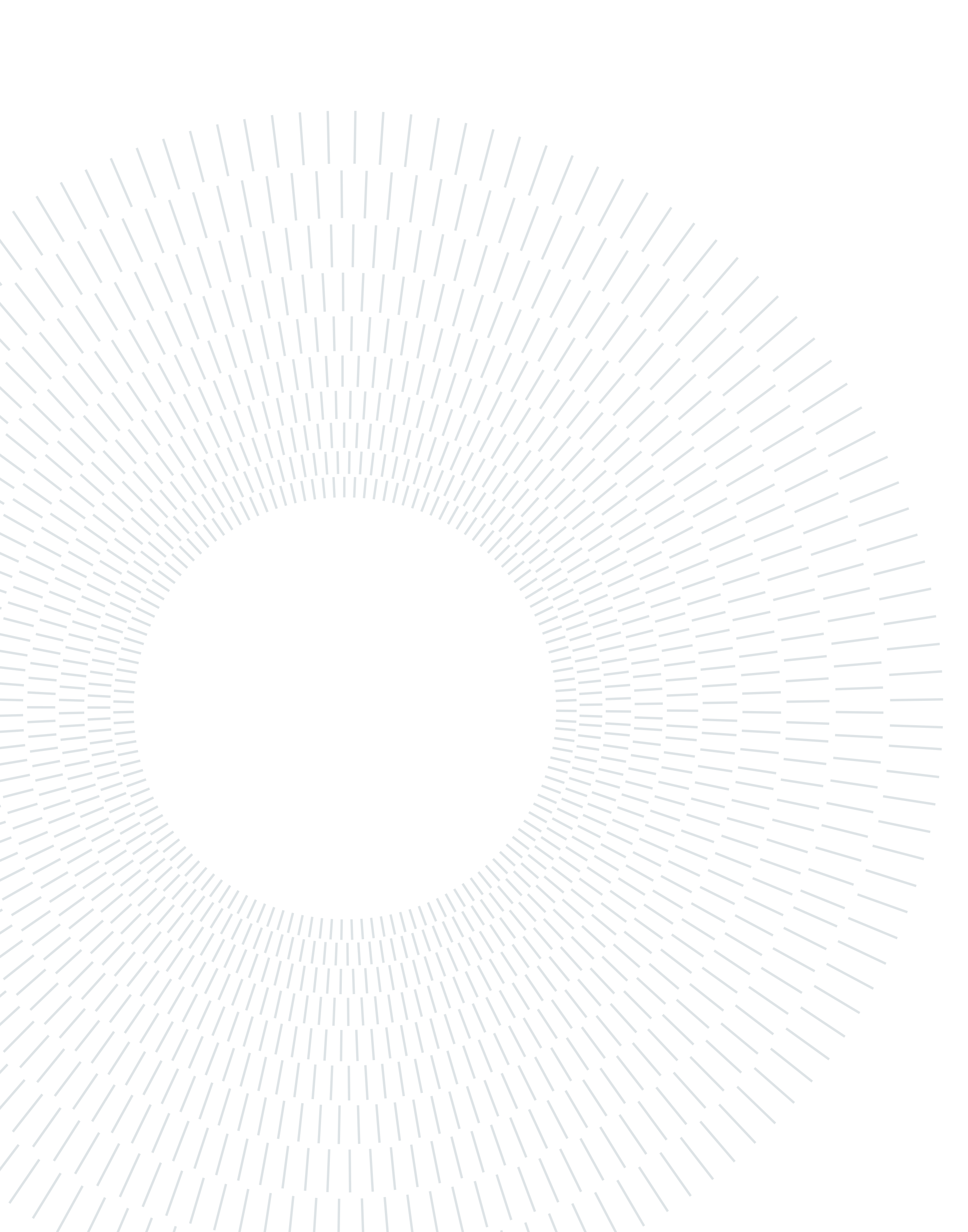
[40] G. Tamura, N. M. Villegas, H. A. Muller, L. Duchien, and L. Seinturier. Improving context-awareness in self-adaptation using the dynamico reference model. In *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 153–162, 2013. doi: 10.1109/SEAMS.2013.6595502. URL https://doi.org/10.1109/SEAMS.2013.6595502.

[41] D. Weyns. Engineering self-adaptive software systems – an organized tour. In *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 1–2, 2018. doi: 10.1109/FAS-W.2018.00012. URL https://doi.org/10.1109/FAS-W.2018.00012.

[42] D. Weyns and R. Calinescu. Tele assistance: A self-adaptive service-based system exemplar. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 88–92, 2015. doi: 10.1109/SEAMS.2015.27. URL https://doi.org/10.1109/SEAMS.2015.27.

[43] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. *On Patterns for Decentralized Control in Self-Adaptive Systems*, pages 76–107. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35813-5. doi: 10.1007/978-3-642-35813-5_4. URL https://doi.org/10.1007/978-3-642-35813-5_4.

[44] T. Wong, M. Wagner, and C. Treude. Self-adaptive systems: A systematic literature review across categories and domains. *Information and Software Technology*, 148:106934, 2022. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2022.106934. URL https://www.sciencedirect.com/science/article/pii/S0950584922000854.



A | Appendix A - Experimental results

The experiments are grouped by scenarios, as follows:

- Scenario S1 - Some QoS indicators are not satisfied
  - E1 - Choice of the experiment duration
  - E2 - Benefits of the adaptation
  - E3 - Impact of the Metrics Window Size
  - E4 - Impact of the Analysis Window Size
  - E5 - Impact of the number of users
- Scenario S2 - Service unavailable
  - E1 - Analysis of the self-healing capabilities
  - E2 - Analysis of the failure tendency detection
  - E3 - Impact of the Monitor Period
- Scenario S3 - Better service implementation available
  - E1 - Analysis of the self-optimization capabilities
- Reusability of the Managing System
  - E1 - Analysis of the Managing System reusability

A.1. Scenario S1 - QoS not satisfied

A.1.1. E1 - Choice of the experiment duration

RESTAURANT-SERVICE

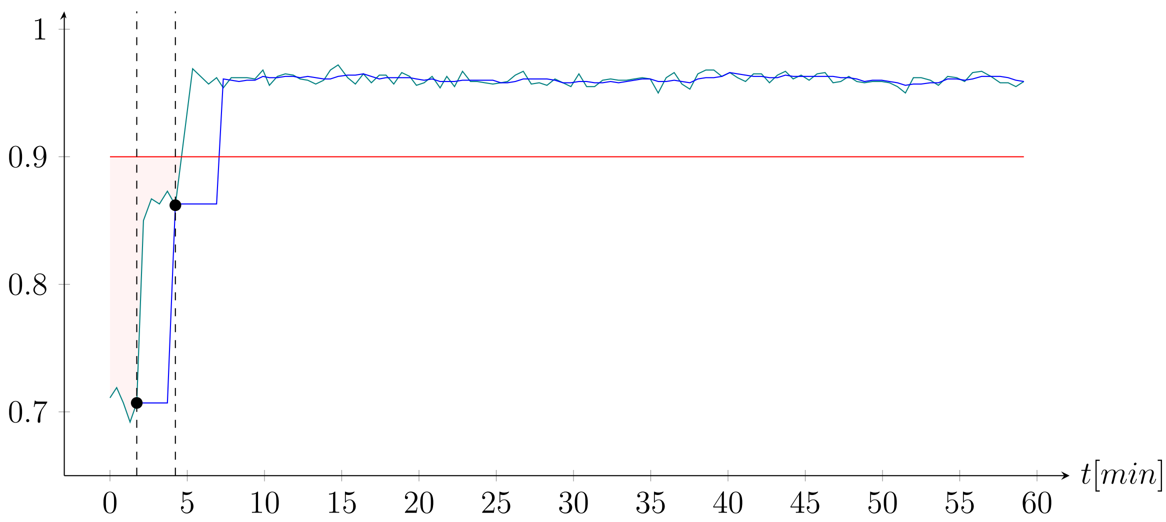


Figure A.1: S1E1 – Restaurant Service availability

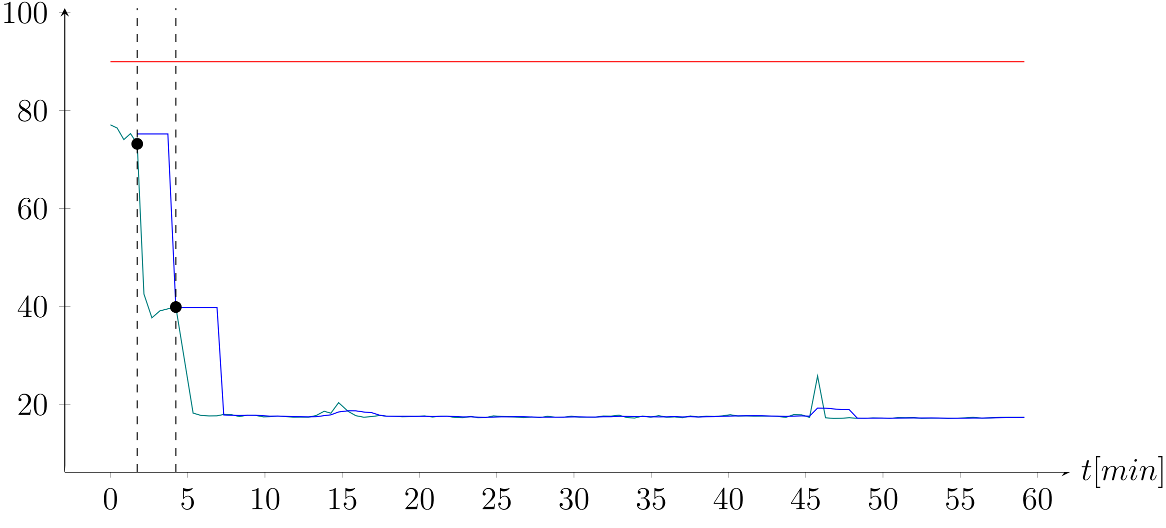


Figure A.2: S1E1 – Restaurant Service average response time

ORDERING-SERVICE

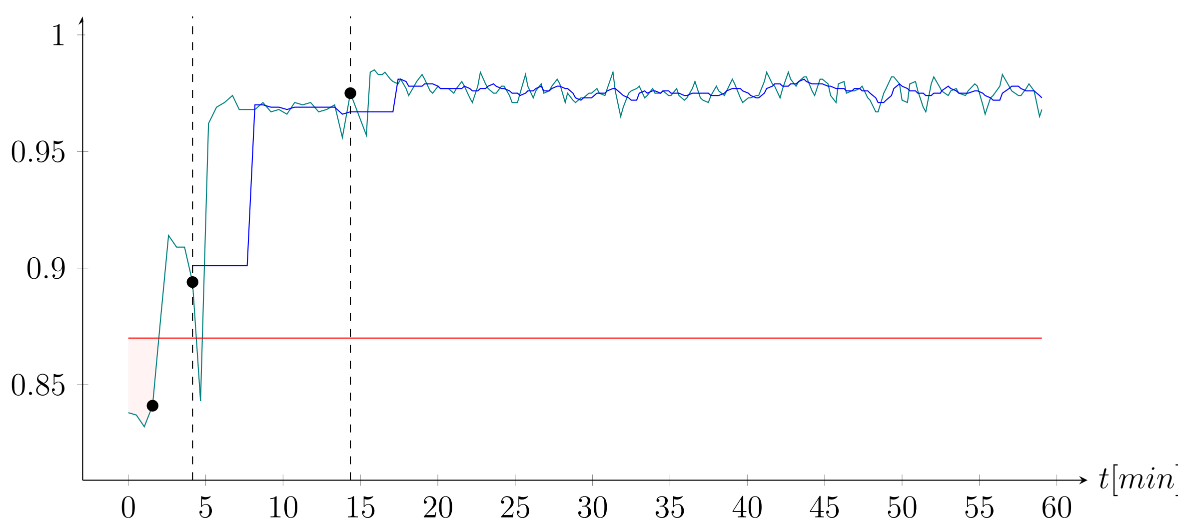


Figure A.3: S1E1 – Ordering Service availability

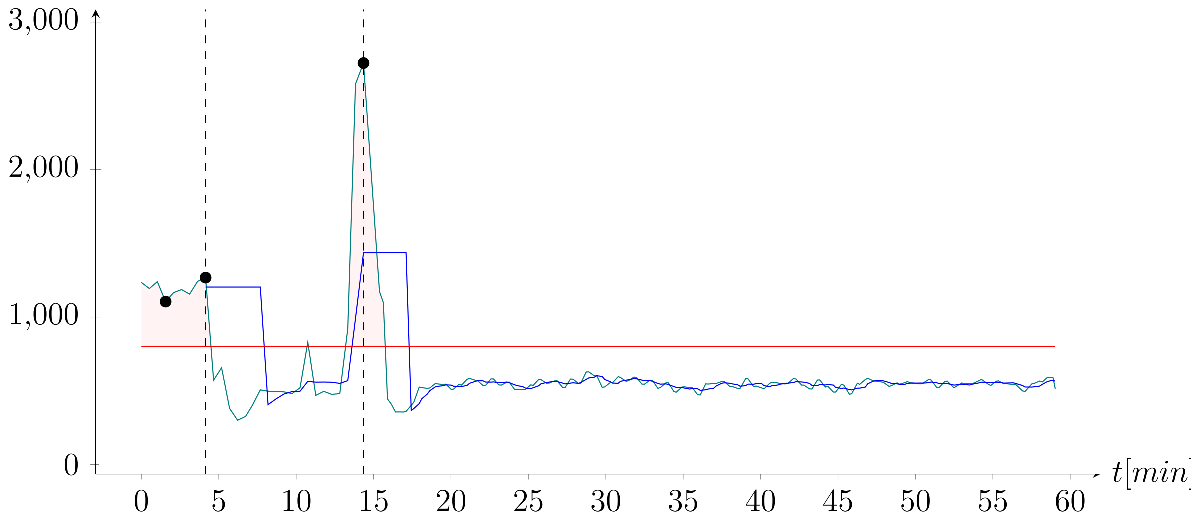


Figure A.4: S1E1 – Ordering Service average response time

PAYMENT-PROXY-SERVICE

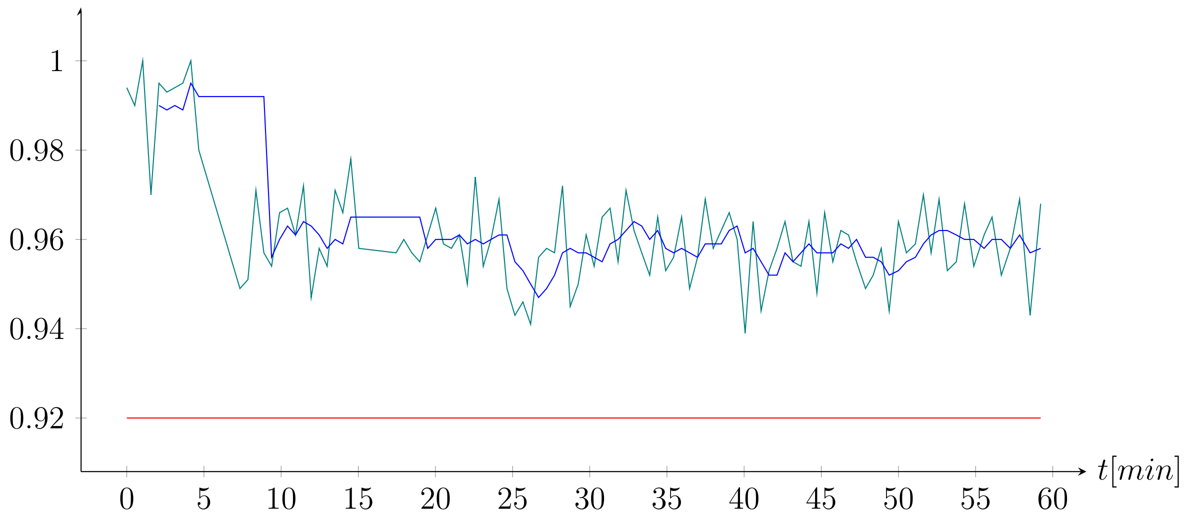


Figure A.5: S1E1 – Payment Proxy Service availability

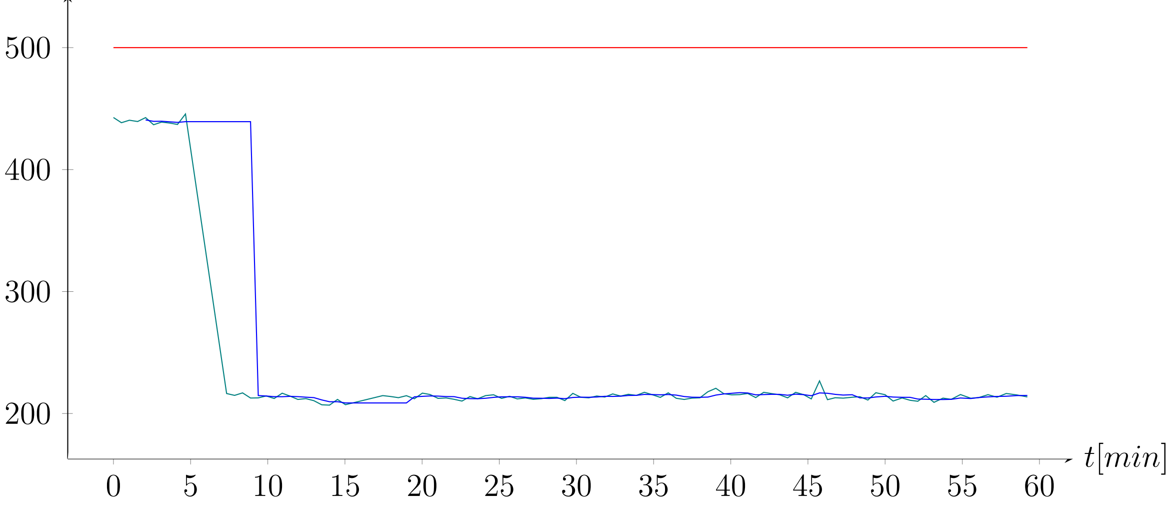


Figure A.6: S1E1 – Payment Proxy Service average response time

DELIVERY-PROXY-SERVICE

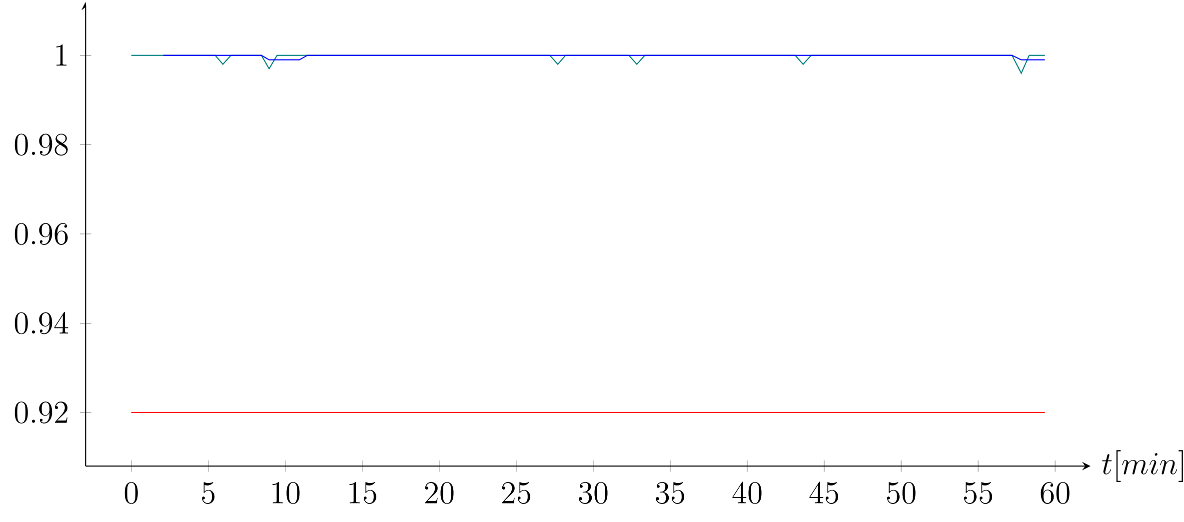


Figure A.7: S1E1 – Delivery Proxy Service availability

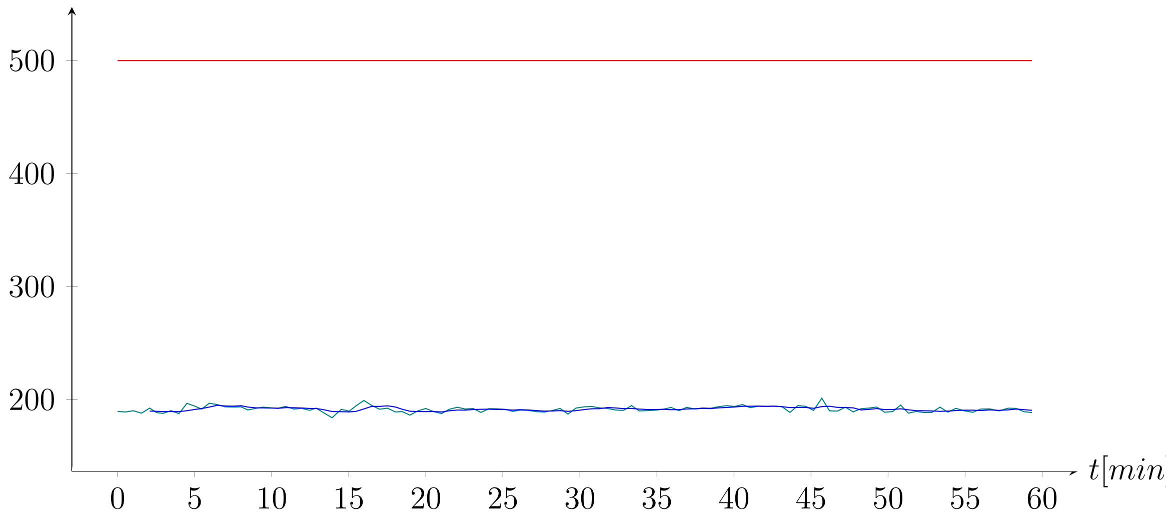


Figure A.8: S1E1 – Delivery Proxy Service average response time

A.1.2. E2 - Benefits of the adaptation

Without adaptation

RESTAURANT-SERVICE

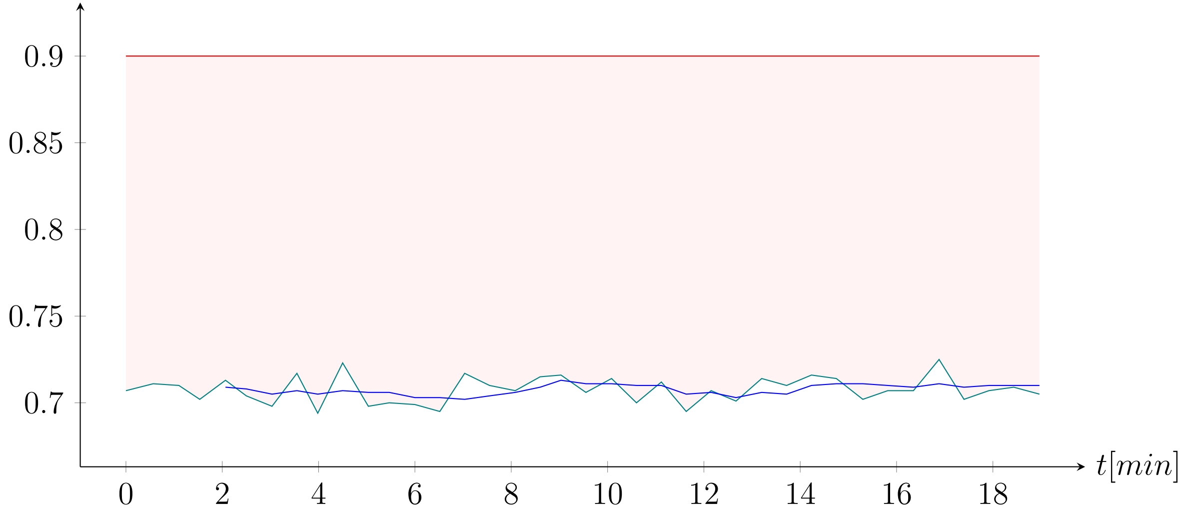


Figure A.9: S1E2 – Restaurant Service availability – without adaptation

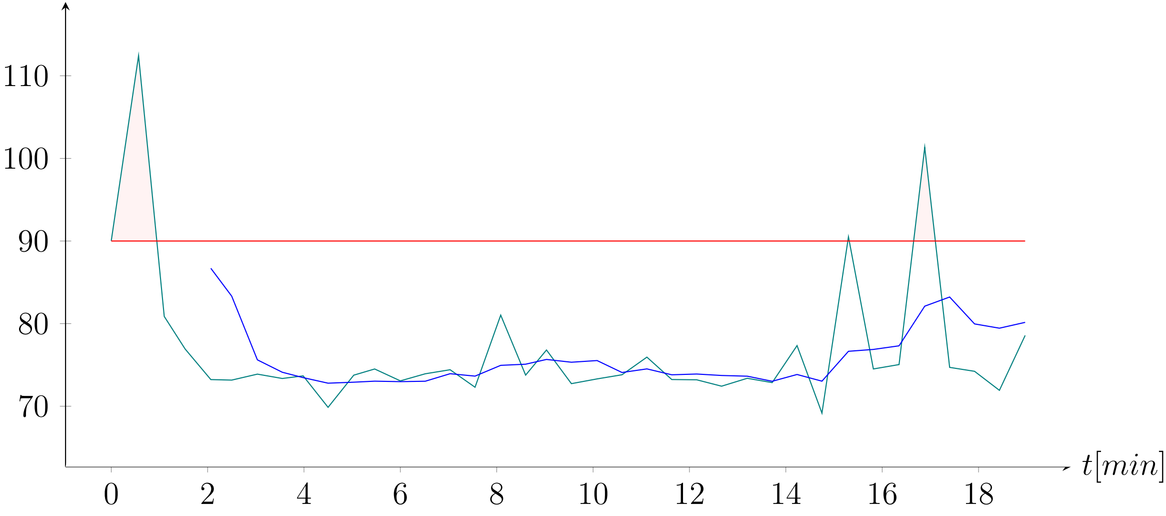


Figure A.10: S1E2 – Restaurant Service average response time – without adaptation

ORDERING-SERVICE

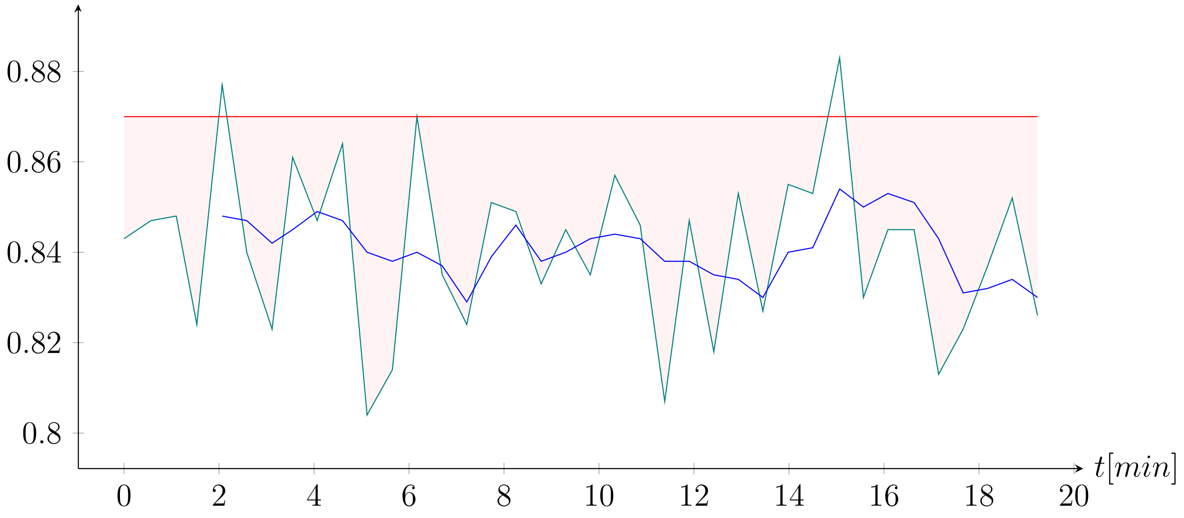


Figure A.11: S1E2 – Ordering Service availability – without adaptation

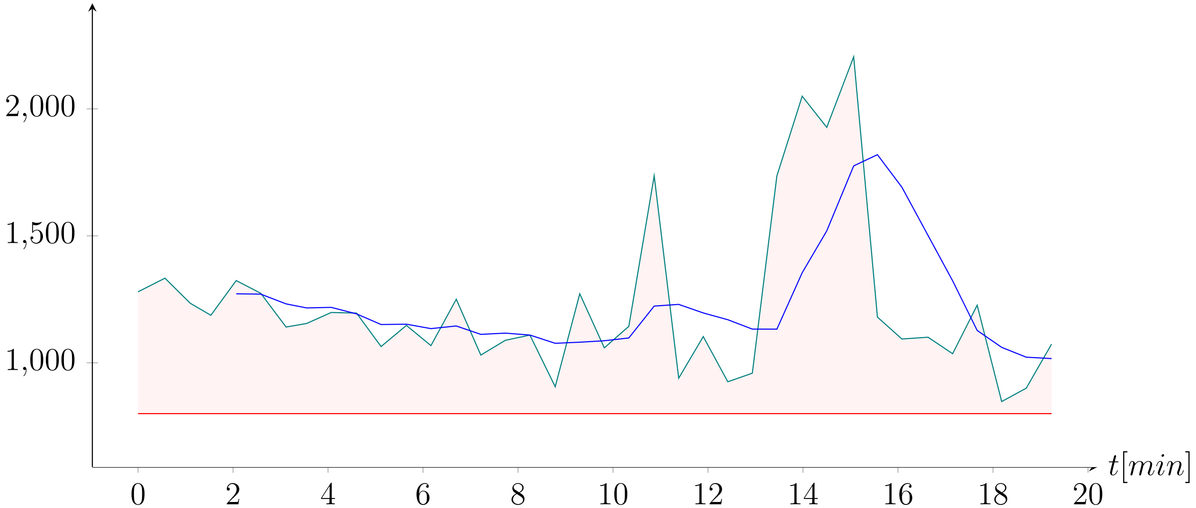


Figure A.12: S1E2 – Ordering Service average response time – without adaptation

PAYMENT-PROXY-SERVICE

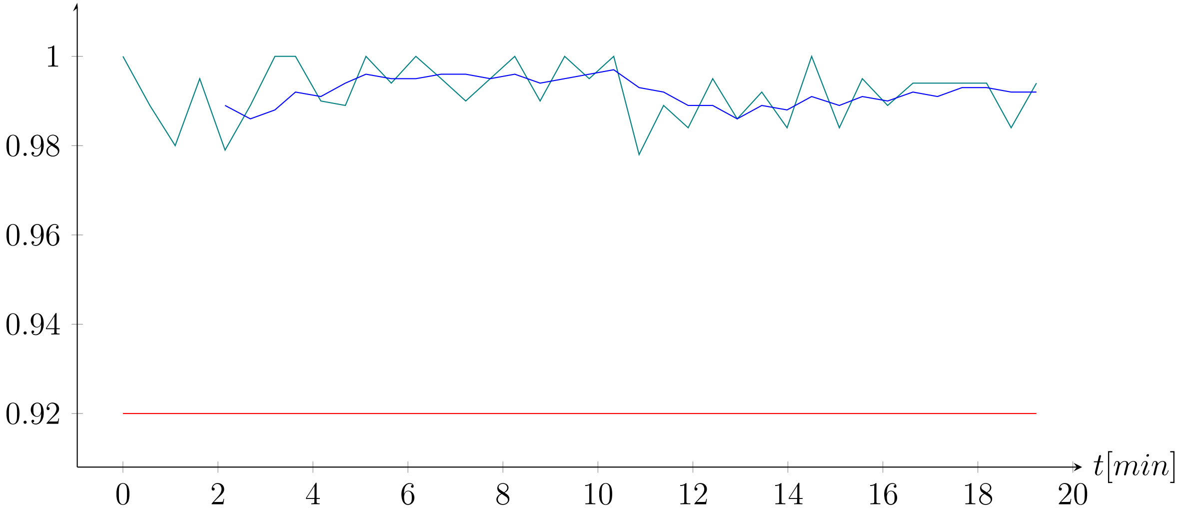


Figure A.13: S1E2 – Payment Proxy Service availability – without adaptation

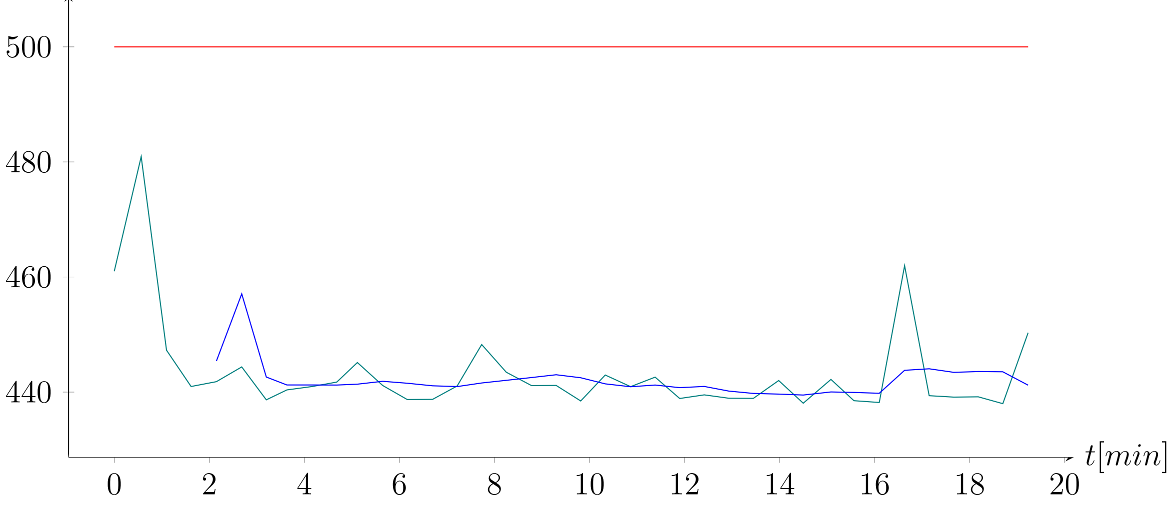


Figure A.14: S1E2 – Payment Proxy Service average response time – without adaptation

DELIVERY-PROXY-SERVICE

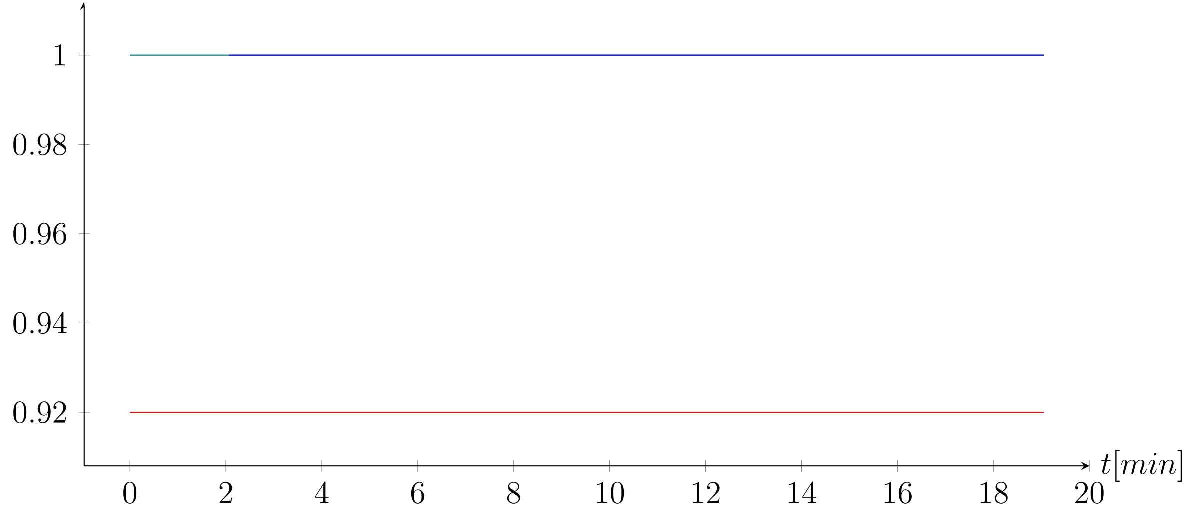


Figure A.15: S1E2 – Delivery Proxy Service availability – without adaptation

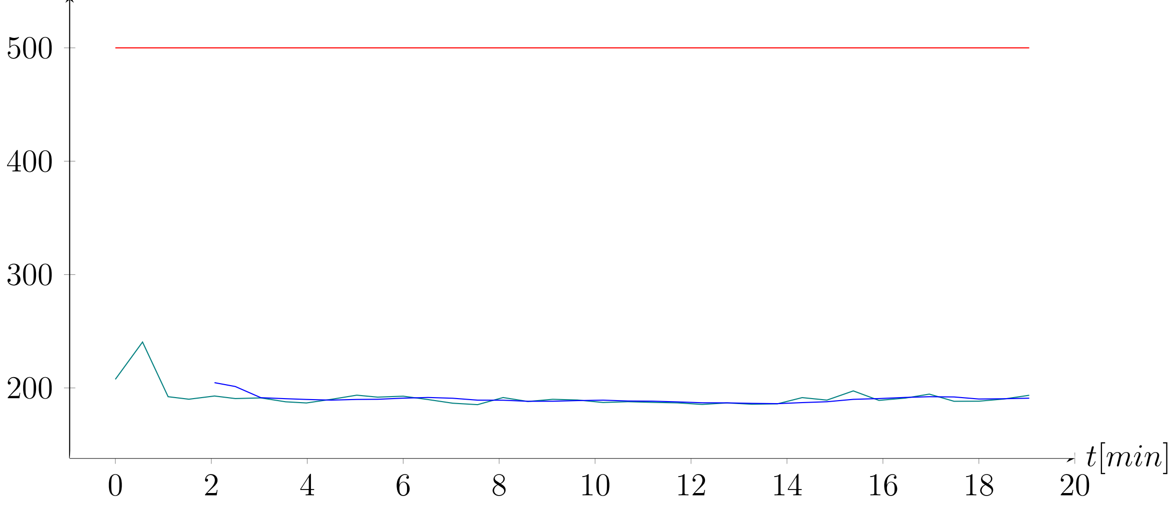


Figure A.16: S1E2 – Delivery Proxy Service average response time – without adaptation

With adaptation

RESTAURANT-SERVICE

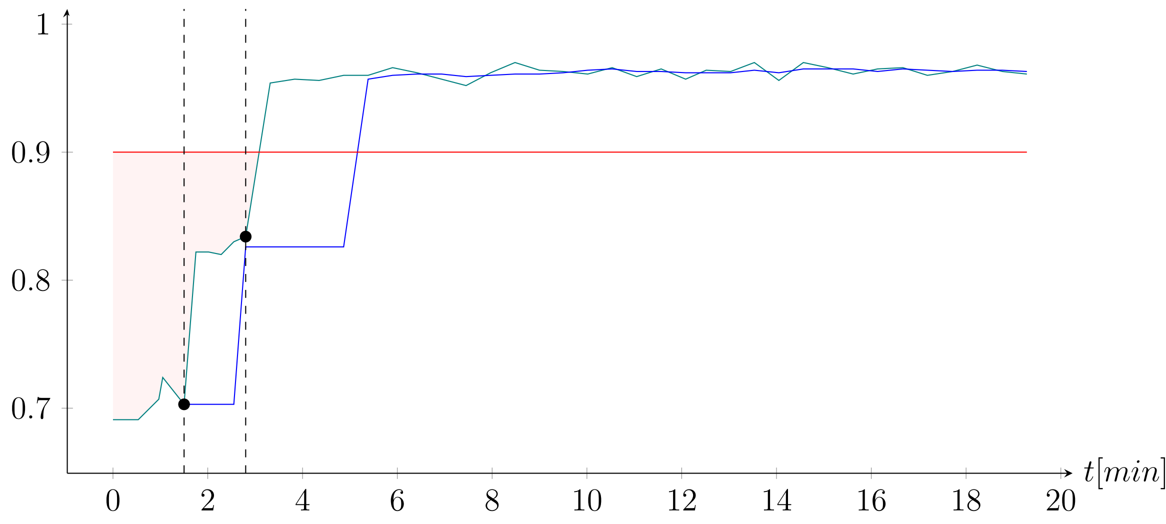


Figure A.17: S1E2 – Restaurant Service availability

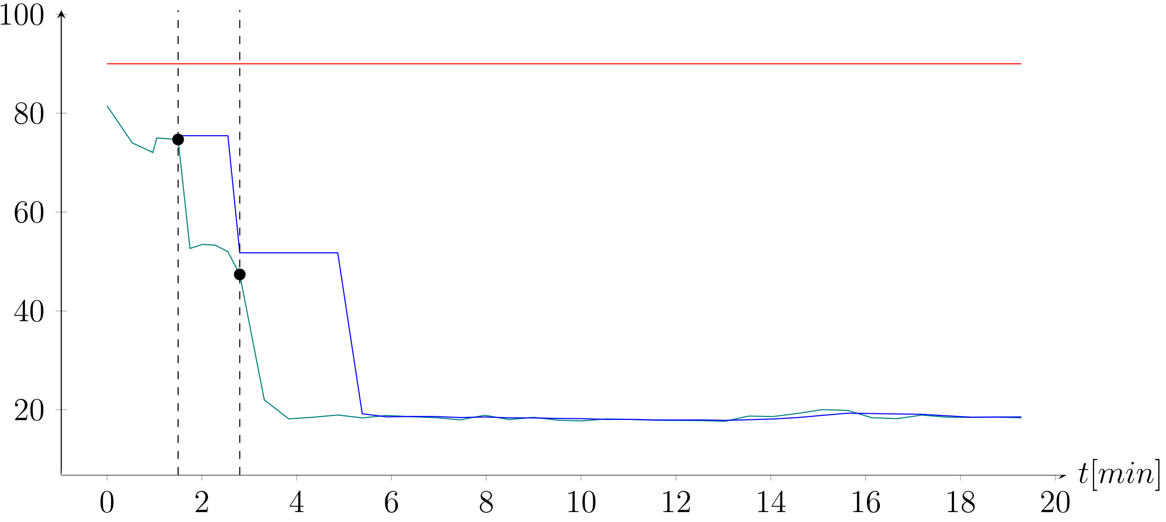


Figure A.18: S1E2 – Restaurant Service average response time

ORDERING-SERVICE

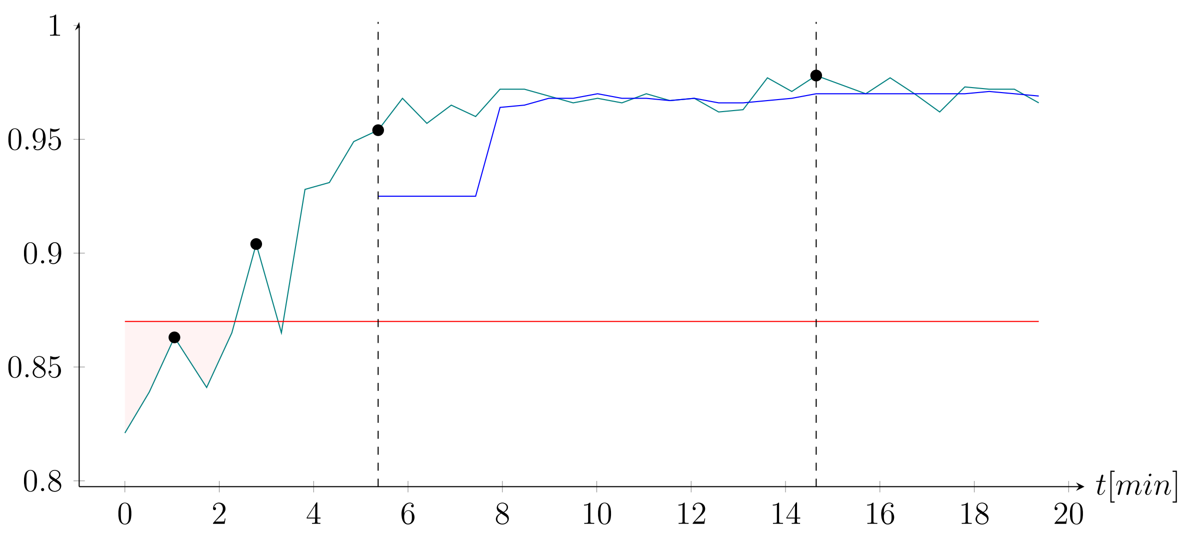


Figure A.19: S1E2 – Ordering Service availability

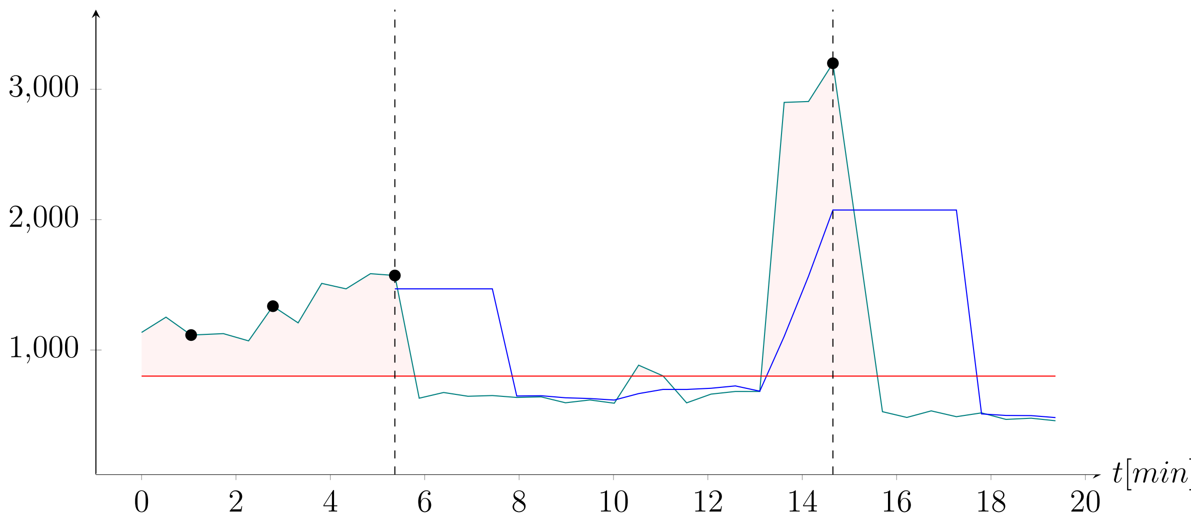


Figure A.20: S1E2 – Ordering Service average response time

PAYMENT-PROXY-SERVICE

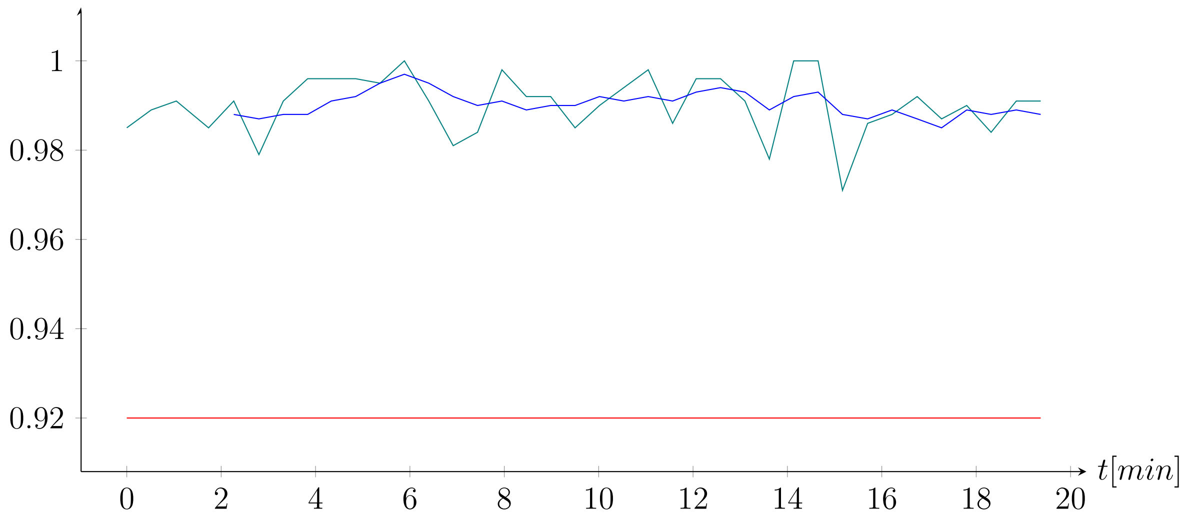


Figure A.21: S1E2 – Payment Proxy Service availability

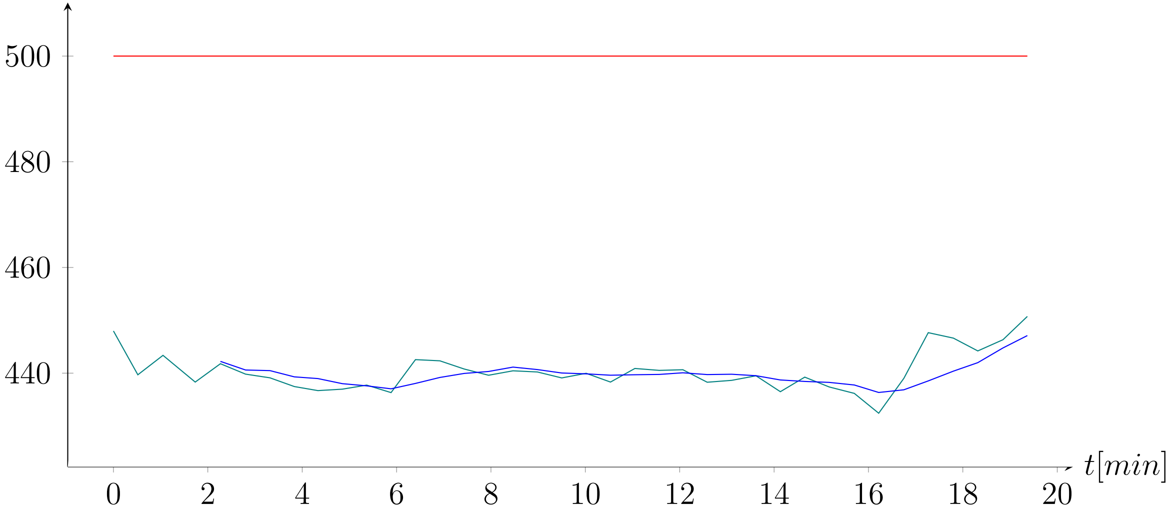


Figure A.22: S1E2 – Payment Proxy Service average response time

DELIVERY-PROXY-SERVICE

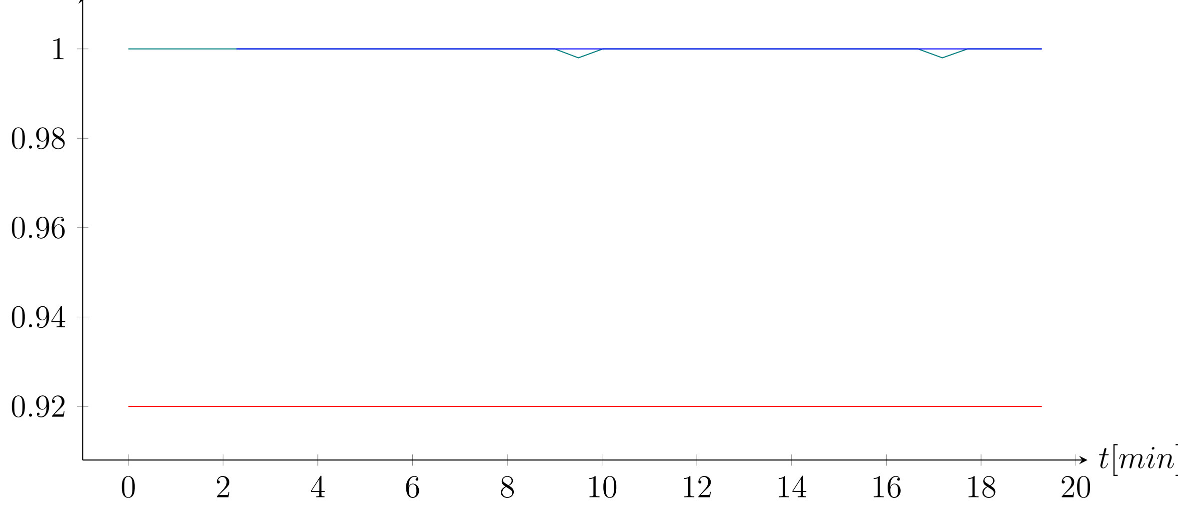


Figure A.23: S1E2 – Delivery Proxy Service availability

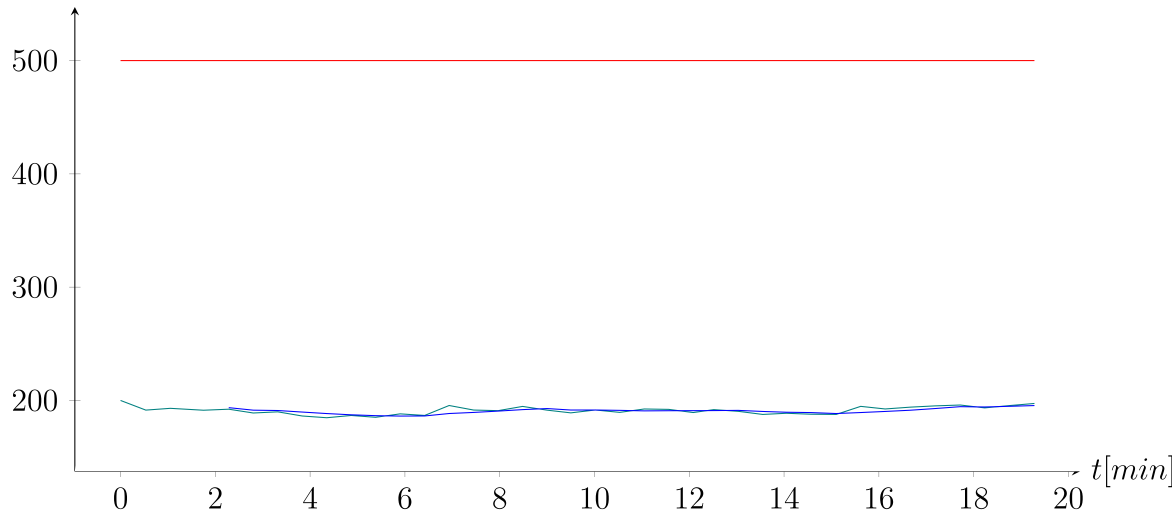


Figure A.24: S1E2 – Delivery Proxy Service average response time

A.1.3. E3 - Impact of the Metrics Window Size

RESTAURANT-SERVICE

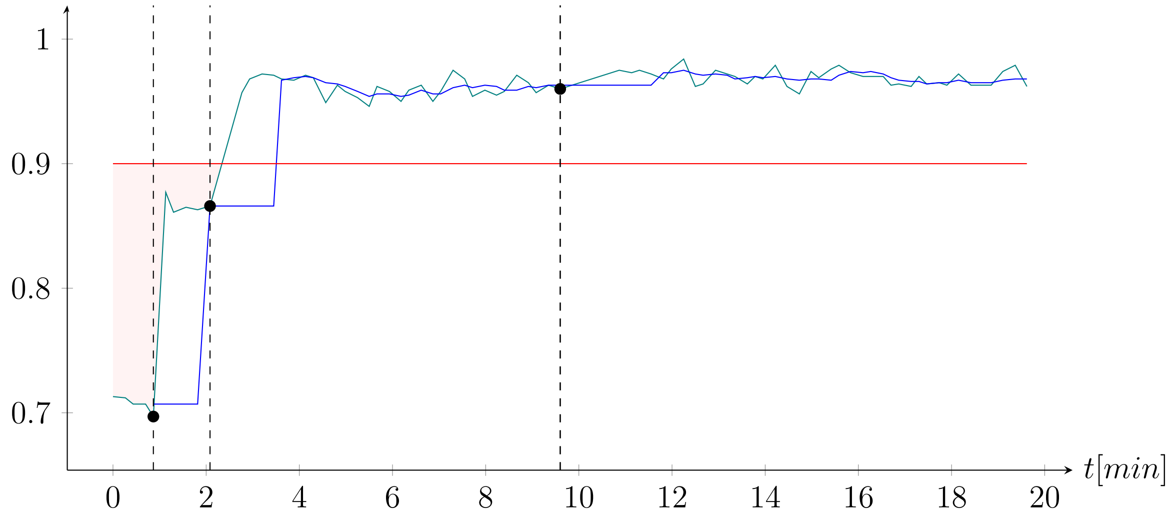


Figure A.25: S1E3 – Restaurant Service availability

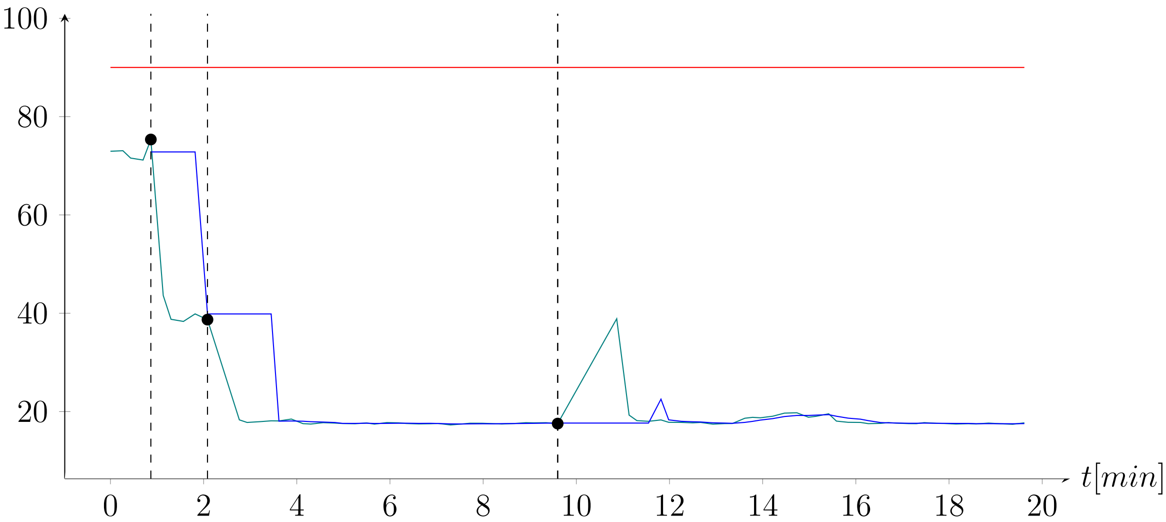


Figure A.26: S1E3 – Restaurant Service average response time

ORDERING-SERVICE

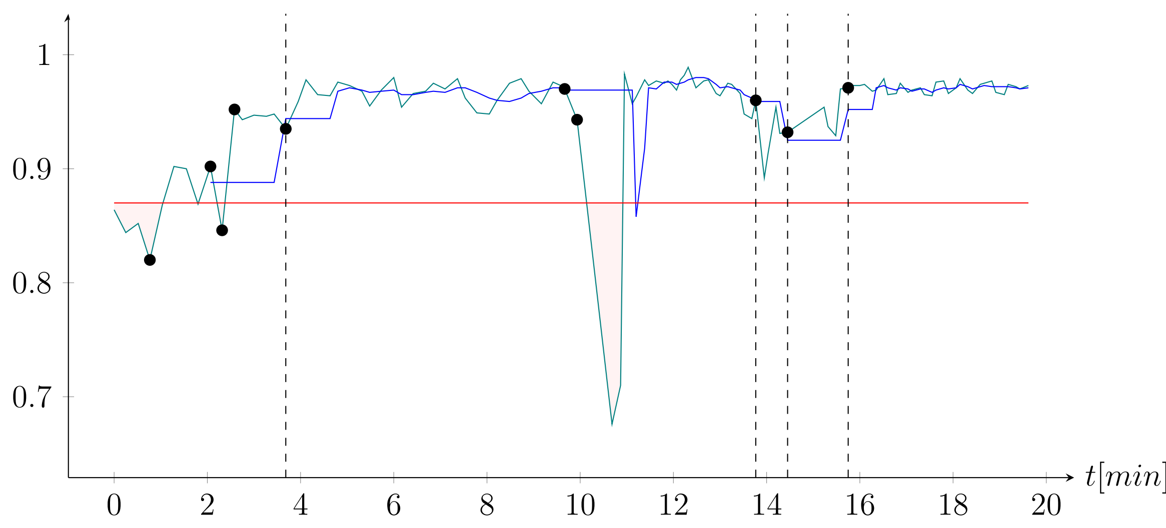


Figure A.27: S1E3 – Ordering Service availability

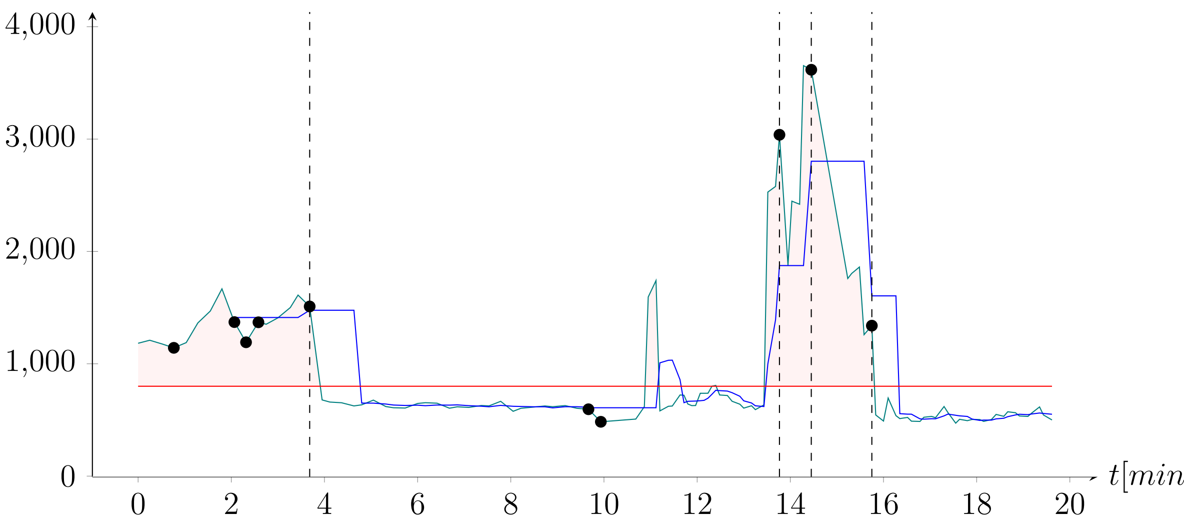


Figure A.28: S1E3 – Ordering Service average response time

PAYMENT-PROXY-SERVICE

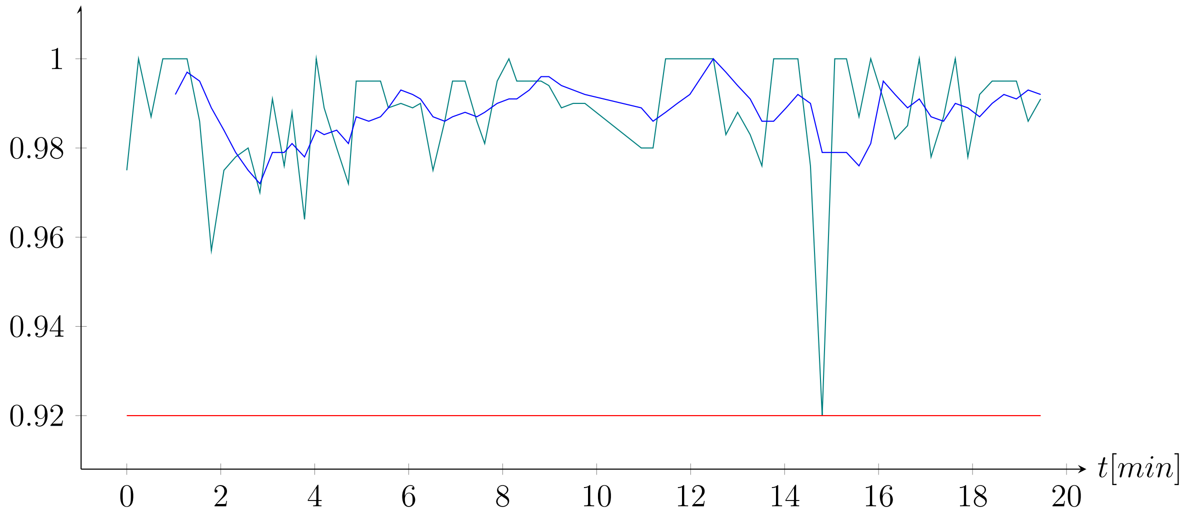


Figure A.29: S1E3 – Payment Proxy Service availability

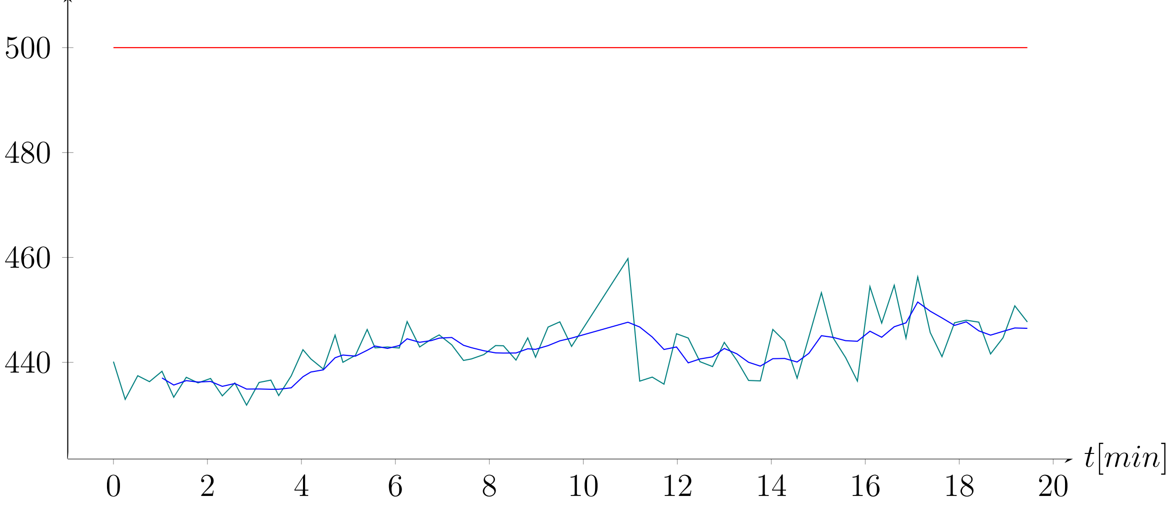


Figure A.30: S1E3 – Payment Proxy Service average response time

DELIVERY-PROXY-SERVICE

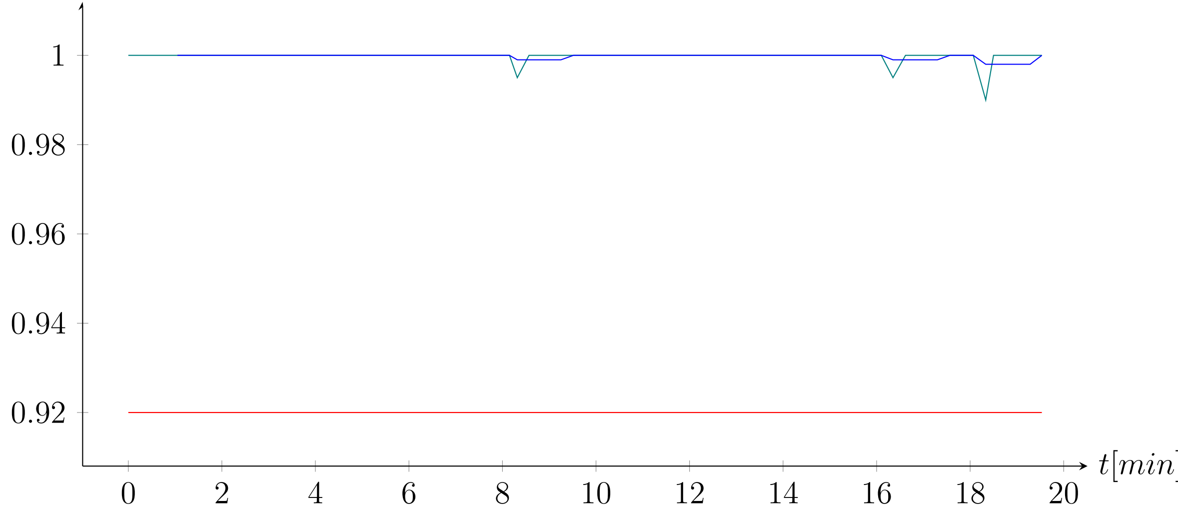


Figure A.31: S1E3 – Delivery Proxy Service availability

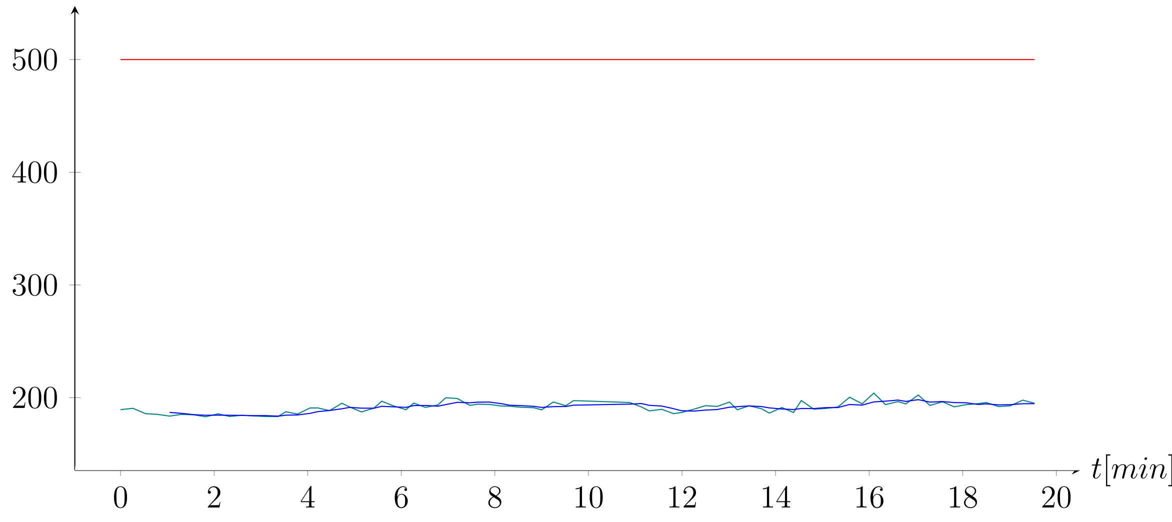


Figure A.32: S1E3 – Delivery Proxy Service average response time

A.1.4. E4 - Impact of the Analysis Window Size

RESTAURANT-SERVICE

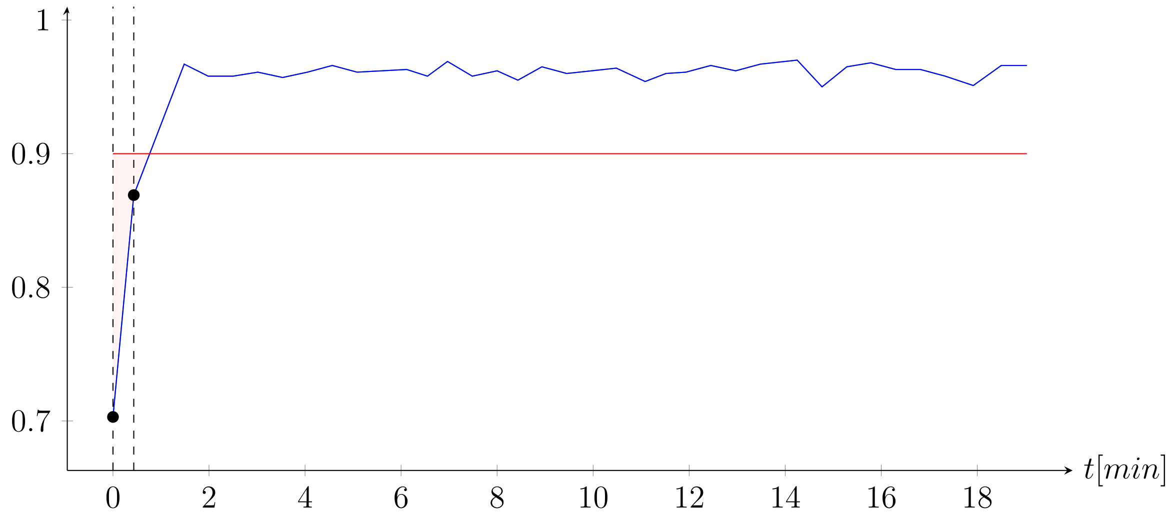


Figure A.33: S1E4 – Restaurant Service availability

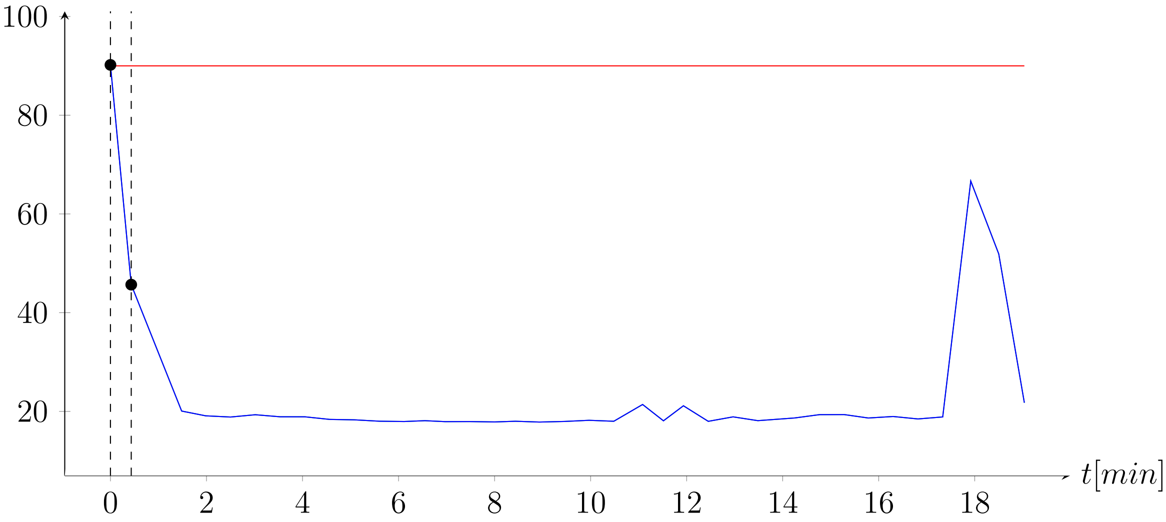


Figure A.34: S1E4 – Restaurant Service average response time

ORDERING-SERVICE

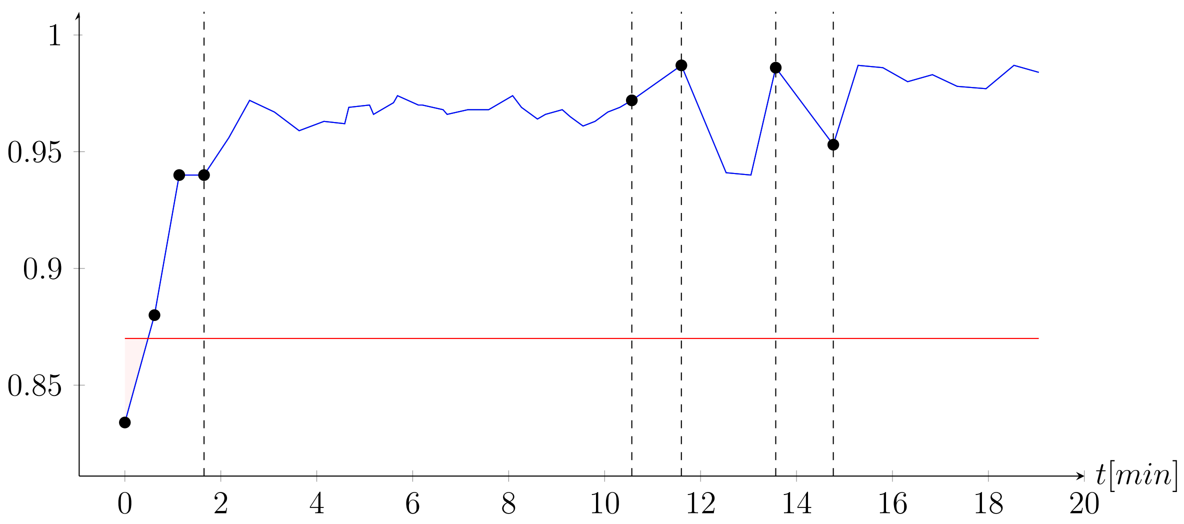


Figure A.35: S1E4 – Ordering Service availability

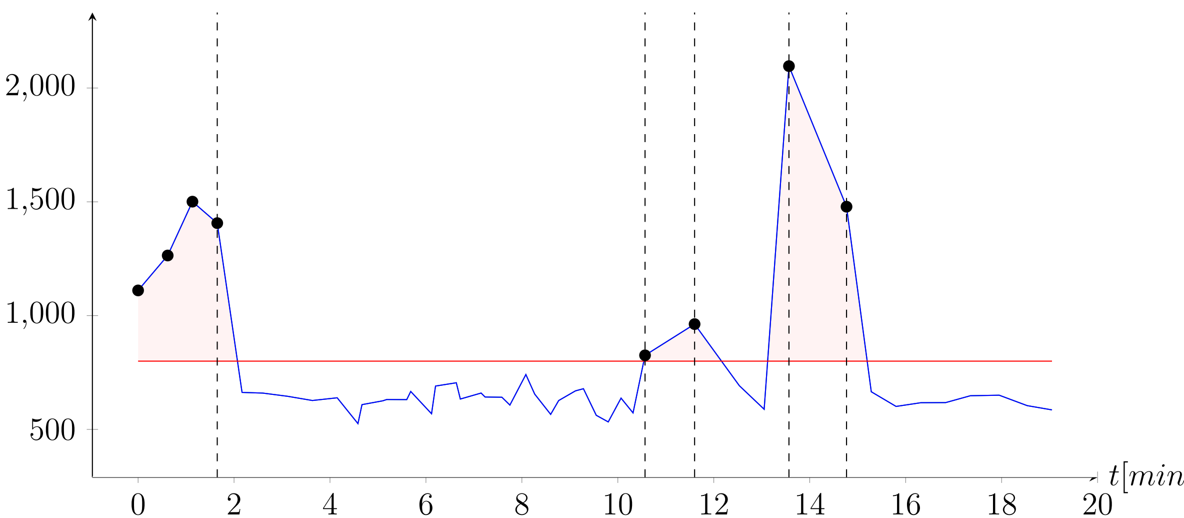


Figure A.36: S1E4 – Ordering Service average response time

PAYMENT-PROXY-SERVICE

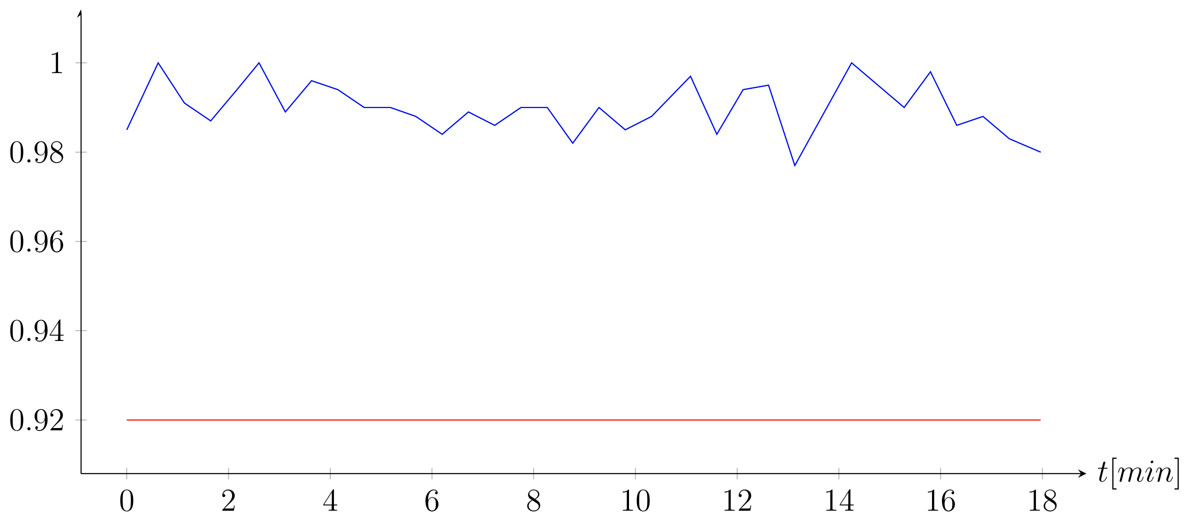


Figure A.37: S1E4 – Payment Proxy Service availability

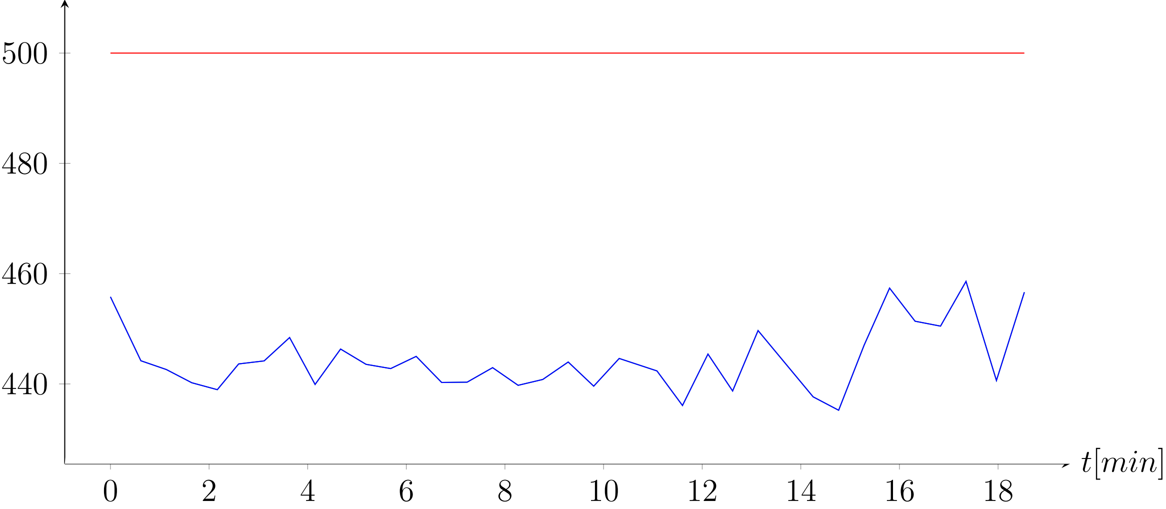


Figure A.38: S1E4 – Payment Proxy Service average response time

DELIVERY-PROXY-SERVICE

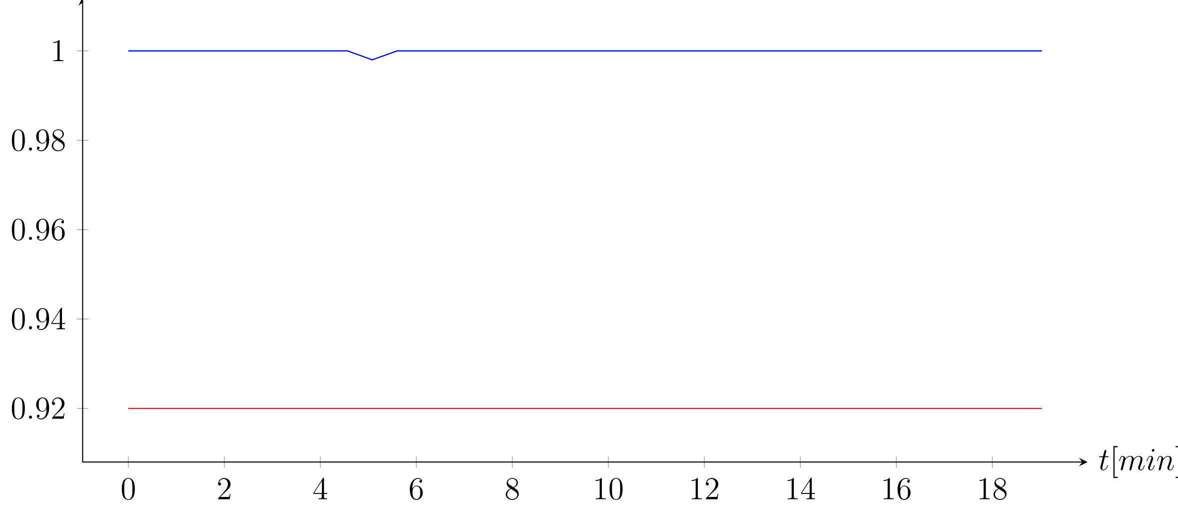


Figure A.39: S1E4 – Delivery Proxy Service availability

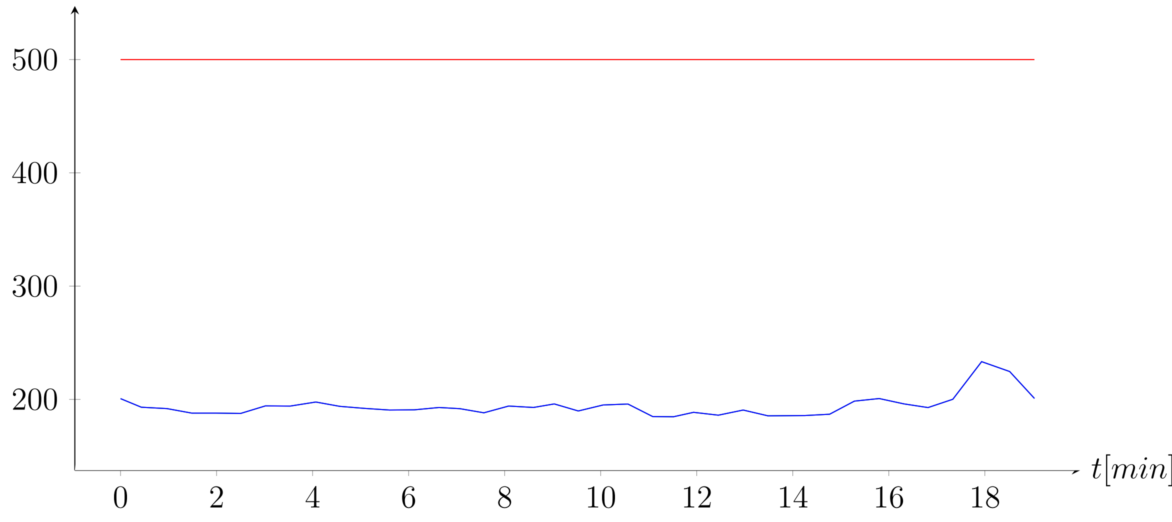


Figure A.40: S1E4 – Delivery Proxy Service average response time

A.1.5. E5 - Impact of the number of users

RESTAURANT-SERVICE

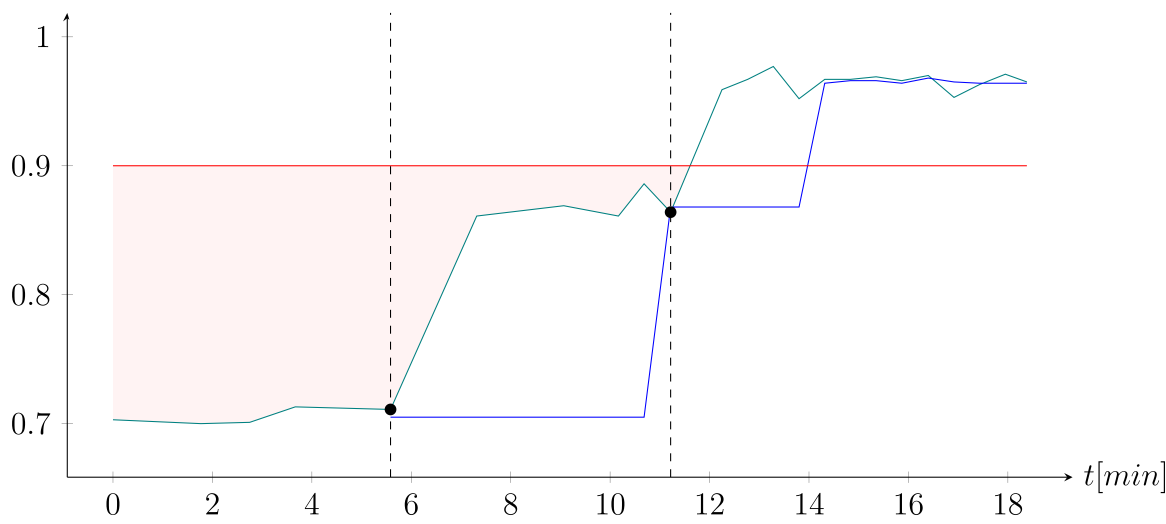


Figure A.41: S1E5 – Restaurant Service availability

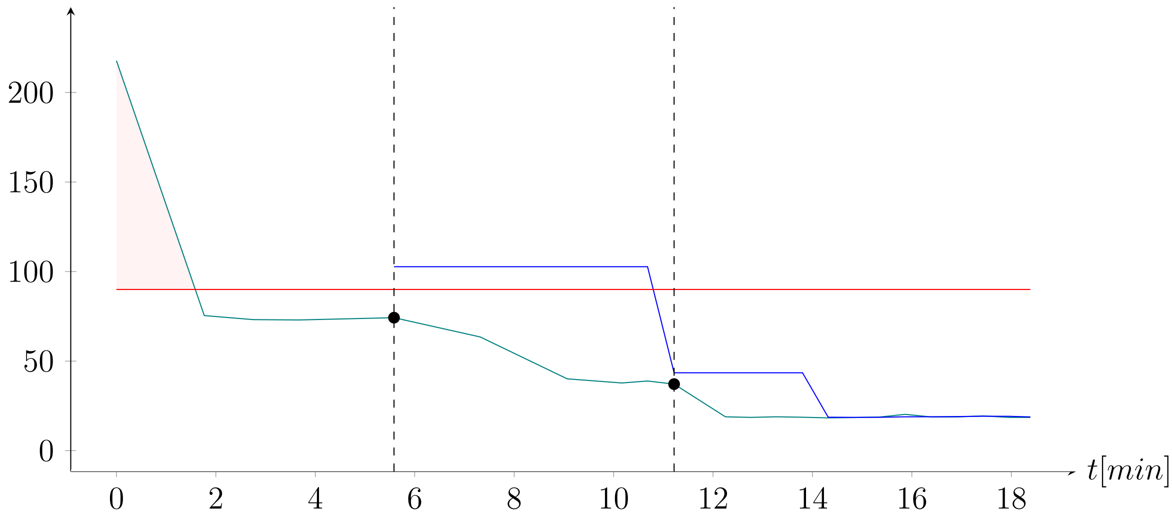


Figure A.42: S1E5 – Restaurant Service average response time

ORDERING-SERVICE

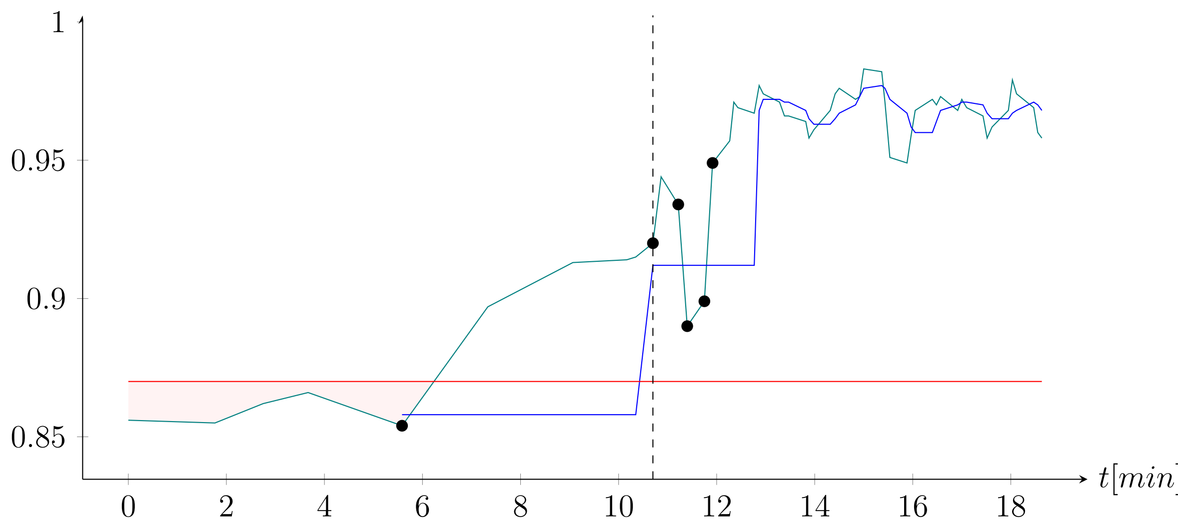


Figure A.43: S1E5 – Ordering Service availability

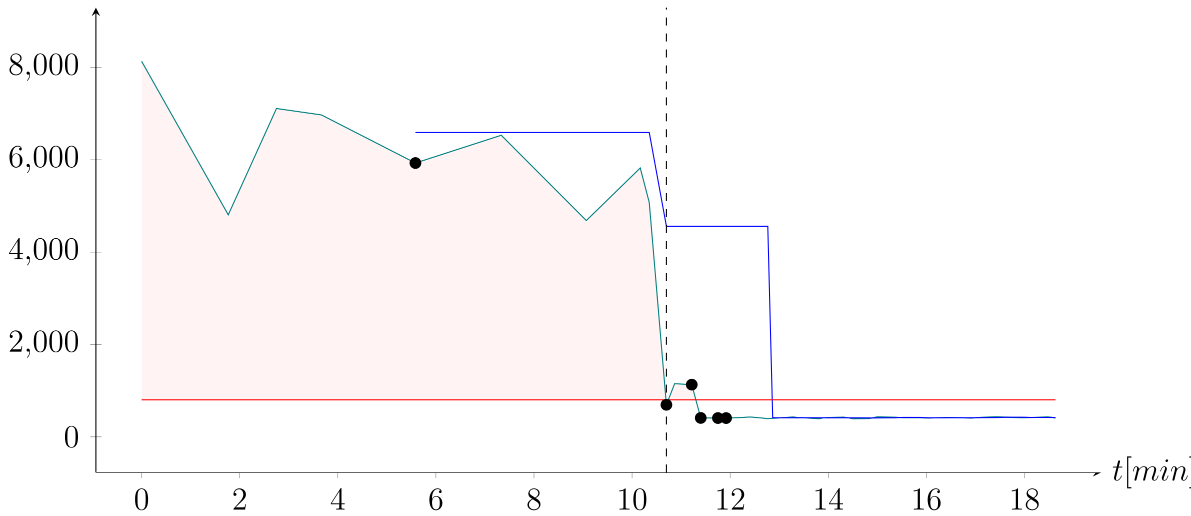


Figure A.44: S1E5 – Ordering Service average response time

PAYMENT-PROXY-SERVICE

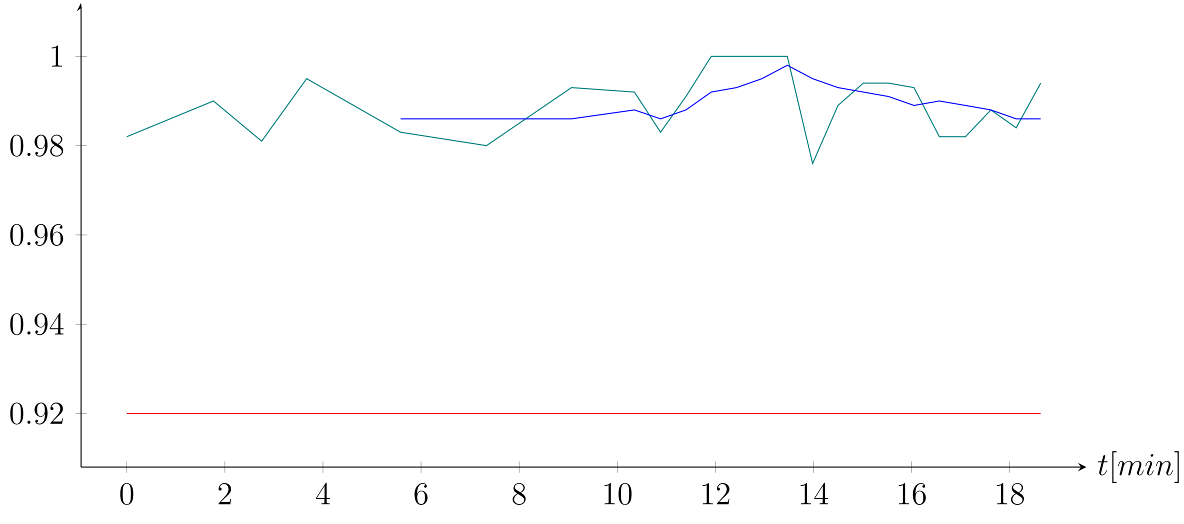


Figure A.45: S1E5 – Payment Proxy Service availability

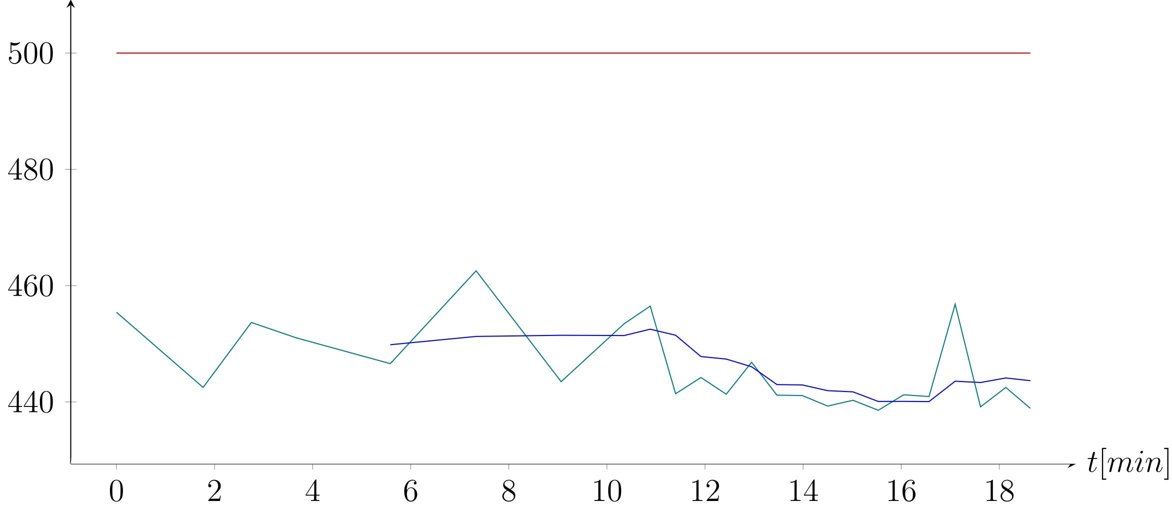


Figure A.46: S1E5 – Payment Proxy Service average response time

DELIVERY-PROXY-SERVICE

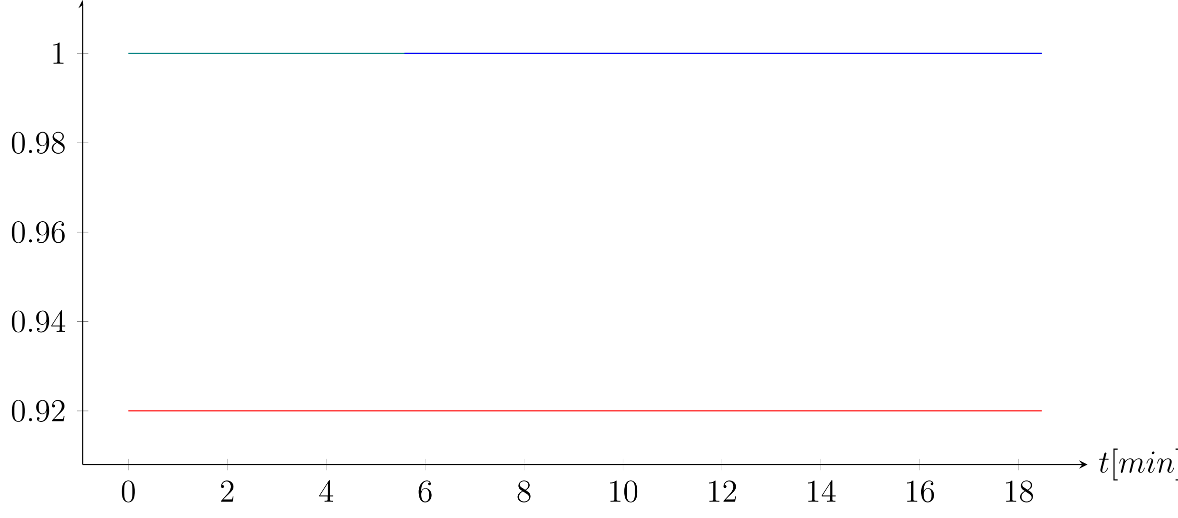


Figure A.47: S1E5 – Delivery Proxy Service availability

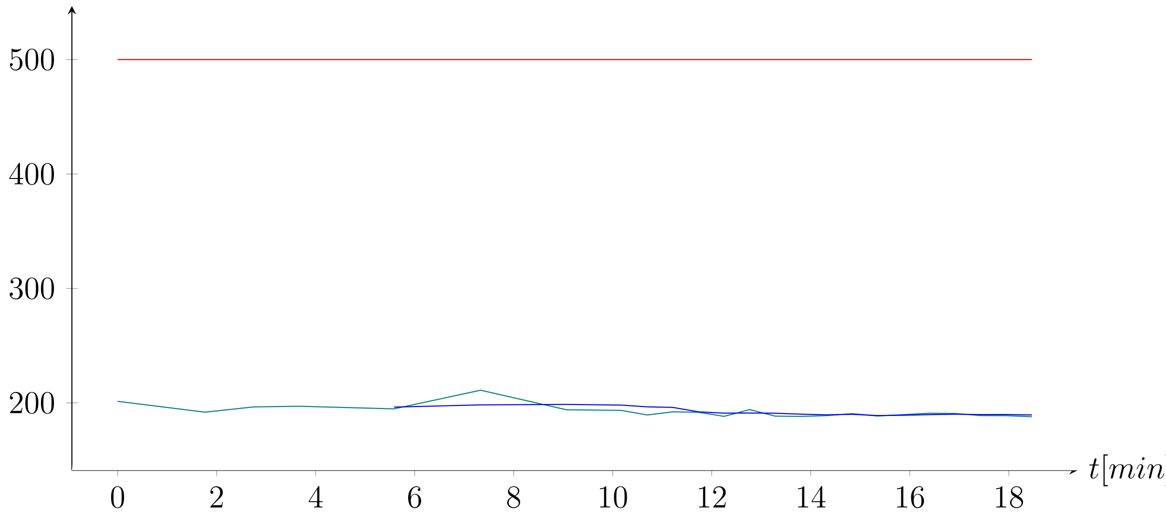


Figure A.48: S1E5 – Delivery Proxy Service average response time

A.2. Scenario S2 - Service unavailable

A.2.1. E1 - Analysis of the self-healing capabilities

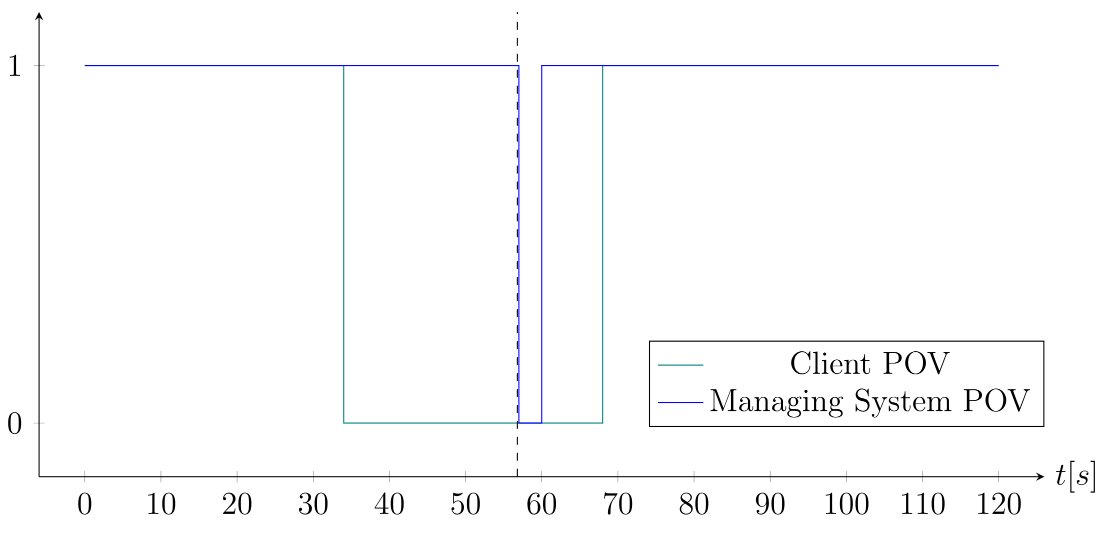


Figure A.49: S2E1 – Payment Proxy Service – Number of instances

A.2.2. E2 - Analysis of the failure tendency detection

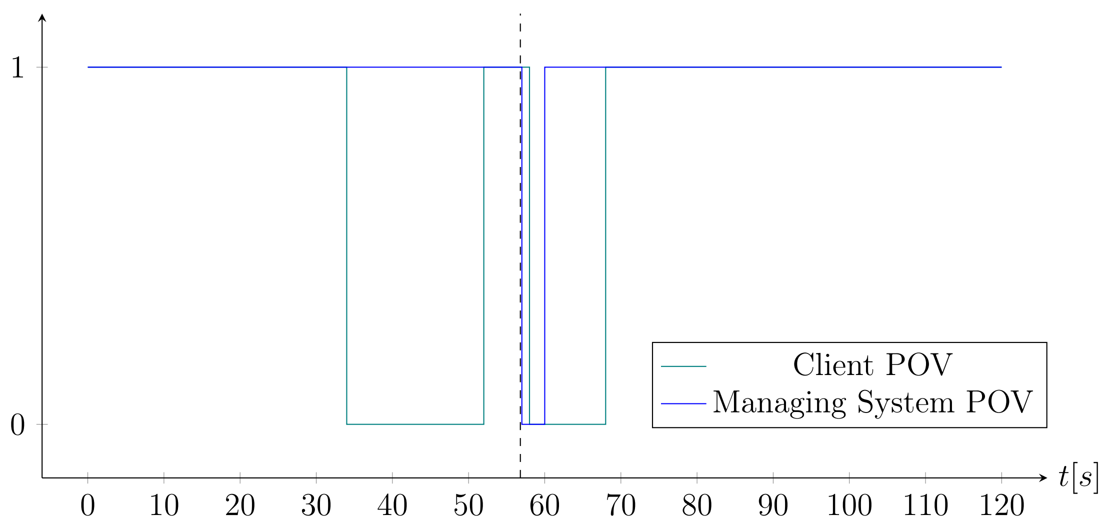


Figure A.50: S2E2 – Payment Proxy Service – Number of instances

A.2.3. E3 - Impact of the Monitor Period

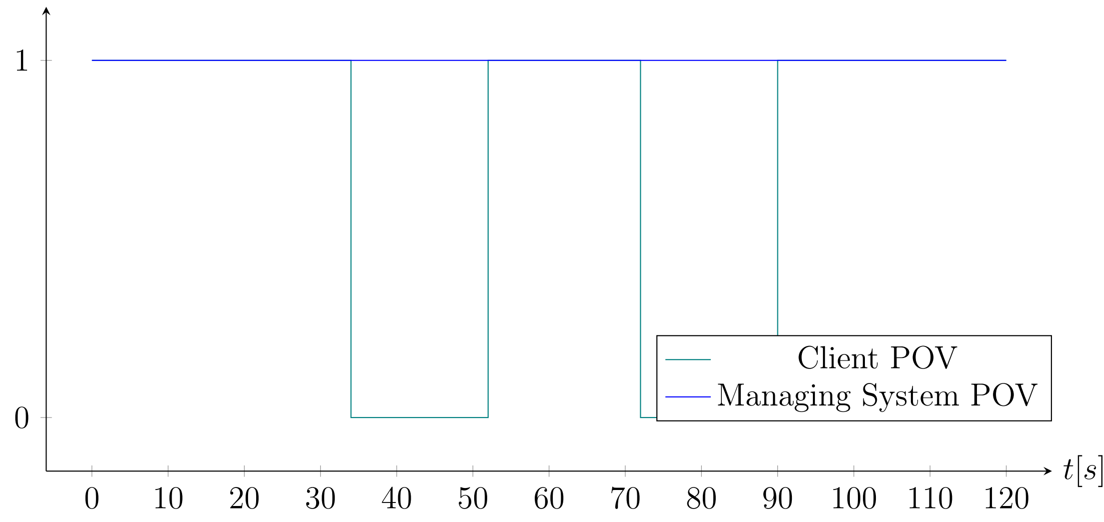


Figure A.51: S2E3 – Payment Proxy Service – Number of instances

A.3. Scenario S3 - Better implementation available

A.3.1. E1 - Analysis of the self-optimization capabilities

RESTAURANT-SERVICE

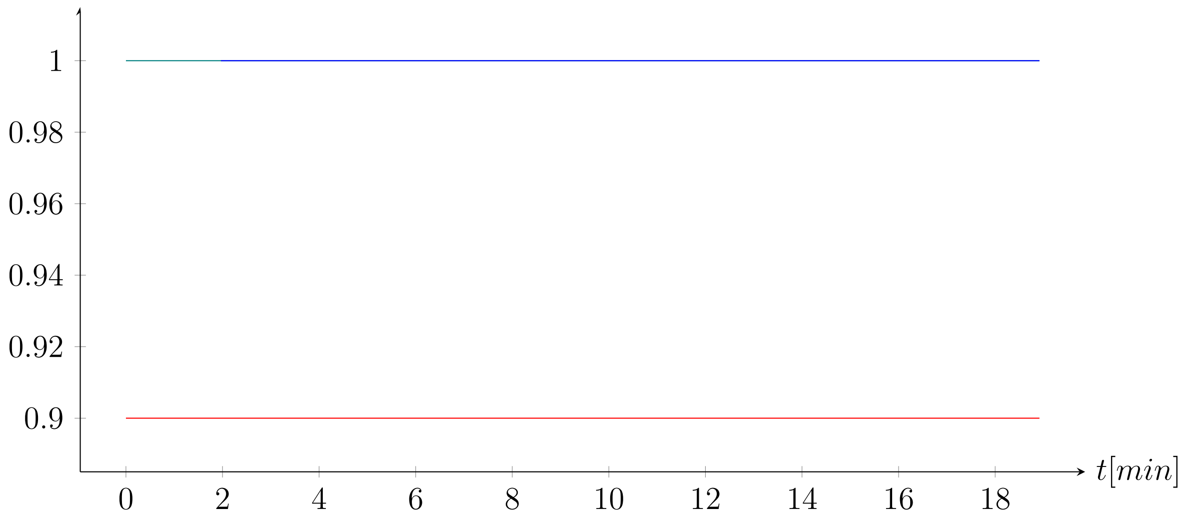


Figure A.52: S3E1 – Restaurant Service availability

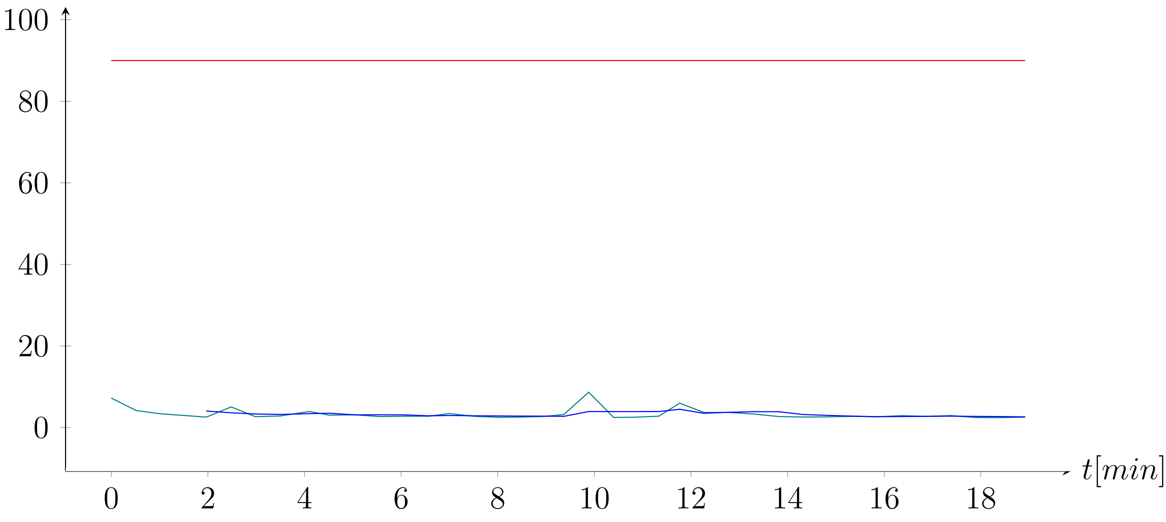


Figure A.53: S3E1 – Restaurant Service average response time

ORDERING-SERVICE

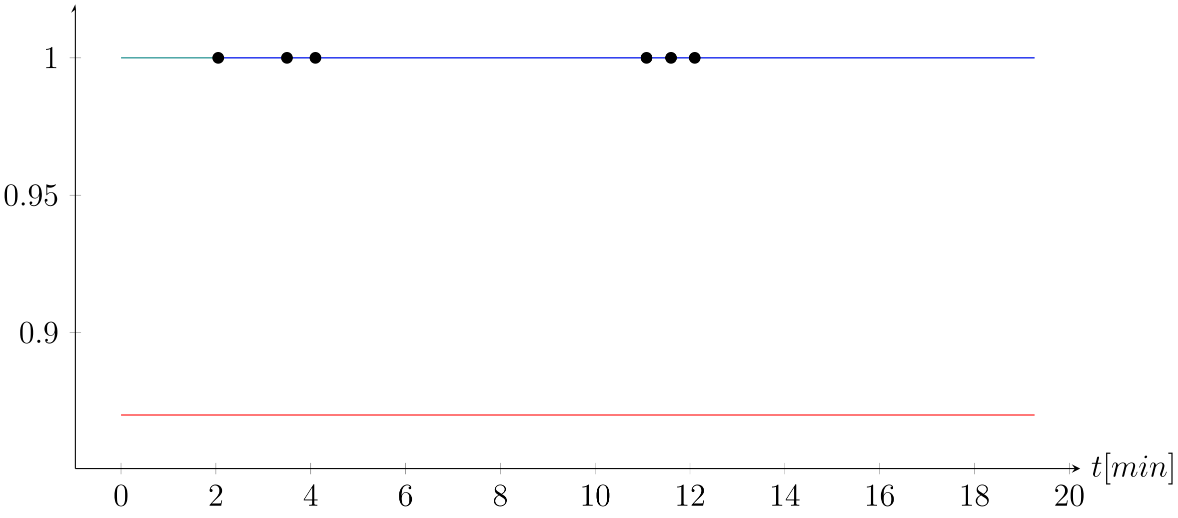


Figure A.54: S3E1 – Ordering Service availability

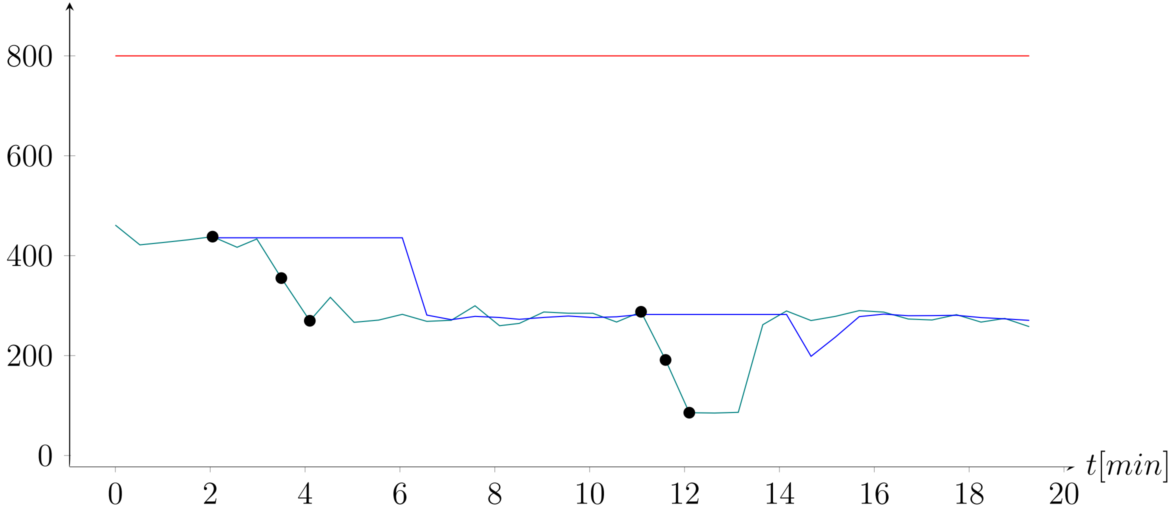


Figure A.55: S3E1 – Ordering Service average response time

PAYMENT-PROXY-SERVICE

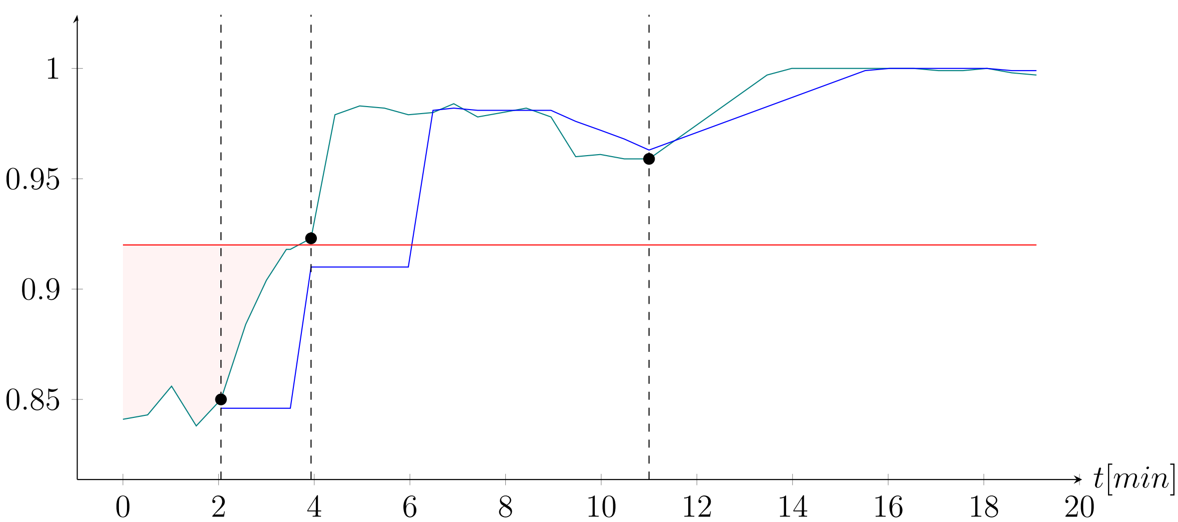


Figure A.56: S3E1 – Payment Proxy Service availability

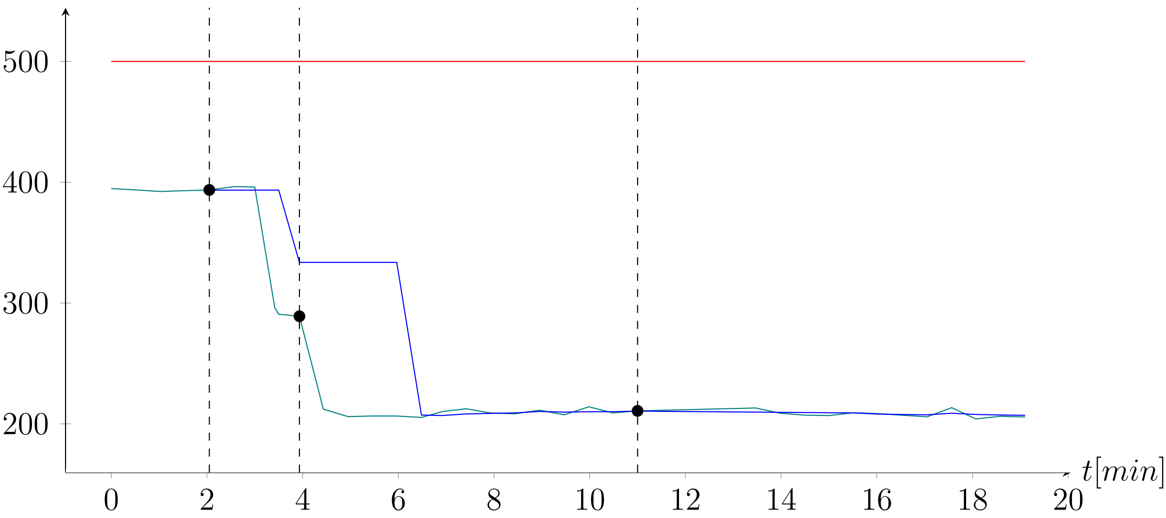


Figure A.57: S3E1 – Payment Proxy Service average response time

DELIVERY-PROXY-SERVICE

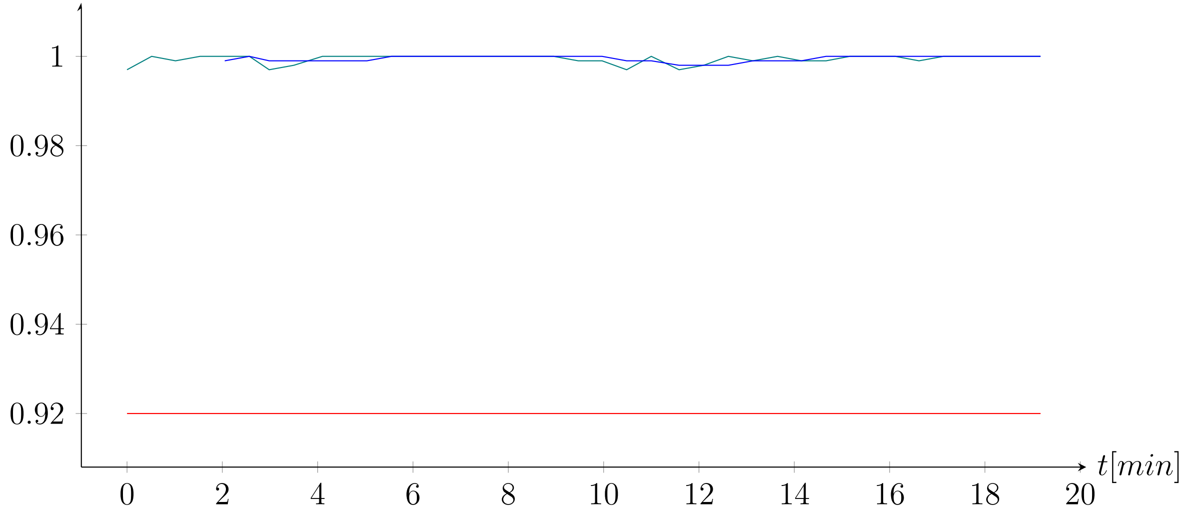


Figure A.58: S3E1 – Delivery Proxy Service availability

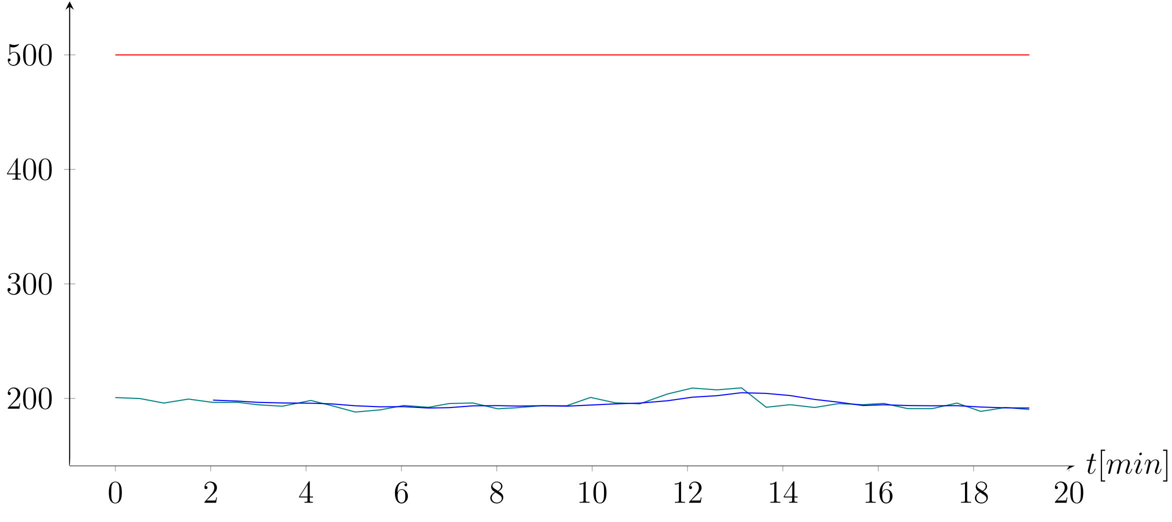


Figure A.59: S3E1 – Delivery Proxy Service average response time

A.4. Reusability of the Managing System

A.4.1. E1 - Analysis of the Managing System reusability

RANDINT-VENDOR-SERVICE

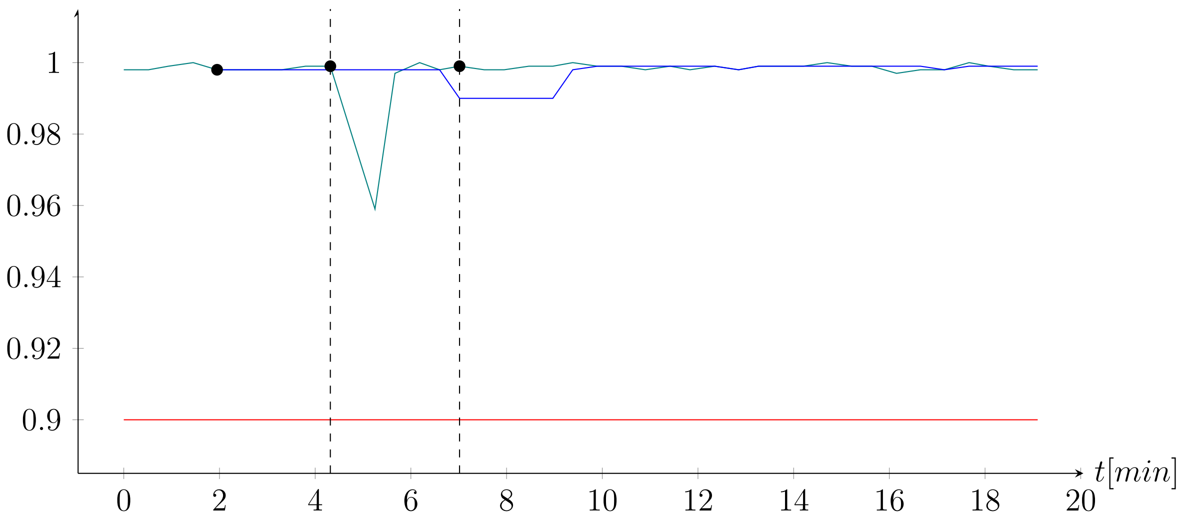


Figure A.60: S4E1 – Randint Vendor Service availability

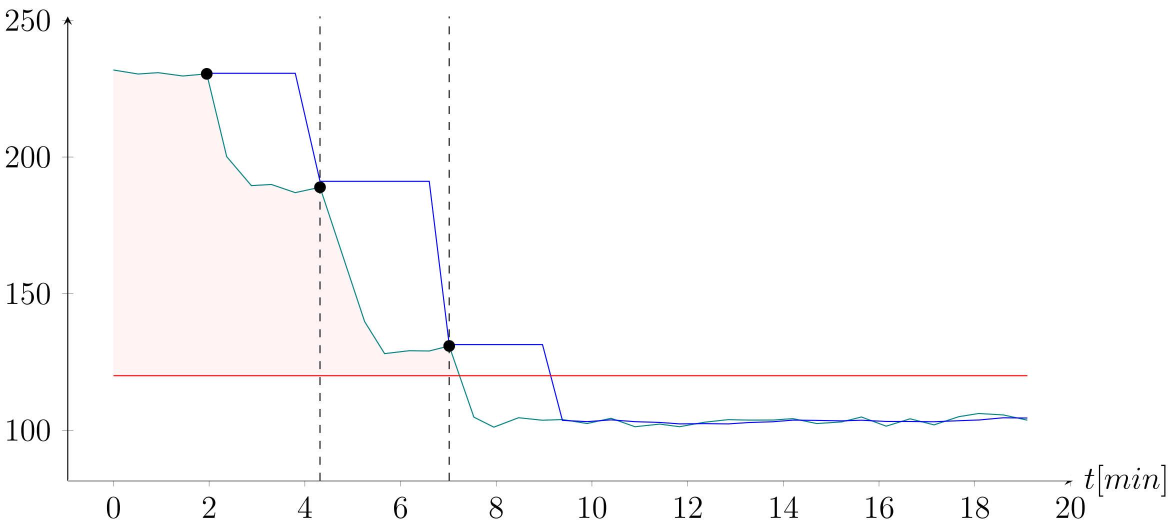


Figure A.61: S4E1 – Randint Vendor Service average response time

RANDINT-PRODUCER-SERVICE

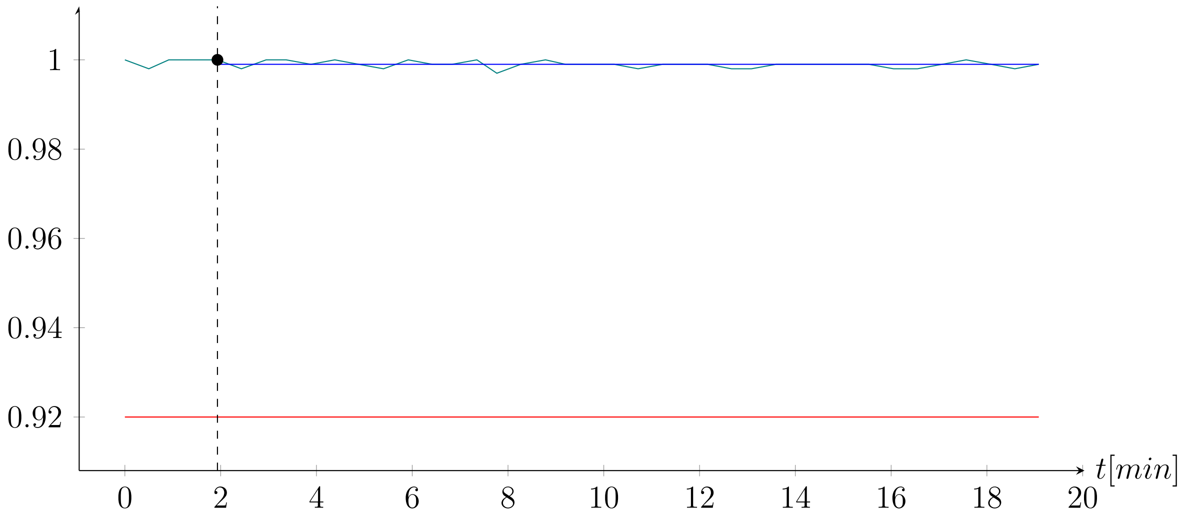


Figure A.62: S4E1 – Randint Producer Service availability

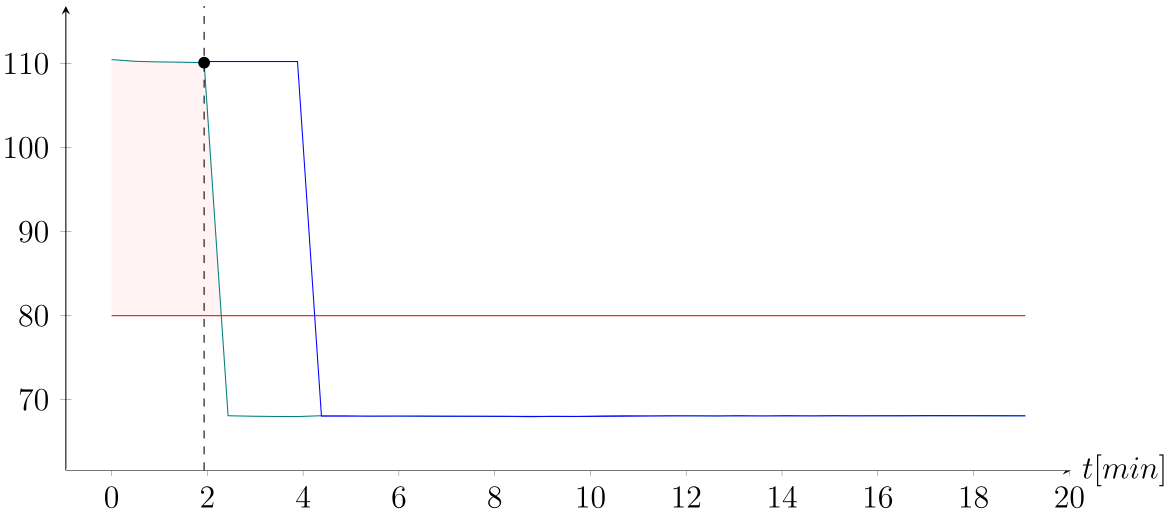
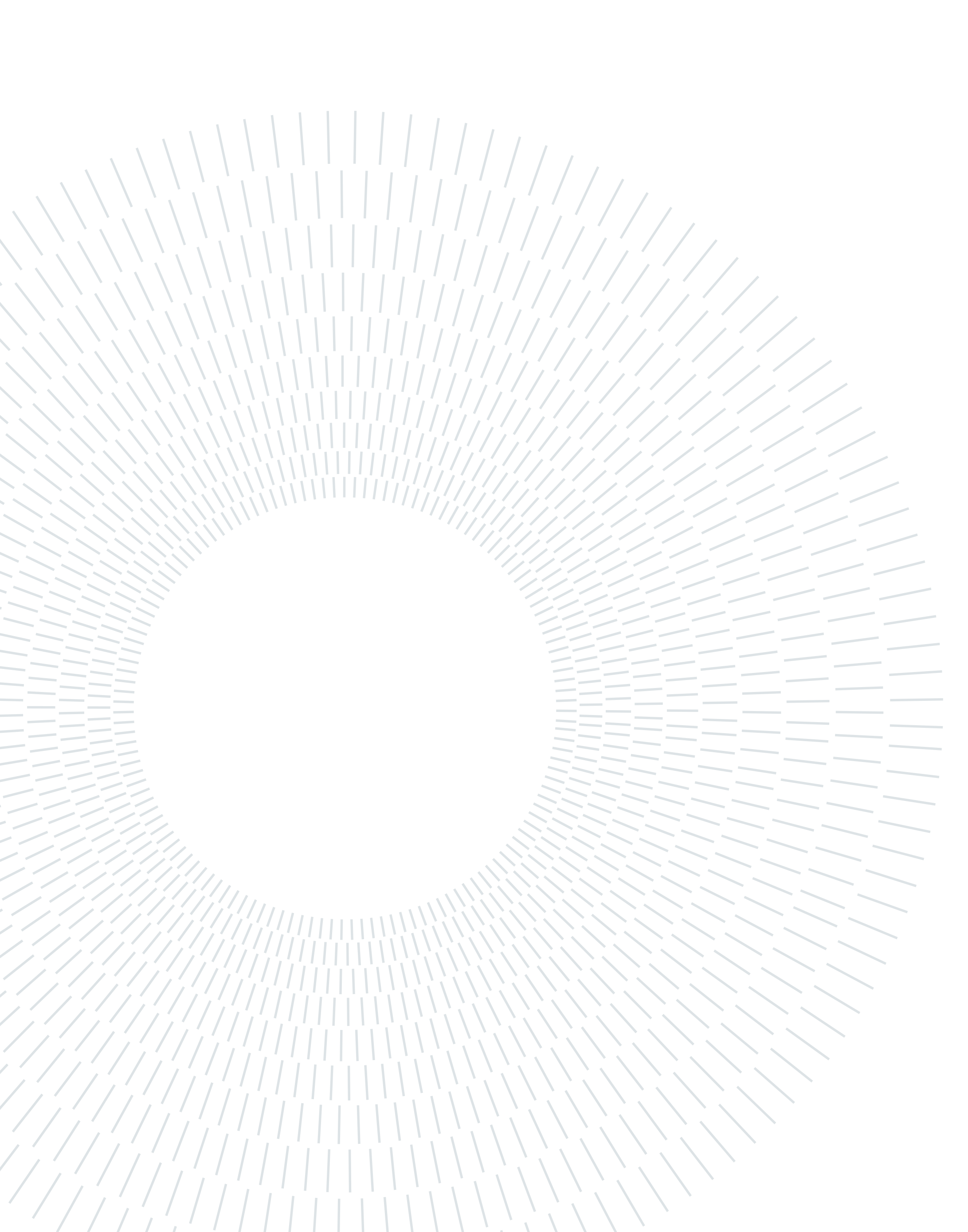


Figure A.63: S4E1 – Randint Producer Service average response time



B | Appendix B - Components interfaces

This appendix contains the list of interfaces offered by the components treated through this paper.
In order to schematize the interfaces, a UML diagram of the APIs is provided for each component, describing the endpoint, the request and the response of each API.

B.1. Managed System

All the services implementing the application logic of the proposed Managed System offer REST APIs over the HTTP protocol.
This section includes the UML diagram of the APIs of each service.

B.1.1. Restaurant Service

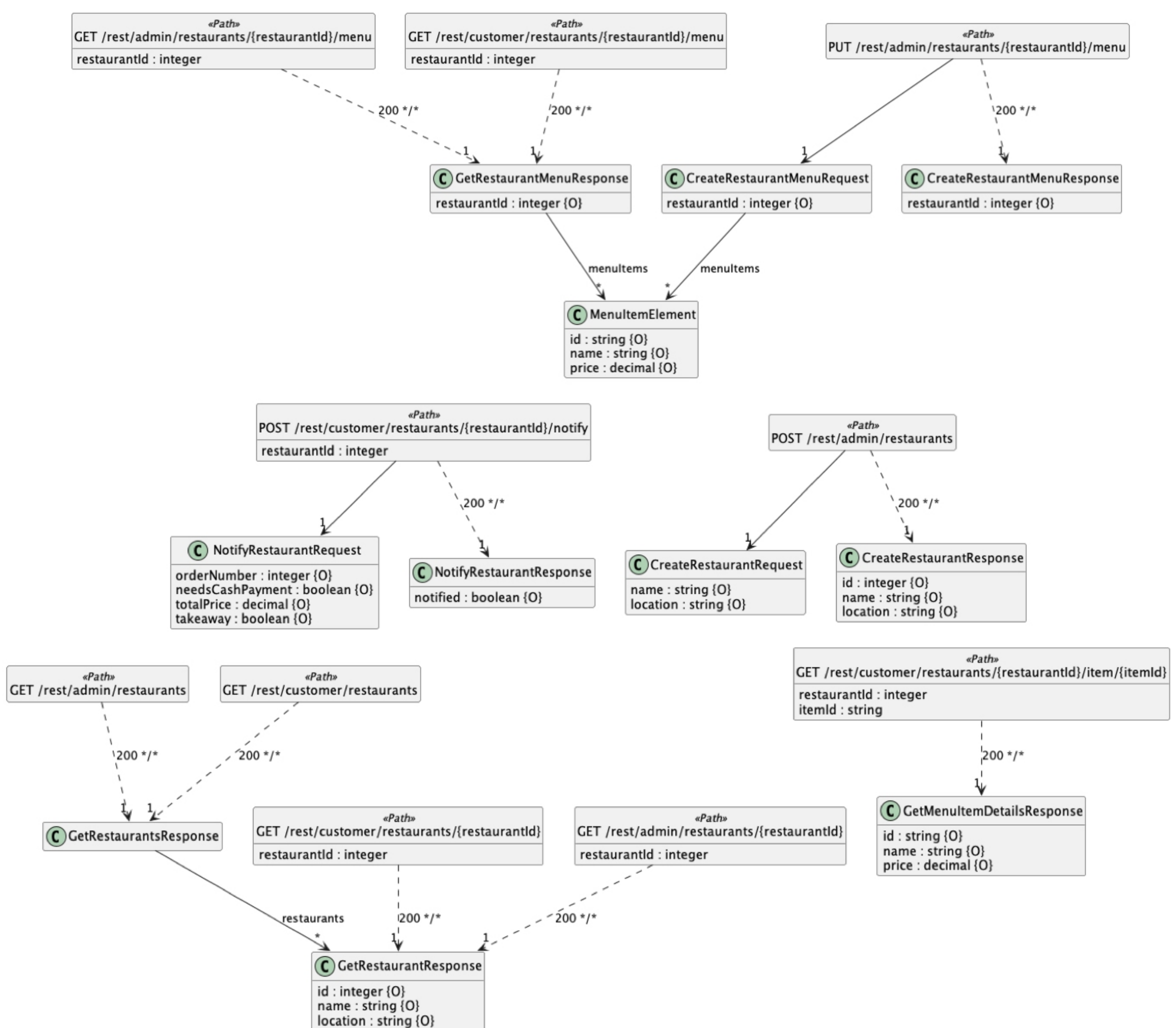


Figure B.1: UML diagram of the API offered by the Restaurant Service

B.1.2. Ordering Service

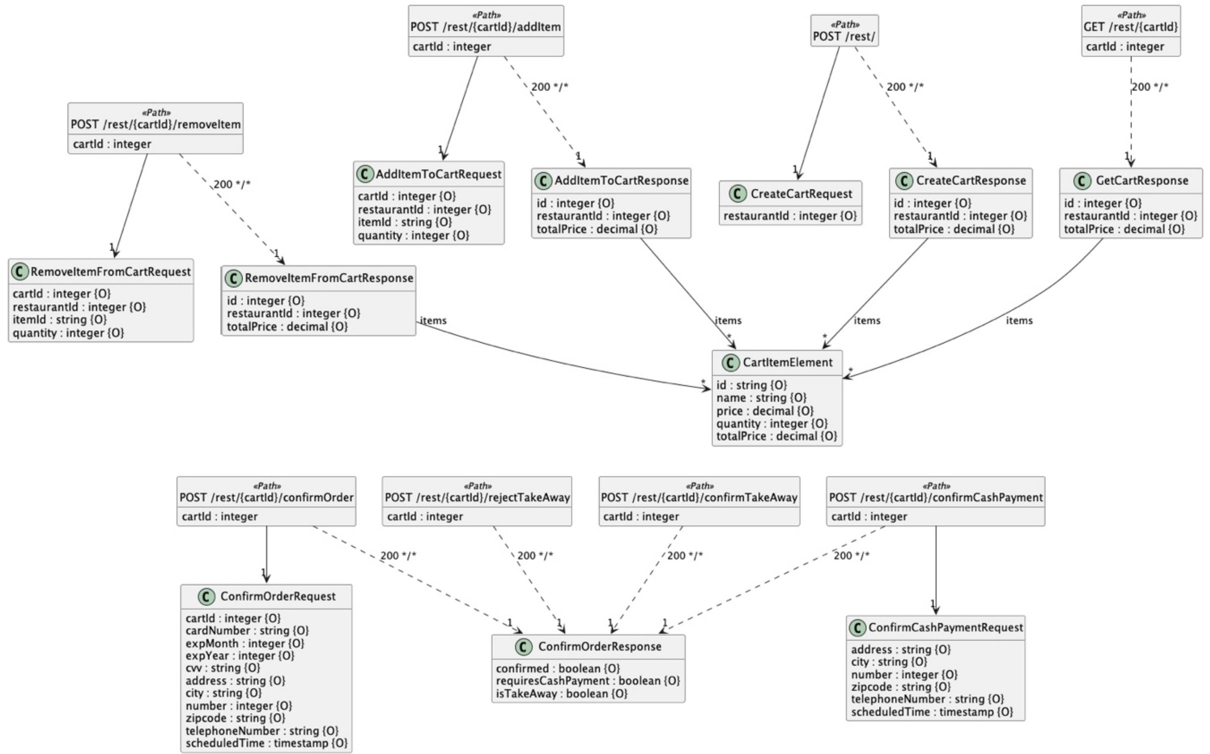


Figure B.2: UML diagram of the API offered by the Ordering Service

B.1.3. Payment Proxy Service

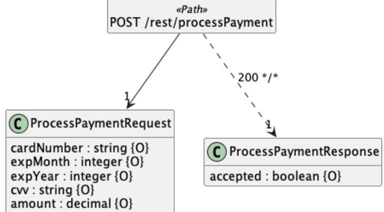


Figure B.3: UML diagram of the API offered by the Payment Proxy Service

B.1.4. Delivery Proxy Service

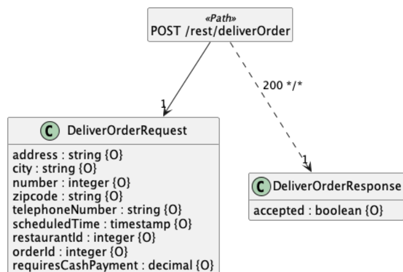


Figure B.4: UML diagram of the API offered by the Delivery Proxy Service

B.2. Probe

The Probe component must offer a REST API which is compliant to the following one.

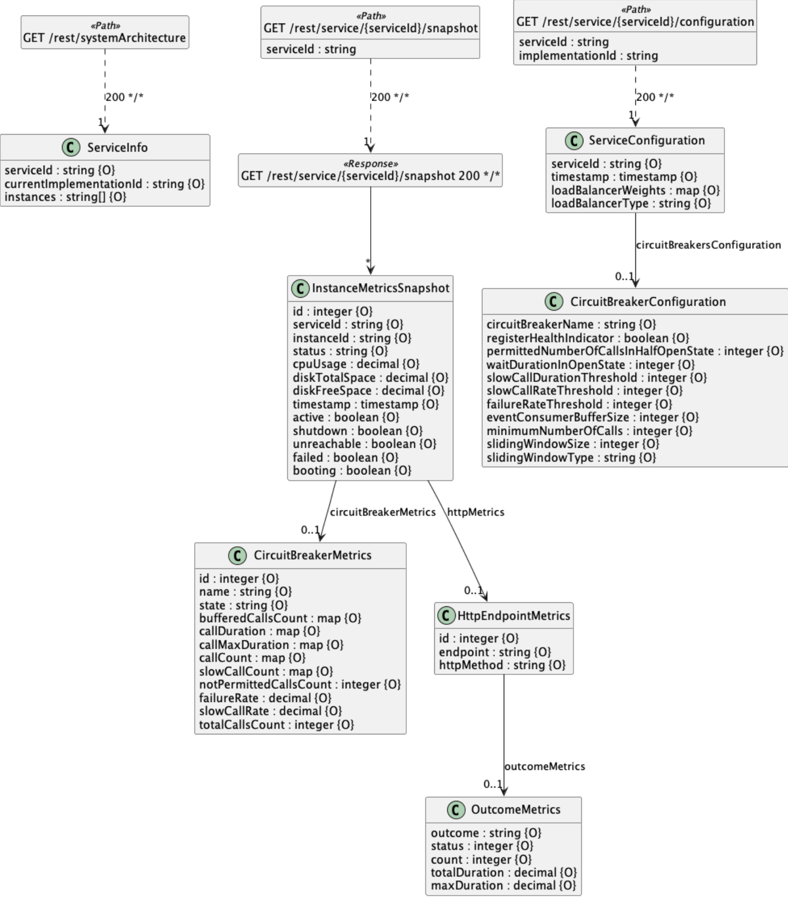


Figure B.5: UML diagram of the API offered by the Probe

B.3. Actuator

The Actuator component must offer a REST API which is compliant to the following one.

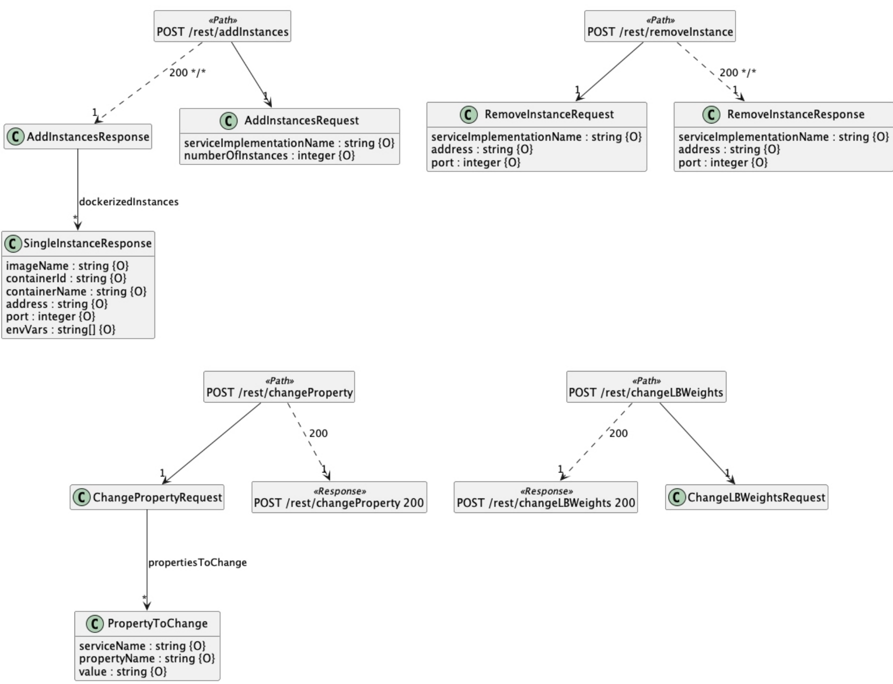


Figure B.6: UML diagram of the API offered by the Actuator

B.4. Managing System

The Managing System exposes to the system admin a REST API to change the configuration parameters specific to each component.

B.4.1. Monitor Component

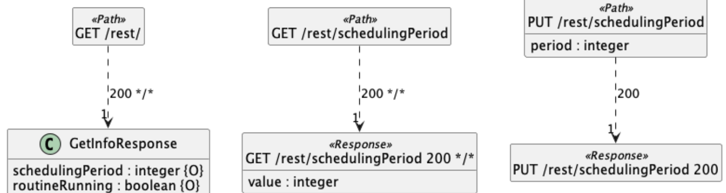


Figure B.7: UML diagram of the API offered by the Monitor Component

B.4.2. Analyse Component

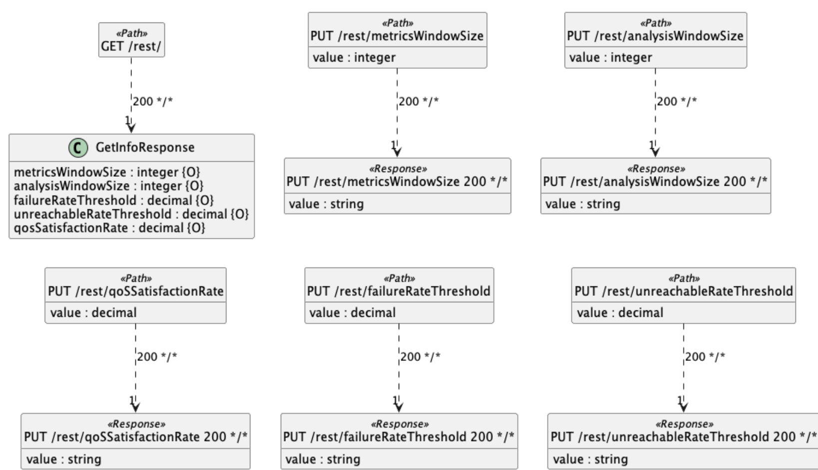


Figure B.8: UML diagram of the API offered by the Analyse Component

B.4.3. Plan Component

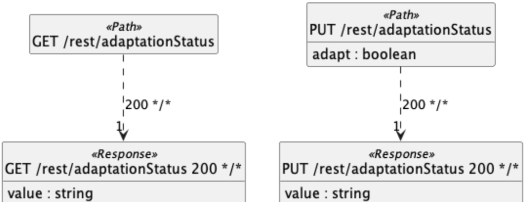


Figure B.9: UML diagram of the API offered by the Plan Component

C | Appendix C - Structure of JSON configuration files

This appendix illustrates the structure of the configuration files, in JSON format, needed by RAMSES to initialize the Knowledge.

C.1. System Architecture

File `system_architecture.json`

```
1  {
2    "services" : [
3      {
4        "service_id" : "SERVICE_ID",
5        "implementations" : [
6          {
7          "implementation_id" : "IMPLEMENTATION_A_ID",
8          "implementation_trust" : VALUE,
9          "preference" : VALUE,
10         "instance_load_shutdown_threshold" : VALUE
11         },
12         ...
13       ],
14       "dependencies" : [
15         {
16         "name" : "DEPENDENCY_A_SERVICE_ID"
17         },
18         ...
19       ]
20     },
21     ...
```

```
  ]
}
```

C.2. System Benchmarks

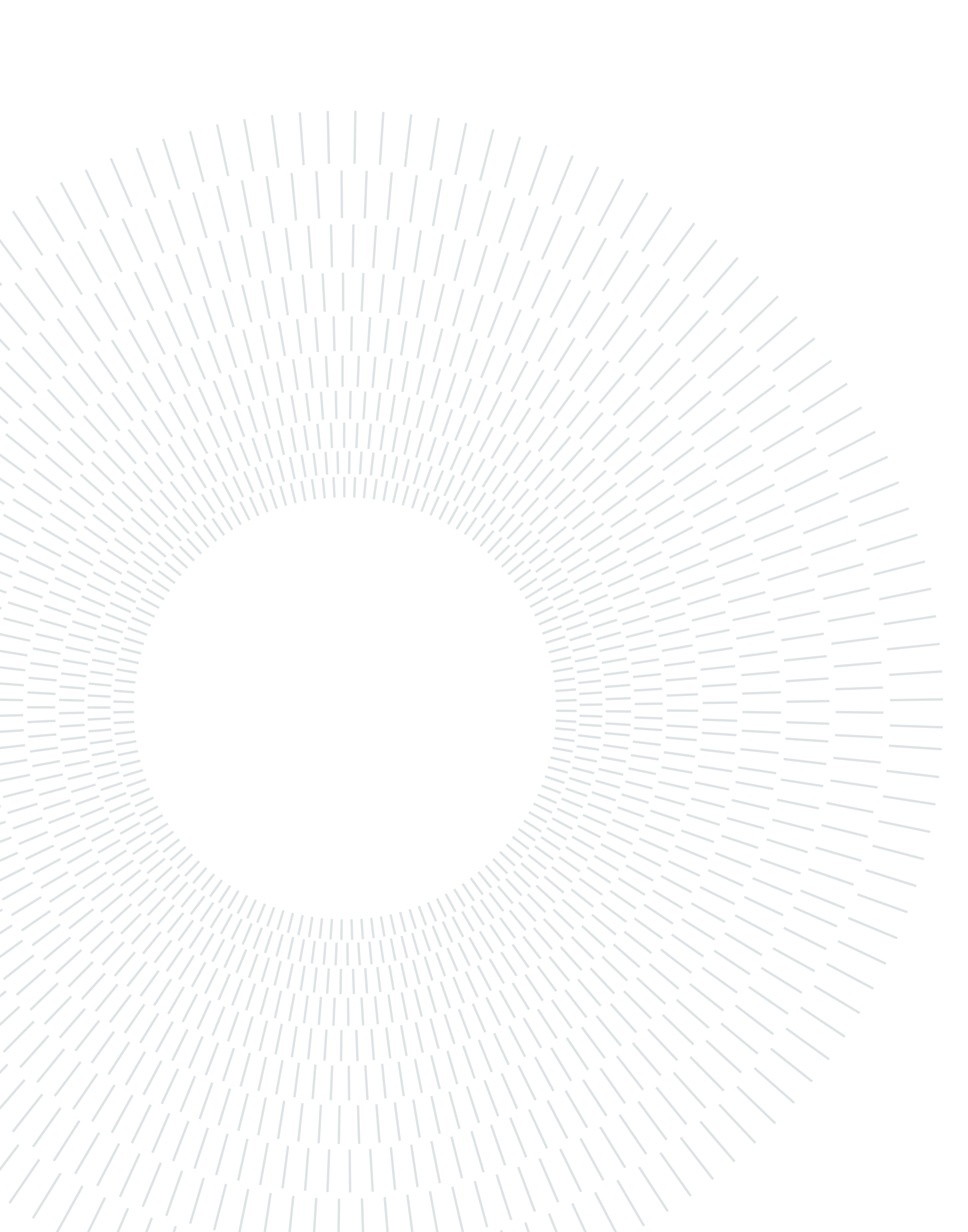
File system_benchmarks.json

```
{
  "services" : [
    {
      "service_id" : "SERVICE_ID",
      "implementations" : [
        {
          "implementation_id" : "IMPLEMENTATION_A_ID",
          "adaptation_benchmarks" : [
            {
              "name": "average_response_time",
              "benchmark": VALUE
            },
            {
              "name": "availability",
              "benchmark": VALUE
            }
          ]
        },
        ...
      ]
    },
    ...
  ]
}
```

C.3. QoS Specification

File qos_specification.json

```
{
  "services" : [
    {
      "service_id" : "SERVICE_ID",
      "qos" : [
        {
          "name" : "availability",
          "weight" : VALUE,
          "min_threshold" : VALUE
        },
        {
          "name" : "average_response_time",
          "weight" : VALUE,
          "max_threshold": VALUE
        }
      ]
    },
    ...
  ]
}
```



List of Figures

2.1 A Conceptual Architecture for Self-Adaptive Software Systems [1] 5
2.2 Structure of a SAS implementing a MAPE-K loop (based on [41]) 6
2.3 Comparison of adaptation logic structures [21] 7

3.1 Use case of a complete interaction with the application – Sequence diagram 15
3.2 Event Diagram of a property change 17
3.3 SEFA Dashboard 18
3.4 Software architecture - Microservices diagram 19
3.5 Workflow of the Probe main task – Sequence diagram 24
3.6 Workflow of the Actuator main task – Sequence diagram 25
3.7 Workflow of a MAPE-K loop iteration – Sequence diagram 26
3.8 UML class diagram describing the System Model 28
3.9 Computation process of a QoS latest value 42
3.10 Computation process of a QoS current value 43
3.11 RAMSES Dashboard 58

4.1 Example of graph used in the experiments 64
4.2 Timeline of injected issues 66
4.3 S1E1 – Restaurant Service availability 67
4.4 S1E1 – Ordering Service average response time 67
4.5 S1E2 – Restaurant Service availability 69
4.6 S1E2 – Ordering Service average response time 70
4.7 S1E3 – Restaurant Service availability 72
4.8 S1E3 – Ordering Service average response time 73
4.9 S1E4 – Restaurant Service availability 75
4.10 S1E4 – Ordering Service average response time 76
4.11 S1E5 – Restaurant Service availability 78
4.12 S1E5 – Ordering Service average response time 79
4.13 S2E1 – Payment Proxy Service – Number of instances 81
4.14 S2E2 – Payment Proxy Service – Number of instances 82

4.15 S2E3 – Payment Proxy Service – Number of instances 84
4.16 S3E1 – Payment Proxy Service availability 86
4.17 S4E1 – Randint Producer Service average response time 90
4.18 S4E1 – Randint Vendor Service average response time 90

A.1 S1E1 – Restaurant Service availability 104
A.2 S1E1 – Restaurant Service average response time 104
A.3 S1E1 – Ordering Service availability 105
A.4 S1E1 – Ordering Service average response time 105
A.5 S1E1 – Payment Proxy Service availability 106
A.6 S1E1 – Payment Proxy Service average response time 106
A.7 S1E1 – Delivery Proxy Service availability 107
A.8 S1E1 – Delivery Proxy Service average response time 107
A.9 S1E2 – Restaurant Service availability – without adaptation 108
A.10 S1E2 – Restaurant Service average response time – without adaptation . . 108
A.11 S1E2 – Ordering Service availability – without adaptation 109
A.12 S1E2 – Ordering Service average response time – without adaptation . . . 109
A.13 S1E2 – Payment Proxy Service availability – without adaptation 110
A.14 S1E2 – Payment Proxy Service average response time – without adaptation 110
A.15 S1E2 – Delivery Proxy Service availability – without adaptation 111
A.16 S1E2 – Delivery Proxy Service average response time – without adaptation 111
A.17 S1E2 – Restaurant Service availability 112
A.18 S1E2 – Restaurant Service average response time 112
A.19 S1E2 – Ordering Service availability 113
A.20 S1E2 – Ordering Service average response time 113
A.21 S1E2 – Payment Proxy Service availability 114
A.22 S1E2 – Payment Proxy Service average response time 114
A.23 S1E2 – Delivery Proxy Service availability 115
A.24 S1E2 – Delivery Proxy Service average response time 115
A.25 S1E3 – Restaurant Service availability 116
A.26 S1E3 – Restaurant Service average response time 116
A.27 S1E3 – Ordering Service availability 117
A.28 S1E3 – Ordering Service average response time 117
A.29 S1E3 – Payment Proxy Service availability 118
A.30 S1E3 – Payment Proxy Service average response time 118
A.31 S1E3 – Delivery Proxy Service availability 119
A.32 S1E3 – Delivery Proxy Service average response time 119

A.33 S1E4 – Restaurant Service availability 120
A.34 S1E4 – Restaurant Service average response time 120
A.35 S1E4 – Ordering Service availability 121
A.36 S1E4 – Ordering Service average response time 121
A.37 S1E4 – Payment Proxy Service availability 122
A.38 S1E4 – Payment Proxy Service average response time 122
A.39 S1E4 – Delivery Proxy Service availability 123
A.40 S1E4 – Delivery Proxy Service average response time 123
A.41 S1E5 – Restaurant Service availability 124
A.42 S1E5 – Restaurant Service average response time 124
A.43 S1E5 – Ordering Service availability 125
A.44 S1E5 – Ordering Service average response time 125
A.45 S1E5 – Payment Proxy Service availability 126
A.46 S1E5 – Payment Proxy Service average response time 126
A.47 S1E5 – Delivery Proxy Service availability 127
A.48 S1E5 – Delivery Proxy Service average response time 127
A.49 S2E1 – Payment Proxy Service – Number of instances 128
A.50 S2E2 – Payment Proxy Service – Number of instances 128
A.51 S2E3 – Payment Proxy Service – Number of instances 129
A.52 S3E1 – Restaurant Service availability 130
A.53 S3E1 – Restaurant Service average response time 130
A.54 S3E1 – Ordering Service availability 131
A.55 S3E1 – Ordering Service average response time 131
A.56 S3E1 – Payment Proxy Service availability 132
A.57 S3E1 – Payment Proxy Service average response time 132
A.58 S3E1 – Delivery Proxy Service availability 133
A.59 S3E1 – Delivery Proxy Service average response time 133
A.60 S4E1 – Randint Vendor Service availability 134
A.61 S4E1 – Randint Vendor Service average response time 134
A.62 S4E1 – Randint Producer Service availability 135
A.63 S4E1 – Randint Producer Service average response time 135

B.1 UML diagram of the API offered by the Restaurant Service 138
B.2 UML diagram of the API offered by the Ordering Service 139
B.3 UML diagram of the API offered by the Payment Proxy Service 139
B.4 UML diagram of the API offered by the Delivery Proxy Service 140
B.5 UML diagram of the API offered by the Probe 140

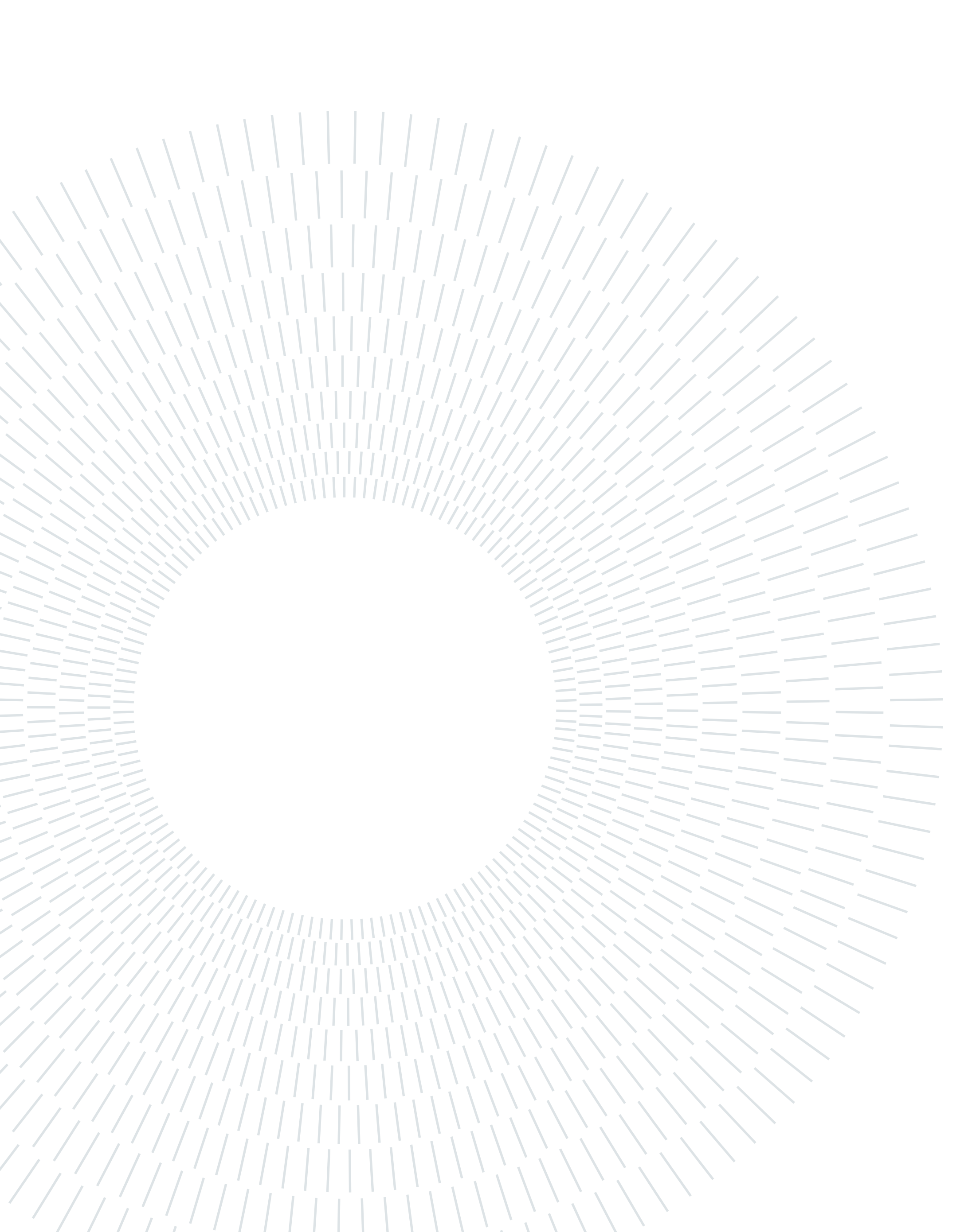
B.6 UML diagram of the API offered by the Actuator 141
B.7 UML diagram of the API offered by the Monitor Component 141
B.8 UML diagram of the API offered by the Analyse Component 142
B.9 UML diagram of the API offered by the Plan Component 142

List of Tables

2.1 Relation of the taxonomy dimensions and the questions [21] 5

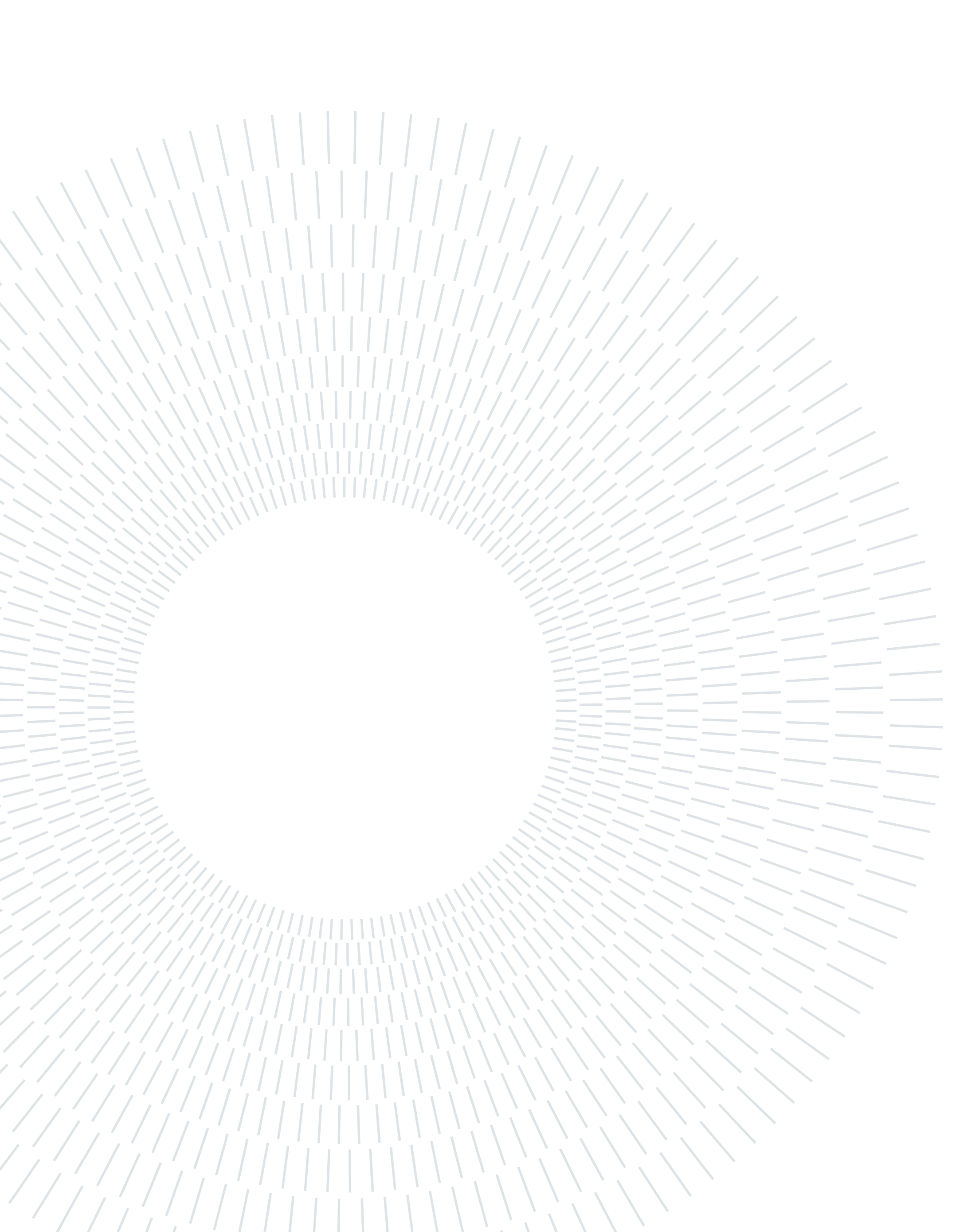
3.1 Adaptation scenarios 13

4.1 Restaurant Service manipulation table 65
4.2 Ordering Service manipulation table 65
4.3 Payment Proxy Service manipulation table 65
4.4 S1 – Configuration parameters of experiment E1 66
4.5 S1 – Configuration parameters of experiment E2 68
4.6 S1 – Configuration parameters of experiment E3 71
4.7 S1 – Configuration parameters of experiment E4 75
4.8 S1 – Configuration parameters of experiment E5 77
4.9 S2 – Configuration parameters of experiment E1 81
4.10 S2 – Configuration parameters of experiment E2 82
4.11 S2 – Configuration parameters of experiment E3 83
4.12 Payment Proxy Service manipulation table 85
4.13 S3 – Configuration parameters of experiment E1 85
4.14 Randint Producer Service manipulation table 88
4.15 Randint Vendor Service manipulation table 89
4.16 Reusability experiment – Configuration parameters 89



List of Algorithms

1 Monitor 37
2 Knowledge – Add Metrics 38
3 Analysis – First Subphase 44
4 Analysis – Second Subphase 46
5 Plan 53
6 Execute 55



Acknowledgements

We would like to spend few words to thank all the people who contributed, directly or indirectly, to the realization of this work.

We deeply acknowledge our advisor and mentor, Prof. Raffaela Mirandola, for her kind attention towards us, her careful guidance and her inspiring suggestions.

Our appreciation and gratitude also go to our co-advisor, Prof. Matteo Camilli, for his attention to detail and his precious advice, that undoubtedly elevated the quality of our work.

Finally, we would also thank our dear families and beloved ones, for motivating and supporting us during our academic path.