An Extended Study of the Performance of Flexible Controllers Composed of Micro-controllers

BENTO RAFAEL SIQUEIRA, Federal University of Lavras, Brazil
FABIANO CUTIGI FERRARI, Federal University of São Carlos, Brazil
ROGÉRIO DE LEMOS, University of Kent, United Kingdom

Generic controllers for self-adaptive systems can be configured parametrically according to system needs, even though their reuse is restricted because of the wide range of services that may be provided by each stage of a feedback control loop, like MAPE-K. Rainbow is a typical example of such a generic, monolithic controller. This article revisits and extends prior work that advocates structurally flexible controllers, as ensembles of micro-controllers each providing specific services. We experimented with our approach with three different architectural configurations for the controller: monolithic, decentralised, and decentralised with a meta-controller. Our results indicate that despite the decentralised configuration with a meta-controller demanding more computational resources, it performed comparatively well compared to the other configurations, including when measuring the target system's response time. Moreover, we found that variations of the control loop timing at the different layers of the controller impact the stability of the target system. We have also evolved the controller by adding a new micro-controller, which caused no impact on the other micro-controllers, and mostly kept the target system's performance. We conclude that a multi-layered controller design, based on micro-controllers, provides the basis for defining structurally flexible controllers at operational-time, and may promote reuse at development-time.

CCS Concepts: • **Software and its engineering → Layered systems**.

Additional Key Words and Phrases: self-adaptive software systems; feedback control loop; flexible controller; microservices

1 INTRODUCTION

In self-adaptive software systems, Rainbow [20] is one of the few examples of a *generic* controller that has been used in several application domains [47] and by different researchers [11]. A key factor restricting the reuse of controllers is that they are intrinsically dependent on the target system (*i.e.,* the software system to be controlled). To mitigate this issue, Rainbow raises its abstraction of a target system to the software architecture, reducing the coupling between the controller and the system [20]. However, the wide range of applications, *i.e.,* target software systems, may also impose a wide range of services expected from the different stages of a controller (such as controller based on the MAPE-K reference architecture [31]). For instance, a safety-critical target system may need an analysis stage based on model checking, instead of a simpler architectural analysis.

Adopting the Rainbow approach to develop a *truly generic* controller that can provide such a wide range of services is quite challenging, and perhaps counter-productive since the unit of reuse would be a monolithic controller that is only parametrically configurable, which is the case for most existing controllers [32]. In contrast, a controller whose services are provided according to the actual needs of the target system can achieve greater

Authors' addresses: Bento Rafael Siqueira, bento.siqueira@ufla.br, Federal University of Lavras, São Sebastião do Paraiso, Minas Gerais, Brazil; Fabiano Cutigi Ferrari, fcferrari@ufscar.br, Federal University of São Carlos, São Carlos, São Paulo, Brazil; Rogério de Lemos, R.Delemos@kent.ac.uk, University of Kent, Canterbury, Kent, United Kingdom.

ACM 1556-4703/2025/1-ART
https://doi.org/10.1145/3715145

flexibility and variability than a monolithic controller [35]. To achieve a modular and structurally flexible approach, loosely coupled controller components implementing specific services should provide the necessary structural independence between one another. This structural flexibility offers a significant advantage over monolithic designs by enabling the development of controllers characterized by low coupling and high cohesion, ensuring better adaptability and maintainability. Examples of such micro-controllers include various tests to be executed on the target system before deployment, alternative solutions for detecting system intrusions, or diverse algorithms to support decision-making during synthesis. The claim is that controllers for self-adaptive software systems should consist of a collection of independent *micro-controllers*. To coordinate these micro-controllers, we envisage *meta-controllers* for configuring the controller according to services required by the target system, thus achieving structure flexibility at the controller level [35].

These micro-controllers would consist of microservices. In a preliminary work [55], we have demonstrated and evaluated an embryonic multi-layer controller using the PhoneAdapter exploratory study in which micro-controllers are configured during system operation depending on changes affecting the system or its environment. In a more recent work [54], we described our view of multi-layer controllers. We evaluated whether a controller, based on micro-controllers, has the advantage of being structurally flexible without compromising the performance of the target system. To evaluate our proposed approach, we have employed as an exploratory study a variant of ZNN.com [14], namely, Kube-ZNN [1], using three different architectural configurations for the controller: monolithic, decentralised, and decentralised with a meta-controller.

In this paper, compared with Siqueira et al. [54], we have made three significant contributions. First, we redid all the experiments, included response times for comparison with existing solutions, and observed that our proposed multi-layered controller solution did not affect the overall performance of the target system. Second, we analysed the timings of the different control loops in our multi-layer controller and noticed that their absolute and relative timings impacted the stability of the target system. Third, we added a new micro-controller that acts directly on the infrastructure of the target system without changing the other micro-controllers, and we observed an improvement in the quality attributes of the target system, including stability.

In the context of promoting flexible controllers for self-adaptive software systems, our key contributions are:

- Definition of an approach, based on micro-controllers implemented as microservices, for enabling structurally flexible controllers.
- Evaluation and comparison of three different architectural configurations for controllers.
- Evaluation of the impact on the stability of the target system when changing the control loop timings within a multi-layer controller.
- Evaluation of the impact on the quality attributes of the target system when adding a new micro-controller to the controller.

The evaluations and comparisons have indicated that our multi-layer approach has little impact on the target system's performance. In some situations, the decentralised controller with a meta-controller outperformed the other two configurations. We have also observed that the timings of the control loops at the different layers of the controller may affect the performance and the stability of the target system. The evolution of the controller by adding a new micro-controller that acts directly on the target system's infrastructure was straightforward. Overall, no impact on the other micro-controllers was observed, and just little impact on the target system's response time was observed, whereas other target system's quality attributes were preserved.

The remainder of this article is organised as follows. Section 2 presents an overview of the main concepts of Rainbow, Docker, Kubernetes and Kubow, and APIs and communication. Section 3 presents the proposed approach for architecting flexible controllers based on micro-controllers. Section 4 introduces an exploratory study based on ZNN.com. In Section 5, through the exploratory study, we evaluate the proposed in three experiments which

were designed based on the GQM template. Section 6 discusses the threats and limitations of our research. Finally, we discuss related work in Section 7 and conclude the article in Section 8.

2 BACKGROUND

This section describes the foundations of our work. First, Section 2.1 describes the Rainbow architecture, its main components and technologies. Next, Section 2.2 describes the main concepts related to Docker and Kubernetes. Section 2.3 describes the Kubow architecture and some useful tools for running applications. Finally, Section 2.4 describes the main characteristics and technologies that are used, which allow the different components to perform communication between themselves.

2.1 Rainbow

Rainbow [20] provides a reusable infrastructure and tools for supporting self-adaptation, thus allowing for its effortless customisation across different target systems. Rainbow relies on the architecture model of the target system (represented in Acme language), and a collection of adaptation strategies defined in the Stitch language (Acme and Stitch are described further in this section). Rainbow contains several components to store the architectural model, evaluate the need for an adaptation, select the appropriate adaptation, and execute the adaptation.

Figure 1 illustrates the Rainbow framework's control loop that enables self-adaptation. Given that it uses an abstract architectural model, a Rainbow-based control loop monitors, evaluates, and performs adaptations on the running target system. The monitoring at runtime uses model properties; the evaluation analyses if a problem occurs, regarding model constraint violations; and finally, adaptations involving system needs are performed [20]. As it is illustrated in Figure 1, there are two main layers: *System-layer infrastructure* and *Architecture-layer infrastructure*. The *System-layer infrastructure* provides the system access interface, using *probes*, a system measurement mechanism, responsible for observing and measuring system states. There is also a mechanism (the *effectors*) that carries out the current system modification. On the other hand, the *Architecture-layer infrastructure* uses *gauges* that aggregate data coming from the *probes* and update the properties in the architectural model. These properties are handled and provided by the *model manager*, and a *constraint evaluator* checks the model periodically, triggering actions in case a constraint violation occurs. After that, the *adaptation engine* determines the system adaptation needs.

Although Rainbow supports the development of generic controllers, Rainbow-based controllers are typically monolithic since all their components are used as a whole. They can be configured, but not changed (for example, to incorporate new services).

2.1.1 Acme and Stitch Languages. Acme is an architectural language created by Garlan et al. [21] to describe structures, properties, constraints and styles for software architectures. Structures provide the organisation of the system based on its constituent parts. Properties provide information about the system and its parts that allow abstract reasoning about overall behaviour (both functional and nonfunctional). Constraints provide guidelines for how the architecture can change over time. Lastly, Styles defines classes and families of architecture. Figure 2 illustrates the Acme elements in which *System* represents the entire abstraction of a particular system, *Component* represents particular parts of the system, *Connector* represents the relationship between components, and *Role/Port* represent the interface between the components. These organisation parts are illustrated in the example in Figure 3. In this work, Acme is used as the architectural language to allow the systems to adapt.

Stitch is a language created by Cheng and Garlan [13] that is used to define adaptations *strategies* as decision trees built up from adaptation *tactics*, which are in turn defined in terms of primitive *operators*. This language takes into consideration the current system state, applicability conditions of individual strategies, their expected costs and benefits, and prior history of strategy invocation. *Operator* is the most primitive unit of execution for an

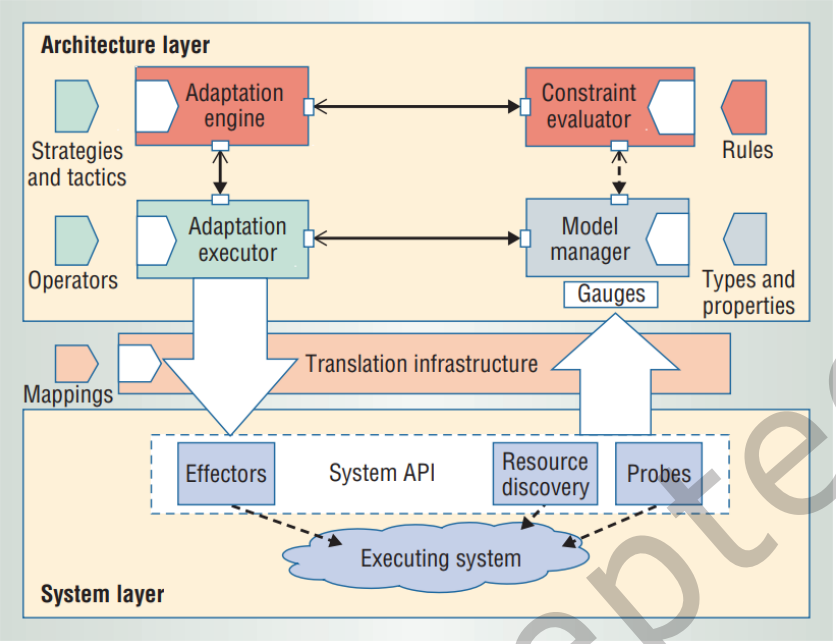


Fig. 1. Rainbow architecture [20].

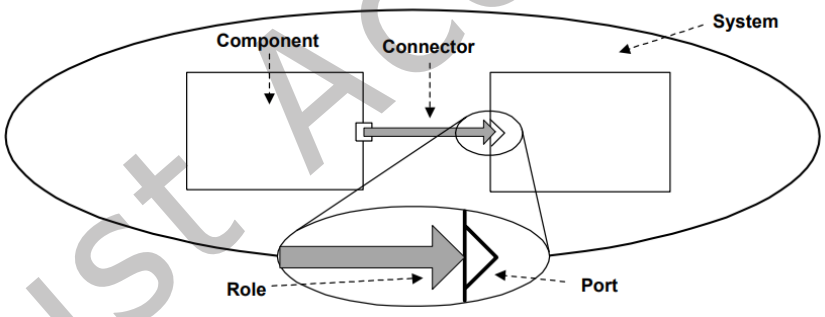


Fig. 2. Acme Elements [21].

adaptation process, representing a basic configuration command provided by the target system. *Tactics* provide an abstraction that packages operations into larger units of change to steps of adaptation used by strategies; providing logical units for specifying the cost-benefit impact of an adaptation step concerning quality dimensions of the system. *Strategies* encapsulate a dynamic adaptation process in which each step is the conditional execution of some tactic. A strategy is also characterised as a tree of condition-action-delay decision nodes, with explicitly defined probability for conditions and a delay time-window for observing tactic effects. An example of stitch code is illustrated in Figure 8. In this work, Acme and Stitch are used together. The system architectures are defined using Acme, and these architectures represent the system state for Stitch strategies to be able to work.

```
System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc  = { Roles {caller, callee} }
  Attachments : {
      client.sendRequest to rpc.caller ;
      server.receiveRequest to rpc.callee }
}
```

Fig. 3. Acme Example [21].

2.2 Docker and Kubernetes

The Docker[1] platform automates the packaging, deployment and execution of applications. These applications are also named docker *containers*. They are based on the client-server architectural style. For each application, REST APIs are defined to allow the applications to be accessed.

Kubernetes[2] is an open source-based container orchestration platform for automating deployment, scaling, and management of containerised applications [9]. In Kubernetes, the main object is a *pod*, which aggregates a set of one or more containers. These pods can use a unique IP and ports to perform communication between themselves and shared storage. Any application can be deployed in a Kubernetes cluster provided all its dependencies are specified in a configuration file. In other words, a Kubernetes cluster allows one to define a complete software ecosystem composed of several sets of applications (*i.e.,* pods) that communicate with one another. To support the pods, Kubernetes provides resources such as the *service*. The service helps the developer focus on development applications to be deployed in Kubernetes Clusters without demanding the management of IP addresses and resolution of names. Kubernetes lets one deploy microservices that are able to make global decisions, such as resource allocation, health verification, recovery, and self-scaling of Kubernetes pods [9].

2.3 Kubow

Kubow[3] is the implementation of Rainbow in Kubernetes [2]. As such, a Kubow instance (*i.e.,* a controller that is based on Rainbow) is also defined as a microservice.

In general, Kubow was implemented by customising and extending Rainbow with support for docker containers and Kubernetes [2]. These customisations and extensions involved modifications, such as Rainbow probes and effectors using the Kubernetes APIs. These APIs allow access to different types of resources managed by Kubernetes (*e.g.,* pods). Consequently, all Rainbow components are Kubow instances within a Kubernetes cluster. Moreover, Kubow provides tools for integrating other kinds of Kubernetes applications. For example, access metrics are collected by Kubernetes' monitoring services (*e.g.,* Kubelet and cAdvisor), and other external monitoring tools (*e.g.,* Prometheus).

Figure 4 depicts the three main modules representing the architecture. The *Kubernetes master* module comprises native resources from Kubernetes, proving artefacts, APIs and the basic infrastructure for running applications. The *Kubernetes extensions* module comprises tools, APIs and other resources that are aggregated to the environment to provide mechanisms for control loops (*e.g.,* monitoring). Finally, the *Kubow* module contains all the Rainbow components customised for working with Kubernetes.

[1]https://docs.docker.com/get-started/overview/ – accessed in December 2024.
[2]https://kubernetes.io/ – accessed in December 2024.
[3]https://github.com/ppgia-unifor/kubow – accessed in December 2024.

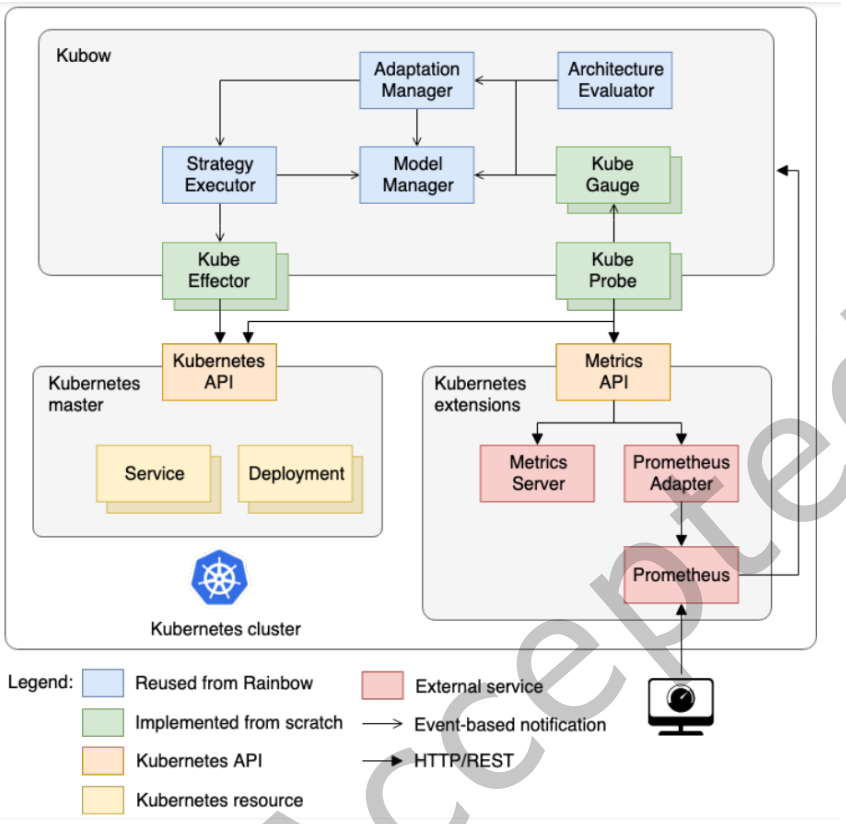


Fig. 4. Kubow architecture [2].

2.4 APIs and Communcation

In this work, all the applications (*e.g.*, meta-controller, micro-controllers etc.) are structured and deployed as microservices. The different components of the system were designed to use Prometheus[4] as a central component to facilitate communication between the different applications. The applications are deployed as microservices within a Kubernetes cluster, which ensures that each service can interact seamlessly with the others. This influences mechanisms such as service discovery features to maintain the high availability of the microservices architecture. In addition, the microservices (to which interactions happen through REST APIs) are also used with tools involving the Acme and Stitch languages.

Prometheus is employed to monitor and manage the interactions among the microservices. Each microservice that uses APIs within the system is configured to collect and expose metrics,[5] which Prometheus then scrapes at regular intervals. This approach not only enables real-time monitoring of the system's health and performance but also allows the APIs to query Prometheus for specific metrics and data related to the operations of other microservices. Through this integration, the microservices can retrieve comprehensive information about the state and behaviour of the entire system. This includes data on resource usage, request rates, error rates, and

[4]https://prometheus.io/ – accessed in December 2024
[5]https://github.com/kubernetes-sigs/metrics-server – accessed in December 2024

other critical metrics. By having access to this information, each microservice can make informed decisions and adapt its behaviour based on the current state of the system, leading to improved efficiency and resilience. The communication that occurs within the different microservices in Kubow [2] is entirely based on Prometheus. As an example of using Prometheus as a global communication tool among the microservices, the FailureManager micro-controller (whose details are presented later in this article) is instrumented with Prometheus to generate messages whenever a CPU failure is detected within the Kubernetes cluster.

In this article, Section 3 describes different levels and layers. As an example, a higher layer (*e.g., meta-controller*) uses APIs and Prometheus to perform changes in a lower layer (*e.g., controller*). These levels, layers and APIs are organised and structured using the Acme and Stitch specifications to perform architectural adaptations in the system.

3 APPROACH

Our approach considers multi-layered controllers for promoting reuse at development-time and structural flexibility at operational-time. For promoting controller reuse, our approach combines generic controllers, that can be easily configured, and whose role is to manage a collection of specific controllers. Structural flexibility is achieved by employing very simple and specific controllers that can be replaced and orchestrated by the generic controller.

This paper focuses on demonstrating the structural flexibility of multi-layered controllers. The idea of multi-layered controllers is not new [8, 27, 40], and it fits quite well into the hierarchical control pattern [59]. We promote multi-layered controllers to decouple the controller from the target system because of the target system's intrinsic intricacies and complexities. The tight coupling between the target system and the controller can impede the reuse of controllers across different applications, as systems offer varied services with differing quality levels.

A self-adaptive system is composed of a target system and a controller, and it should be considered in the context of its environment. The loci of change are the target system and its environment (*i.e.,* external phenomena related to other systems and humans). The locus of adaptation is the controller, which empirically observes changes to build reasoning models that enable it to control the target system. Adaptations can either be parametric or structural [3, 35].

The proposed approach for decoupling the controller from the target system is to introduce a two-layer controller, as shown in Figure 5. The controller layer is a structurally flexible controller that should be able to adjust quite easily to the needs of the target system. The meta-controller layer is a generic controller, like Rainbow [20], that should be able to control the controller layer by adapting its structure and parameters. Although parametric adaptations of monolithic controllers are more common [32], structural adaptations are also possible, but these may require re-evaluation and redeployment of the controller. In this context, the main idea being promoted in this paper is that controllers do not need to be structurally static at operational-time. Controllers can both be the locus of change *and* the locus of adaptation.

3.1 Micro-controllers

The key idea being promoted is to use an ensemble of micro-controllers instead of a monolithic controller, similar to an implementation of a MAPE-K loop [20, 31].

These micro-controllers would not be limited to the services provided by the individual stages of a MAPE-K loop [41]. Instead, the micro-controllers would be associated with specific services that make up the individual stages of a controller [16]. For example, for the Analysis stage of the MAPE-K loop, a micro-controller would implement the services associated with integration testing [53], or model checking [50]. The number of micro-controllers for implementing a controller would depend on the needs of a target system, and the granularity of

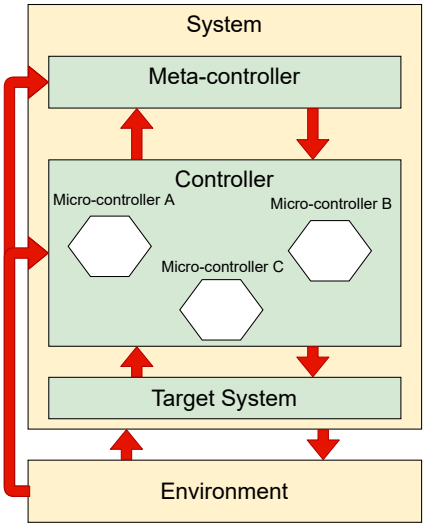


Fig. 5. Multi-layered controller.

the available micro-controllers in terms of the services they provide. Therefore, in our proposed approach, a controller for a self-adaptive system would be implemented as an ensemble of service-specific micro-controllers.

These micro-controllers can either be open or closed-loop. An open-loop micro-controller requires predefined knowledge about the specific target system. In contrast, a closed-loop micro-controller dynamically takes the needs of the specific target system into account (*e.g.,* a closed-loop micro-controller for online testing can refine its strategy if the achieved test coverage of the target system is insufficient). This suggests that while a closed-loop micro-controller needs to be configured with generic techniques (*e.g.,* to explore various testing strategies), an open-loop micro-controller requires a target system-specific configuration (*e.g.,* a testing strategy suited for the specific target system).

Micro-controllers can be implemented using different software technologies that can orchestrate and choreograph execution flows. In the context of controllers that need to be adaptable at run-time and evolvable during development time, microservices offer a modular and scalable architecture that enables dynamic updates, seamless integration of new functionalities, and improved maintainability without disrupting ongoing operations. That said, we highlight that each micro-controller can be developed as a separate process for maximising independent deployment. Implementing controllers as a collection of micro-controllers offers several benefits, including independent development and deployment, and dynamic operational support, such as versioning and scaling. In this context, concerning the granularity of the micro-controllers, one particular micro-controller can embed (a) a given MAPE-K stage; (b) a full MAPE-K control loop for an adaptive property of the target system; or even (c) a very fine-grained component that is responsible for a specific activity within a controller. In the exploratory study we present in Section 4, the granularity of the designed micro-controllers follows option (b), that is, the micro-controllers embed each adaptive property of the target system. Nevertheless, each micro-controller in our exploratory study could have been divided into smaller micro-controllers, thus matching either option (a) or (c) as well.

3.2 Meta-controller

If the controller is realized as an ensemble of micro-controllers and serves as both the locus of adaptation and the locus of change, there may be a need for an additional higher-level controller, depending on the system's complexity. This *meta-controller* would manage the changes occurring at the controller by adapting the ensemble of micro-controllers. An example of such a controller could be Rainbow [20], or any of its variants [2].

In the context of self-adaptive software systems, the tailoring of a meta-controller to different ensembles of micro-controllers would be simpler because the 'target system' of the meta-controller would be just a collection of micro-controllers, instead of a wide range of software components that usually characterises a target system.

The meta-controller orchestrates the services provided by micro-controllers that implement the controller. It should maintain a consistent view that micro-controllers have of the target system and environment [39], including the state of the micro-controllers, allowing them to be stateless. Although micro-controllers should be independent, the coordination between micro-controllers could follow the control flow of a MAPE-K loop [31]. Several micro-controllers do not preclude that all micro-controllers can access the target system. Conflicts might occur amongst micro-controllers and should be handled by the meta-controller.

In summary, the proposed multi-layered controller approach is beneficial because it addresses the different needs of the target system with specific micro-controllers. This is achieved because of the structural flexibility of the micro-controller ensemble, which is managed by a meta-controller. Thus, the controller consisting of an ensemble of micro-controllers reflects on changes affecting the target system and environment. In contrast, the meta-controller reflects on changes affecting the controller and the system environment, which makes the overall system meta-self-adaptive. Although this paper describes a multi-layer controller consisting of two layers, conceptually, a controller could have more than two layers [40].

To achieve a modular and structurally flexible approach, the micro/meta-controller design emphasizes loosely coupled components that implement specific services, providing the necessary structural independence. An additional advantage of this approach, particularly when contrasted with monolithic designs, is the ability to develop controllers with low coupling and high cohesion. Such components are inherently more adaptable and have a greater likelihood of being assembled and reused without requiring substantial modifications, enhancing both development efficiency and system scalability.

4 EXPLORATORY STUDY

This section describes the implementation of our multi-layer controller approach. First, in Section 4.1, we present the controller configurations employed for evaluating our approach that was described in Section 3. Then, in Section 4.2, we describe the exploratory study. In the study, we explore Kube-ZNN [1], which is a containerised deployment of the ZNN.com system [14] that delivers multimedia news content to its clients. Finally, in Section 4.3, we describe the environmental setup used for performing the experiments. The artefacts used in our research are available online[6].

4.1 Controller Configurations

To demonstrate and evaluate our approach, we have employed three functionally equivalent controllers configurations (named Mon-KZ, Des-KZ, and Meta-KZ) which are representative of existing solutions. Before we provide details of the three configurations, we describe basic concepts of ZNN.com [14] and Kube-ZNN [1] (more details of ZNN.com and Kube-ZNN are provided in Section 4.2). In ZNN.com and Kube-ZNN, a *server replica* (or simply a *server*) represents an instance of a web server that delivers media contents as requested by the users. A server is *active* when it is operational (*i.e.,* it is responding to user requests). *Scalability* is an adaptive property of the system that allows for the addition or removal of server replicas when the rate of user requests increases

[6]https://github.com/californi/ExperimentingKube-TAAS2023

or decreases. *Fidelity* is another adaptive property of the system that either increases or reduces the quality of the media (which is associated with delivered news content) as a way to help the system fulfil the service demand. The scalability and fidelity properties are adapted by a managing system which relies on the Rainbow infrastructure [20].

Figures 6(a), 6(b), and 6(c) illustrate the structures and the relationship between the main components of the Mon-KZ, Des-KZ and Meta-KZ configurations. In addition, a fourth configuration, which is named Meta-KZ-New and is an evolution of Meta-KZ, is described to demonstrate the flexibility of the micro-controller-based approach. Meta-KZ-New is shown in Figure 6(d).

Figure 7 shows a detailed view of the micro-controllers that compose each configuration. In particular, the variants of the Scalability and Fidelity micro-controllers, as well as the Meta-Controller, have common internal structures (which are realised as Kubow controllers, as explained later in this section). The FailureManager and FailureHandler micro-controllers, on the other hand, have tailor-made internal structures (not based on Kubow). Details of each configuration are next provided.

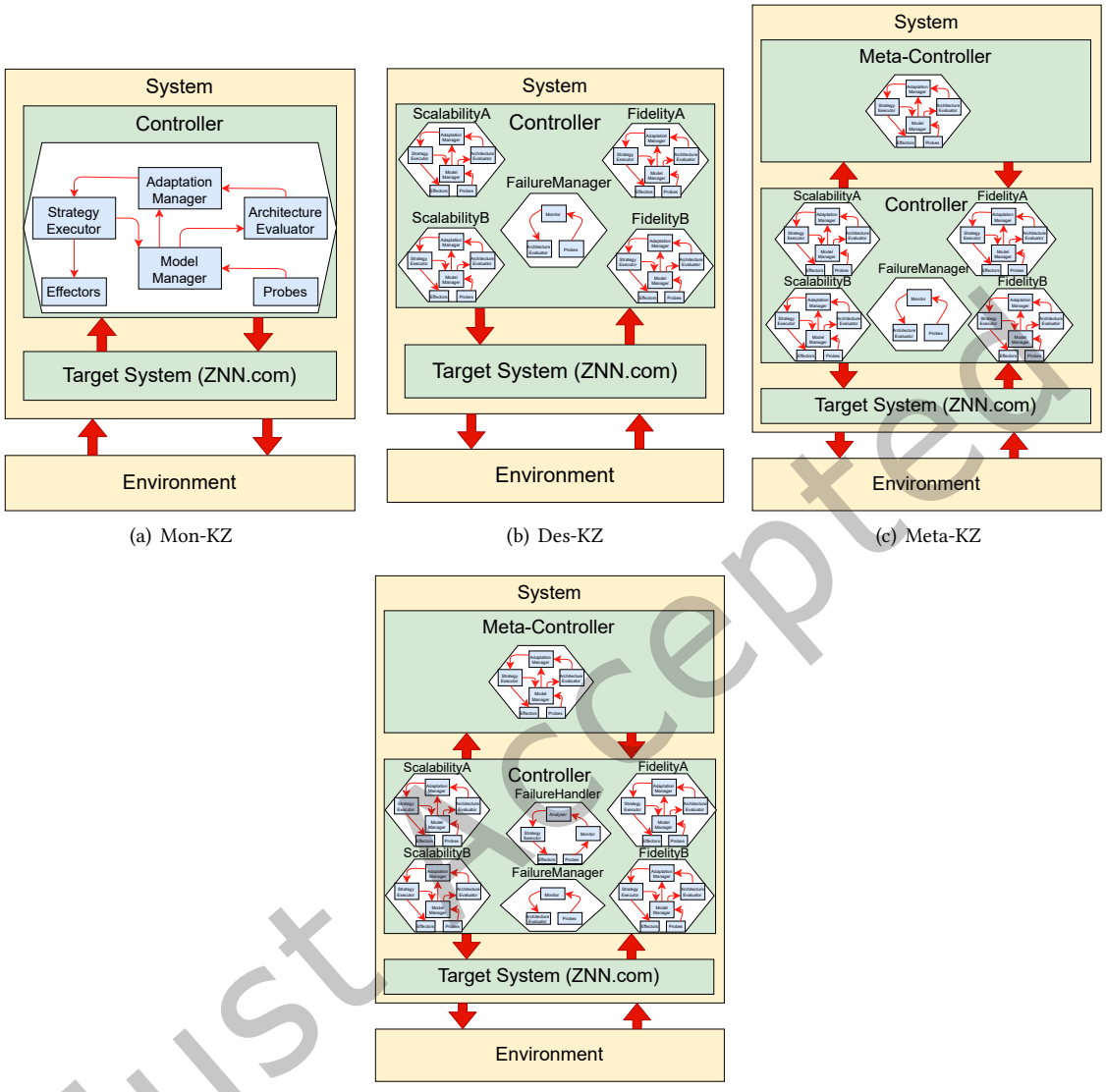
Mon-KZ: This configuration is identified as a monolithic controller since a single controller implements all the adaptation strategies and tactics. It is an evolution of the original implementation by Aderaldo et al. [2] to include the new set of strategies and tactics to deal with the number of failures in the server replicas. As shown in Figure 6(a), Mon-KZ includes a single Kubow instance (hexagon labelled with Controller).

The strategies that take into account the variations in the failure rate aim to adapt the behaviour of the controller for dealing with a higher or lower number of active servers in the target system (*i.e.,* the adaptations concerning the scalability property), and higher or lower quality of media delivered to the clients (*i.e.,* the adaptations concerning the fidelity property).

Des-KZ: This is a decentralised implementation of the Mon-KZ configuration, as shown in Figure 6(b). It consists of five micro-controllers. Four of these micro-controllers are implemented as Kubow instances (namely, ScalabililyA, ScalabililyB, FidelityA and FidelityB). The remaining one (namely, the FailureManager), on the other hand, is tailor-made (that is, it is not a Kubow instance) as a way of demonstrating the possibility of integrating heterogeneous controller designs. The different adaptation strategies for Mon-KZ have been reproduced by using decentralised variants. In other words, four adaptation strategies in Mon-KZ have turned into four different micro-controllers; the micro-controllers of Des-KZ that deal with scalability and fidelity have two variants each, as follows: ScalabililyA and FidelityA, respectively, deal with a high number of active servers and high media quality; their variants, ScalabililyB and FidelityB, handle lower numbers of servers and reduced media quality.[7] Specific adaptation strategies guarantee only a single variant of each micro-controller is active at a particular moment, depending on the failure rate gathered by the FailureManager micro-controller. Note that the failures considered in our implementations concern Kube-ZNN failures related to the lack of CPU resources; despite that, the FailureManager micro-controller could be easily tailored to handle other kinds of failures. Details of the target system quality attributes addressed in our implementations, including the number of CPU failures, are described further in Section 4.2.

Meta-KZ: This configuration includes a meta-controller for managing the micro-controllers, as shown in Figure 6(c). Differently from Des-KZ, in which the choreography of micro-controllers is responsible for adaptations at the controller layer, in the Meta-KZ configuration the meta-controller orchestrates adaptation. Depending on the state (*i.e.,* the failure rate) gathered by the FailureManager micro-controller, the Meta-controller – which is also implemented as a Kubow controller – selects which Scalability and Fidelity micro-controller variant to activate. The strategies to adapt the controller are listed in Table 2.

[7] In our experiments, which are described in Section 5, we set ScalabitityA and ScalabitityB to operate with a maximum of 10 and 4 active servers, respectively. For FidelityA and FidelityB, the sizes of media range from 400 to 800 KB, and from 20 to 200 KB, respectively. Note that the inferior number of active servers and the inferior media quality are temporary; they hold until the moment the system restores its default operational capacity. When this happens, the controller is reconfigured to have ScalabilityA and FidelityA active again.



(a) Mon-KZ (b) Des-KZ (c) Meta-KZ

(d) Meta-KZ-New

Fig. 6. Architectural configurations. In parts (a), (b), (c), and (d): yellow rectangles refer to more general parts of the context (in this case, System and Environment); green rectangles refer to the particular contexts of the target system and the managing system; white hexagons refer to components that have been deployed in microservices; blue rectangles refer to components and classes of particular stages of MAPE-K, or to functionalities important to be used in a control loop; red thick outer arrows refer to external communication (*e.g.*, usage of APIs) performed between the target system and the controller; and red thin inner arrows refer to internal communication (*e.g.*, usage of APIs or methods calls) performed between the internal components.

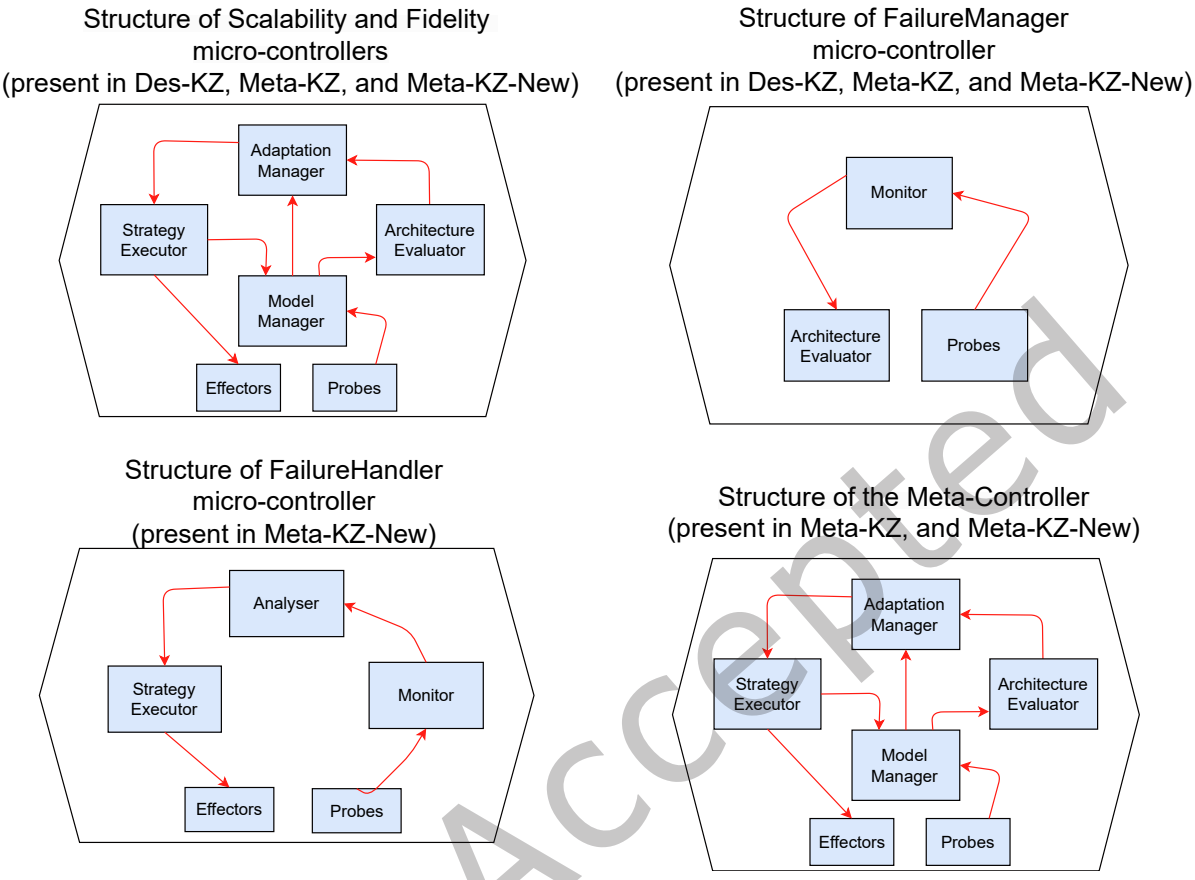


Fig. 7. Detailed view of the micro-controllers and meta-controller.

Meta-KZ-New: This configuration is an evolved version of the Meta-KZ configuration for dealing with CPU failures. Meta-KZ-New includes all features designed for Meta-KZ. Besides this, at the controller layer of Meta-KZ-New there is a new component, named FailureHandler and shown in Figure 6(d), which is responsible for identifying high failure rate situations. After identifying the rates of failures, decisions are made concerning the disposal of unavailable resources from the cluster (which, in terms of Kubernetes, consists of deleting Kube-ZNN server pods that do not become operational due to limited CPU availability). The strategy to deal with CPU failures in Meta-KZ-New is listed in the last line of Table 2.

The motivation for introducing this additional micro-controller is to demonstrate the structural flexibility of a controller architecture based on micro-controllers. The Meta-KZ-New is a prime example of a specialised micro-controller that seamlessly integrates additional services into the controller without causing disruptions during its incorporation or altering the behaviour of other micro-controllers.

Decomposing Mon-KZ to create Des-KZ and Meta-KZ: The process for decomposing the monolithic controller to build decentralised micro-controllers (for Des-KZ and Meta-KZ), as well as to build a meta-controller (for Meta-KZ) relied on extracting micro-controllers directly from the Rainbow Stitch strategies. Figure 8 shows Stitch code snippets illustrating the decomposition. For instance, the conditional sentence `t0` in Figure 8(a) has been extracted

into a variant of the scalability micro-controller, as illustrated in Figures 8(b) and 8(d). Specifically, Figure 8(a) shows the `ImproveSlo` strategy implemented in Mon-KZ; it combines adding server replicas and lowering media quality when user requests are high and requests are lost. Figure 8(b) shows the same `ImproveSlo` strategy for the ScalabilityA micro-controller in Des-KZ, which in that case only focuses on adding replicas. Figure 8(b) also shows the `activateScalabilityB` strategy that activates the variant ScalabilityB when the failure rate starts to be different from NoFailureRate (*i.e.,* the rate starts to increase). In Meta-KZ, the management of the variants of specific micro-controllers is concentrated in the meta-controller. This is illustrated in Figure 8(c); such strategies are responsible for activating the variants of the scalability and fidelity micro-controllers. Finally, Figure 8(d) shows that the ImproveSlo strategy implemented in the ScalabilityA micro-controller of Meta-KZ is the same as in Des-KZ, but the strategy for activating the variant of the micro-controller is placed in the meta-controller. In particular, the meta-controller replaces ScalabilityA, which operates with a maximum of 10 active servers, with ScalabilityB, which operates with a maximum of four active servers. Note that all functionalities that were implemented in Mon-KZ also were implemented in the Des-KZ and Meta-KZ configurations. As an example, Mon-KZ has two adaptation strategies to deal with scalability with four and ten servers. In particular, for Des-KZ, we created one variant for four servers and another variant for ten servers.

```
strategy ImproveSlo [ sloRed ] {
 t0: (sloRed && canAddReplica) ->
       addReplica() @[20000 /*ms*/]
       { t0a: (success) -> done; }
 t1: (sloRed && !canAddReplica) ->
       lowerFidelity() @[20000 /*ms*/]
       { t1a: (success) -> done; }
 t2: (default) -> TNULL;
}
```

(a)

```
strategy ImproveSlo [ canAddReplica
                      && sloRed ] {
t0: (sloRed && canAddReplica) ->
       addReplica() @[20000 /*ms*/]
       { t0a: (success) -> done; }
 t1: (default) -> TNULL;
}

strategy activateScalabilityB [
    LowFailureRate || HighFailureRate ] {
 t0: (LowFailureRate || HighFailureRate) ->
       addLowScalability()
       { t0a: (success) -> done; }
 t1: (default) -> TNULL;
}
```

(b)

```
strategy activateNoFailureRate
                  [ NoFailureRate ] {
 t0: (NoFailureRate) ->
  addHighScalabilityHighQuality()
                    @[40000 /*ms*/]
  { t0a: (success) -> done; }
 t1: (default) -> TNULL;
}

strategy activateLowFailureRate
     [ LowFailureRate && IsScalabilitya ] {
 t0: (LowFailureRate) ->
  addLowScalabilityHighQuality()
                    @[40000 /*ms*/]
       { t0a: (success) -> done; }
 t1: (default) -> TNULL;
}
```

(c)

```
strategy ImproveSlo [
    canAddReplica && sloRed ] {
 t0: (sloRed && canAddReplica) ->
     addReplica() @[20000 /*ms*/]
     { t0a: (success) -> done; }
 t1: (default) -> TNULL;
}
```

(d)

Fig. 8. Stitch code snippets that illustrate the differences of strategy implementations in the controllers for Mon-KZ (a), Des-KZ (b), and Meta-KZ (c and d).

We highlight the decomposition process as a key technical challenge to transform a monolithic controller into a decentralised one. One of the primary difficulties was the testing and comparison of different versions of the newly formed components, that is, the micro-controllers. In a monolithic architecture, the testing process is relatively straightforward, as all functionalities reside within a single codebase. However, by using a microservices architecture, each service must be tested independently, as well as in conjunction with other services, to ensure

seamless integration and functionality. This fragmentation required the employment of a complex testing strategy to validate the correctness and performance of each component individually and collectively.

The characteristics of choreography and orchestration: the Mon-KZ configuration was decomposed into two distinct versions (*i.e.,* Des-KZ and Meta-KZ) to explore different communication paradigms: choreography and orchestration. Des-KZ was designed using a choreography-based communication model. In this approach, each component is aware of its own state and the state of only one other component (*e.g.,* ScalabilityA is aware of ScalabilityB and vice versa). This limited visibility enables each component to make autonomous decisions based on the state information of the other component, resulting in a decentralised and loosely coupled interaction pattern. This model emphasises localised decision-making and reduces dependencies on a central coordinator, thereby promoting flexibility and resilience in the system's operations. Table 2 presents the rules responsible for doing the Des-KZ choreography, implemented as Stitch strategies. Note that, as an example, the [FidelityB] micro-controller has the strategies [ActivateFidelityA] and [DeactivateFidelityA] associated to it, likewise the strategies [ActivateFidelityB] and [DeactivateFidelityB] are associated to the [FidelityA] micro-controller.

In contrast, the Meta-KZ employed an orchestration-based communication model. Thus, the components (*e.g.,* [ScalabilityA] and [ScalabilityB]) are entirely unaware of each other's states. Instead, a new component (*i.e.,* the meta-controller) is introduced to manage the interactions. The meta-controller possesses complete visibility over the states of all other components, orchestrating their behaviours by performing the necessary state readings and modifications. This centralised coordination facilitates a more structured and controlled interaction pattern, enabling a more straightforward implementation of complex workflows and ensuring a higher degree of consistency across the system.

Architectural Elements and Required Resources for each Configuration: Figure 6 illustrates the various architectures that we developed in this work. Regarding the semantics of the elements used, the yellow rectangular shape refers to a more general part of the context, in this case, System and Environment. The green rectangular shape refers to the particular context of the target system (*i.e.,* ZNN.com) and the managing system (*i.e.,* the Controller). The white hexagon shape refers to components that have been deployed in microservices. The blue rectangular shape refers to components and classes of particular stages of MAPE-K, or to functionalities important to be used in a control loop. The red thick outer arrows refer to external communication (*e.g.,* usage of APIs) performed between the target system and the controller. Finally, the red thin inner arrows refer to internal communication (*e.g.,* usage of APIs or methods calls) performed between the internal components.

The two last columns of Table 2 summarise, for each architectural configuration, which controller component is responsible for implementing the adaptation strategies (refer to Figure 6). Since decentralisation tends to increase the number of pods, Table 1 captures the number of pods associated with each configuration. For the evaluation of different configurations, this information is important to understand the impact of decentralisation. In particular, regarding the Kube-ZNN target system, the number of pods may vary from 1 to N. In our experiments, we defined the upper limit of 10 (more details about this in Section 5.1). As Table 1 summarises, the other components require one pod each, except FailureManager which requires two pods. One pod is responsible for identifying the CPU inconsistency error; the other pod is responsible for registering the findings via Prometheus to make the data available for the entire system. Regarding how much resource is demanded for each pod, the first two columns in Table 1 summarise such usage in terms of CPU and memory. These amounts of resources were defined based on the original Kubow implementation [2]. In fact, Table 1 only presents a minimal estimate; variations are possible during the execution in a Kubernetes cluster. The units used for representing the allocation are milliCPUs (m) and Gigabytes (Gi). In the CPU column, 1000m represents a full CPU allocation and 250m represents 1/4 of a CPU, whereas in the Memory column, 100mi represents 100 megabytes. The ranges of the total required memory and CPU for each configuration are listed in the last two lines of Table 1.

Timing Setup for Control Loops: We established the timing for the control loops of the three configurations (Mon-KZ, Des-KZ, and Meta-KZ) based on Meta-KZ. The hierarchical control model of Meta-KZ was expected to

Table 1. Architectural configurations and their resources.

Resource per component			Number of Kubernetes pods*			
CPU	Mem.	Component	**Mon-KZ**	**Des-KZ**	**Meta-KZ**	**Meta-KZ-New**
250m	100mi	Target System (Kube-ZNN)	[1 ... 10]	[1 ... 10]	[1 ... 10]	[1 ... 10]
1000m	1Gi	Controller	1			
1000m	1Gi	ScalabilityA		1	1	1
1000m	1Gi	ScalabilityB		1	1	1
1000m	1Gi	FidelityA		1	1	1
1000m	1Gi	FidelityB		1	1	1
200m	64mi	FailureManager		2	2	2
200m	64mi	FailureHandler				1
1000m	1Gi	Meta-controller			1	1
Range of Kubernetes pods			[2 ... 11]	[7 ... 16]	[8 ... 17]	[9 ... 18]
Range of Required Memory			[1.1Gi ... 2Gi]	[2.228Gi ... 3.128Gi]	[3.228Gi ... 4.128Gi]	[3.292Gi ... 4.192Gi]
Range of Required CPU units			[1250m ... 3500m]	[2650m ... 4900m]	[3650m ... 5900m]	[3850m ... 6100m]

* Note that only one variant of a given component is included in a configuration at a particular moment. Therefore, the ranges above consider a single variant of micro-controllers for dealing with scalability and fidelity of the target system.

require longer cycles when compared to cycles for Mon-KZ and Des-KZ. On an empirical basis (that is, after several observations of the running system), we concluded that Meta-KZ requires 65 seconds and 45 seconds to allow both controllers (namely, the meta-controller and the controller, respectively) to fully perform their jobs (that is, to monitor their respective target systems and, eventually, adapt them). We consider these times as the *default* timing setup for the control loops.

4.2 ZNN.com and Kube-ZNN

Our study uses Kube-ZNN [1] as an infrastructure in which ZNN.com[8] executes and is considered part of the target system. Kube-ZNN is based on ZNN.com and reproduces a typical infrastructure for a news website [14]. The original ZNN.com (illustrated in Figure 9) is implemented in a three-tier architecture consisting of servers providing news content to clients (or, users, which are represented by the c_i elements in Figure 9). The goal of ZNN.com is to provide content with maximum quality to clients within a reasonable response time, while keeping the cost of the service within an operating budget. ZNN.com has a web-based client-server architecture that uses a load balancer (the lbproxy element in Figure 9) to deal with requests across a pool of replicated servers (the s_i elements in Figure 9). The number of servers can be adapted according to service demand.

In our experiment setup (described in Section 5), we have observed the media quality (*i.e.,* contents delivered to the clients), the size of the server pool while adapting the system according to the demand (*i.e.,* number of client requests), and response time. Our measurements regarding the size of the server pool, the quality of the media, and response time at particular moments (namely, right after the peak of demand, and right after the demand decrease) provide evidence of the system's capacity to accomplish its goals.

Quality Attributes: Three ZNN.com key quality attributes (which are measurable at run-time) are:

- **Scalability:** when a server becomes overloaded, new replicas (*i.e.,* copies) of the server are created. On the other hand, when a server replica ceases to be in demand, it can be destroyed. This attribute captures the elasticity of the system.

[8]https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/model-problem-znn-com/ – accessed in December, 2023.

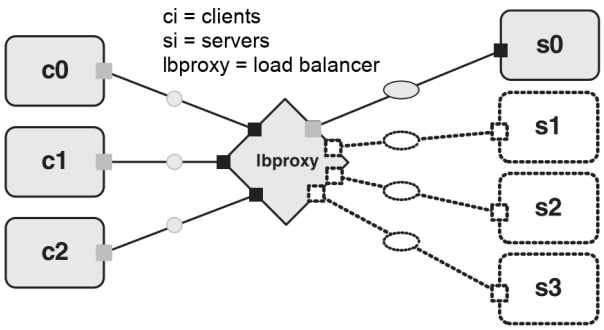


Fig. 9. ZNN.com system architecture [12].

- **Fidelity:** depending on the costs associated with service demand, different levels of fidelity (also referred to as *quality of media*) are provided for supporting client experience (*e.g.,* usage of different resources, such as *text*, *images*, or *videos*, implies different costs). For instance, if the maximum limit on the number of servers is reached, the servers start providing images instead of videos, as a way to properly fulfil the service demand. Like elasticity handles scalability, content fidelity can be increased when the service demand decreases.
- **Response Time:** it measures how long it takes for ZNN.com to respond to client demands. The intent is to adapt scalability and fidelity to keep the response time acceptable to the client.

In this paper, we have adopted Kube-ZNN [1], which is a new deployment of ZNN.com based on containerised applications, using Docker and Kubernetes. Each server in Kube-ZNN (*i.e.,* either a physical or virtual machine) is a pod in Kubernetes. The client requests are handled by a service, which is responsible for distributing and delivering requests at run-time according to the load.

Strategies: There are four adaptation strategies associated with the Kube-ZNN exploratory study:

- ImproveSlo-addReplica: it increases the number of servers available and relies on the addReplica tactic. Since there are more resources, the response time of Kube-ZNN decreases.
- ReduceCost-removeReplica: in contrast to ImproveSlo, ReduceCost decreases the number of servers available, and uses the removeReplica tactic. Thus, the response time of Kube-ZNN tends to increase when the number of user requests is increased.
- ReduceCost-raiseFidelity: it improves the media quality provided by the target system.[9] In Kube-ZNN the delivered media quality is normally high, but the response time may increase when the number of requests increases.
- ImproveSlo-lowerFidelity: in contrast to the previous strategy, lowerFidelity ensures that the response time is kept at an acceptable level by reducing the media quality when the number of client requests starts increasing the response time of Kube-ZNN .

Additional quality attribute: in this paper, we introduce a third quality attribute:

- **Number of Failures**: Depending on the number of failures affecting the server replicas of Kube-ZNN, different strategies can be selected for improving the resource allocation to mitigate the high incidence of failures that may affect the goals of the target system. In other words, the number of failures becomes an observable variable that can influence our adaptation strategies.

[9]Note that the quality of the media is directly related (and proportional to) the size of the media, *e.g.,* , in Kbytes.

Resources like CPU, memory and storage are associated with Kubernetes clusters. When resources are not available, a failure is triggered by Kubernetes. For example, when a pod is being created, Kubernetes services verify which resources are available to be used. During this creation, if no CPUs are available, then a CPU failure is raised. The same occurs with memory and storage, once a pod reaches the minimal requirements. All the raised failures may be handled by the FailureManager, even though in this exploratory study we focus on CPU failures, which are handled by using the Kubernetes API.

Additional strategies: Based on the number of failures attribute, three additional strategies are defined:

- ActivateNoFailureRate: if there are no server failures, Kube-ZNN provides high scalability and fidelity.
- ActivateLowFailureRate: if the server failure rate is low, scalability should be low, but the fidelity can still be high.
- ActivateHighFailureRate: if the server failure rate is high, Kube-ZNN should provide low scalability and fidelity.

In particular the *ActiveHighFailureRate*, when this strategy is activated, there are also mechanisms able to perform adaptations to improve the usage of the system resources.

Table 2 shows the adaptation strategies and their respective tactics and predicates defined for our Kube-ZNN exploratory study. For the execution of a strategy, its associated predicate must be satisfied. For example, for the execution of ImproveSlo, and its addReplica tactic, both variables lowSLO and canAddReplica must be true. The lowSLO variable shows whether the number of user requests lost is high, whereas canAddReplica shows when the limit of servers is reached. The last column of the table lists the components in charge of managing (*e.g.,* triggering) the adaptation.

The predicates that take into account the number of CPU failures are defined as follows:

- noFailureRate: it is *true* when there is no CPU failure for 30 seconds.
- lowFailureRate && isScalabilityA: it is *true* when the failure rate is 0.5 for 30 seconds, and the micro-controller ScalabilityA is activated.
- highFailureRate && isScalabilityA: it is *true* when the failure rate is 1.0 for 30 seconds, and the micro-controller ScalabilityA is activated.

Note that for the decentralised configuration (*i.e.,* Des-KZ), we derived eight specific strategies that are embedded in the micro-controllers of that configuration. It was necessary to implement the choreography among micro-controllers at the controller level in the decentralised configuration. We highlight that those specific strategies capture precisely the rationale embedded in the strategies ActivateNoFailureRate, ActivateLowFailureRate, and ActivateHighFailureRate. The strategies enable the micro-controllers to adapt based on the current rate of failures affecting the server replicas. Those specific strategies are listed in the last lines of Table 2 (except the last one, which concerns the FailureHandler micro-controller). To illustrate a strategy associated with the number of failures attribute, the ActivateLowFailureRate strategy concerns situations in which CPU failures started to happen (even if not in high number) but the quality of the media may still be kept high. Therefore, the ScalabilityA micro-controller is replaced with ScalabilityB and, as such, the system will temporarily operate with fewer active servers until the number of failures decreases. Note that the tactics related to the adaptation strategies that trigger the reconfiguration of the controller have associated tactics with self-explanatory names. Examples are the addLowScalability and removeLowScalability tactics; they respectively concern the replacement of ScalabilityA with ScalabilityB and vice versa. Similar rationale applied to other tactics such as addHighScalabilityHighQuality, addLowScalabilityLowQuality, addLowQuality, and removeLowQuality. In addition, whenever the controller needs to let the target system operate with fewer active servers, the strategy AdjustDefaultReplicas is triggered to decrease the servers until the number of failures decreases.

From Table 2, it is important to mention that the strategies concerning the Number of Failures attribute relate to characteristics of control and meta-control layers. Moreover, note that predicates are associated with a particular

Table 2. Rules in the adaptation strategies.

Stitch				Architecture	
Attribute	Strategy	Tactic	Predicate	Config.	Component in charge
Scalability	ImproveSlo	addReplica	lowSLO && canAddReplica	Mon-KZ	Controller
				Des-KZ	ScalabilityA, ScalabilityB
				Meta-KZ	ScalabilityA, ScalabilityB
Scalability	ReduceCost	removeReplica	highSLO && canRemoveReplica	Mon-KZ	Controller
				Des-KZ	ScalabilityA, ScalabilityB
				Meta-KZ	ScalabilityA, ScalabilityB
Scalability	AdjustDefaultReplicas	adjustReplicas	desiredReplicas > maxReplicas;	Mon-KZ	Controller
				Des-KZ	ScalabilityB
				Meta-KZ	ScalabilityB
Fidelity	ReduceCost	raiseFidelity	highSLO && lowFidelity	Mon-KZ	Controller
				Des-KZ	FidelityA, FidelityB
				Meta-KZ	FidelityA, FidelityB
Fidelity	ImproveSlo	lowerFidelity	lowSLO && cannotAddReplica	Mon-KZ	Controller
				Des-KZ	FidelityA, FidelityB
				Meta-KZ	FidelityA, FidelityB
Failures	ActivateNoFailureRate	addHighScalabilityHighQuality	noFailureRate	Mon-KZ	Controller
				Meta-KZ	Meta-controller
Failures	ActivateLowFailureRate	activateLowScalabilityHighQuality	lowFailureRate && isScalabilityA	Mon-KZ	Controller
				Meta-KZ	Meta-controller
Failures	ActivateHighFailureRate	addLowScalabilityLowQuality	highFailureRate && isScalabilityA	Mon-KZ	Controller
				Meta-KZ	Meta-controller
Failures	ActivateFidelityB	addLowQuality	highFailureRate	Des-KZ	FidelityA
Failures	DeactivateFidelityB	removeLowQuality	noFailureRate || lowFailureRate	Des-KZ	FidelityA
Failures	ActivateFidelityA	addHighQuality	noFailureRate || lowFailureRate	Des-KZ	FidelityB
Failures	DeactivateFidelityA	removeHighQuality	highFailureRate	Des-KZ	FidelityB
Failures	ActivateScalabilityB	addLowScalability	lowFailureRate || highFailureRate	Des-KZ	ScalabilityA
Failures	DeactivateScalabilityB	removeLowScalability	noFailureRate && (!canAddReplica || isScalabilityB)	Des-KZ	ScalabilityA
Failures	ActivateScalabilityA	addHighScalability	noFailureRate	Des-KZ	ScalabilityB
Failures	DeactivateScalabilityA	removeHighScalability	lowFailureRate || highFailureRate	Des-KZ	ScalabilityB
Failures	DisposeResourcesCpuAllocationError	removeInconsistentPods	highFailureRate	Meta-KZ-New	FailureHandler

strategy and tactic; it means that the adaptation would occur once the system state (which is observed through the evaluation of the predicates) is true. Given that some rules comprise predicates to deal with the control level, some micro-controllers and meta-controller deal with this information to make adaptation decisions (*e.g.*, replacing FidelityA with FidelityB either during a choreography or orchestration).

Figure 10 depicts a chart that illustrates one execution of the Meta-KZ configuration, showcasing the behaviour of the controller and target system. The execution is divided into two main phases (namely, load and unload), each lasting 8 minutes. During the load phase, 601 user requests are distributed. The unload phase, which starts after the load ends, is characterized by a waiting period with no new requests. The figure highlights the system's behaviour over time.

The chart consists of three key elements represented by lines:

(1) Blue line: shows the number of servers in Kube-ZNN during the load and unload periods, reflecting the controller and target system's adjustments to the overload caused by user requests.
(2) Green line: depicts the media quality maintained throughout the phases, illustrating how the controller ensures the quality of service.
(3) Dashed red line: represents the load of user requests, highlighting the active period of incoming demands.

These elements demonstrate how the target system adapts to load variations while maintaining media quality levels. Additionally, transparent rectangles in the graph indicate which system components (*i.e.*, , the micro-controllers) were operational at specific times.

Notably, the initial controller architecture includes the ScalabilityA and FidelityA micro-controllers. During the load phase, the meta-controller (which operates continuously) adapts the controller by replacing ScalabilityA and FidelityA with ScalabilityB and FidelityB, respectively, in response to the changing demands. By the end of the unload phase, the return of the FidelityA micro-controller is observed, as FidelityB is no longer required.

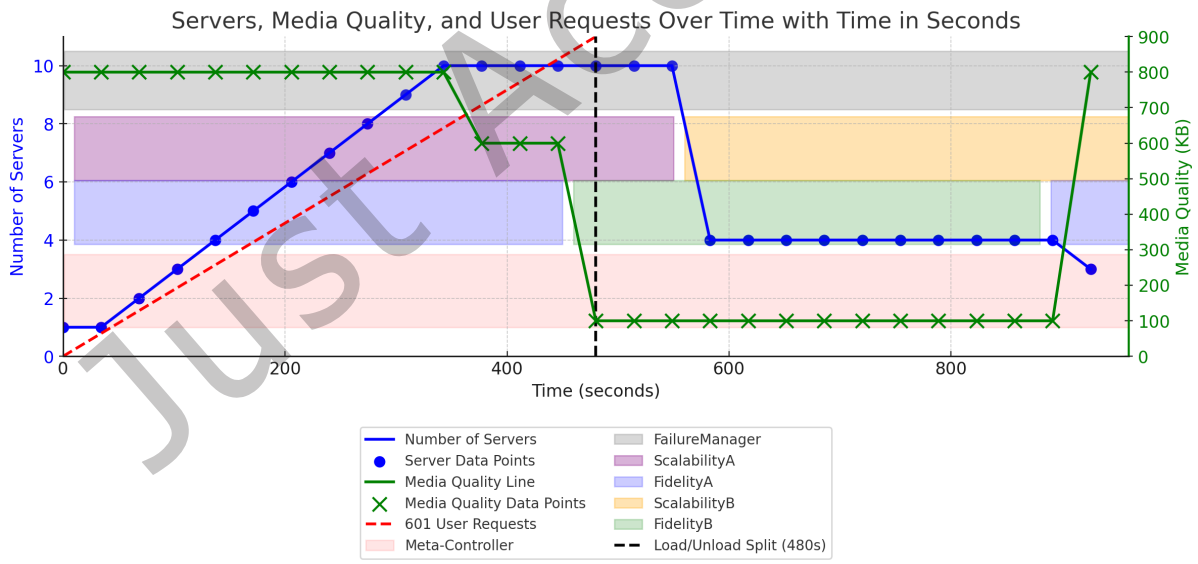


Fig. 10. Characteristic run of the Meta-KZ configuration.

4.3 Experimental Infrastructure

The experimental infrastructure was the same for the evaluation and comparison of the designed architectural configurations. Experiments were conducted on a single physical machine equipped with two AMD Ryzen 5, 3.6 GHz processors, 16GB of memory and 250GB SSD. MS Windows 10 was used, supported with the following tools: Hyper-V for creating a new local instance of a Kubernetes cluster; Minikube version v1.23.1 to set up locally a single-node Kubernetes cluster; and Kubernetes CLI version v1.22.2 command line tools for controlling the Kubernetes cluster. To simulate a typical cloud environment and dynamically change the resource allocation, each application was deployed with all its tiers inside its virtual machine, *e.g.,* using a Minikube as a Kubernetes cluster. Finally, for conducting the experiments, we followed six steps:

(1) configure the computational infrastructure, including, the computer, virtual machines, Kubernetes CLI and Minikube;
(2) deploy the monitoring tools, which involves the tools to support logging and monitoring (*i.e.,* metrics-server, Prometheus and kube-state-metric);
(3) deploy the target system objects, which involves deploying objects responsible for managing the entry point of the target system (*i.e.,* kube-znn-svc and ngnix), at run-time, and storing the current media to be accessed by the users (*i.e.,* db-svc);
(4) select the structure of the controller, *i.e.,* selecting which controller is used in the experiment (Mon-KZ, Des-KZ, or Meta-KZ).
(5) deploy the overload simulator, which involves deploying tools responsible for simulating a load of user requests (*i.e.,* the k6 tool); and
(6) collect execution data, which involves the deployment of tailor-made tools (*e.g.,* Python scripts and shellscrips) responsible for collecting data from the Kubernetes cluster.

5 EVALUATION

The evaluation we present in this section consists of three experiments we performed to check whether a controller based on micro-controllers has the advantage of being structurally flexible without compromising the target system's performance.

The experiments involve the controller configurations presented in Section 4 (namely, Mon-KZ, Des-KZ, Meta-KZ and Meta-KZ-New) and utilise the same experimental infrastructure described in Section 4.3. The experiments follow the GQM (Goal/Question/Metrics) paradigm [5]. Figures 11, 15 and 19 depict the design of the experiments in terms of the GQM elements. According to Basili [5], in a typical GQM diagram, the flow from the goal(s) to metrics can be viewed as a directed graph (the flow is from the goal nodes to the question nodes to the metric nodes).

5.1 General Experimental Execution Setup and Analysis Procedures

We defined an execution setup that considers a maximum number of 10 active Kube-ZNN servers to collect evidence to answer all research questions (RQ). The setup involves the predefined time during which a load/unload of user requests is performed,[10] thus simulating how the controller and the target system are working. In our experiments, we have used the size of the server pool, the media quality, and the non-operational Kubernetes pods (being the latter only present in the Meta-KZ-New configuration) for adapting the system according to demand (in particular, according to the number of user requests and the current rate of CPU failures). Our measurements regarding these variables at particular moments (namely, right after the peak of demand, and right after the

[10]In this work, we use Grafana k6, available at https://k6.io/docs/#what-is-k6, as a tool to perform loads. It is an open-source load testing tool used in performance testing. As an example, the tool helps to verify performance regression.

demand decrease, *i.e.,* right after unload), as well as the measurement of the response time to user requests, provide evidence of the capacity of the system to accomplish its goals.

To assess the statistical relevance of the achieved results from the perspective of comparing the different controller configurations explored in this research, we apply the Mann-Whitney U test [34] using the 5% confidence level (*i.e., p-value* < 0.05). The Mann-Whitney U test indicates whether there is a significant difference between the two sample medians; in that case, the conclusion is that there is a high likelihood that the samples represent populations with different median values [52]. Note that the Mann-Whitney U test does not assume a normal distribution since it is non-parametric. The normality of data distribution was checked with the Shapiro-Wilk test for normality [49].

5.2 Experiment 1 (GQM-1): Evaluation Regarding Original ZNN Quality Attributes

This experiment compares the Mon-KZ, Des-KZ and Meta-KZ configurations. The GQM design for this experiment is displayed in Figure 11. In the context of this experiment, the performance of the target system is considered higher when the system responds to a given demand by using fewer active servers (related to RQ1), by delivering higher media quality (related to RQ2), or by responding to user requests in a shorter time (related to RQ3) than it would do in different execution setups.

Answers we derive for the three following research questions (RQs) help us to draw a general conclusion regarding the experiment goal:

- **RQ1:** How does a decentralised controller perform regarding structural adaptations of the target system when compared to a monolithic controller? Note that structural adaptations, in the case of Kube-ZNN , directly relate to the scalability (increase or decrease of the server pool size) of the target system.
- **RQ2:** How does a decentralised controller perform regarding parametric adaptations of the target system when compared to a monolithic controller? Note that parametric adaptations, in the case of Kube-ZNN , directly relate to the fidelity attribute (increase or decrease of quality of the delivered media) of the target system.
- **RQ3:** How does a decentralised controller impact the system response time?

We characterise the adaptation as structural and parametric due to their characteristics. The scalability quality attribute can be modified when the composition of the pool of Kube-ZNN servers is modified. On the other hand, the fidelity quality attribute can be modified when internal parameters are changed, similar to a conditional choice of media quality. Thus, to provide answers to the RQs, we collect the following set of metrics from the subject software systems (namely, Mon-KZ, Des-KZ, and Meta-KZ).

- **M1:** Number of active servers at the peak state.
- **M2:** Number of active servers at the final state.
- **M3:** Media fidelity level at the peak state.
- **M4:** Media fidelity level at the final state.
- **M5:** Median response time for user requests during the load period.

The *peak* state is reached after exposing the target system (*i.e.,* the Kube-ZNN) to a load of customer requests for a particular time interval. The *final* state represents the end of the unload period. By unload period we mean the time interval after the load has ceased so that the target system is expected to scale down incrementally. The evaluation of the state of the target system, in terms of scalability (M1 and M2) and fidelity (M3 and M4), at particular execution moments (namely, peak and final), provides evidence to answer questions RQ1 and RQ2, respectively. Furthermore, the observation of the median response time for user requests during the load period underlies our answer to RQ3.

Gathering evidence for answering RQ1, RQ2, and RQ3: We executed each configuration (*i.e.,* Mon-KZ, Des-KZ and Meta-KZ) 40 times (3 x 40 = 120 executions, in total). For each execution, we defined 8 minutes

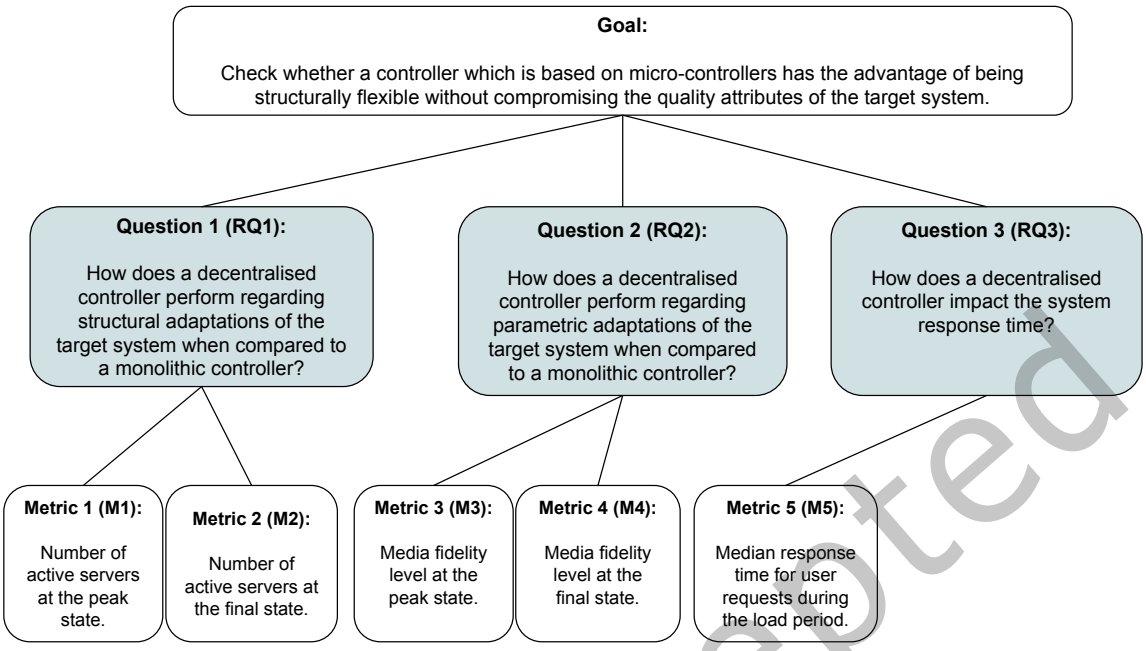


Fig. 11. Design of the first experiment based on the GQM paradigm.

of load/unload, and collected metrics M1, M2, M3, M4 and M5. This allows us to characterise the performance of a given configuration regarding the observed attributes (*i.e.,* scalability, fidelity, and response time). The distributions of values with respect to each set of 40 executions are represented as boxplot charts in Figures 12, 13, and 14. In Figure 12, the *Y* axes in the charts represent the number of active Kube-ZNN servers, while the *Y* axes in the charts of Figure 13 represent the quality of the media delivered to the customers (ranging from 400KB to 800KB), and the Y axes in the charts of Figure 14 represent the median response time to user requests. As an example, let us consider the results for the scalability property shown in Figure 12 . For the 40 executions of Meta-KZ, at the end of the load period (*i.e.,* the "Peak" column for the Meta-KZ configuration), we may observe a larger concentration of results between the 4 to 6 interval, whereas the minimum and maximum numbers of active servers were 2 and 10, respectively).

5.2.1 Scalability Results (Related to RQ1). Figure 12 summarises the results regarding the scalability of Mon-KZ, Des-KZ, and Meta-KZ (40 executions each). We next describe and analyse the results with a focus on: (1) the end of the load period (*i.e.,* at the peak of demand), and (2) the end of the unload period. At the end, we compare the results for the three configurations.

Scalability-related results at the peak of demand: the Mon-KZ configuration had a concentration between 6 and 10 active servers across the 40 executions, with a range of variations from 2 to 10. Des-KZ reached 10 active servers in all executions. Finally, Meta-KZ had concentrated between 4 and 8 active servers, while the variation ranged from 2 to 10.

Scalability-related results at the end of the unload period: In general, configurations Des-KZ and Meta-KZ restored to 1 active server in the executions (the latter with a few outliers). The number of active servers for Mon-KZ at the end of the unload period varied between 1 and 2 (also with a few outliers).

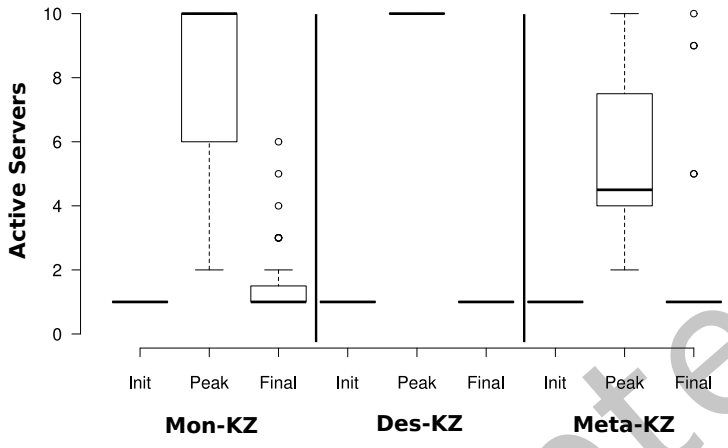


Fig. 12. **GQM-1 - RQ1:** Results regarding **scalability** (init = initial state; peak = middle state (peak of system demand); final = final state (just after unload period).

Comparison: at the peak of demand, results varied across the configurations. In particular, Des-KZ always reached 10 active servers, whereas both Mon-KZ and Meta-KZ required fewer resources to respond to the load of user requests. Despite this, after the unload period, the results for all configurations mostly reached 1 active server, thus restoring to the original target system state (concerning the server pool size) in all executions.

When the demand is very high (*e.g.,* at load peak), we noticed that 10 servers may be active and many failures start to be generated by the Kubernetes environment and captured by the FailureManager component, irrespective of the configuration. Thus, as explained in Section 4.1, stitch adaptation strategies define a new capacity to respond to requests that will start working; more specifically, ScalabilityA, which operates with a maximum of 10 active servers, is replaced with ScalabilityB, which operates with a maximum of 4 active servers, until the failure rate returns to a low value. In particular, whenever the target system operates with fewer active servers, the AdjustDefaultReplicas strategy, described in Table 2, is triggered to decrease the servers; it holds until the number of failures decreases.

When the system capacity changes, it is noted that the Meta-KZ configuration was able to manage more efficiently the controller reconfiguration, and thus the reaction to the decrease in system capacity was faster in the Meta-KZ configuration.

The Meta-KZ outperforms the Mon-KZ at peak times because the latter relies on a single controller, where the activation of different tactics related to scalability and fidelity occurs sequentially. In contrast, the Meta-KZ employs multiple controllers, enabling parallel activation of these tactics. Furthermore, the Meta-KZ outperforms the Des-KZ due to reduced intercommunication between micro-controllers, as the meta-controller efficiently orchestrates the activation of tactics, ensuring smoother coordination. In other words, results indicate that having a dedicated controller to perform adaptation improves the overall results concerning the scalability quality attribute.

We applied the Shapiro-Wilk test to the results regarding scalability which were collected at the peak of demand. The tests revealed a non-normal distribution for Mon-KZ ($p\text{-}value = 9.527 * 10^{-8}$), Des-KZ (constant values), and Meta-KZ ($p\text{-}value = 1.432 * 10^{-3}$). Subsequently, the comparison of Meta-KZ and Mon-KZ through the application of the Mann-Whitney U test revealed a statistically significant difference between the results for both configurations ($p\text{-}value = 6.086 * 10^{-6}$). The Mann-Whitney U test applied to results of Meta-KZ and Des-KZ ($p\text{-}value = 9.548 * 10^{-15}$), and to results of Des-KZ and Mon-KZ ($p\text{-}value = 4.922 * 10^{-6}$), also revealed statistically significant differences among results for those configurations.

Regarding RQ1, in terms of structural adaptations of the target system, the Mon-KZ and Meta-KZ configurations performed better than Des-KZ, particularly at the peak of demand. Moreover, there is a noticeable difference between Mon-KZ and Meta-KZ: while Mon-KZ concentrated its results between 6 and 10 active servers, the values ranged from 4 to 8 for Meta-KZ.

5.2.2 Fidelity Results (Related to RQ2) . Figure 13 summarises the results regarding the fidelity of configurations Mon-KZ, Des-KZ, and Meta-KZ. Similar to the discussion presented for scalability, we next describe and analyse the results with a focus on: (1) the end of the load period, and (2) the end of the unload period. We also compare the results for all configurations.

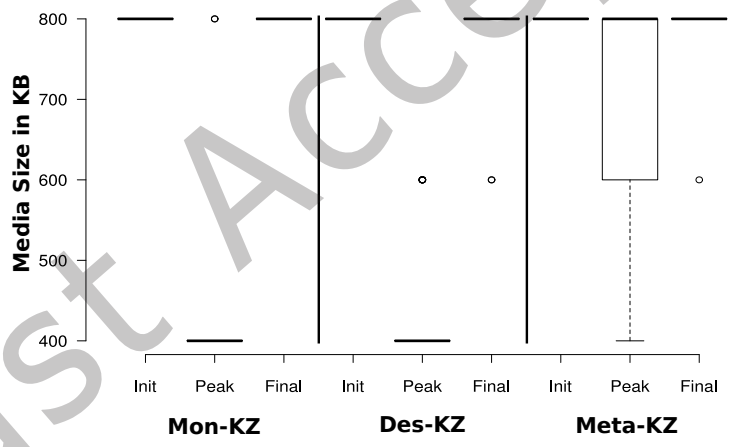


Fig. 13. **GQM-1 - RQ2:** Results regarding **fidelity** (init = initial state; peak = middle state (peak of system demand); final = final state (just after unload period).

Fidelity-related results at the peak of demand: both Mon-KZ and Des-KZ decreased the fidelity to the minimum level (*i.e.,* 400KB) when the highest number of user requests were reached (*i.e.,* peak of demand), whereas the concentration in higher levels of fidelity was observed for Meta-KZ (media quality mostly varied between 600KB and 800KB).
Fidelity-related results at the end of the unload period: with respect to the final states, all configurations restored the fidelity level to 800KB after the unload period (with just a few outliers, as depicted in Figure 13).

Comparison: For fidelity, Meta-KZ outperformed Des-KZ and Mon-KZ which is reinforced by the results of the statistical tests next described. Particularly at the peak of demand, both Mon-KZ and Des-KZ decreased the level of fidelity to the minimum, whereas Meta-KZ kept the fidelity level between 600KB and 800KB. Similarly to what happened with the variants of scalability micro-controllers, replacements of FidelityA by FidelityB and vice versa in all configurations were observed in the experiment runs.

Similarly as we described for scalability, the results indicate that Meta-KZ can manage the adaptations better than the other configurations. Note that during the peak of load, Meta-KZ was able to keep a higher quality of media, due to the same reason we identify for scalability, mainly when compared to Mon-KZ and Des-KZ.

We applied the Shapiro-Wilk test to the results regarding fidelity which were collected at the peak of demand. The tests revealed a non-normal distribution for Mon-KZ ($p\text{-}value = 3.733 * 10^{-13}$), Des-KZ ($p\text{-}value = 5.616 * 10^{-11}$), and Meta-KZ ($p\text{-}value = 4.481 * 10^{-8}$). Subsequently, the application of the Mann-Whitney U test to compare results of Meta-KZ and Mon-KZ revealed a statistically significant difference ($p\text{-}value = 2.801 * 10^{-10}$). The Mann-Whitney U test applied to results of Meta-KZ and Des-KZ ($p\text{-}value = 2.276 * 10^{-9}$) also revealed statistically significant differences among results for those two configurations, whereas the results for the comparison between Des-KZ and Mon-KZ revealed non-statistically significant differences ($p\text{-}value = 1.043 * 10^{-1}$).

Regarding RQ2, in terms of parametric adaptations of the target system, the Meta-KZ configuration performed better than Mon-KZ and Des-KZ, particularly at the peak of demand, without a distinction between the performance of Mon-KZ and Des-KZ.

5.2.3 Response time Results (Related to RQ3). Figure 14 shows the results regarding response time for Mon-KZ, Des-KZ and Meta-KZ. Data points in the boxplots represent the median response time for each of the 40 executions.

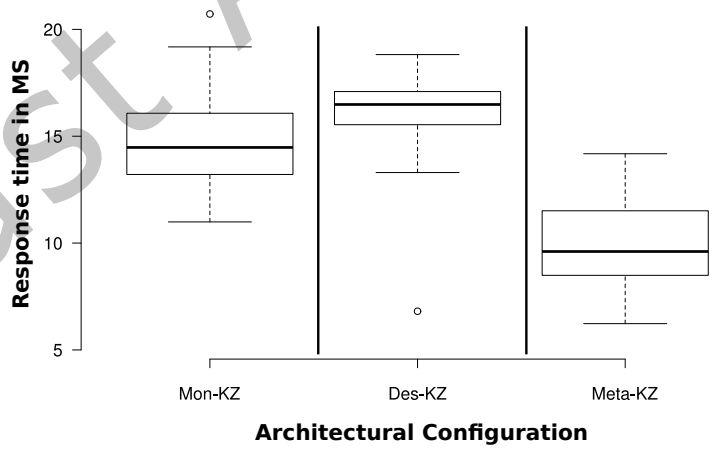


Fig. 14. **GQM-1 - RQ3:** Results regarding **response time**.

Comparison: For response time, Meta-KZ outperformed Des-KZ and Mon-KZ. From Figure 14, it is noticeable Meta-KZ responded to user requests faster than the other configurations. The application of the Shapiro-Wilk test reveals that data regarding response time for Meta-KZ has a normal distribution ($p\text{-}value = 1.625 * 10^{-1}$), whereas data for Mon-KZ ($p\text{-}value = 3.848 * 10^{-2}$), and Des-KZ ($p\text{-}value = 5.843 * 10^{-6}$) have non-normal distributions. Subsequently, the comparison of Meta-KZ and Mon-KZ through the application of the Mann-Whitney U test revealed a statistically significant difference between the results for both configurations ($p\text{-}value = 3.356 * 10^{-12}$). The Mann-Whitney U test applied to results of Meta-KZ and Des-KZ ($p\text{-}value = 3.113 * 10^{-13}$), and to results of Des-KZ and Mon-KZ ($p\text{-}value = 1.497 * 10^{-4}$), also revealed statistically significant differences among results for those configurations.

Regarding RQ3, the Meta-KZ configuration performed better than Mon-KZ and Des-KZ, whereas Des-KZ was the slowest configuration in terms of response time among the three configurations that were evaluated in our experiment.

5.3 Experiment 2 (GQM-2):: Evaluation Regarding the Stability of the Target System

In the context of this work, "stability" refers to the system's ability to maintain consistent performance and reliability despite environmental or internal changes. For the target system, stability ensures the website remains accessible and functional during fluctuations in user traffic or updates. In this work, the stability of the target system depends on the feasibility of adaptations initiated by control loops within the controller layer, which monitor and adjust the system to meet the desired objectives. An adaptation is feasible if it can be implemented without causing further disruption. However, if the timing of control loops is too short, it can introduce significant overhead and instability due to the high frequency of modifications in the pods. Therefore, stability is achieved when control loops within the intermediate controller layer trigger feasible adaptations at an optimal frequency, balancing responsiveness with the system's capacity to handle changes, thus maintaining overall performance and reliability. In this sense, this experiment also compares the Mon-KZ, Des-KZ and Meta-KZ configurations, but now from the perspective of the impact on the system stability caused by variations of the timings of control loops. The GQM design for this experiment is displayed in Figure 15, which includes the following research question:

- **RQ4:** How does the timing of the different control loops impact the stability of the target system?

The evidence we collected to answer RQ4 is based on the following set of metrics:

- **M1:** Number of active servers at the peak state.
- **M2:** Number of active servers at the final state.
- **M3:** Media fidelity level at the peak state.
- **M4:** Media fidelity level at the final state.
- **M6:** Number of unavailable servers at the peak state.
- **M7:** Number of unavailable servers at the final state.

We analyse both scalability (M1 and M2) and fidelity (M3 and M4) when we change (decreasingly) the control loop timings, which may impact the stability of the target system with respect to the number of available and unavailable Kube-ZNN servers (M6 and M7).

Gathering evidence for answering RQ4: A justification to decrease the time in the control loop has emerged from empirical studies that demonstrate when we decrease the frequencies in the control loop, we face target system instability. To analyse the stability of target system from the perspective of required resources (namely, required Kube-ZNN servers), in contrast to the previous experiment reported in Section 5.2, we have performed 24

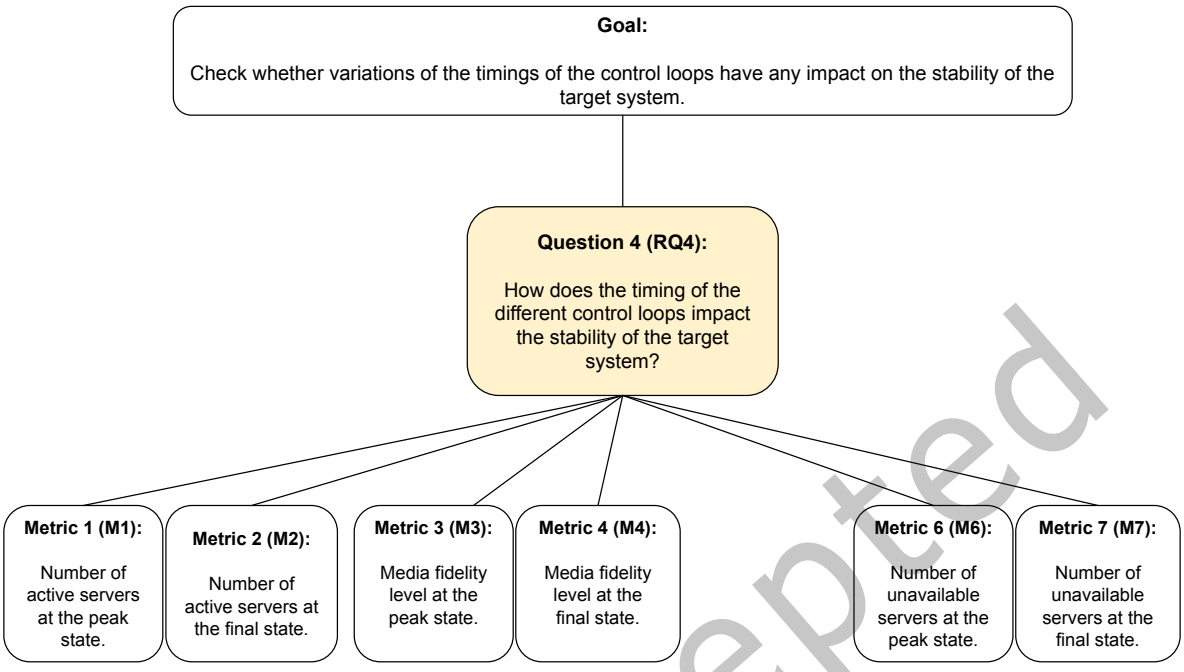


Fig. 15. Design of the second experiment based on the GQM paradigm.

new executions of each configuration (72 executions in total). Each set of 24 executions was split into 8 executions for each of the following variations of the timing setups:

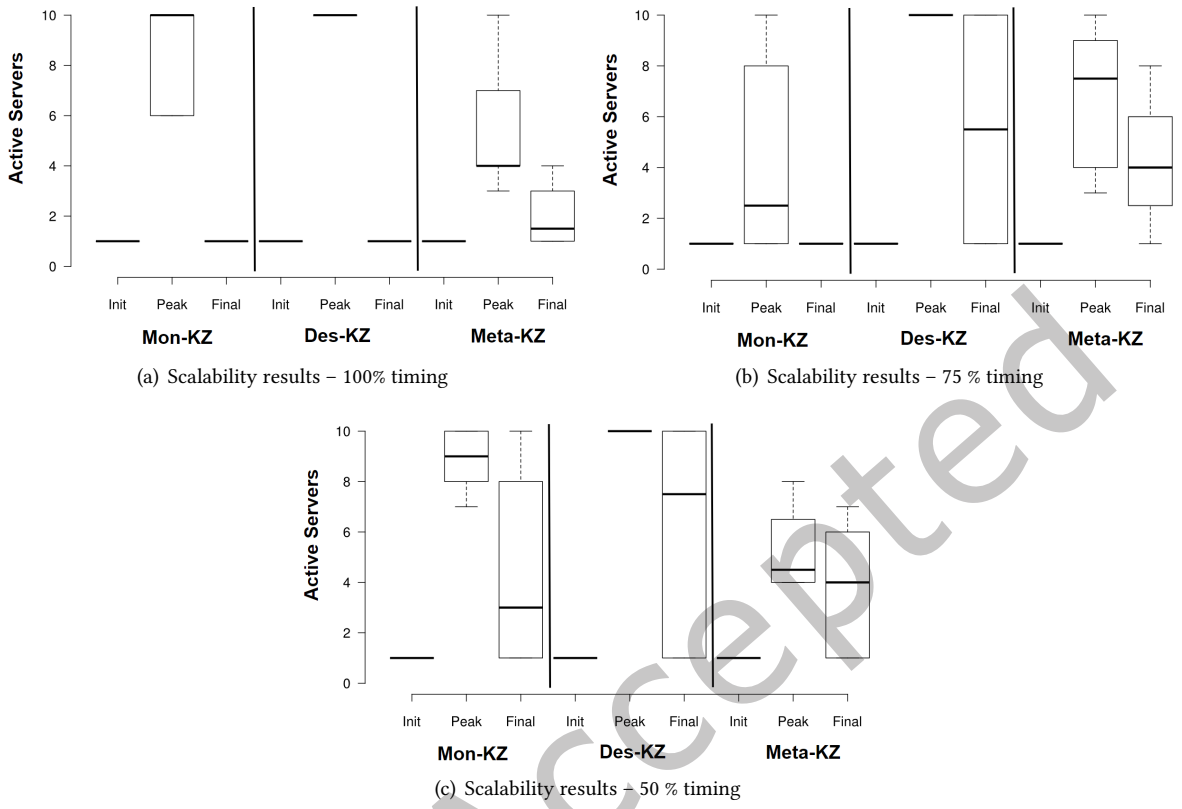
- *Default timing* (or *100% timing*): 45 seconds for the controller loop, and 65 seconds for the meta-controller loop.
- *75% timing*: 34 seconds for the controller loop, and 49 seconds for the meta-controller loop.
- *50% timing*: 23 seconds for the controller loop, and 33 seconds for the meta-controller loop.

The default timing represents the same timing used to gather evidence to answer the RQs of GQM-1. In this case, we identify not only how the controller and target system operate at different control loop timings, but also how stable the target system is.

We set 8 minutes of load and 20 minutes of unload of user requests.

Figures 16, 17 and 18 summarise, respectively, the results regarding the scalability, fidelity, and number of unavailable servers for Mon-KZ, Des-KZ, and Meta-KZ. We next describe and analyse the results concerning the three observed variables, namely, the server pool size, the size of delivered media, and the number of unavailable servers.

Scalability-related results for the three variations of timing setup: Based on the results presented in Figure 16, Des-KZ resulted in 10 active servers at the peak of demand irrespective of control loop timings. At the end of the unload period, Des-KZ faced restoration difficulties when the timing was shortened to 75% and 50% (in other words, the target system was not restored to its original state concerning the number of active servers). On the other hand, the executions of Mon-KZ and Meta-KZ, overall, resulted in slightly lower numbers of active servers at the peak state. At the final state, Meta-KZ faced some difficulty in restoring the target system to its original state in the three timing setups. Mon-KZ faced restoration difficulty only in executions with 50% timing.



(a) Scalability results – 100% timing

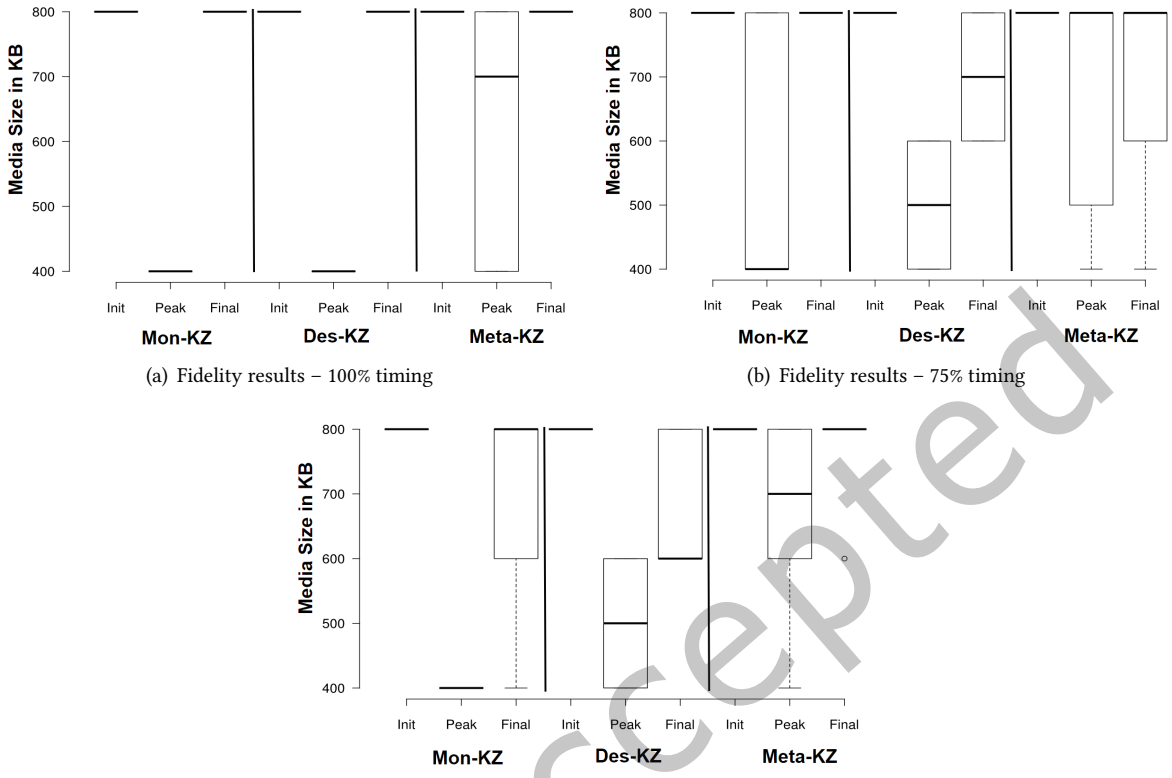
(b) Scalability results – 75 % timing

(c) Scalability results – 50 % timing

Fig. 16. **GQM-2 - RQ4:** Results regarding **scalability** when shortening the control loop timing (init = initial state; peak = middle state (peak of system demand); final = final state (just after unload period).

Fidelity-related results for the three variations of timing setup: Based on the results presented in Figure 17, Mon-KZ resulted in the minimum media quality (*i.e.,* 400KB) at the peak of demand irrespective of control loop timings (with a few exceptions in 75% timing). At the end of the unload period, Mon-KZ restored the media quality to the maximum level in almost all cases (with a few exceptions when the timing was shortened to 50%). For Des-KZ, the media quality at the peak state increased slightly when the timing was reduced to 75% and 50%, whereas some restoration problems were observed at the end of the unload period when considering the same reduced timing setups. Similarly, for Meta-KZ the results point to slight improvements of the media quality at the peak of demand when timing was shortened to 75% and 50%, whereas a few restoration problems were observed at the end of the unload period for reduced timings.

While comparing the three approaches, we observe that in some executions, Meta-KZ was able to maintain fidelity at 800KB even at the 50% timing. The media quality also increased in the other two configurations as long as the timing was shortened. We note that both fidelity and scalability adaptations affect the results discussed here. Given that adaptations related to both scalability and fidelity (in all configurations) run independently, and both share the same target system state, when some servers are added (*i.e.,* ImproveSlo strategy in Table 2), the "highSLO && low-Fidelity" predicate may become *true*, and the media size naturally is increased through the (*i.e.,* ReduceCost strategy).



(a) Fidelity results – 100% timing

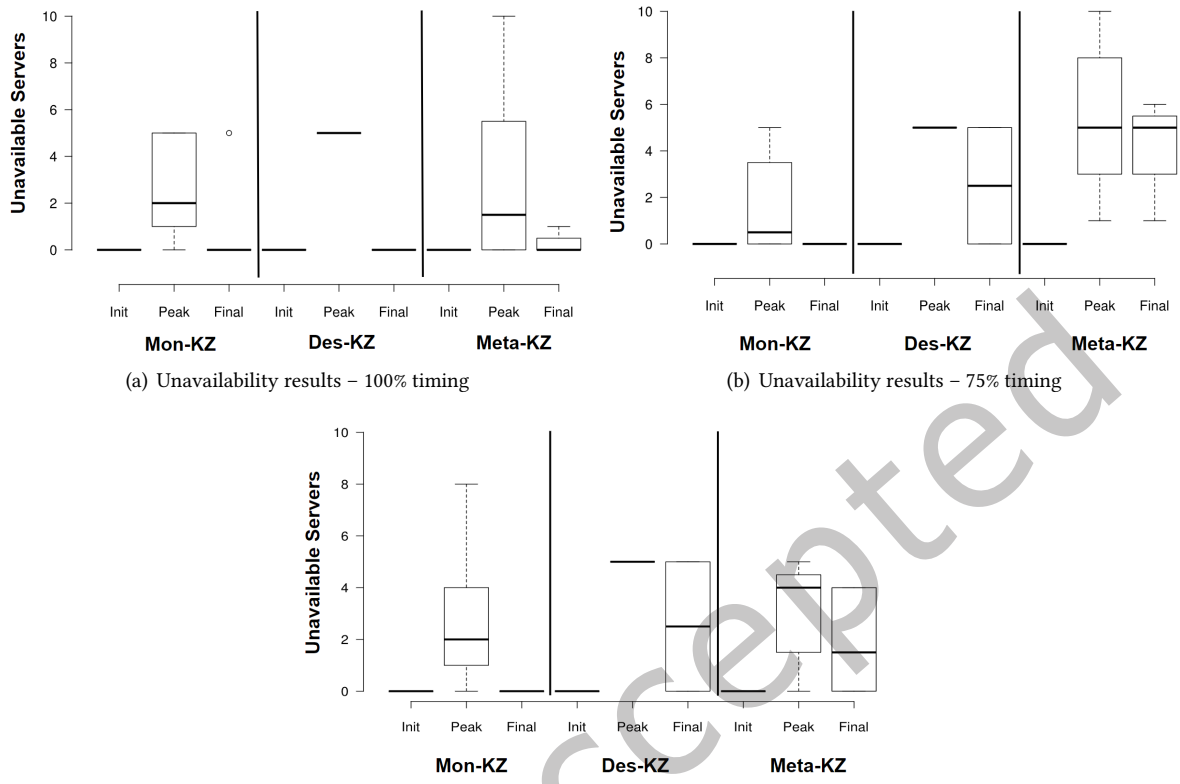
(b) Fidelity results – 75% timing

(c) Fidelity results – 50% timing

Fig. 17. **GQM-2 - RQ4:** Results regarding **fidelity** when shortening the control loop timing (init = initial state; peak = middle state (peak of system demand); final = final state (just after unload period).

Unavailability-related results for the three variations of timing setups: In 100% and 50% timing setups, the Des-KZ configuration had higher numbers of unavailable servers at the peak state, whereas this number was higher for Meta-KZ in the 75% timing. When comparing the three configurations, while shortening the timings of control loops, it is noticeable that: (1) in contrast to the monolithic configuration, the two decentralised configurations (namely, Des-KZ and Meta-KZ) seem to lose performance in terms of waste of resources (*i.e.,* servers that are deployed but do not become active) either at the peak state or at the final state (or both); and (2) Meta-KZ showed higher instability from the perspective of the unavailability of servers than the other two configurations.

Regarding RQ4, in terms of the impact on the stability of the target system, shortening the control loop timings results in: (1) an increase in the number of active servers after the unload period (that is, restoration is negatively affected concerning structural adaptations of the target system); (2) the media quality begins to exhibit variations at both the peak and final states, even without a clear trend; and (3) even though Meta-KZ is the most affected in terms of unavailable servers, the number of active servers as well as the media quality level surpass Mon-KZ and



(a) Unavailability results – 100% timing

(b) Unavailability results – 75% timing

(c) Unavailability results – 50% timing

Fig. 18. **GQM-2 - RQ4:** Results regarding **unavailability** of Kube-ZNN servers when shortening the control loop timing (init = initial state; peak = middle state (peak of system demand); final = final state (just after unload period).

Des-KZ. Note that, even though we have not included response time as a dependent variable in this experiment, it is well-expected that the variations in control loop timings would also have impacted this target system's goal.

5.4 Experiment 3 (GQM-3):: Evaluation Regarding the Evolved Meta-KZ Configuration

This experiment evaluates the Meta-KZ-New configuration in contrast to the original Meta-KZ. The goal of this experiment is to check whether adding a new micro-controller impacts the quality attributes of the target system. Therefore, the comparison revisits the same research questions we established for the previous experiments (Sections 5.2 and 5.3), but now we only consider the two configurations that include the meta-controller layer. Figure 19 depicts the design of the experiment in terms of the GQM elements. The following research questions are established:

- **RQ5:** How does a new micro-controller that deals with unavailable resources impact the structural adaptations of the target system?
- **RQ6:** How does a new micro-controller that deals with unavailable resources impact the parametric adaptations of the target system?

- **RQ7:** How does a new micro-controller that deals with unavailable resources impact the system response time?
- **RQ8:** How does a new micro-controller that deals with unavailable resources impact the stability of the target system?

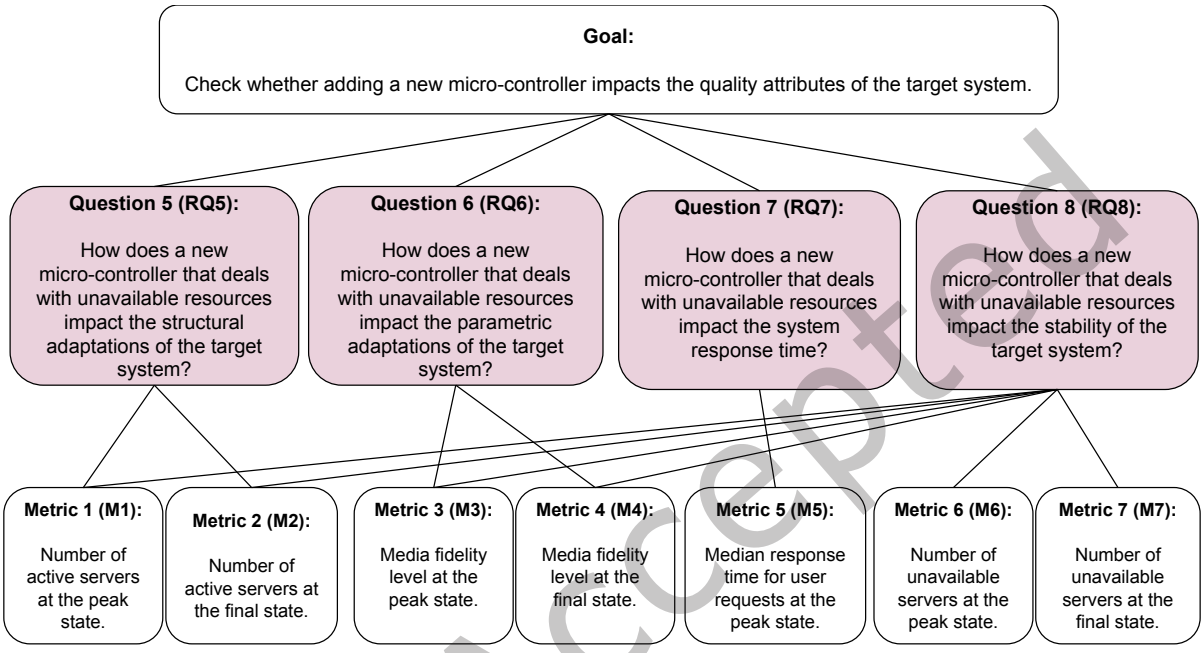


Fig. 19. Design of the third experiment based on the GQM paradigm.

The evidence we collected to answer RQ5, RQ6, RQ7, and RQ8 are based on the following set of metrics:

- **M1:** Number of active servers at the peak state.
- **M2:** Number of active servers at the final state.
- **M3:** Media fidelity level at the peak state.
- **M4:** Media fidelity level at the final state.
- **M5:** Median response time for user requests during the load period.
- **M6:** Number of unavailable servers at the peak state.
- **M7:** Number of unavailable servers at the final state.

Gathering evidence for answering RQ5, RQ6, RQ7, and RQ8: Data concerning Meta-KZ executions is the same as collected for the first two experiments; in summary, the answers we derive for RQ5, RQ6 and RQ7 rely on data collected from 40 executions of Meta-KZ, whereas the answer we derive for RQ8 relies on data collected from 8 executions of Meta-KZ, specifically the executions that concern the *default timing*.

As for Meta-KZ-New, data was collected from the same number of executions we performed for Meta-KZ. In particular, we performed 40 executions that yielded novel data to be analysed for RQ5, RQ6, and RQ7. Besides that, we randomly selected 8 (out of 40) executions to extract data to be analysed for RQ8. In both cases (40 executions and 8 executions) we set 8 minutes of load/unload of user requests.

5.4.1 Scalability-related results (related to RQ5): Figure 20 summarises the results regarding the scalability of Meta-KZ and Meta-KZ-New (40 executions each). At the end of the load period (*i.e.,* at the peak of demand), Meta-KZ-New was replying to user requests with a smaller pool of servers (ranging from 3 to 5 servers) when compared to Meta-KZ (ranging from 4 to 8 servers). At the end of the unload period, both configuration restored the pool size to its original state with 1 active server (with a few observed outliers).

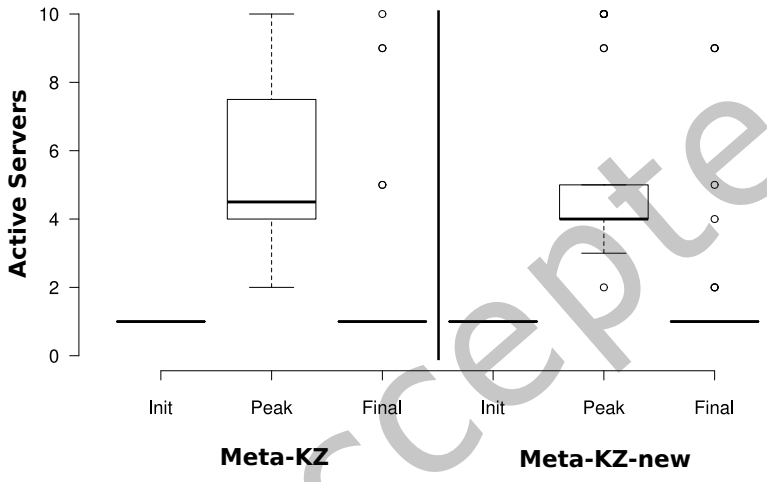


Fig. 20. **GQM-3 - RQ5:** Results regarding **scalability** for Meta-KZ and Meta-KZ-New (init = initial state; peak = middle state (peak of system demand); final = final state (just after unload period).

Concerning the results regarding scalability which were collected at the peak of demand, as reported in Section 5.2.1, data has a non-normal distribution for the Meta-KZ configuration. The application of the Shapiro-Wilk test to results for Meta-KZ-New also indicated a non-normal distribution ($p\text{-}value = 7,701 * 10^{-8}$). The application of the Mann-Whitney U test revealed there is not a statistically significant difference between the results for both configurations ($p\text{-}value = 6.622 * 10^{-1}$).

Regarding RQ5, the Meta-KZ-New configuration performed similarly to Meta-KZ with respect to the number of active servers at the peak of demand, even with the introduction of a new micro-controller, which in turn increased the complexity of the system.

5.4.2 Fidelity-related results (related to RQ6): Figure 21 summarises the results regarding the fidelity of Meta-KZ and Meta-KZ-New (40 executions each). At the peak of demand, the media quality delivered by Meta-KZ-New ranged from 500KB to 800KB (with a higher concentration of 600KB). In comparison, the figures concerning Meta-KZ ranged from 600KB to 800KB (with a higher concentration of 800KB). At the end of the unload period, both configuration restored the media quality to the maximum level.

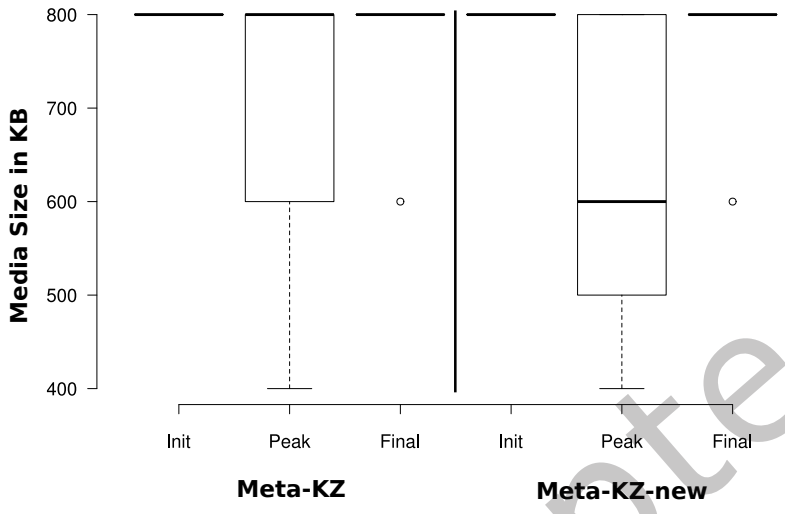


Fig. 21. **GQM-3 - RQ6:** Results regarding **fidelity** for Meta-KZ and Meta-KZ-New (init = initial state; peak = middle state (peak of system demand); final = final state (just after unload period).

Regarding the results for fidelity which were collected at the peak of demand, data has a non-normal distribution for the Meta-KZ configuration (Section 5.2.2), as well as for Meta-KZ-New ($p\text{-}value = 3.547 * 10^{-6}$). Thus, we applied the Mann-Whitney U test to compare results from Meta-KZ and Meta-KZ-New. The test revealed the absence of a statistically significant difference between the results for both configurations ($p\text{-}value = 1.250 * 10^{-1}$).

Regarding RQ6, the Meta-KZ-New configuration performed similarly to Meta-KZ with respect to the quality of media delivered at the peak of demand, even with the introduction of a new micro-controller, which in turn increased system complexity.

5.4.3 Response time-related results (related to RQ7): Figure 22 shows the results regarding response time for Meta-KZ and Meta-KZ-New. The boxplots depict data regarding the median response time to user requests during the load period for the 40 executions. The results presented in the charts indicate that Meta-KZ responded to user requests faster than Meta-KZ-New.

Given that data regarding response time for Meta-KZ has a normal distribution (Section 5.2.3), but data concerning Meta-KZ-New has a non-normal distribution ($p\text{-}value = 3.256 * 10^{-2}$), we applied the Mann-Whitney U test to compare results for the two configurations. The results indicate there is a statistically significant difference between the results for both configurations ($p\text{-}value = 4.598 * 10^{-7}$), which confirms what is suggested by the analysis of Figure 22.

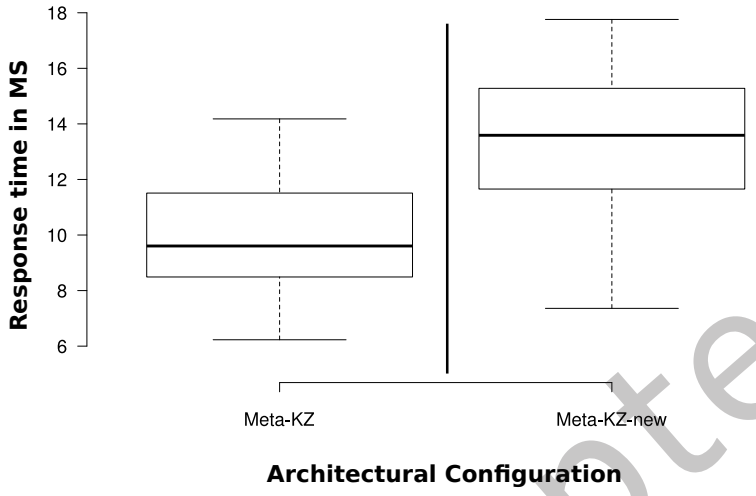


Fig. 22. **GQM-3 - RQ7:** Results regarding **response time** for Meta-KZ and Meta-KZ-New.

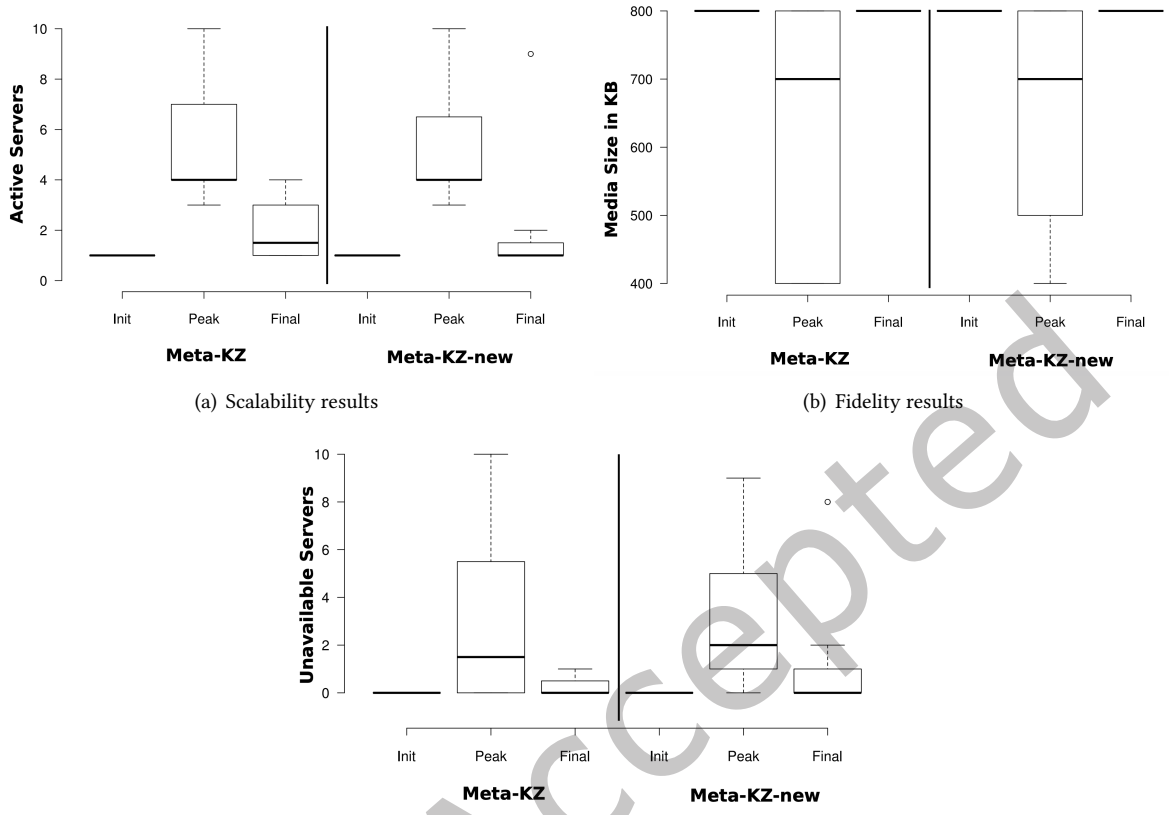
Regarding RQ7, the Meta-KZ configuration performed better than Meta-KZ-New concerning the time to respond to user requests during the load period. This may be a side effect of introducing a new micro-controller, thereby increasing system complexity.

5.4.4 Target System stability-related results (related to RQ8): Results related to stability of the target system (which comprise 8 executions of each configuration) are shown in Figure 23. The charts of Figures 23(a), 23(b), and 23(c), respectively, show the distribution of the three observed variables, namely, the server pool size, the size of delivered media, and the number of unavailable servers. Note that the data for Meta-KZ is the same as the one analysed in Section 5.3. Overall, results for both configurations are very similar. For the three observed variables, small gains were observed when the FailureHandler was introduced into the controller, particularly at the peak moment.

Regarding RQ8, the performances of Meta-KZ and Meta-KZ-New were similar with respect to the stability of the target system, even with the introduction of a new micro-controller, which in turn increased system complexity.

5.5 Further discussion

When comparing the four configurations, increasing the number of Kube-ZNN servers can negatively impact the restoration time for configurations that need to manage a larger number of components (namely, Des-KZ, Meta-KZ, and Meta-KZ-New). In our study, restoration regarding scalability means returning the target system



(a) Scalability results

(b) Fidelity results

(c) Server unavailability results

Fig. 23. **GQM-3 - RQ8:** Results regarding **the stability of the target system** (init = initial state; peak = middle state (peak of system demand); final = final state (just after unload period).

to only one active server. In contrast, restoration regarding fidelity means increasing the quality of the media delivered to the customers to its maximum (*i.e.,* 800KB).

In our previous work [54], we highlighted the importance of conducting studies focused on system restoration. Thus, the RQ4 and RQ8 (established and handled in Sections 5.3 and 5.4, respectively) address specific rounds of executions demonstrating that restoration is directly related to stability, so an approach that brings stability problems would consequently result in restoration problems. In this context, considering conflicts and synchronisation of the target system state, the Mon-KZ configuration contains a single set of adaptation strategies. As such, the strategies are executed by the Kubow controller without the need to solve issues related to conflicts among strategies, or issues related to the synchronisation of the state of the target system. For the Des-KZ, Meta-KZ and Meta-KZ-New configurations, on the other hand, even though the variants of micro-controllers are mutually exclusive (*i.e.,* either ScalabilityA or ScalabilityB is active at a particular time, but not both; likewise for fidelity micro-controllers), conflicts may arise due to the inherent characteristics of pod management performed by Kubernetes. In summary, considering the equal importance of all QoS attributes (scalability, fidelity and

response time), evidence from the experiments shows that Meta-KZ and Meta-KZ-New outperforms the other two configurations.

As shown in Table 1, even though Meta-KZ and Meta-KZ-New demand more resources (at least 3650m of CPU and 3.228Gi of Memory) when compared with the Mon-KZ configuration (at least 1250m of CPU and 1.1Gi of Memory) and with the Des-KZ configuration (at least 2650m of CPU and 2.228Gi of Memory), results for Meta-KZ and Meta-KZ-New were not compromised (according to the results summarised in the previous sections).

6 THREATS TO VALIDITY

This section discusses the threats to the validity of our experimental work, according to the categories of threats listed by Wohlin et al. [60]. Two of our experiments involved one *independent variable* (the controller design approach) to which three *treatments* were applied (monolithic, decentralised, and decentralised with a meta-controller), four *dependent variables* (scalability, fidelity, response time, and stability), and a single subject (Kube-ZNN). The third experiment is similar to the other two experiments except for *independent variable*, which compares two different configurations of the decentralised with a meta-controller, without and with the FailureHandler.

Internal validity: In our study, the software under analysis presents some factors of uncertainty which may lead to variations in the observed outcomes for the given treatments. In particular, uncertainty is present in the controller related to processing uncertainty of the periodic monitoring of the components of the controller and target system components, which is directly associated with the control loop timings). Uncertainty is also present in the communication between components (regarding the inconsistency between the observed states from the target system to allow for decisions taken by the controller), and in the target system and its environment (variations in the available resources to run a target system inside a Kubernetes cluster deployed in a single physical machine). To mitigate the uncertainty threat, we performed 40 executions of each scenario and based our analysis on the median value of collected quantitative data. In addition, there is an experiment that addresses the stability of the target system when analysing the impact of control loop timings on the physical machine performance.

Construct validity: In our study, a threat could be the lack of prior knowledge to design and operate with the adopted technologies (*i.e.,* Kubow and Kube-ZNN). To mitigate this threat, we communicated with the developers of Kubow and Kube-ZNN [2], who helped us improve their implementations (the fixes were propagated to all configurations we used). In addition, there was a natural evolution of the architectural configurations used in the experiment. We started with the original Kubow and Kube-ZNN implementations [2] and evolved them to create the other two configurations which included micro-controllers (Des-KZ and Meta-KZ) and meta-controller (Meta-KZ). To mitigate this threat, we applied good design practices, performed code revisions, and assessed the implementations with support from the Kubow and Kube-ZNN developers.

External validity: Our study used a single subject (Kube-ZNN) and a small set of micro-controllers. Thus, our results may not translate to other settings that, for instance, use different target systems, sets of micro-controllers, or even execution infrastructures.

7 RELATED WORK

In this section, we discuss four topics of related work.

(1) Microservices for self-adaptive systems: Baylov and Dimov [6] provided a reference architecture supporting the design of self-adaptive microservices, however, it implies a specific structure that does not offer the same level of flexibility as our approach. Hassan and Bahsoon [26] proposed the creation of a controller for microservice-based self-adaptive applications. Sampaio Jr. et al. [46] advocated the reconfiguration of microservice-based

systems using affinities and historical resource usage patterns. In contrast, we promote architecting controllers based on micro-controllers (as microservices).

Mendonça et al. [36, 37] discussed issues to build self-adaptive systems based on microservices (concerning how to deploy controllers). Without a detailed solution, they suggest controllers as monolithic services or decomposed into collections of independently developed and managed microservices. The same research group have created Kubow [2] as an extended and customised version of the Rainbow framework to provide self-adaptation support to containerised applications. Thus, the main focus is on realising self-adaptation for microservice-based target systems. Aligned with Mendonça et al.'s suggestion [37], we propose and assess a concrete solution for deploying microservice-based controllers that can adapt at run-time.

Rossi et al. [45] proposed *me-kube* that is a Kubernetes extension introducing a hierarchical architecture for controlling the elasticity of microservice-based applications. Their extension aims to support approaches that are either centralised or fully decentralised. In general, Rossi et al. defined a hierarchical architecture, where a high-level controller coordinates the run-time adaptation of target systems. Compared to our work, we note that they also use adaptations that are based on microservices. However, there is no exploration concerning how the controller can be adapted at the meta-level.

Finally, based on some relatively recent literature reviews on microservice-based software systems [17, 30], we were unable to identify any approach for composing controllers from micro-controllers or decomposing controllers into micro-controllers to address the specific needs of individual target systems. This emphasises the originality of our work.

(2) Flexible controllers and meta-controllers: Banijamali et al. [4] highlighted the increasing interest in microservices in self-adaptive systems, particularly, in the automotive domain. Similarly to us, they consider a controller consisting of microservices. However, they did not address the role of a coordinating entity for managing those microservices.

Similar to our approach in defining a meta-controller, Gerostathopoulos et al. [22] proposed to generate adaptation strategies at run-time to reflect environment changes and increase the system utilities. In particular, the authors highlighted utilities regarding robustness, resilience, safety, performance, and availability. In their work, they introduced the concept of meta-adaptation strategies. The evolution of their work [23] incorporated homeostasis [51] as an additional control layer. As our meta-controller layer, this homeostasis layer changes adaptation strategies at the lower layer, at run-time. Compared to our approach, we defined not only meta-adaptation strategies but also the concept of different levels and architectures involving a meta-controller; we focus on micro-controller-based controllers as a bottom-up approach, so that micro-controllers could have more specific characteristics and functionalities.

Pereira et al. [41] proposed the development of flexible controllers based on MAPE-K (each state as a single microservice), whereas our approach is not limited to the MAPE-K components. We also include the meta-controller as an additional layer, which can reconfigure the controller.

For defining a meta-level (analogously to a meta-controller), Glazier and Garlan [25] conceptually conceived a component that they called meta-manager. This component is responsible for managing collections of components, which is similar to the idea of micro-controllers. However, Glazier and Garlan did not demonstrate details involving the different levels, nor presented studies to demonstrate the feasibility of their approach.

(3) Decentralised controllers: Florio and Di Nitto [19] proposed adding autonomic capabilities (in a decentralised way) to containerised and microservice-based systems not originally designed to be autonomic. Their approach consists of a decentralised controller implemented as a multi-agent system. Each agent realises a control loop responsible for managing a subset of microservices. Nallur and Bahsoon [38] also developed a decentralised, multi-agent self-adaptation approach for web service-based systems deployed in the cloud. Regarding both studies [19, 38] and ours, key differences are: (i) both [19, 38] were specifically designed to be applied to particular

domains (namely, systems based on microservices, and systems based on web services, respectively); and (ii) the controllers based on multi-agent systems are non-reconfigurable.

Dragan et al. [18] proposed CoADAPT as a technique for decentralised coordination for self-adaptive systems. The authors addressed the challenges associated with both local and shared data used by various distributed and parallel components of a self-adaptive system. To deal with such difficulties, their technique introduces two types of constraints: preference constraints and consistency constraints. Regarding the decentralised coordination, preference constraints express local concerns for each self-adaptive system, whereas consistency constraints express shared concerns. Thus, in their technique, the sub-systems engage in joint decision-making, conflicts or other shared concerns that are explicitly accounted for, and information about local concerns remains private during coordination.

Calinescu et al. [10] proposed the DECIDE approach, which uses distributed communication in a self-adaptive system. For mission-critical self-adaptive systems, the DECIDE approach addresses the decentralisation of control loops and practical run-time verification and validation. Regarding decentralised control loops, the authors addressed eliminating the single point of failure created by centralised control loops. In particular for run-time verification and validation, they focused on guaranteeing compliance with quality of services requirements of the mission-critical self-adaptive system. Compared to our work, which evaluated different architectures, their focus was solely on run-time verification and validation within the domain of mission-critical self-adaptive systems.

Ismail and Cardellini [29] proposed an architectural model for decentralised self-adaptation to support the development of cloud-based systems. In particular, their architecture allows the developer to define a self-adaptive system capable of supporting information-sharing patterns between a set of self-adaptive subsystems. Thus, such subsystems can manage the cross-layer and a multi-cloud environment. Compared to our approach, we defined not only this information-sharing pattern (*i.e.,* the components responsible for managing shared Knowledge data that belongs to both micro-controllers and the meta-controller); we also established architectures and concepts to define flexible controllers.

Finally, in a systematic literature review, Quin et al. [43] characterised the state-of-the-art of decentralised self-adaptive systems. They highlighted the increasing ubiquity and scale of self-adaptive systems. Differently from our work, their study has a particular interest in systems that have controllers exclusively based on the MAPE-K, with the MAPE-K stages (*e.g.,* monitoring and planning) realised by multiple inter-coordinated components. In that direction, the micro-controllers proposed in our work may also be split into inter-communicating components, hence turning themselves into decentralised controllers.

(4) Reuse for self-adaptive systems:

Some studies [36, 37] addressed the usage of containerised microservices as a primary abstraction to build target systems. Other studies addressed controllers by using MAPE-K blueprint [31], design patterns [44], and control patterns [59]. Technical reuse of the controller is also addressed. For instance, in Rainbow [20], the level of abstraction is raised to the software architecture so that a controller performs generic architectural adaptation by adding, removing, and reconfiguring components in a target system. Rainbow has to be tailored with target system-specific gauges and effectors that bridge the abstraction gap, using model-driven engineering techniques [7, 56].

Some approaches address execution engines for controllers specified by models [28, 57, 58]. While such reuse eases the development of controllers [58], the models must be created for each specific target system. To create models, reusable templates for each MAPE-K controller stage exist [15]. Similarly but for code-based development, Krupitzer et al. [33] provided reusable templates for these stages. The controllers resulting from these approaches fail to address the diverse needs of target systems, as the unit of reuse is limited to either a monolithic controller (consisting of templates that typically restrict the structure or behaviour of controllers), or model execution engines. In contrast, we address a wide range of needs of target systems by orchestrating a controller from a collection of generic micro-controllers.

8 CONCLUSIONS AND FUTURE WORK

This paper has presented and evaluated a novel approach, based on micro-controllers implemented as microservices, to design and deploy multi-layered controllers for self-adaptive software systems. The feasibility of the approach, which promotes flexibility and reuse, has been demonstrated and evaluated using three different controller architectural configurations (monolithic, decentralised, and decentralised with a meta-controller), and deployed on the Kube-ZNN exploratory study. The evidence collected for the evaluation consists of four key Kube-ZNN attributes regarding the target system, namely: structural adaptation in terms of the number of servers (*i.e.,* scalability of resources), parametric adaptation related to the content provided by the servers (*i.e.,* fidelity of content), the response time of the target system, and the stability of the target system. We also performed experiments regarding the evolution of the controller by adding a new micro-controller. From the results obtained, we conclude that designing structurally flexible controllers based on micro-controllers promotes flexibility without compromising target system quality attributes, including response time and resource stability. Even though the evolved version of the controller that includes a meta-controller (*i.e.,* Meta-KZ-New) was outperformed by its simpler version (*i.e.,* Meta-KZ), Meta-KZ-New was still faster than Mon-KZ ($p\text{-}value$=$4.040 * 10^{-2}$ for Mann-Whitney U test) and Des-KZ ($p\text{-}value$=$1.056 * 10^{-7}$ for Mann-Whitney U test) and has the benefit of cleaning up the Kubernetes execution environment with the removal of non-operational pods.

Based on the results achieved with the approach proposed and explored in this article, we envision a wide range of self-adaptive software systems for which our proposal would be suitable. Taking as a reference the Self-Adaptive Systems Artifacts [48], several exemplars could make use of micro-controllers. For example, in the Body Sensor Network exemplar [24], both the Strategy Manager and the Strategy Enactor could be implemented as micro-controllers. Since there are just two of them, a simple choreography, without a meta-controller, could be implemented to coordinate their activities. In the case of the SEAByTE exemplar [42], instead of having a single Feedback Loop managing to perform A/B testing on several components, we could have several micro-controllers each implementing the Feedback Loop to manage the A/B testing of each component.

In future work, a key challenge to be handled is the potential increase in the risk of having multiple feedback control loops associated with the different micro-controllers. Moreover, coordination mechanisms are needed for dealing with potential conflicts amongst the micro-controllers' decisions. The stability investigated in this work also demonstrated the need to conduct future studies that address different comparisons of timing changes. Another challenge that needs to be addressed relates to the actual controller reuse. This requires a different kind of study involving different applications and repositories of micro-controllers aiming to collect evidence for substantiating the benefits that multi-layered controllers might bring.

ACKNOWLEDGEMENTS

We sincerely thank the anonymous reviewers for their detailed and insightful feedback, which has greatly enriched the quality of this article.

This work was partly supported by: Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CNPq (grant #312086/2021-0), FAPESP (grant #2019/25307-8), and Federal University of São Carlos. We would also like to express our gratitude to Carlos Aderaldo and Nabor Mendonça (Unifor, Brazil), who supported us to configure and customise Kubow to allow us to conduct our experiments, and to Thomas Vogel (Humboldt-Universität zu Berlin, Germany), who contributed to devise the approach for developing flexible controllers presented in our prior papers.

REFERENCES

[1] C. M. Aderaldo and N. C. Mendonça. 2022. Kube-ZNN Repository. Online. https://github.com/ppgia-unifor/kubow/tree/master/samples/samples/kube-znn - accessed in February, 2023.

[2] C. M. Aderaldo, N. C. Mendonça, B. Schmerl, and D. Garlan. 2019. Kubow: An Architecture-Based Self-Adaptation Service for Cloud Native Applications. In *Proceedings of the 13th European Conference on Software Architecture (ECSA) – Demos Track*. ACM, Paris, France, 42–45.

[3] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. 2009. Reflecting on Self-Adaptive Software Systems. In *Proceedings of the 4th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, Vancouver, BC, Canada, 38–47.

[4] A. Banijamali, P. Kuvaja, M. Oivo, and P. Jamshidi. 2020. Kuksa*: Self-adaptive Microservices in Automotive Systems. In *Product-Focused Software Process Improvement*, M. Morisio, M. Torchiano, and A. Jedlitschka (Eds.). Springer International Publishing, Cham, 367–384.

[5] V. R. Basili. 1992. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm.* Technical Report. University of Maryland at College Park, College Park, MD, USA.

[6] K. Baylov and A. Dimov. 2017. Reference Architecture for Self-adaptive Microservice Systems. In *Intelligent Distributed Computing XI*. Springer, Belgrade, Serbia, 297–303.

[7] A. Bennaceur, R. France, G. Tamburrelli, T. Vogel, P. J. Mosterman, W. Cazzola, F. M. Costa, A. Pierantonio, M. Tichy, M. Akşit, P. Emmanuelson, H. Gang, N. Georgantas, and D. Redlich. 2014. Mechanisms for Leveraging Models at Runtime in Self-adaptive Software. In *Models@run.time: Foundations, Applications, and Roadmaps*. Springer, Cham, Switzerland, 19–46.

[8] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel. 2015. MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaptation. In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering* (Bergamo, Italy) *(CTSE 2015)*. ACM, New York, NY, USA, 9–16.

[9] B. Burns, J. Beda, and K. Hightower. 2019. *Kubernetes: Up and Running: Dive into the Future of Infrastructure.* O'Reilly Media, O'Reilly Media. https://books.google.com.br/books?id=-5izDwAAQBAJ

[10] R. Calinescu, S. Gerasimou, and A. Banks. 2015. Self-adaptive Software with Decentralised Control Loops. In *Fundamental Approaches to Software Engineering*, A. Egyed and I. Schaefer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 235–251.

[11] J. Cámara, P. Correia, R. de Lemos, D. Garlan, P. Gomes, B. Schmerl, and R. Ventura. 2013. Evolving an Adaptive Industrial Software System to Use Architecture-based Self-adaptation. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, San Francisco, CA, USA, 13–22.

[12] J. Cámara, R. de Lemos, N. Laranjeiro, R. Ventura, and M. Vieira. 2014. Testing the Robustness of Controllers for Self-Adaptive Systems. *Journal of the Brazilian Comp. Society* 20, 1 (2014), 1–14.

[13] S. W. Cheng and D. Garlan. 2012. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software* 85, 12 (2012), 2860–2875. https://doi.org/10.1016/j.jss.2012.02.060 Self-Adaptive Systems.

[14] S-W. Cheng, D. Garlan, and B. Schmerl. 2009. Evaluating the Effectiveness of the Rainbow Self-adaptive System. In *Proceedings of the 4th International Workshop on Software Engineering for Adaptive and Self-managing Systems (SEAMS)*. IEEE, Vancouver, 132–141.

[15] D. G. de la Iglesia and D. Weyns. 2015. MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 10, 3 (2015), 15:1–15:31.

[16] R. de Lemos and P. Potena. 2017. Identifying and Handling Uncertainties in the Feedback Control Loop. In *Managing Trade-Offs in Adaptable Software Architectures* (1st ed.). Science Direct, Boston, 353–367.

[17] P. Di Francesco, P. Lago, and Iv. Malavolta. 2019. Architecting with microservices: A systematic mapping study. *Jounal of Systems and Software* 150 (2019), 77–97.

[18] P. Dragan, A. Metzger, and K. Pohl. 2023. Towards the Decentralized Coordination of Multiple Self-adaptive Systems. In *Proceedings of the 4th International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, Toronto, ON, Canada, 107–116. https://doi.org/10.1109/ACSOS58161.2023.00028

[19] L. Florio and E. Di Nitto. 2016. Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures. In *Proceedings of the 13th IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, Wurzburg, Germany, 357–362.

[20] D. Garlan, S-W. Cheng, A-C. Huang, B. R. Schmerl, and P. Steenkiste. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer* 37, 10 (2004), 46–54.

[21] D. Garlan, R. T. Monroe, and D. Wile. 2000. *Acme: architectural description of component-based systems.* Cambridge University Press, USA, 47–67.

[22] I. Gerostathopoulos, T. Bures, P. Hnetynka, A. Hujecek, F. Plasil, and D. Skoda. 2017. Strengthening Adaptation in Cyber-Physical Systems via Meta-Adaptation Strategies. *ACM Trans. Cyber-Phys. Syst.* 1, 3 (apr 2017), 25 pages. https://doi.org/10.1145/2823345

[23] I. Gerostathopoulos, D. Skoda, F. Plasil, T. Bures, and A. Knauss. 2019. Tuning self-adaptation in cyber-physical systems through architectural homeostasis. *Journal of Systems and Software* 148 (2019), 37–55.

[24] E. B. Gil, R. Caldas, A. Rodrigues, G. L. G. da Silva, G. N. Rodrigues, and P. Pelliccione. 2021. Body Sensor Network: A Self-Adaptive System Exemplar in the Healthcare Domain. In *Proceedings of the 16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) – Artifact Track*. IEEE, Madrid, Spain, 224–230.

[25] T. Glazier and D. Garlan. 2019. An Automated Approach to Management of a Collection of Autonomic Systems. In *Proceedings of the 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. IEEE, Umea, Sweden, 110–115. https:

//doi.org/10.1109/FAS-W.2019.00038
[26] S. Hassan and R. Bahsoon. 2016. Microservices and Their Design Trade-offs: A Self-Adaptive Roadmap. In *Proceedings of the 13th IEEE International Conference on Services Computing (SCC)*. IEEE, San Francisco, CA, USA, 813–818.
[27] IBM. 2005. *An Architectural Blueprint for Autonomic Computing*. Technical Report. IBM.
[28] M. U. Iftikhar and D. Weyns. 2014. ActivFORMS: Active Formal Models for Self-Adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, Hyderabad, India, 125–134.
[29] A. Ismail and V. Cardellini. 2015. Decentralized Planning for Self-Adaptation in Multi-cloud Environment. In *Advances in Service-Oriented and Cloud Computing*, Guadalupe Ortiz and Cuong Tran (Eds.). Springer International Publishing, Cham, 76–90.
[30] P. Jamshidi, C. Pahl, C. N. Mendonça, J. Lewis, and S. Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (2018), 24–35.
[31] J. O. Kephart and D. M. Chess. 2003. The Vision of Autonomic Computing. *IEEE Computer* 36, 1 (2003), 41–50.
[32] C. Krupitzer, F. M. Roth, M. Pfannemüller, and C. Becker. 2016. Comparison of Approaches for Self-Improvement in Self-Adaptive Systems. In *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, Wurzburg, Germany, 308–314.
[33] C. Krupitzer, F. M. Roth, S. Vansyckel, and C. Becker. 2015. Towards Reusability in Autonomic Computing. In *Proceedings of the 2015 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, Grenoble, France, 115–120.
[34] Henry B. Mann and Douglas R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics* 18 (1947), 50–60.
[35] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. 2004. Composing Adaptive Software. *IEEE Computer* 37, 7 (2004), 56–64.
[36] N. C. Mendonça, D. Garlan, B. R. Schmerl, and J. Cámara. 2018. Generality vs. Reusability in Architecture-based Self-adaptation: The Case for Self-adaptive Microservices. In *Proceedings of the 1st International Workshop on Architectural Knowledge for Self-adaptive Systems (AKSAS)*. ACM, Madrid, Spain, 18:1–18:6.
[37] N. C. Mendonça, P. Jamshidi, D. Garlan, and C. Pahl. 2019. *Developing Self-Adaptive Microservice Systems: Challenges and Directions*. Technical Report arXiv:1910.07660v2. University of Fortaleza (Fortaleza, CE, Brazil); University of South Carolina (Columbia, SC, USA); Carnegie Mellon University (Pittsburgh, PA, USA); Free University of Bozen-Bolzano (Bozen-Bolzano, Italy).
[38] V. Nallur and R. Bahsoon. 2013. A Decentralized Self-adaptation Mechanism for Service-based Applications in the Cloud. *IEEE Transactions on Software Engineering (TSE)* 39, 5 (2013), 591–612.
[39] H. P. Nii. 1986. Blackboard Systems Part Two: Blackboard Application Systems. *AI Magazine* 7, 3 (1986), 82–106.
[40] T. Patikirikorala, A. Colman, J. Han, and L. Wang. 2012. A Systematic Survey on the Design of Self-Adaptive Software Systems Using Control Engineering Approaches. In *SEAMS*. IEEE Press, Zurich, Switzerland, 33–42.
[41] J. D. Pereira, R. Silva, N. Antunes, J. L. M. Silva, B. de França, R. Moraes, and M. Vieira. 2020. A Platform to Enable Self-Adaptive Cloud Applications Using Trustworthiness Properties. In *Proceedings of the 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, New York, NY, USA, 71–77.
[42] F. Quin and D. Weyns. 2022. SEAByTE: A Self-adaptive Micro-service System Artifact for Automating A/B Testing. In *Proceedings of the 17th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE Computer Society, Pittsburgh, PA, USA, 77–83. https://doi.org/10.1145/3524844.3528081
[43] F. Quin, D. Weyns, and O. Gheibi. 2021. Decentralized Self-Adaptive Systems: A Mapping Study. In *Proceedings of the 16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, Madrid, Spain, 18–29.
[44] A. J. Ramirez and B. H. C. Cheng. 2010. Design Patterns for Developing Dynamically Adaptive Systems. In *Proceedings of the 5th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, Cape Town, South Africa, 49–58.
[45] F. Rossi, V. Cardellini, and F. Presti. 2020. Hierarchical Scaling of Microservices in Kubernetes. In *Proceedings of the 1st International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, Washington, USA, 28–37. https://doi.org/10.1109/ACSOS49614.2020.00023
[46] A. R. Sampaio Jr., J. Rubin, I. Beschastnikh, and N. S. Rosa. 2019. Improving Microservice-based Applications with Runtime Placement Adaptation. *Journal of Internet Services and Applications (JISA)* 10, 4 (2019), 1–30.
[47] B. Schmerl, J. Cámara, G. A. Moreno, D. Garlan, and A. Mellinger. 2014. *Architecture-Based Self-Adaptation for Moving Target Defense*. Technical Report CMU-ISR-14-109. Carnegie Mellon University.
[48] self-adaptive.org. 2023. Software Engineering for Self-Adaptive Systems. Online. https://www.hpi.uni-potsdam.de/giese/public/selfadapt/ - accessed in February, 2023.
[49] S. S. Shapiro and M. B. Wilk. 1965. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika* 52, 3-4 (1965), 591–611.
[50] A. M. Sharifloo and A. Metzger. 2013. MCaaS: Model Checking in the Cloud for Assurances of Adaptive Systems. In *Software Engineering for Self-Adaptive Systems III. Assurances*. Springer, Cham, 137–153 (LCNS 9640).
[51] M. Shaw. 2002. "Self-Healing": Softening Precision to Avoid Brittleness: Position Paper for WOSS '02: Workshop on Self-Healing Systems. In *Proceedings of the 1st Workshop on Self-Healing Systems*. ACM, New York, NY, USA, 111–114.

[52] David J. Sheskin. 2000. *Handbook of Parametric and Nonparametric Statistical Procedures* (2nd ed.). Chapman & Hall/CRC, Boca Raton, FL, USA.
[53] C. E. Silva and R. de Lemos. 2011. Dynamic Plans for Integration Testing of Self-adaptive Software Systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, Waikiki, Honolulu, HI, USA, 148–157.
[54] B. R. Siqueira, F. C. Ferrari, and R. de Lemos. 2023. Design and Evaluation of Controllers based on Microservices. In *Proceedings of the 18th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, Melbourne, Australia, 13–24. https://doi.org/10.1109/SEAMS59076.2023.00013
[55] B. R. Siqueira, F. C. Ferrari, T. Vogel, and R. de Lemos. 2020. Micro-controllers: Promoting Structurally Flexible Controllers in Self-Aware Computing Systems. In *Proceedings of the 1st Workshop on Self-Aware Computing (SeAC'20)*. IEEE, Washington DC, USA, 188–193.
[56] T. Vogel and H. Giese. 2010. Adaptation and Abstract Runtime Models. In *Proceedings of the 5th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, Cape Town, South Africa, 39–48.
[57] T. Vogel and H. Giese. 2012. A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, Zurich, Switzerland, 129–138.
[58] T. Vogel and H. Giese. 2014. Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 8, 4 (2014), 18:1–18:33.
[59] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. 2013. On Patterns for Decentralized Control in Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw (Eds.). Springer, Heidelberg, Germany, 76–107.
[60] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering* (1st ed.). Springer, Springer.

Received 19 December 2023; revised 12 December 2024; accepted 16 December 2024