# Design and Evaluation of Controllers based on Microservices

Bento R. Siqueira
*Federal University of São Carlos*
São Carlos, SP, Brazil
brsiqueira@estudante.ufscar.br

Fabiano C. Ferrari
*Federal University of São Carlos*
São Carlos, SP, Brazil
fcferrari@ufscar.br

Rogério de Lemos
*University of Kent*
Canterbury, United Kingdom
r.delemos@kent.ac.uk

***Abstract*—In self-adaptive software systems, generic controllers can be configured parametrically according to system needs, even though their reuse is restricted because of the wide range of services that can be provided by each of the stages of a feedback control loop, like MAPE-K. Rainbow is a typical example of such a generic, monolithic controller. This paper advocates controllers that are structurally flexible, and which are composed from micro-controllers, each providing specific services (e.g., based on microservices). To provide evidence on the feasibility of our approach, we compare three different architectural configurations for the controller: monolithic, decentralised, and decentralised with a meta-controller. Results from our experiments indicate that even though the decentralised configuration with a meta-controller demanded more computational resources, it performed comparatively well when compared to the other configurations. 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.**

***Index Terms*—self-adaptive software systems; feedback control loop; flexible controller; microservices**

## I. Introduction

In self-adaptive software systems, Rainbow [14] is one of the few examples of a *generic* controller that has been used in several application domains [34] and by different researchers [9]. Many of the existing controllers cannot be reused across different applications and by different developers. In general, 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, which successfully reduces the coupling between the controller and system [14]. However, the wide range of applications, *i.e.,* target software systems, may also impose a wide range of services that are expected from the different stages of a controller (such as controller based on the MAPE-K reference architecture [20]). 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 is able to 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 [21]. In contrast, a controller whose services are composed according to the actual needs of the target system may achieve a higher flexibility and variability than a monolithic controller [23]. For achieving such modular and structurally flexible approach, controller components implementing specific services should become structurally independent from each other. Essentially, the claim being made in this paper 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* that are able to configure the controller according to services required by the target system, thus achieving structure flexibility at the controller level [23].

These micro-controllers would consist of microservices, and in a previous work, 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 [39]. In this paper, we describe in detail our view of multi-layer controllers, and evaluate whether a controller which is based on micro-controllers has the advantage of being structurally flexible without compromising the performance of the target system. For evaluating the proposed approach, we have employed as an exploratory study a variant of ZNN.COM [10], namely, Kube-ZNN [1], using three different architectural configurations for the controller: monolithic, decentralised, and decentralised with a meta-controller. In the context of promoting flexible controllers in self-adaptive software systems, the key contributions of this paper are:

- Definition of an approach, based on microservices, for enabling structurally flexible controllers.
- Evaluation and comparison of three different architectural configurations for controllers.

Results from the experiments we performed have indicated that our multi-layer approach has little impact on performance of the target system. It was also noticed that there are some situations in which the decentralised controller with a meta-controller outperformed the other two configurations.

The remainder of this paper is organised as follows. Section II presents an overview of the main concepts about Rainbow, Docker, Kubernetes and Kubow. Section III presents the approach. Section IV introduces an exploratory study. In Section V, we demonstrate and evaluate the proposed approach using the exploratory study. Section VI discusses threats and limitations of our research. Finally, we discuss related work in Section VII, and conclude the paper in Section VIII.

## II. Background

### A. Rainbow

Rainbow provides a reusable infrastructure and a set of tools for supporting self-adaptation, thus allowing for its effortless customisation across different target systems [14]. Rainbow relies on the architecture model of the target system, and a collection of adaptation strategies defined in the Stitch language. Rainbow contains several components to store the architectural model, evaluating the need for an adaptation, selecting the appropriate adaptation, and executing the adaptation.

Despite the fact that Rainbow supports the development of generic controllers, Rainbow-based controllers are typically monolithic since all its components are used as a whole – they can be configured, but they cannot be changed (for example, for incorporating new services).

### B. 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, and for each application, REST APIs are defined for allowing the applications to be accessed.

Kubernetes[2] is an open source based container orchestration platform for automating deployment, scaling, and management of containerised applications [8]. In this platform, the main object is a *pod* that 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 is able to be deployed in a Kubernetes cluster, given that all its dependencies are specified in a configuration file. In other words, a Kubernetes cluster allows to define a complete software ecosystem composed of several sets of applications (*i.e.,* pods) that communicate with one another.

### C. 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 customisation and extension involve implementations, such as, Rainbow probes and effectors using the Kubernetes APIs. These APIs allow to access 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 that allows the integration of other kinds of Kubernetes applications. For example, access metrics collected by Kubernetes' own monitoring services (*e.g.,* Kubelet and cAdvisor), as well as, by other external monitoring tools (*e.g.,* Prometheus).

[1]https://docs.docker.com/get-started/overview/ – accessed in Feb, 2023.
[2]https://kubernetes.io/ – accessed in Feb, 2023.
[3]https://github.com/ppgia-unifor/kubow – accessed in Feb, 2023.

## III. 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 easily 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 [7, 18, 28], and it fits quite well into the hierarchical control pattern [43]. Promoting multi-layered controllers is to decouple the controller from the target system because of the target system's intrinsic intricacies and complexities. The tight coupling between target system and controller may hinder the reuse of controllers across different applications since systems provide different services with different levels of quality.

A self-adaptive system is composed by 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). Whereas the locus of adaptation is the controller, which empirically observes changes for building reasoning models that allows it to control the target system. Adaptations can either be parametric or structural [3, 23].

The approach being proposed, for decoupling the controller from the target system, is to introduce a two-layer controller, as shown in Figure 1. 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 [14], that should be able to control the controller layer by adapting its structure and parameters. Although parametric adaptations of monolithic controllers are more common [21], 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 (*i.e.,* once deployed, they are not expected to change): controllers can both be the locus of change *and* the locus of adaptation.

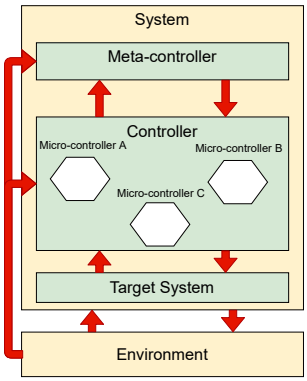



Fig. 1. Multi-layered controller.

### A. Micro-controllers

The key idea being promoted is to use an ensemble of micro-controllers instead of a monolithic controller, like an implementation of a MAPE-K loop [14, 20]. These micro-controllers would not be restricted to the implementation of services provided by the distinct stages of a MAPE-K loop [29]. Instead, the micro-controllers would be associated with specific services that make up the individual stages of a controller [12]. For example, for the Analysis stage of the MAPE-K loop, a micro-controller would implement the services associated with integration testing [38], or model checking [36]. The number of micro-controllers for implementing a controller would depend on the needs of a target system, and the granularity of 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 testing strategy if the achieved test coverage of the target system is not sufficient). 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 realised as microservices. Each micro-controller can be developed as a separate process for maximising independent deployment. Some of the benefits of implementing controllers as a collection of micro-controllers, include, independent development and deployment, dynamic operational support, like, versioning and scaling.

### B. Meta-controller

If the controller, realised as an ensemble of micro-controllers, in addition of being the locus of adaptation is also the locus of change, there may be the need for an additional controller, at a higher-level, depending on its complexity. This *meta-controller* would manage the changes occurring at the controller by adapting the ensemble of micro-controllers. An example of such controller could be Rainbow [14], 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 is responsible for orchestrating 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 [27], including the state of the micro-controllers, thus allowing the micro-controllers to be stateless. Although micro-controllers should be independent, the coordination between micro-controllers could follow the control flow of a MAPE-K loop [20]. Several micro-controllers do not preclude that all micro-controllers are able to access the target system. Conflicts might occur amongst micro-controllers and should be handled by the meta-controller.

In summary, the multi-layered controllers approach being proposed is beneficial because the different needs of a target system are addressed by specific micro-controllers. This is achieved because of the structural flexibility of the ensemble of micro-controllers, 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, whereas the meta-controller reflects on changes affecting the controller and the system environment, which makes the overall system meta-self-adaptive. Although in this paper we have described a multi-layer controller consisting of two layers, conceptually, a controller could have more than two layers [28].

## IV. Exploratory Study

This section describes the implementation of our multi-layer controller approach. First, we describe the exploratory study. Then, we present the controller configurations employed for evaluating our approach. Finally, we describe the environmental set up used for performing the experiments. The artefacts used in our research are available online[4].

### A. Exploratory Study: ZNN.com and Kube-ZNN

We use the target system Kube-ZNN [1], which is based on the ZNN.com [10] which reproduces a typical infrastructure for a news website. ZNN.com is a three-tier architecture consisting of servers providing news content to clients. The goal of ZNN.com is to provide content to customers 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 to deal with requests across a pool of replicated servers. The number of servers can be adapted according to service demand. In our experiment setup (described in Section V), we have used media quality (*i.e.,* contents to user) and size of the server pool for adapting the system according to demand (*i.e.,* number of user requests). Our measurements regarding the size of the server pool and the quality of the media delivered at particular moments (namely, right after the peak of demand, and right after the demand decrease) provide evidence on the capacity of the system to accomplish its goals.

**Quality Attributes:** Two ZNN.com key quality attributes are:

- **Scalability:** In case 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 brings elasticity to the system.
- **Fidelity:** Depending on the costs associated with service demand, different levels of fidelity are provided for supporting client experience (*e.g.,* usage of different resources, such as, *text*, *images*, or *videos* implies different costs). For

[4]https://github.com/californi/ExperimentsForMicrocontrollers/

TABLE I
RULES IN THE ADAPTATION STRATEGIES.

| Stitch | | | | Architecture | |
|---|---|---|---|---|---|
| **Attribute** | **Strategy** | **Tactic** | **Predicate** | **Configuration** | **Component** |
| Scalability | IncreaseServers | addReplica | lowSLO && canAddReplica | Mon-KZ<br>Des-KZ<br>Meta-KZ | Controller<br>ScalabilityA, ScalabilityB<br>ScalabilityA, ScalabilityB |
| Scalability | DecreaseServers | removeReplica | highSLO && canRemoveReplica | Mon-KZ<br>Des-KZ<br>Meta-KZ | Controller<br>ScalabilityA, ScalabilityB<br>ScalabilityA, ScalabilityB |
| Fidelity | IncreaseMediaSize | raiseFidelity | highSLO && lowFidelity | Mon-KZ<br>Des-KZ<br>Meta-KZ | Controller<br>FidelityA, FidelityB<br>FidelityA, FidelityB |
| Fidelity | DecreaseMediaSize | lowerFidelity | lowSLO && cannotAddReplica | Mon-KZ<br>Des-KZ<br>Meta-KZ | Controller<br>FidelityA, FidelityB<br>FidelityA, FidelityB |
| Failures | ActivateNoFailureRate | addHighScalabilityHighQuality | noFailureRate | Mon-KZ<br>Meta-KZ | Controller<br>Meta-controller |
| Failures | ActivateLowFailureRate | activateLowScalabilityHighQuality | lowFailureRate && isScalabilityA | Mon-KZ<br>Meta-KZ | Controller<br>Meta-controller |
| Failures | ActivateHighFailureRate | addLowScalabilityLowQuality | highFailureRate && isScalabilityA | Mon-KZ<br>Meta-KZ | Controller<br>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 |

instance, if the maximum limit on the number of servers is achieved, the servers start providing images instead of videos, as a way to properly fulfil the service demand. Similar to the elasticity for scalability, once the service demand has decreased, the fidelity of content can be increased.

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 object in Kubernetes. The client requests are handled by a service object which is responsible for distributing and delivering at run-time the requests according to the load.

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

- IncreaseServers: it increases the number of servers available, and relies on tactic addReplica. Since there are more resources, the latency of Kube-ZNN is decreased.
- DecreaseServers: in contrast to IncreaseServers, it decreases the number of servers available, and uses the tactic removeReplica. Thus, the latency of Kube-ZNN tends to increase when the number of user requests is increased.
- IncreaseMediaSize: it improves the media quality provided by the target system. In Kube-ZNN the delivered media quality is normally high, but when the number of requests increases the latency may increase.
- DecreaseMediaSize: in contrast to the previous strategy, this strategy ensures that the latency is kept at an acceptable level by reducing media quality, when the number of client requests starts increasing the latency 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 trade-off between scalability and fidelity. In other words, the number of failures becomes an observable variable that can influence our adaptation strategies.

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 there is no CPU available, then a CPU failure is raised. The same occurs with memory and storage, once that each pod has minimal requirements. All the raised failures may be handled by the FailureManager, even though in this evaluation study we focus on CPU failures, which are handled by using the Kubernetes API.

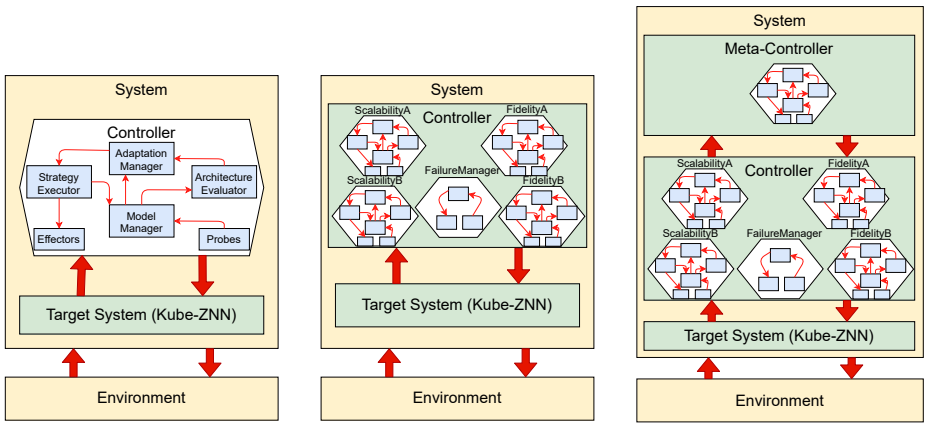
**Additional strategies:** Based on the number of failures attribute, three additional strategies are defined:

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

Table I 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 IncreaseServers, and its tactic addReplica, both variables lowSLO and canAddReplica must be true. The lowSLO variable shows whether the number of user requests lost is low, whereas canAddReplica shows when the limit of servers is achieved.

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.




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

Fig. 2. Architectures for configurations Mon-KZ, Des-KZ and Meta-KZ

- 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 (named Des-KZ, described in the next section), we derived eight specific strategies, which are embedded in the micro-controllers of that configuration. We highlight that those specific strategies precisely reflect the rationale embedded in the ActivateNoFailureRate, ActivateLowFailureRate, and ActivateHighFailureRate aforementioned strategies. These strategies enable the micro-controllers to adapt themselves, depending on the current rate of failures affecting the server replicas. Those specific strategies are listed in the last eight lines of Table II. 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 work with a lower number of active servers until the number of failures decreases again.

### B. Controller Configurations

To demonstrate and evaluate our approach, we have employed three functionally equivalent controller configurations that are representative of existing solutions. Figure 2 illustrates the structures and the relationship between the main components of the three architectural configurations.

**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] in order to include the new set of strategies and tactics to deal with the number of failures in the server replicas. As shown in Figure 2(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 higher or lower number of active servers in the target system (*i.e.,* the adaptations w.r.t. scalability property), and higher or lower quality of media delivered to the users (*i.e.,* the adaptations w.r.t. fidelity property).

**Des-KZ**: This is a decentralised implementation of configuration Mon-KZ, as shown in Figure 2(b). It consists of five micro-controllers, from which four are implemented as Kubow instances (namely, ScalabililyA, ScalabililyB, FidelityA and FidelityB), and one is tailor-made (FailureManager) in order to demonstrate the possibility of integrating heterogeneous controller designs. To reproduce the behaviour of the Mon-KZ controller, the micro-controllers of Des-KZ that deal with scalability and fidelity have two variants each: ScalabililyA and FidelityA, respectively, deal with high number of active servers and high media quality. Their variants, ScalabililyB and FidelityB, handle lower numbers of servers and reduced media quality. According to the strategies described earlier, only a single variant of each micro-controller is active at a particular moment, depending on the failure rate gathered by the FailureManager. As previously mentioned, the failures considered in our implementations concern Kube-ZNN failures related to the lack of CPU resources, even though FailureManager could be easily tailored to handle other kinds of failures.

**Meta-KZ**: This configuration includes a meta-controller for controlling the configuration of micro-controllers, as shown in Figure 2(c). Differently from Des-KZ, in which the choreography of micro-controllers is responsible for adaptation, 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 in Table I.

The process for decomposing the monolithic controller to build decentralised micro-controllers (for Des-KZ and

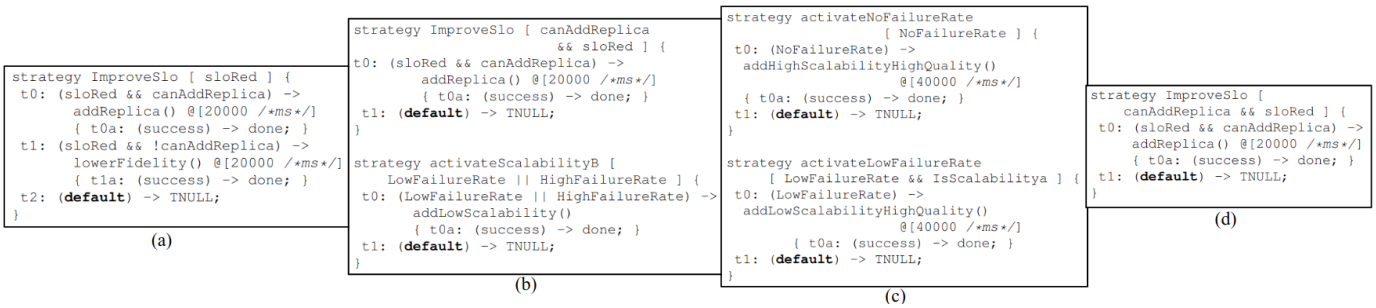



Fig. 3. Stitch snippets that illustrate the differences of strategy implementations in the controllers for Mon-KZ (a), Des-KZ (b), and Meta-KZ (c) and (d).

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 3 shows Stitch code snippets that illustrate the decomposition. For instance, the conditional sentence t0 in Figure 3(a) has been extracted into a variant of scalability micro-controller, as illustrated in Figures 3(b) and Figures 3(d). Specifically, Figure 3(a) shows the `ImproveSlo` strategy implemented in Mon-KZ; it mixes the addition of server replicas and the lowering of media quality when user requests are high and requests are lost. Figure 3(b) shows the same `ImproveSlo` strategy for the ScalabilityA micro-controller in Des-KZ, which now only focuses on adding replicas. Figure 3(b) also shows the `activateScalabilityB` strategy that activates the variant ScalabilityB when the failure 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 3(c); such strategies are responsible for activating the variants of the scalability and fidelity micro-controllers. Finally, Figure 3(d) shows that the ImproveSlo strategy implemented in the ScalabilityA micro-controller of Meta-KZ is exactly the same as in Des-KZ, but now the strategy for activating the variant of micro-controller is placed in the meta-controller.

**Resources for each configuration:** The two last columns of Table I summarise, for each architectural configuration, which controller component is responsible for implementing the adaptation strategies (refer to Figure 2).

Since decentralisation tends to increase the number of pods, Table II 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 limits of 10 (more details about this in Section V-A). As Table II summarises, the other components require one pod each, except FailureManager that requires two pods.

Regarding how much resource is demanded for each pod, the first two columns in Table II summarise such usage in terms of CPU and memory. These amounts of resources were defined based on the original Kubow implementation [2]. The units used to represent 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 total required memory and CPU for each configuration are listed in the last two lines of Table II.

TABLE II
ARCHITECTURAL CONFIGURATIONS AND THEIR RESOURCES.

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

* 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.

### C. Experimental Infrastructure

The experimental setup was the same for the demonstration, evaluation and comparison of the three 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 locally set up a single-node Kubernetes cluster; and Kubernetes CLI version v1.22.2 command line tools for controlling the Kubernetes clusters. To simulate a typical cloud environment and also to dynamically change the resource allocation, each application was deployed with all its tiers, inside its own virtual machine, *e.g.,* using a Minikube as a Kubernetes cluster.

For conducting the experiments, these steps we followed: 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 tool k6); 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.

## V. Evaluation

The evaluation consists in checking whether a controller which is based on micro-controllers has the advantage of being structurally flexible without compromising the performance of the target system. The performance of the target system is considered higher when the system responds to a given demand by using fewer active servers, or by delivering higher media quality than it would do in different execution setups. The evaluation comprises two research questions:

- RQ1: How does a decentralised controller perform regarding architectural adaptations of the target system when compared to a monolithic controller?
- RQ2: How does a decentralised controller perform regarding parametric adaptations of the target system when compared to a monolithic controller?

To answer these questions, we experimented with the three system configurations described in Section IV-B, namely, Mon-KZ, Des-KZ, and Meta-KZ. The evaluation of the state of the target system, in terms of scalability and fidelity, at particular execution moments, provides us with evidence to answer questions RQ1 and RQ2, respectively. Evidence was collected from the target system through the following metrics:

M1: The middle state regarding the number of active servers.
M2: The final state regarding the number of active servers.
M3: The middle state regarding the fidelity.
M4: The final state regarding the fidelity.

By middle state, we mean the state of the target system at the moment that represents the peak of system demand (*i.e.,* number of user requests). In our experiments, the peak happens at the end of the load period (which is described below, in Section V-A). The final state, on the other hand, represents the state of the target system right after the unload period.

### A. Experimental Execution Setups

We defined two execution scenarios that consider each a maximum number of 10 active Kube-ZNN servers. The scenarios involve the predefined times (namely, *short* and *long*) during which a load/unload of user requests are performed, therefore simulating how the controller and the target system are working. In this work we have used media quality and size of the server pool for adapting the system according to demand (*i.e.,* number of user requests). Our measurements regarding the size of the server pool and the quality of the media delivered at particular moments (namely, right after the peak of demand, and right after the demand decrease) provide evidence on the capacity of the system to accomplish its goals.

For *short* executions, we defined 2 minutes of load/unload, whereas for *long* executions, we defined 5 minutes. For each predefined time (*i.e.,* short or long), in each particular configuration (*i.e.,* Mon-KZ, Des-KZ and Meta-KZ), we ran 40 executions. For each set of 40 executions (80 executions of each configuration; 240 executions, in total), we gathered metrics M1, M2, M3 and M4. This allows us to to characterise the performance of a given configuration regarding the observed properties (*i.e.,* scalability and fidelity).

The distributions of values with respect to each set of 40 executions are represented as boxplot charts in Figures 4 to 5. In Figure 4, the *Y* axes in the charts represent the number of active Kube-ZNN servers, while the *Y* axes in the charts of Figure 5 represent the quality of the media delivered to the clients (ranging from 400KB to 800KB).

An example, let us consider the results for the scalability property that are shown in Figure 4(a) (Short Executions). For the 40 executions of Mon-KZ, at the end of the unload period (*i.e.,* the "Final" column for the Mon-KZ configuration), we may observe a larger concentration of results between the 2 to 5 interval, whereas the minimum and maximum numbers of active servers were 1 and 7, respectively).

### B. Scalability Results

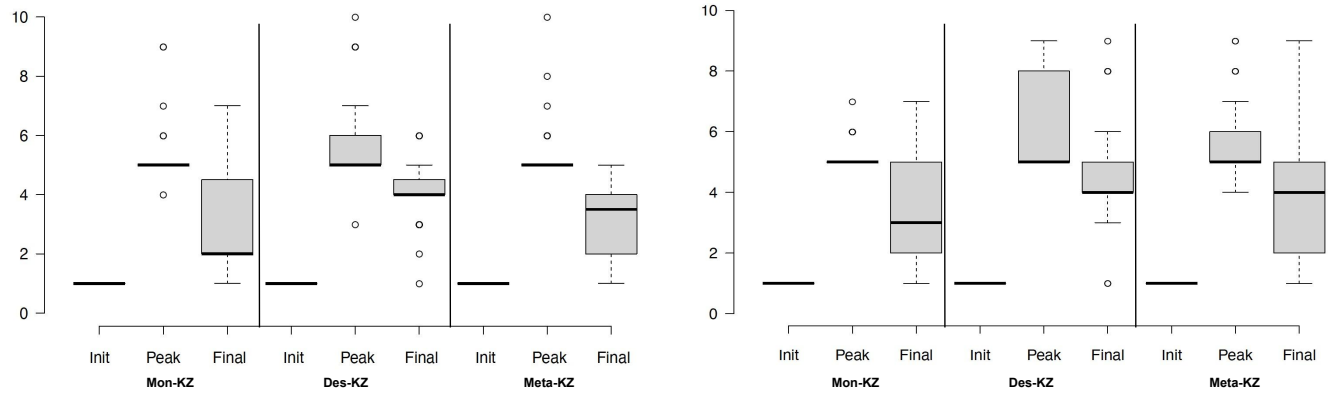
Figures 4(a) and 4(b) summarise the executions regarding the scalability of Mon-KZ, Des-KZ, and Meta-KZ. We next describe the results with focus on: (1) end of the load period, and (2) end of the unload period. We also compare the results for the three configurations.

**Scalability-related results at the peak of demand**: the Mon-KZ configuration reached 5 active servers in both short and long executions, across the 40 executions for each execution time. For the same scenarios, Des-KZ had a variation between 5 and 7 for short executions and a variation between 5 and 9 for long executions. Finally, Meta-KZ reached 5 active servers in short executions, and a variation between 4 and 7 for long executions.

**Scalability-related results at the end of the unload period:** Regarding short executions, Mon-KZ has a variation between 1 and 7 active servers (with higher concentration between 2 and 4). For Des-KZ, the final state has variation and concentration between 4 and 5 servers. Finally, for Meta-KZ the final state varies and concentrates between 2 e 5 servers.

Regarding long executions, Mon-KZ showed a variation between 1 and 7 servers (mostly concentrated between 2 and 5). Des-KZ results varied and concentrated between 3 and 6 servers, whereas Meta-KZ varied between 1 and 9 Kube-ZNN servers (with a concentration between 2 and 5).

**Comparison:** For the short executions, at the peak of demand, very similar results were observed for Mon-KZ and Meta-KZ, whereas Des-KZ required a slightly higher number of active servers. After the unload period, on the other hand, results for Meta-KZ produced better results (lower number of active servers). For the long executions, at demand peak Meta-KZ mostly required up to 6 active servers; in most Mon-KZ runs, 5 servers were required, while Des-KZ mostly required from 5 to 8 servers. On the other hand, after the unload period, Des-KZ once again kept more active servers than Mon-KZ and Meta-KZ; the two latter performed similarly.




(a) Short executions

(b) Long executions

Fig. 4. Results regarding scalability (init = initial state; peak = middle state (peak of system demand); final = final state (just after unload period).

We noticed that, when the demand is very high (*e.g.,* at peak load), 10 servers are active and many failures start to be generated by the Kubernetes environment and captured by FailureManager. Thus, as explained in Secton IV-B, the meta-controller adaptation rules – included as Stitch strategies – define that a new capacity to respond to requests will start working; more specifically, the meta-controller replaces ScalabilityA, which operates with a maximum of 10 active servers, with ScalabilityB, which operates with a maximum 4 active servers, until the failure rate returns to a low value. 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.

Regarding RQ1, in terms of structural adaptations of the target system, the Mon-KZ and Meta-KZ configurations performed better than Des-KZ, without a clear distinction between the performance of Mon-KZ and Meta-KZ.

### C. Fidelity Results

Figures 5(a), and 5(b) summarise the executions regarding the fidelity of configurations Mon-KZ, Des-KZ, and Meta-KZ. Similarly to the discussion presented for scalability, we next describe the results with focus on: (1) end of the load period, and (2) end of the unload period. We also compare the results for all configurations.

**Fidelity-related results at the peak of demand:** irrespective of the execution time (short or long), the Mon-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 higher levels of fidelity were observed for Des-KZ and Meta-KZ. For Des-KZ, the values are highly concentrated in 600KB, wheres the values range from 400KB to 600KB for Meta-KZ.

**Fidelity-related results at the end of the unload period:** with respect to the final states, in Mon-KZ the fidelity level ranged from 400KB and 600KB after the unload period. For Des-KZ the level was concentrated again in 600KB, and for Meta-KZ the fidelity level ranged from 400KB to 800KB.

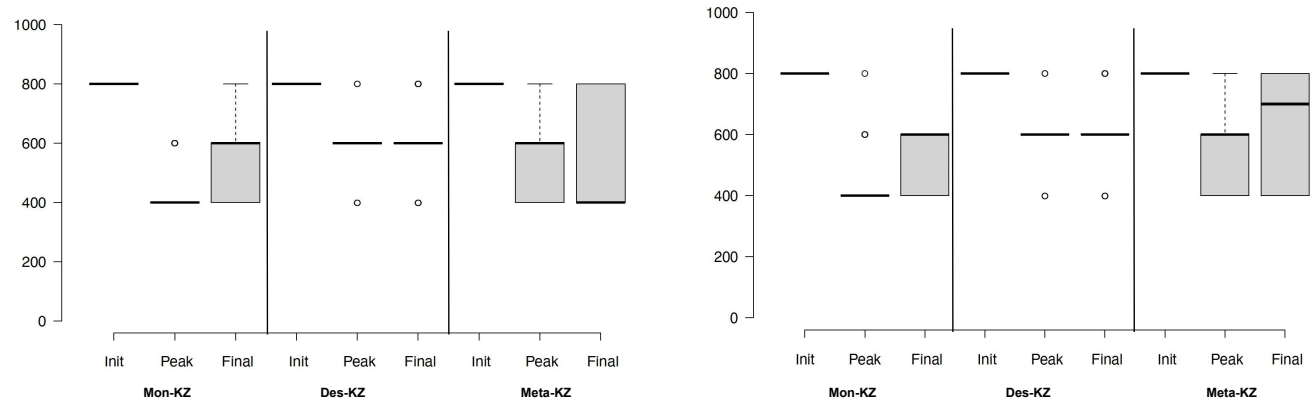
**Comparison:** For fidelity, Des-KZ and Meta-KZ outperformed Mon-KZ. From the results depicted in Figures 5(a) and 5(b), the Mon-KZ has decreased the level of fidelity to the minimum at the peak of demand, and was not able to restore it to the maximum level after the unload period. Regarding Des-KZ, it was able to keep the fidelity level in 600KB, despite the fact that no increase in the final state was observed. For Meta-KZ, when compared to Des-KZ, we observed slightly inferior results at the peak of demand, while slightly better results were achieved in the final state. Similarly to what happened with the variants of scalability micro-controllers, replacements of FidelityA by FidelityB and vice versa in Des-KZ and Meta-KZ were observed in the experiment runs.

Regarding RQ2, in terms of parametric adaptations of the target system, the Des-KZ and Meta-KZ configurations performed better than Mon-KZ, without a clear distinction between the performance of Des-KZ and Meta-KZ.

### D. Further discussion

When comparing the three configurations, setting up a higher number of Kube-ZNN servers may negatively impact the restoration time for the configurations that need to deal with larger numbers of components (namely, Des-KZ and Meta-KZ). In the context of our study, restoration regarding scalability means returning the target system to only one active server, whereas restoration regarding fidelity means increasing the quality of the media delivered to the clients to its maximum (*i.e.,* 800KB). Experiments to address the restoration time will be addressed in future work.

From a perspective of conflicts and synchronisation of the target system state, the Mon-KZ configuration contains a single set of adaptation strategies. As such, the Kubow controller was in charge to apply the strategies, without the need to solve issues related to conflicts among strategies,




(a) Short executions

(b) Long executions

Fig. 5. Results regarding fidelity (init = initial state; peak = middle state (peak of system demand); final = final state (just after unload period).

neither issues related to the synchronisation of the state of the target system. For the Des-KZ and Meta-KZ 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. That said, we have not noticed any significant impact regarding this issue on our experimental results

As shown in Table II, even though Meta-KZ demanded more resources (at least 3650m of CPU and 3.228Gi of Memory) when compared with configurations Mon-KZ (at least 1250m of CPU and 1.1Gi of Memory) and Des-KZ (at least 2650m of CPU and 2.228Gi of Memory), results for Meta-KZ were not compromised (cf. Figures 4(a), 4(b), 5(a), and 5(b)).

## VI. Threats to Validity

This section discusses the threats to the validity of our experiment according to the categories listed by Wohlin et al. [44]. Our experiment involved one *independent variable* (namely, the controller design approach) to which three *treatments* were applied (namely, monolithic, decentralised, and decentralised with a meta-controller). The *dependent variables* are scalability and fidelity observed in the target system when the different treatments are applied. The Kube-ZNN system was a single subject handled in the experiment.

**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 (processing uncertainty related to the periodical monitoring of controller components and target system components), 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 this threat, we performed 40 executions of each scenario and based our analysis on the median value of collected quantitative data.

**Construct validity:** In our study, a threat could be the lack of prior knowledge to design and operate with the adopted technologies, namely, Rainbow / Kubow and ZNN.com / Kube-ZNN . To mitigate it, we established communication with the original developers of Kubow and Kube-ZNN [2], who helped us to improve their implementations (the fixes were propagated to all configurations we used). We also relied on available documentation for other applied technologies.

Another construct threat concerns the interaction of different treatments. In this case, 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 and meta-controller. 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:** In our study, we used a single subject (Kube-ZNN ) and 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.

## VII. Related Work

Here we discuss four topics of related work, as follows:

**(1) Microservices for self-adaptive systems:** Baylov and Dimov [5] provided a reference architecture supporting the design of self-adaptive microservices, however it implies a specific structure that does not provide the same level of flexibility compared to our approach.

Hassan and Bahsoon [17] proposed the creation of a controller for microservice-based self-adaptive applications. Sampaio Jr. et al. [33] support the reconfiguration of microservice-based systems based on affinities and history of resource usage. In contrast, we promote architecting controllers based on micro-controllers (as microservices). Aligned with Mendonça et al.'s suggestion [25], we propose and

assess a concrete solution for deploying microservice-based controllers that are able to adapt at run-time.

**(2) Flexible controllers and meta-controllers:** Banijamali et al. [4] emphasise the increasing interest in microservices in self-adaptive systems, in particular, in the automotive domain. In their approach, which is similar to ours, they consider a controller consisting of microservices. However, they do not address the role of a coordinating entity for managing those microservices, which is addressed by Gerostathopoulos et al. [15] on their approach for incorporating homeostasis [37] into an additional control layer. This homeostasis layer is above the controller, as our meta-controller layer, and is responsible for changing adaptation strategies at the lower layer, at run-time. While Gerostathopoulos et al. [15] focused on mechanisms for the homeostasis layer, we focus on micro-controller-based controllers. We address is from as a bottom-up approach, so that micro-controllers could have more specific characteristics and functionalities.

Pereira et al. [29] proposed the development of flexible controllers that are based on MAPE-K (each sate 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 is able to reconfigure the controller.

**(3) Decentralised controllers:** Florio and Di Nitto [13] proposed adding autonomic capabilities (in a decentralised way) to containerised and microservice-based not originally designed to be autonomic. Nallur and Bahsoon [26] also developed a decentralised, multi-agent self-adaptation approach for web service-based systems deployed in the cloud. Considering both pieces of work [13, 26] and ours, key differences are: (i) both [13, 26] 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.

Recently, Quin et al. [31] reported on a systematic literature mapping that focused on characterising the state-of-the-art of decentralised self-adaptive systems. The authors justify the focus of their research by highlighting the increasing ubiquity and scale of self-adaptive systems. Differently from our work, their study has particular interest on systems that have controllers exclusively based on the MAPE-K, and that have the MAPE-K functions (*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 in inter-communicating components, and hence turning themselves into decentralised controllers.

**(4) Reuse for self-adaptive systems:** Mendonça et al. [24] discussed difficulties for reusing self-adaptation services and frameworks across different self-adaptive software systems. Their proposed solution is to use containerised microservices as a primary abstraction to build target systems. Other pieces of work support conceptual reuse of controllers by using MAPE-K blueprint [20], design patterns [32], and control patterns [43]. Technical reuse of the controller is also addressed. For instance, in Rainbow [14], 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. For this, Rainbow has to be tailored with target system-specific gauges and effectors that bridge the abstraction gap, which could be eased by using model-driven engineering techniques [6, 40].

Other approaches enable reuse of execution engines for controllers specified by models [19, 41]. The resulting models are specific for each target system but the engines executing these models are generic and reusable. This principle has been extended to individual controller stages, which allows reuse of engines at a more fine-grained level [42]. While such a reuse ease developing controllers, the models have to be created for each specific target system. To ease the creation of models, reusable templates for each MAPE-K controller stage exist [11]. Similarly but for code-based development, Krupitzer et al. [22] provided reusable templates for components of these stages. However, the controllers resulting from these approaches do not address the wide range of needs of target systems since the unit of reuse is either a monolithic controller, templates typically restricting the structure or behaviour of controllers, or model execution engines. In contrast, we address the wide range of needs of target systems by orchestrating a controller from a collection of generic micro-controllers.

## VIII. Conclusion and Future Work

This paper presented a novel approach, based on microservices, for the design and deployment of multi-layered controllers for self-adaptive software systems. The feasibility of the approach, that 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 was related to two key Kube-ZNN attributes: structural flexibility in terms of the number of servers (*i.e.,* scalability of resources), and parametric adaptation related to the content provided by the servers (*i.e.,* fidelity of content). From the results obtained, we conclude that the design of controllers based on micro-controllers allows for the definition of controllers that are structurally flexible, without compromising the overall system performance. Based on this, we envisage a whole range of self-adaptive software systems for which our proposal would be suitable. Taking as a reference the Self-Adaptive Systems Artifacts [35], there are several exemplars that could make use of micro-controllers. For example, in Body Sensor Network exemplar [16], 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 [30], instead of having a single Feedback Loop managing performing 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.

As future work, a key challenge to be handled is the potential increase in risk for having multiple feedback control loops

associated with the different micro-controllers. Moreover, coordination mechanisms need to be defined for dealing with potential conflicts amongst the micro-controllers' decisions. Another challenge that needs to be addressed is related 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.

## REFERENCES


[1] C. M. Aderaldo and N. C. Mendonça, "Kube-znn repository," Online, 2022, 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, "Kubow: An architecture-based self-adaptation service for cloud native applications," in *Proceedings of the 13th European Conference on Software Architecture (ECSA) – Demos Track*. Paris, France: ACM, 2019, pp. 42–45.

[3] J. Andersson, R. de Lemos, S. Malek, and D. Weyns, "Reflecting on self-adaptive software systems," in *SEAMS*. Vancouver, BC, Canada: IEEE, 2009, pp. 38–47.

[4] A. Banijamali, P. Kuvaja, M. Oivo, and P. Jamshidi, "Kuksa*: Self-adaptive microservices in automotive systems," in *Product-Focused Software Process Improvement*, M. Morisio, M. Torchiano, and A. Jedlitschka, Eds. Cham: Springer International Publishing, 2020, pp. 367–384.

[5] K. Baylov and A. Dimov, "Reference architecture for self-adaptive microservice systems," in *Intelligent Distributed Computing XI*. Belgrade, Serbia: Springer, 2017, pp. 297–303.

[6] 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, "Mechanisms for leveraging models at runtime in self-adaptive software," in *Models@run.time: Foundations, Applications, and Roadmaps*. Cham, Switzerland: Springer, 2014.

[7] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel, "Morph: A reference architecture for configuration and behaviour self-adaptation," in *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*, ser. CTSE 2015. New York, NY, USA: ACM, 2015, p. 9–16.

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

[9] J. Cámara, P. Correia, R. de Lemos, D. Garlan, P. Gomes, B. Schmerl, and R. Ventura, "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)*. San Francisco, CA, USA: IEEE, 2013, pp. 13–22.

[10] S.-W. Cheng, D. Garlan, and B. Schmerl, "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)*. Vancouver: IEEE, May 2009, pp. 132–141.

[11] D. G. de la Iglesia and D. Weyns, "MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 10, no. 3, pp. 15:1–15:31, 2015.

[12] R. de Lemos and P. Potena, "Identifying and handling uncertainties in the feedback control loop," in *Managing Trade-Offs in Adaptable Software Architectures*, 1st ed. Boston: Science Direct, 2017.

[13] L. Florio and E. Di Nitto, "Gru: An approach to introduce decentralized autonomic behavior in microservices architectures," in *Proceedings of the 13th IEEE International Conference on Autonomic Computing (ICAC)*. Wurzburg, Germany: IEEE, 2016, pp. 357–362.

[14] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, 2004.

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

[16] E. B. Gil, R. Caldas, A. Rodrigues, G. L. G. da Silva, G. N. Rodrigues, and P. Pelliccione, "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)*, 2021.

[17] S. Hassan and R. Bahsoon, "Microservices and their design trade-offs: A self-adaptive roadmap," in *Proceedings of the 13th IEEE International Conference on Services Computing (SCC)*. San Francisco, CA, USA: IEEE, 2016, pp. 813–818.

[18] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., Jun. 2005.

[19] M. U. Iftikhar and D. Weyns, "Activforms: Active formal models for self-adaptation," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Hyderabad, India: ACM, 2014, pp. 125–134.

[20] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[21] C. Krupitzer, F. M. Roth, M. Pfannemüller, and C. Becker, "Comparison of approaches for self-improvement in self-adaptive systems," in *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC)*. Wurzburg, Germany: IEEE, 2016, pp. 308–314.

[22] C. Krupitzer et al., "Towards reusability in autonomic computing," in *Proceedings of the 2015 IEEE International Conference on Autonomic Computing (ICAC)*. Grenoble, France: IEEE, 2015, pp. 115–120.

[23] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *IEEE Computer*, vol. 37, no. 7, p. 56–64, 2004.

[24] N. C. Mendonça, D. Garlan, B. R. Schmerl, and J. Cámara, "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)*. Madrid, Spain: ACM, 2018, pp. 18:1–18:6.

[25] N. C. Mendonça, P. Jamshidi, D. Garlan, and C. Pahl, "Developing self-adaptive microservice systems: Challenges and directions," 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), Tech. Rep. arXiv:1910.07660v2, 2019.

[26] V. Nallur and R. Bahsoon, "A decentralized self-adaptation mechanism for service-based applications in the cloud," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 5, pp. 591–612, 2013.

[27] H. P. Nii, "Blackboard systems part two: Blackboard application systems," *AI Magazine*, vol. 7, no. 3, 1986.

[28] T. Patikirikorala et al., "A systematic survey on the design of self-adaptive software systems using control engineering approaches," in *SEAMS*. Zurich, Switzerland: IEEE Press, 2012, p. 33–42.

[29] J. D. Pereira, R. Silva, N. Antunes, J. L. M. Silva, B. de França, R. Moraes, and M. Vieira, "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)*. New York, NY, USA: ACM, 2020, p. 71–77.

[30] F. Quin and D. Weyns, "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)*. Pittsburgh, PA, USA: IEEE Computer Society, 2022, pp. 77–83.

[31] F. Quin, D. Weyns, and O. Gheibi, "Decentralized self-adaptive systems: A mapping study," in *Proceedings of the 16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Madrid, Spain: IEEE, 2021, pp. 18–29.

[32] A. J. Ramirez and B. H. C. Cheng, "Design patterns for developing dynamically adaptive systems," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Cape Town, South Africa: ACM, 2010, pp. 49–58.

[33] A. R. Sampaio Jr., J. Rubin, I. Beschastnikh, and N. S. Rosa, "Improving microservice-based applications with runtime placement adaptation," *Journal of Internet Services and Applications (JISA)*, vol. 10, no. 4, pp. 1–30, 2019.

[34] B. Schmerl, J. Cámara, G. A. Moreno, D. Garlan, and A. Mellinger, "Architecture-based self-adaptation for moving target defense," Carnegie Mellon University, Tech. Rep. CMU-ISR-14-109, 2014.

[35] self-adaptive.org, "Software engineering for self-adaptive systems," Online, 2023, https://www.hpi.uni-potsdam.de/giese/public/selfadapt/ - accessed in February, 2023.

[36] A. M. Sharifloo and A. Metzger, "Mcaas: Model checking in the cloud for assurances of adaptive systems," in *Software Engineering for Self-Adaptive Systems III. Assurances*. Cham: Springer, 2013, pp. 137–153 (LCNS 9640).

[37] M. Shaw, ""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*. New York, NY, USA: ACM, 2002, p. 111–114.

[38] C. E. Silva and R. de Lemos, "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)*. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 148–157.

[39] B. R. Siqueira, F. C. Ferrari, T. Vogel, and R. de Lemos, "Micro-controllers: Promoting Structurally Flexible Controllers in Self-Aware Computing Systems," in *Proceedings of the 1st Workshop on Self-Aware Computing (SeAC'20)*. Washington DC, USA: IEEE, 2020, pp. 188–193.

[40] T. Vogel and H. Giese, "Adaptation and abstract runtime models," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Cape Town, South Africa: ACM Press, 2010, pp. 39–48.

[41] ——, "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)*. Zurich, Switzerland: IEEE, 2012, pp. 129–138.

[42] ——, "Model-driven engineering of self-adaptive software with EUREMA," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 8, no. 4, pp. 18:1–18:33, 2014.

[43] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Heidelberg, Germany: Springer, 2013, pp. 76–107.

[44] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, 1st ed. Springer: Springer, 2012.