RAMSES: an Artifact Exemplar for Engineering Self-Adaptive Microservice Applications

Vincenzo Riccio
Giancarlo Sorrentino
Ettore Zamponi
Matteo Camilli
{first.last}@mail.polimi.it
matteo.camilli@polimi.it
Politecnico di Milano
Milano, Italy

Raffaela Mirandola
raffaela.mirandola@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Patrizia Scandurra
patrizia.scandurra@unibg.it
University of Bergamo
Bergamo, Italy

ABSTRACT

This paper introduces RAMSES, an exemplar tailored for both practitioners and researchers working on self-adaptive microservice applications. By emphasizing a clear separation of concerns between the application and its adaptation logic, RAMSES realizes a reusable autonomic manager that implements a MAPE-K feedback loop whose components are microservices themselves. Its primary focus lies in addressing user-defined QoS attributes at runtime, like availability and performance. To illustrate its usage, we provide a practical example showing its mechanics in an e-food microservice application. Initial experiments indicate the advantages of utilizing RAMSES, as shown by a comparative analysis of the quality properties of a microservice application with and without self-adaptation.

CCS CONCEPTS

• **Computer systems organization** → *Self-organizing autonomic computing*; *Distributed architectures*; • **Software and its engineering** → *Software verification and validation*; *Extra-functional properties.*

KEYWORDS

Microservice applications, self-adaptation, MAPE-K, exemplar

ACM Reference Format:
Vincenzo Riccio, Giancarlo Sorrentino, Ettore Zamponi, Matteo Camilli, Raffaela Mirandola, and Patrizia Scandurra. 2024. RAMSES: an Artifact Exemplar for Engineering Self-Adaptive Microservice Applications. In *19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '24), April 15–16, 2024, Lisbon, AA, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3643915.3644110

1 INTRODUCTION

Microservice architectures have gained widespread popularity due to their scalability, flexibility, and ability to facilitate continuous deployment. Existing development frameworks (like Spring, Flask and GoKit, to name a few) and management infrastructures simplify the whole development and management process also offering some forms of self-adaptation built-in mechanisms (such autoscaling and circuit breaking for resilience)[6]. However, the dynamic and heterogeneous nature of modern computing environments calls for the capacity to employ self-adaptation strategies that extend beyond these built-in capabilities and can achieve arbitrary adaptation.

As a step in addressing this challenge, this paper proposes the exemplar RAMSES to facilitate practitioners and researchers in engineering a self-adaptive microservice application by clearly separating the microservice application and self-adaptation concerns. RAMSES provides a reusable autonomic manager (a managing subsystem) conforming to the well-known feedback control loop model MAPE-K [4] (Monitor-Analyse-Plan-Execute over a Knowledge base) to make a microservice application self-adaptive. Adaptation concerns are mainly aimed at satisfying user-defined QoS attributes (e.g., availability and response time) and self-* autonomic properties (self-configuring, self-healing, and self-optimizing) of a microservice application at runtime. RAMSES's control loop components themselves are microservices. RAMSES is designed to ease its reuse across microservice applications through an *API-led integration* that exploits a *Contract-First* approach to specify probing/actuating RESTful APIs for connecting the RAMSES autonomic manager to any managed microservice application. RAMSES was implemented using the Java-based Spring Boot and Spring Cloud frameworks[1].

To illustrate RAMSES, as part of this exemplar, an e-food microservice application has been developed and released to be used as target managed system. This paper concretely reports some preliminary evaluation results about the capability of RAMSES to make such a microservice application self-adaptive by enforcing the adaptation goals at runtime.

Concerning the timeliness of the problem addressed by the proposed artifact and the overall discussion and considerations in [7] regarding generality (*"ability to support a variety of architectural models and adaptation mechanisms"*) and reusability (*"ability to be reused without requiring substantial effort from software developer"*) in self-adaptive microservices, RAMSES has been conceived with a focus on decoupling managed and managing systems. This is achieved by defining RESTful probing and actuating APIs, providing a means for managing systems to interact with managed systems that expose the same interfaces, thereby facilitating reusability and reuse by-design. At the same time, to better handle and speed up the

[1]https://spring.io/microservices

SEAMS '24, April 15–16, 2024, Lisbon, AA, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0585-4/24/04.
https://doi.org/10.1145/3643915.3644110

implementation of various self-adaptation strategies, the managing system in RAMSES has been developed leveraging microservices principles and mainstream microservice frameworks and infrastructure platforms. In-house mechanisms have also been implemented to enhance its autonomy from the management infrastructure itself. This situates RAMSES at the *Cross-Layer level* in the MAPE integration patterns as defined in [7].

The main contributions of this artifact paper for the community of *Software Engineering for Adaptive and Self-Managing Systems* are: (i) a non-simulated exemplar of self-adaptive microservice application; (ii) a standalone autonomic manager for microservices characterised by a Contract-First and API-led approaches for reusability across other microservice applications; (iii) an in-house developed managed microservice application that can be also reused in a standalone manner as a use case for other benchmarks. Section 5 concludes the paper.

2 RELATED WORK

Exemplars development is an active line of research in the self-adaptive community [2]. We report here the exemplars most related to our work. Hogna [1] 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. It provides a modular managing system, which is however specifically focused on the deployment and configuration of cloud applications on platforms such as *Amazon EC2*. Another well-known exemplar is *TAS* [13]. It provides a useful (simulated) service-based system to be used as a managed subsystem. Moreover, it defines a set of generic adaptation scenarios applicable to service-based systems, to deal with uncertainties of the system itself and of the environment it is set in, which inspired the one proposed in Table 1. Another related exemplar is *SEAByTE* which 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 [10]. 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.

Other exemplars (e.g., *SWIM* [9], *RDMSim* [12] and *EWS* [2]) implement specific managed subsystems that can be reused by researchers to evaluate and compare different adaptation logic). In different contexts, [3] proposes a framework that implements automatic container sizing and self-healing features for a microservice-based application deployed in Docker containers by exploiting MAPE-K loops. In [14], an extension of Kubernetes has been developed to monitor microservices data and manage aspects of scalability. When compared to RAMSES, the key differences are non-simulated managed and managing systems, the adoption of approaches of *Contract-First* and *API-led* for reusability across multiple platforms. This last aspect is also facilitated by our design choice of not depending to much on the self-adaptation capabilities supported by some service management infrastructures natively (such us Kubernetes autoscaling), to avoid conflicts/interferences with the adaptation decisions of our autonomic manager.

[2]www.hpi.uni-potsdam.de/giese/public/selfadapt/category/exemplar/

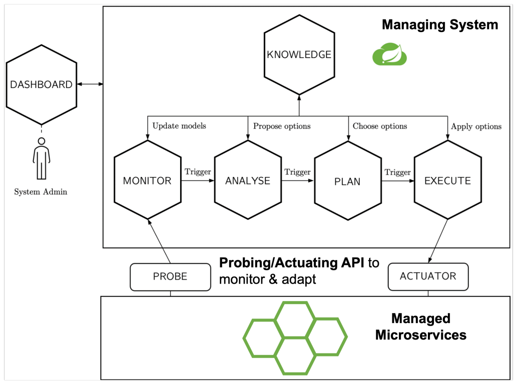


Figure 1: RAMSES architecture overview

3 EXEMPLAR DESCRIPTION

RAMSES is about an autonomic manager (or managing system) for making microservice applications self-adaptive. Once connected to a target microservice application (the managed subsystem) through probing/actuating APIs, the overall self-adaptive microservice application (see Figure 1) is a two-layer architecture that, according to the principles of architecture-based self-adaptation [5], decouples the managed microservice application from its control loop layer. RAMSES control loop components are microservices themselves that execute the monitoring and adaptation activities conforming to the well-known MAPE-K feedback loop model [4]. The goal of RAMSES is to enforce the satisfaction of user-defined QoS attributes (e.g., availability and response time) and self-* properties at run-time, without human oversight. RAMSES is designed to ease its reuse across microservice applications thanks to an *API-led integration* with the managed microservices. An additional dashboard for configuring admin preferences and inspecting the managed microservices at run-time is also supported. RAMSES has been implemented using the Java-based frameworks Spring Boot and Spring Cloud [3] for developing microservice applications. The exemplar is publicly available (See Section 6), including all software artifacts for injecting and simulating fictional service failures or slowdowns for test scenarios.

3.1 Adaptation concerns and strategies

RAMSES is primarily tailored to automate typical service maintenance tasks. The main adaptation concerns are summarized in the Table 1, together with the adaptation goals that the system adaptations should achieve, and related strategies. These last could be the result of a combined application of single adaptation actions. The current prototype supports the following adaptation actions: *addInstance*, *removeInstance*, *changeConfiguration* (e.g., for tuning the service weights of the load balancing, or changing the circuit breaker's parameters and timeout thresholds), and *selectNewImplementation*(to replace all instances of a given service type with the spawned instances of another implementation of the same service).

[3]https://spring.io/microservices

Table 1: Adaptation concerns, goals, and strategies in RAMSES

Concern	Goal	Strategies (examples)
C1: Individual service failure	Recover from a silent or crashed service (self-healing)	Start a new service instance
C2: Search for better service implementations	Select a better service implementation (self-optimizing)	Change the current service implementation
C3: Workload distribution upon a service instance activation/deactivation	Configure load balancing (self-configuring)	Change service configuration for load balancing weights
C4: Violation of QoS requirements	Maintain QoSs (availability and response time)	Add service instance(s) and/or change weights for load balancing (e.g., to lighten or even shutdown an instance with low performance)

3.2 Probing and actuating APIs

The probing/actuating interfaces have been defined using a *Contract-First approach* to promote reusability and then implemented as a RESTful API. Such APIs are consumed by the managing layer and provided by the managed microservice application.

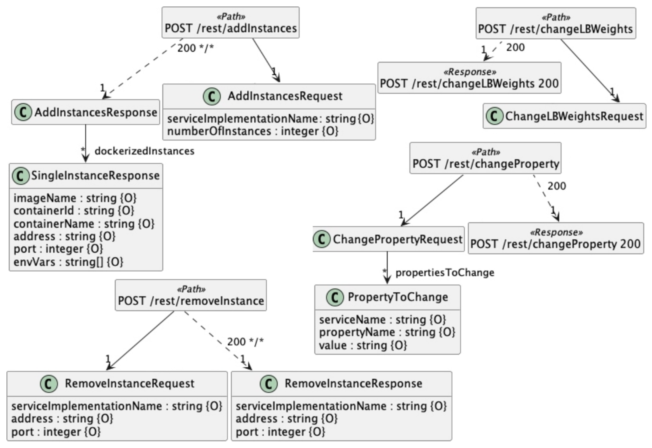


Figure 2: Actuating REST/JSON API

Figure 2 shows the actuating API of RAMSES using the PlantUML class diagram as generated from the API description in the standard format OpenAPI. The actuating API exposes low-level operations to be applied to the managed microservices. Below we provide a description of the main operations exposed by the APIs.

- POST `/rest/addInstances`: allocates and starts a number of new instances of a given service implementation, and returns the details on each new instance (such as its address and port).
- POST `/rest/removeInstance`: stops and deallocates an instance which is currently running.
- POST `/rest/changeLBWeights`: updates the load balancer weights of the instances of a given service.
- POST `/rest/changeProperty`: updates the value of a global property or of a property for a certain service (includes the addition/removal of the property)

The probing API for RAMSES has been defined as well. They are responsible for collecting metrics from all the service instances as a collection of objects `Instance Metrics Snapshot` (or simply *snapshot*). Each service snapshot includes a timestamp, the status of the instance, metrics related to resource usage (e.g., CPU usage), metrics related to HTTP requests (e.g., number of server errors), and circuit breakers. In the current prototype of RAMSES only availability and average response time are concretely used to extract QoS indicators for adaptations.

3.3 Managing subsystem

This section details the behavior of the MAPE-K microservices of RAMSES. The *Knowledge* microservice maintains an up-to-date data structure with all relevant information about the managed microservices as collected by *Monitor* component through the probing API. These include data about service implementations and their associated operational profile, current values for service configuration parameters (e.g., for load balancing and circuit breaking), running service instances and their snapshots series with the collected metrics (CPU usage, HTTP requests stats, etc.) useful for calculating QoS indicators. Other useful information computed by the MAPE components for coordinating the loop execution itself is also stored in the knowledge.

Through the probing API, the *Monitor* microservice periodically collects snapshots of all running service instances and their associated quality metrics, and stores them in the knowledge. The *Monitor* executes asynchronously w.r.t. the rest of the control loop and its sampling period can be dynamically configured via a REST API exposed by the *Monitor* microservice itself or the admin dashboard. Once a control loop execution terminates, the *Executor* microservice notifies the *Monitor* so that the *Monitor* can invoke and request the *Analyze* microservice to start a new asynchronous adaptation loop with an up-to-date data collection for service snapshots.

The *Analyze* microservice exploits the updated metrics stored in the knowledge for the QoS values computations (e.g., availability and response time) adopting a *sliding window* approach, with configurable window dimensions. Data are collected in the last window of observation and a weighted average of the value of interest is then computed. The weights are values in [0, 1] dynamically assigned to microservice instances by a load balancer. If the evaluated QoS is below the user-defined reference threshold, then a set of adaptation actions with different priority levels are selected and stored in the knowledge and the *Plan* microservice is triggered.

The *Plan* microservice compares the proposed adaptation actions and selects the one that best fits the user requirements. The benefit is calculated depending on the adaptation strategy and by estimating the QoS values obtained after applying the corresponding adaptation actions. For example, if the availability of a service does not satisfy the requirement, an *addInstance* adaptation action can be undertaken followed by a *changeConfiguration* action with the computation of the new weighted average availability of the current instances using the new weights. In each case, the best adaptation strategy is selected using, for example, the option with the associated highest estimated utility or the one that maximizes

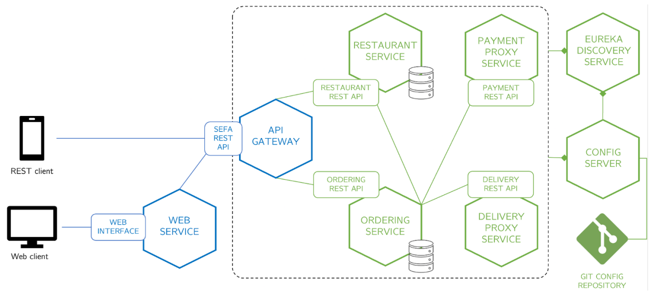


Figure 3: Managed e-food microservice application

multiple objectives. The selected strategy becomes the adaptation plan in the knowledge and then the *Execute* microservice is invoked.

The *Execute* microservice retrieves the adaptation plan from the knowledge, if any. Then, for each adaptation action of the plan, it invokes the actuating API as exposed by the managed microservice application. All processed adaptations are persisted into the knowledge. Once the *Execute* microservice terminates, it notifies the completion of the current loop iteration to the *Monitor*.

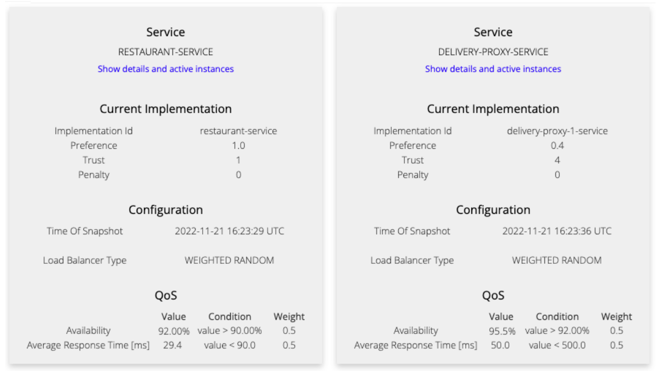
3.4 Managed microservice application

Our use case is an e-food microservice application (see Figure 3) for ordering food from an online restaurant. It is made of a small set of microservices running in Docker containers, an API gateway that acts as a single entry point, infrastructure means for service management – Netflix Eureka for service discovery, and a configuration server for storing and serving hot configurations to microservices –, and service *circuit breakers* as supported by Spring Cloud in a native way. A load balancer (not shown in Figure 3) is also used to allocate requests to service instances according to a *roulette wheel selection* policy [8]. Such a module has been developed in-house to better oversight its internal behavior in a white box, thus avoiding external and not so transparent load balancing mechanisms as offered by management infrastructure solutions (e.g., Kubernetes) of containerized applications. To be fully working, some services (e.g., `Restaurant` service and `Ordering` Service) require some data to be persisted. To this end, a *database-per-service* pattern is applied [11], and independent MySQL databases are added. Finally, to make the managed microservice application also usable and inspectable by humans at the edge of the microservices, a reusable web application (also developed as a Spring Boot application) is provided. It serves as the front end of the managed microservice application and acts as a REST client that communicates with the API gateway.

3.5 HMI-dashboard

Once the managed application has been connected (via appropriate configuration property files) to the end-points of the probing/actuating APIs of RAMSES, a web-based HMI dashboard (developed using Spring Boot and the frontend engine Thymeleaf) allows an administrator for configuring (see the screenshot in Figure 4) and inspecting the control loop and the managed microservices at runtime (see the screenshot in Figure 5).

Figure 4a shows the control variables settable from the dashboard for the running managed microservices, including current



Service
RESTAURANT-SERVICE
Show details and active instances

Current Implementation

Implementation Id	restaurant-service
Preference	1.0
Trust	1
Penalty	0

Configuration

Time Of Snapshot	2022-11-21 16:23:29 UTC
Load Balancer Type	WEIGHTED RANDOM

QoS

	Value	Condition	Weight
Availability	92.00%	value > 90.00%	0.5
Average Response Time [ms]	29.4	value < 90.0	0.5

Service
DELIVERY-PROXY-SERVICE
Show details and active instances

Current Implementation

Implementation Id	delivery-proxy-1-service
Preference	0.4
Trust	4
Penalty	0

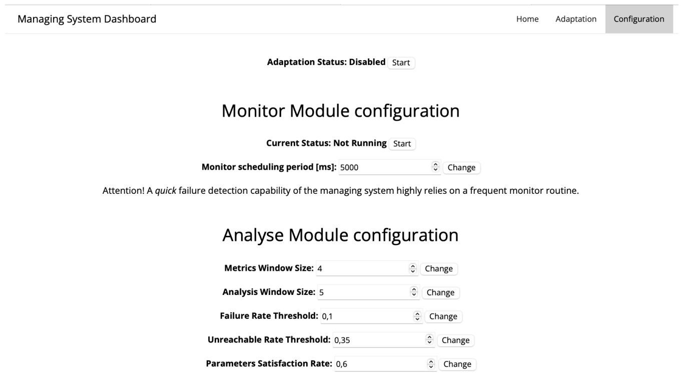
Configuration

Time Of Snapshot	2022-11-21 16:23:36 UTC
Load Balancer Type	WEIGHTED RANDOM

QoS

	Value	Condition	Weight
Availability	95.5%	value > 92.00%	0.5
Average Response Time [ms]	50.0	value < 500.0	0.5

(a) Home



Adaptation Status: Disabled Start

Monitor Module configuration

Current Status: Not Running Start

Monitor scheduling period [ms]: 5000 Change

Attention! A *quick* failure detection capability of the managing system highly relies on a frequent monitor routine.

Analyse Module configuration

Metrics Window Size: 4 Change

Analysis Window Size: 5 Change

Failure Rate Threshold: 0,1 Change

Unreachable Rate Threshold: 0,35 Change

Parameters Satisfaction Rate: 0,6 Change

(b) Parameter Configuration

Figure 4: HMI dashboard

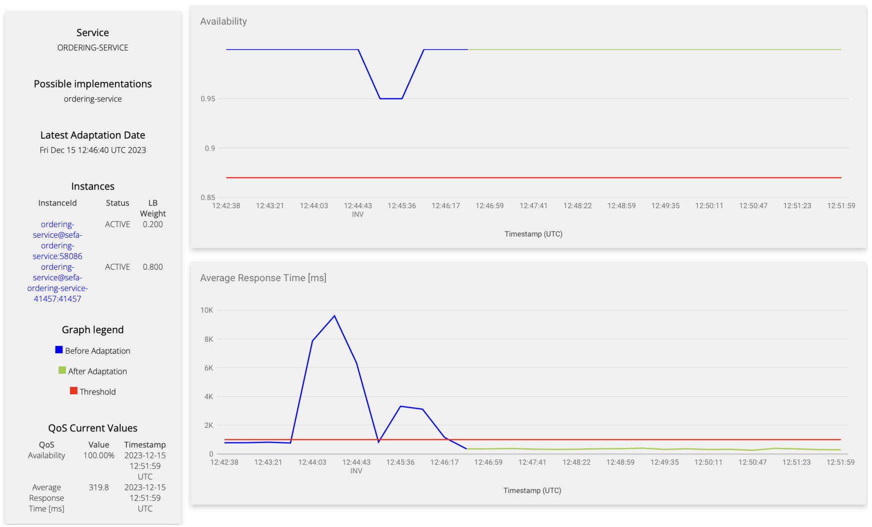
implementations, active service instances and their quality values and QoS requirements. Configuration parameters of the control loop (see Figure 4b) are also visible and settable: the current state of the loop (running or not running), the monitor sampling period, and specific parameters for the analysis. These latter include the *metric window size*, i.e. the number of metric collected and buffered before triggering the analysis; the *analysis window size*, i.e. the number of metric values used to estimate the QoS level of each microservice instance; and the *shutdown threshold*, i.e. the minimum amount of requests that each microservice instance must be able to process to stay alive. The dashboard also allows us to control the execution of the MAPE-K loop, for example, to stop the autonomic manager and simply monitor the managed microservice application.

4 EXEMPLAR EVALUATION

This section describes preliminary results about the efficacy of the RAMSES autonomic manager. We refer the reader to our publicly available package for a comprehensive guideline on utilizing the artifact and conducting independent experimental campaigns with RAMSES (see Section 6).

4.1 Setup of the testbed infrastructure

The experiments have been executed on two physical machines: a *driver node* N_1 and a *subject node* N_2 equipped with an Apple M1


Service
ORDERING-SERVICE

Possible implementations
ordering-service

Latest Adaptation Date
Fri Dec 15 12:46:40 UTC 2023

Instances

InstanceId	Status	LB Weight
ordering-service@sefa-ordering-service:58086	ACTIVE	0.200
ordering-service@sefa-ordering-service-41457:41457	ACTIVE	0.800

Graph legend
🟦 Before Adaptation
🟩 After Adaptation
🟥 Threshold

QoS Current Values

QoS	Value	Timestamp
Availability	100.00%	2023-12-15 12:51:59 UTC
Average Response Time [ms]	319.8	2023-12-15 12:51:59 UTC

Figure 5: Screenshot of the dashboard showing QoS data of the managed microservices at runtime.

(8-core) processor, 16GB RAM LPDDR4, and 256GB NVMe SSD. To run our experiments, we used the *swarm mode* of Docker to manage a cluster of two Docker daemons: DockerManager deployed onto N_1, and a DockerWorker deployed onto N_2. The RAMSES managing subsystem and a software module ScenarioRunner for conducting experiments have been deployed into the driver node N_1. The DockerManager and DockerWorker are in charge of the deployment/undeployment of the containerized microservices of the managed subsystem onto the subject node N_2.

For the sake of simplicity, the artifact comes with instructions to set up and run RAMSES as well as a set of pre-configured scenarios in a single machine. To promote portability and usability, our package includes also instructions for other mainstream platforms other than MacOS (i.e., Windows, and Linux).

The ScenarioRunner software module runs as a stand-alone microservice and has been developed to facilitate and automate the setup of the experimental campaign. It takes as input the specification of the experiments in terms of all configurable factors required to create and reproduce a certain test scenario: the number of instances per microservice and boot time, the observation period T of the experimental campaign, workload (service requests per second to issue to the managed subsystem), synthetic delays in given time intervals (following a given Normal distribution), failures (synthetic exceptions to generate in a given time interval according to a certain failure rate), and network issues (i.e., time intervals of service unreachability) to inject. The specification is provided to ScenarioRunner through a configuration file where the user specifies the aforementioned factors through pre-defined variables as follows:

```
FAILURE_INJECTION = Y
FAILURE_INJECTION_1_START = 180
ID_OF_INSTANCE_TO_FAIL = restaurant:58085
```

In this example, the option failure injection is set to "Yes". Thus, a failure is injected at time 180s targeting the microservice instance with ID restaurant:58085.

Given the specification, ScenarioRunner runs all the experiments following a simple pipeline for each one of them. Such a pipeline is made of three subsequent phases: *setup*, *execution & monitoring*, and *teardown*. During the *setup* phase the RAMSES autonomic manager and the managed microservice application are deployed following the given configuration. The database images are then loaded onto the managed microservices, and the control loop's knowledge is reset to its initial state to make sure that the adaptation decisions undertaken are not influenced by prior knowledge collected from past experiments. During the *execution* phase, the ScenarioRunner replicates the desired operating conditions in N_2 by generating a workload for the managed microservices and by injecting specific issues/failures, as specified in the experimental campaign. After a ramp-up period, the *monitoring* sub-phase starts and the ScenarioRunner computes the metrics of interest to quantify the effectiveness of RAMSES in achieving the adaptation goals. When the observation period T ends, the *teardown* phase terminates and undeploys the managing and the managed subsystems.

In our preliminary experiments, the effects of adaptations applied by the autonomic manager have been measured over the observation period of $T = 20$ minutes for all experiments. Concerning the other configuration parameters of the managing layer, we adopted the following default values (see Figure 4b): metric and analysis window size equal to 5, shutdown threshold 40%, and monitor scheduling period of 5 seconds.

4.2 Preliminary results

We here focus on the concern *C*4 of Table 1 about achieving QoS requirements, and report some outcomes of the comparative analysis of the capability of the system to maintain QoS properties with and without adaptation (see Figure 6 and Figure 7).

To evaluate the system degradation, we introduce the quantitative metric termed the *QoS Degradation Area* (QoSDA). The QoSDA is calculated by measuring the area between the QoS threshold and

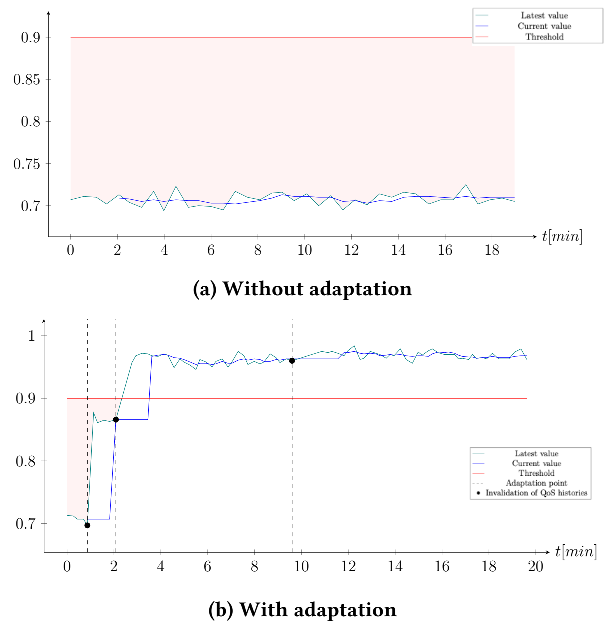


Figure 6: `Restaurant` **service availability**

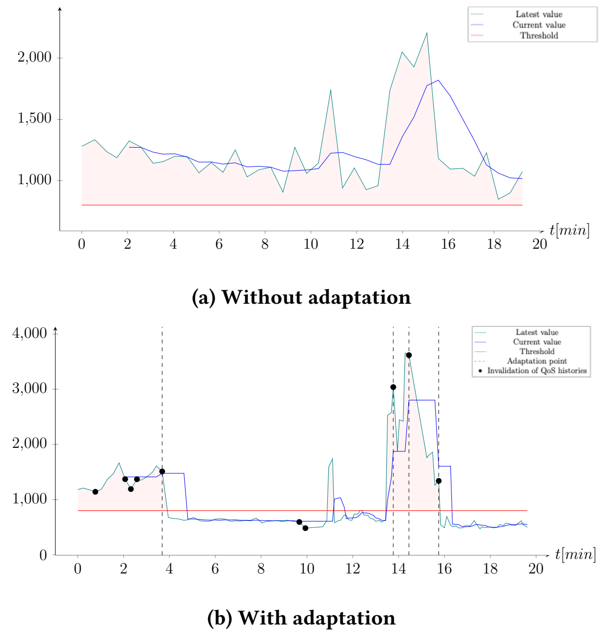


Figure 7: `Ordering` **service response time**

the actual QoS during specific time intervals within the observation period T. These intervals correspond to instances where the operational state is degraded, indicating that the actual QoS falls below the prescribed QoS requirement, such as a target availability level. In essence, the adaptation actions performed by RAMSES aim to minimize the QoSDA. Smaller QoSDA indicates a more effective mitigation of degradation.

Figure 6 shows the results of the `Restaurant` service availability by running the e-food application without and with the RAMSES autonomic manager under a uniform workload intensity of 100 requests per second generated by 50 concurrent users. The application itself is not able to meet the QoS requirement *availability > 0.9* as illustrated in Figure 6a, with a QoSDA equal to 3.65×10^3 (red area). With the introduction of the autonomic manager, RAMSES plans and then actuates two adaptations to satisfy the QoS requirement (adaptation points corresponding to $t = 2$ and $t = 3$). The first action corresponds to the change of the weights of the load balancer to penalize the bad-performing instances. This improves the availability up to $\sim$0.8 at time $t = 3$, but the requirement is still not satisfied. Another adaptation action is then performed: two instances out of three are shut down, due to their poor performances. The microservice finally achieves a stable and desired QoS value from $t = 4$ on. The QoSDA is 4.28×10^2 (red area) in Figure 6b. The adaptation options reduce the QoSDA by 88%.

Similar results can be observed in Figure 7 considering the `Ordering` service response time. In this case, the QoS requirement is *response time < 800 milliseconds*, which is not fulfilled without introducing the managing layer for adaptation. Figure 7a shows a QoSDA of 8.22×10^3. The results of the introduction of the autonomic manager are shown in Figure 7b, where the addition of a new instance and the change of the weights of the load balancer yield QoS requirement satisfaction. QoSDA in this last plot is equal to 6.39×10^3 which leads to a 22% reduction.

5 CONCLUSION AND FUTURE DIRECTIONS

This work describes RAMSES a microservice-based autonomic manager tailored to microservice applications. RAMSES is reusable given that the managed system implements the probing/actuating APIs and the provided load-balancing mechanism. The e-food microservice application developed as the managed application is itself reusable by other applications to experiment with different kinds of adaptation means.

In future work, RAMSES may be extended with: more metrics (e.g., resource usage and circuit breakers metrics) for a more in-depth evaluation; more adaptation actions and scenarios to deal with additional quality like security and with more complex situations; more complex decision-making approaches taking into account also the costs and the risks derived from the application of an adaptation option.

6 ARTIFACT AVAILABILITY

The package of RAMSES, including the sources and the instructions to set up and run it is publicly available at https://zenodo.org/doi/10.5281/zenodo.10400820. We refer the reader to the README file included in the package for further details.

ACKNOWLEDGMENTS

This work has been partially founded from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs.

REFERENCES

[1] C. Barna, H. Ghanbari, M. Litoiu, and M. Shtern. 2015. Hogna: A Platform for Self-Adaptive Applications in Cloud Environments. In *SEAMS*. 83–87.

[2] R. R. Filho, E. Alberts, I. Gerostathopoulos, B. Porter, and F. M. Costa. 2022. Emergent Web Server: An Exemplar to Explore Online Learning in Compositional Self-Adaptive Systems. In *SEAMS*. 36–42.

[3] Luca Florio and Elisabetta Di Nitto. 2016. Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures. In *IEEE ICAC*. 357–362.

[4] Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. *IEEE Computer* 36, 1 (Jan. 2003).

[5] Sara Mahdavi-Hezavehi, Vinicius H.S. Durelli, Danny Weyns, and Paris Avgeriou. 2017. A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. *Information and Software Technology* 90 (2017), 1–26. https://doi.org/10.1016/j.infsof.2017.03.013

[6] N. C. Mendonca, P. Jamshidi, D. Garlan, and C. Pahl. 2021. Developing Self-Adaptive Microservice Systems: Challenges and Directions. *IEEE Software* 38, 2 (2021), 70–79.

[7] Nabor C. Mendonça, David Garlan, Bradley Schmerl, and Javier Cámara. 2018. Generality vs. Reusability in Architecture-Based Self-Adaptation: The Case for Self-Adaptive Microservices. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings* (Madrid, Spain) *(ECSA '18)*. Association for Computing Machinery, New York, NY, USA, Article 18, 6 pages.

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

[9] G. A. Moreno, B. Schmerl, and D. Garlan. 2018. SWIM: An Exemplar for Evaluation and Comparison of Self-Adaptation Approaches for Web Applications *(SEAMS '18)*. ACM, 137–143.

[10] F. Quin and D. Weyns. 2022. SEAByTE: A Self-adaptive Micro-service System Artifact for Automating A/B Testing. In *SEAMS*. 77–83.

[11] C. Richardson. 2022. Pattern: Microservice Architecture. https://microservices.io/patterns/microservices.html

[12] H. Samin, L. H. G. Paucar, N. Bencomo, C. M. C. Hurtado, and E. M. Fredericks. 2021. RDMSim: An Exemplar for Evaluation and Comparison of Decision-Making Techniques for Self-Adaptation. In *SEAMS*. 238–244.

[13] D. Weyns and R. Calinescu. 2015. Tele Assistance: A Self-Adaptive Service-Based System Exemplar. In *SEAMS*. 88–92.

[14] Shuai Zhang, Mingjiang Zhang, Lin Ni, and Peini Liu. 2019. A Multi-Level Self-Adaptation Approach For Microservice Systems. In *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. 498–502.