Cost-Aware Multidimensional Auto-Scaling of Service- and Cloud-Based Dynamic Routing to Prevent System Overload

Amirali Amiri
Software Architecture Research Group
University of Vienna, Vienna, Austria
amirali.amiri@univie.ac.at

Uwe Zdun
Software Architecture Research Group
University of Vienna, Vienna, Austria
uwe.zdun@univie.ac.at

André van Hoorn
Software Engineering and Construction Methods
University of Hamburg, Hamburg, Germany
andre.van.hoorn@uni-hamburg.de

Schahram Dustdar
Distributed Systems Group
Technical University of Vienna, Vienna, Austria
dustdar@dsg.tuwien.ac.at

***Abstract*—Dynamic reconfiguration is commonly used to accommodate the dynamic behavior of today's applications. As cloud-based systems become increasingly complex, it is hard and cost-ineffective to manage them manually. Dynamic routers, such as API Gateways or Message Brokers, in combination with auto-scalers can adapt the system to the resource demands, e.g., when a sudden load spike for a specific part of the system is observed. Without taking costs of cloud resources into account, this reconfiguration can lead to significant increase of charges. We propose a self-adaptive and cost-aware dynamic routing architecture called Adaptive Dynamic Routers. The novel architecture performs a multi-criteria optimization analysis to automatically reconfigure the routers and the services of a cloud-based system considering the costs of reconfiguration. This multidimensional auto-scaling of resources takes incoming load as an input, and uses queuing theory to find an optimal reconfiguration solution. We systematically evaluated our architecture with an extensive number of evaluation cases (9600). On average over cases where an overload is predicted, our approach reduces the overload rate by 46.7% and 61.8% for routers and services, respectively.**

***Index Terms*—Self-Adaptive, System Overload, Cloud Resources, Dynamic Routing Architectures, Cost-Awareness, Automatic Reconfiguration, Multidimensional Auto-Scaling**

I. Introduction

Dynamic routing is an essential part of cloud-based systems. The non-static behaviour of today's applications usually means that dynamic routers are used, e.g., API Gateways [18], Enterprise Service Buses [8], Message Brokers [12], or Sidecars [13]. Cloud computing provides an elastic infrastructure to manage the dynamic behaviour of Internet applications. However, cloud-based systems are becoming increasingly complex, so that it is hard and cost-ineffective to manage them manually. The need for self-management systems is inevitable [6].

Consider for instance an e-commerce shop that offers discounted products for a specific location during a period of time. The application must cope with a sudden incoming load increase that needs to be routed to the services residing in the location. Dynamic routers in combination with auto-scalers can accommodate the system demand. Without such measures a system overload can lead to an application being non-responsive. However, if the cost of cloud resources is not considered, a business may lose profit by inducing significant resource costs when dealing with the sudden load spikes.

Horizontal auto-scaling, i.e., adding or removing replicas, and vertical auto-scaling, i.e., adding or removing resources such as processing power or memory, are commonly used in practice. A newer concept is multidimensional auto-scaling[1], which combines the two previous concepts in one decision-making step. Nontheless, the concept is not fully developed and has limitations such as not considering the incoming load as an input for multidimensional auto-scaling. Thus, we set out to answer the following research questions:

RQ1: *Can we find a multidimensional auto-scaling approach to perform self-adaptive cost-aware dynamic routing for automatically reconfiguring resources to prevent system overload?*

RQ2: *How well does the self-adaptive dynamic routing approach perform with regard to system overload prevention?*

In our prior work [5], we proposed a self-adaptive architecture named Adaptive Dynamic Routers (ADR). This architecture dynamically adapted the number of routers at runtime to adjust system reliability and performance trade-offs. In this paper, we substantially extend our proposed architecture by adding support to reconfigure components and prevent system overload. Moreover, we add cost-awareness to ADR.

For the evaluation of the proposed architecture, we take multiple levels of call frequencies, component configurations and routing profiles into consideration. Our extensive systematic evaluation of 9600 cases shows that the ADR architecture

[1] https://cloud.google.com/kubernetes-engine/docs/how-to/multidimensional-pod-autoscaling

is beneficial in terms of system overload prevention. Our approach can yield up to 100% improvement in reducing the rate, with which the routers and services overload. The mean of the data over all cases where an overload is predicted shows that the architecture reduces the average overload rate by 46.7% and 61.8% for routers and services, respectively.

II. Approach Overview

A. Background

In our prior work [2]–[5], we studied three representative dynamic routing architecture patterns in service- and cloud-based environments. These include the central-entity-based architecture, e.g., an API Gateway [18], or any kind of central service bus [8]; dynamic routers architecture [12] in which multiple routers are responsible for the routing regarding groups of services; and the sidecar architecture that follows the sidecar pattern [13]. In [5], we introduced the Adaptive Dynamic Routers (ADR) that is a self-adaptive routing architecture. ADR abstracts the controlling logic in the afore-mentioned architectures, i.e., the central entity, dynamic routers and sidecars, under a concept called *router*. Then, it automatically adjusts the number of routers based on a multi-criteria optimization [1] analysis.

B. Architecture Extensions

We require a couple of significant extensions to the ADR architecture for our approach presented in this paper, i.e., to prevent system overload. Firstly, we provide support to monitor and reconfigure the services in addition to the routers of a system. We use the term *reconfigurable components* to refer to services and routers collectively. Secondly, we model system overload analytically and use queuing theory [14] in the context of dynamic routing architectures. We identify system overload when a component of a system overloads resulting in an application being non-responsive to requests. Thirdly, the perspective for the reconfiguration is different in this paper. That is, the extended ADR architecture focuses on each reconfigurable component, i.e., a router or a service, separately and reconfigures it individually to prevent the overload of that specific component. As a result, we study reconfiguration measures that we did not investigate before such as auto-scaling. Finally, we consider the reconfiguration cost as a deciding factor, i.e., an optimization criterion.

C. ADR Component Diagram

Figure 1 presents a UML component diagram of the extended ADR architecture with a sample configuration. An *API Gateway*, which is an entry access for clients, publishes monitoring data to the *Quality of Service QoS Monitor* component. Moreover, the API Gateway forwards requests to the current routing architecture. The routers either forward or block the requests to the services they shield. The *QoS Monitor* observes the monitoring data and can trigger a reconfiguration if necessary, e.g., when degradation of quality attributes is detected. A reconfiguration is performed at run-time using a multi-criteria optimization analysis, explained in detail in the remainder of this paper.

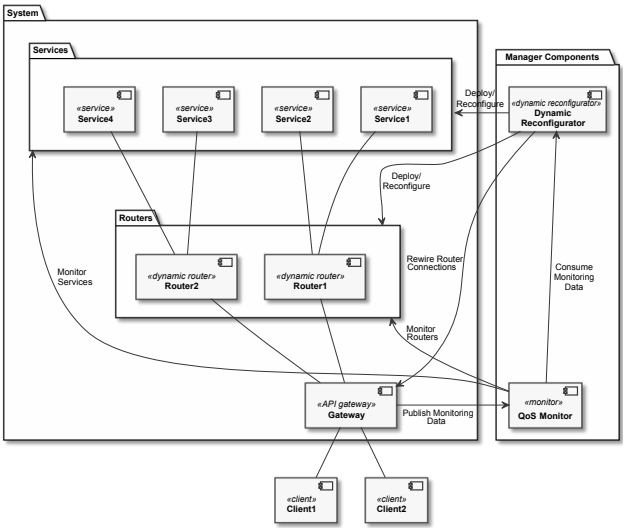


Fig. 1: Component Diagram of the ADR Architecture

D. Reconfigurable Components as Queuing Stations

We model each reconfigurable component, i.e., a router or a service, as a queuing station having two subcomponents, namely a *buffer* and a *processor*. Incoming requests are buffered in a queue and processed by the processor one-by-one according to a queuing discipline, e.g., a first-come-first-served strategy. A buffer has a Length l of the number of requests it can store, and a processor has a Processing Rate μ based on the number of requests per second r/s. There is a standard service offered by many cloud providers to configure containers. For instance, Google Kubernetes Engine Autopilot[2] and Microsoft Azure Container Instances[3] allow the configuration of vCPUs and Memories per containers.

We indicate a system overload when a buffer of a reconfigurable component overloads, that is when a component as a queuing station is not in its steady state [14]. Let the Arrival Rate of a Reconfigurable Component i be λ_i. For a Reconfigurable Component i to be in the steady state, its arrival rate must be lower than its processing rate, i.e., $\lambda_i < \mu_i$. The buffer of Reconfigurable Component i eventually overloads if $\lambda_i > \mu_i$ resulting in a system overload. We only consider homogeneous workload, i.e., single class requests.

E. Measures to Overcome System Overload

When a system overload is predicted for a reconfigurable component (see Section III-A), we consider the following measures to address system overload at the component level:

- Scale-out a router by using replicas, i.e., HAS.
- Scale-out a service by using replicas, i.e., HAS.
- Increase the processing rate of a router, i.e., VAS.
- Increase the processing rate of a service, i.e., VAS.

These measures are associated with cloud costs since they add cloud resources to the system (see Section III-C2). In the

[2]https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview

[3]https://azure.microsoft.com/en-us/services/container-instances/

following, we define an analytical model of system overload and use this model to automatically adjust the cloud resource usage considering the cost of cloud deployment.

III. Approach Details

A. System Overload Model

As shown in Figure 1, the *QoS Monitor* observes each reconfigurable component, i.e., a router or a service, and triggers a reconfiguration for each component separately by monitoring the arrival rate of each component. The monitor also observes the processing rate of all routers and services, e.g., by executing a `docker stats` command. An overload is predicted if an arrival rate of a reconfigurable component is higher than its processing rate, and its buffer is getting full.

1) Arrival Rate of Reconfigurable Components: In order to model the Arrival Rate λ_i of each Reconfigurable Component i, we define the incoming requests from the view point of a component. That is, the requests received by a component. An example ADR configuration is presented in Figure 2. We define Call Frequency cf as the frequency with which the client requests are received by the API Gateway based on requests per second r/s. IR_i is the number of Incoming Requests IR_i for a Reconfigurable Component i:

$$\lambda_i = cf \cdot IR_i \tag{1}$$

That is, the arrival rate for Reconfigurable Component i is the call frequency multiplied by the number of its incoming requests IR_i. To illustrate, in the example ADR configuration presented in Figure 2, $IR_i = 2$ uniformly for all routers. When the application is under stress with, e.g., $cf = 10\ r/s$, each router has an arrival rate of $\lambda_i = 20\ r/s$. Note that IR_i needs to be parameterized for each application separately.

2) Buffer Fill Rate: We define the Buffer Fill Rate BFR_i of a Reconfigurable Component i as:

$$BFR_i = \lambda_i - \mu_i \tag{2}$$

When the buffer fill rate is positive for any reconfigurable component, i.e., a router or a service, it means the arrival rate of the component is higher than its processing rate. In this case, the system eventually overloads if the Call Frequency cf of client requests does not decrease. Using Equation (1):

$$BFR_i = cf \cdot IR_i - \mu_i \tag{3}$$

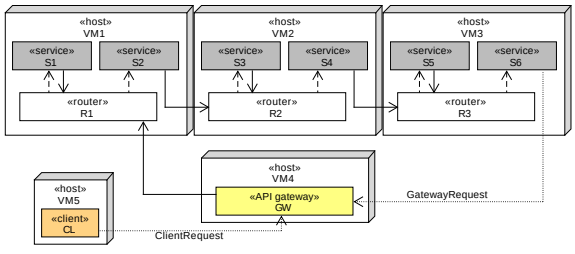


Fig. 2: Example ADR Configuration with Incoming Requests for Routers (Solid Lines) and Services (Dashed Lines)

B. Threshold for Reconfiguration

The severity of the damage of a positive BFR depends on the Full State FS of the buffer indicating how full it is, i.e, $0.0 < FS < 1.0$. An architect can define a Full State Threshold FS_{th} for the buffers of the reconfigurable components of a system. If any buffer reaches this threshold, ADR reconfigures the architecture configuration (see Figure 1). The *Dynamic Reconfigurator* reduces the Buffer Fill Rate BFR_i of a Reconfigurable Component i by performing:

- Scale-out the Reconfigurable Component i by using replicas reducing its Arrival Rate λ_i.
- Increase the Processing Rate μ_i of the Reconfigurable Component i to reduce its BFR_i (see Equation (2)).

Conversely, if the FS of a buffer goes below the threshold, the steps can be reversed e.g., by scaling-in the replicas.

C. Reconfiguration Algorithm

The proposed architecture automatically reconfigures the system at run-time using a multi-criteria optimization analysis [1]. We consider the following optimization criteria: the buffer fill rate prediction and the cost of reconfiguration.

1) Prediction of Buffer Fill Rate: The *Dynamic Reconfigurator* uses the buffer fill rate predictions to decide how much of each reconfiguration measure to perform. Remind that the architecture monitors each reconfigurable component separately and reconfigures it individually. Let $BFR(n_{scal}, n_{pro})$ be the predicted buffer fill rate by improving n units in each reconfiguration measure, i.e., scaling-out the component by n_{scal} replicas, or improving the processing rate of an alarming component by n_{pro} requests per second r/s.

$$BFR(n_{scal}, n_{pro}) = cf_{scal} \cdot IR_i - \mu_{pro} \tag{4}$$

IR_i is the number of incoming requests of a Reconfigurable Component i, cf_{scal} is the call frequency after scaling-out the component and load-balancing the cf among the replicas, and μ_{pro} is the increased processing rate of the component:

$$\mu_{pro} = \mu_i + n_{pro} \tag{5}$$

That is, the Processing Rate μ_i of the Reconfigurable Component i is increased by $n_{pro}\ r/s$.

2) Cost of the Reconfiguration Measures: We consider the cost models of widely-used cloud providers so that our proposed approach can be applied to real-world applications. These cost models relate to the resource usage and are at the container level. That is, customers pay per use of resources their containers need. For instance, Google Kubernetes Engine Autopilot[4] and Microsoft Azure Container Instances[5] charge per use of vCPUs and memories in seconds from the time the container images are pulled until the task is finished.

The reconfiguration measures mentioned in Section II-E require an increase of cloud resource usage. We associate costs to an improvement of n units for each reconfiguration measure, i.e., $C(n_{scal}, n_{pro}) = C(n_{scal}) + C(n_{pro})$. We have the cost

[4]https://cloud.google.com/kubernetes-engine/pricing
[5]https://azure.microsoft.com/en-us/pricing/details/container-instances/

of scaling-out the component by n_{scal} replicas, and the cost of increasing the processing rate by n_{pro} requests per second.

Note that the conversion between vCPU and Processing Rate μ_i depends on the application. There are different CPU types with different processing capabilities (see, e.g., Google Cloud CPU Platforms[6]). Moreover, the requests for each application have different timing needs. We provide a systematic evaluation in Section IV. Also, consider that the ADR cost model is general and is not specific to these cloud-providers or models. Other cloud cost models can be used if necessary.

3) Multi-Criteria Optimization (MCO): The *QoS Monitor* triggers the *Dynamic Reconfigurator* to reconfigure the components (see Figure 1). When the Full State Threshold FS_{th} has reached for a component, the reconfigurator automatically adapts the Buffer Fill Rate BFR of the component based on an MCO analysis [1]. Consider the following optimization problem: the reconfigurator must decide how many units of each measure to improve in order to have the minimum predicted buffer fill rate. However, at the same time, the reconfigurator needs to minimize the cost of cloud resource usage. Let C_{th} be the Cost Threshold:

$$Minimize$$
$$BFR(n_{scal}, n_{pro}) \tag{6}$$
$$C(n_{scal}, n_{pro}) \leq C_{th} \tag{7}$$

Note that any number of constraints can be added. The objective functions are opposing each other: we want to minimize the buffer fill rate but at the minimum cost of reconfiguration. Typically, there is no single answer to an MCO problem but a set of acceptable points called the Pareto front [1].

4) Automatic Reconfiguration: When a reconfiguration is triggered, the *Dynamic Reconfigurator* uses Algorithm 1 to deploy new components or reconfigure the existing ones.

Algorithm 1: ADR Reconfiguration Algorithm for one Overloading Component

```
input: C(n_scal), C(n_pro), IR                    // costs and incoming requests
begin
  cf, μ ← consumeMonitoringData()                 // call frequency and processing rate

  paretoFront ← MCO(cf, IR, μ)                    // Multi−Criteria Optimization
  reconfigSolution ← preferenceFunction(paretoFront)
  reconfigure ( reconfigSolution )

  function preferenceFunction ( paretoFront )
  begin
    R ← 0                                         // reconfiguration ratio
    reconfigSolution ← (0, 0)                     // (scaling replica, processing rate)

    foreach ( solution : paretoFront )
    begin
      R(n_scal, n_pro) ← (BFR(0,0) − BFR(n_scal, n_pro)) / C(n_scal, n_pro)   // buffer fill rate, cost

      if (R(n_scal, n_pro) > R)
          R ← R(n_scal, n_pro)
          reconfigSolution ← (n_scal, n_pro)
    end
    return reconfigSolution                       // final reconfiguration solution
  end
end
```

[6]https://cloud.google.com/compute/docs/cpu-platforms

5) Preference Function: The final reconfiguration choice from the Pareto front (see Section III-C3) is upon the decision maker based on their preference. In order for the *Dynamic Reconfigurator* to automatically choose a point as the selected solution, we define a preference function. We select a final solution based on the Reconfiguration Ratio R, i.e., the ratio of buffer fill rate improvement to the cost of reconfiguration:

$$R = \frac{BFR(0,0) - BFR(n_{scal}, n_{pro})}{C(n_{scal}, n_{pro})} \tag{8}$$

This gives us the amount of BFR improvements for each unit of cost spent on the reconfiguration. The preference function chooses the solution with the highest R on the Pareto front.

IV. Evaluation

A. Experiment Details

We ran an extensive experiment of 200 runs with a total of 1200 hours of runtime [3], [4]. We studied dynamic routing of requests in a call sequence of multiple services (see Figure 2 for an example configuration). We had a private cloud setting with three physical nodes, each having two identical Intel® Xeon® E5-2680 CPUs. On the nodes, we installed Virtual Machines (VMs) with eight vCPUs and 60 GB system memory. Each router or service was containerized in a Docker container. We used five desktop computers for load generation to send HTTP requests to the VMs. For the sake of simplicity, we labeled the routers and services incrementally from 1 and let the incoming requests go through them one-by-one as shown in Figure 2.

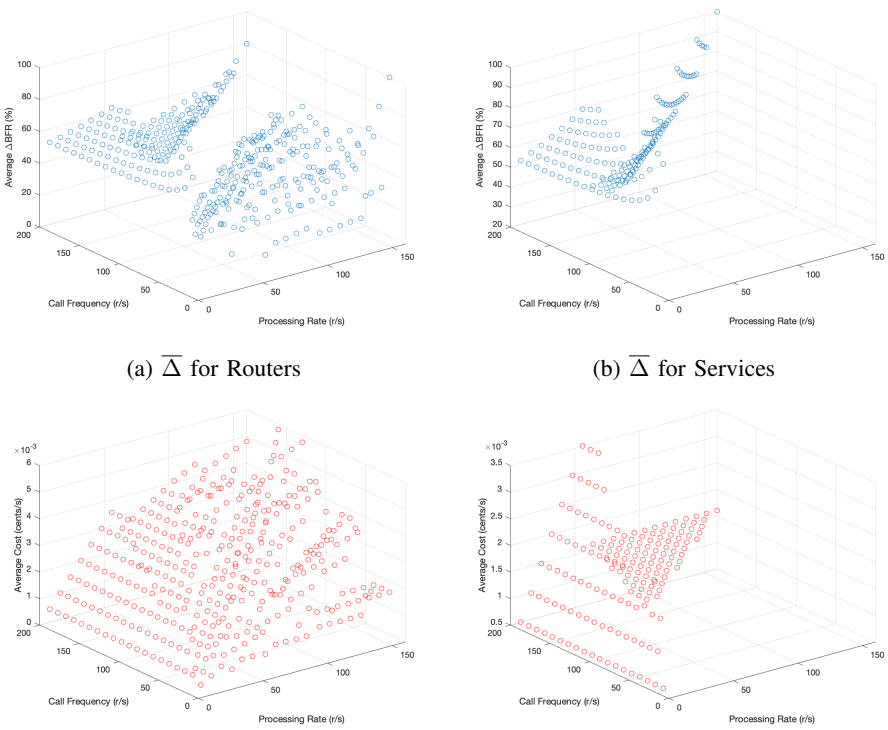
Systematic Analysis

In order to systematically evaluate the proposed ADR architecture, we use our experiment cases: We study three levels of the Number of Services, i.e., $n_{serv} \in (3, 5, 10)$ and four levels of the Number of Routers $n_{rout} \in (1, 3, n_{serv}) = (1, 3, 5, 10)$. We consider 20 levels of the Call Frequency cf in $10 \leq cf \leq 200$ r/s each step increasing 10 r/s. For container configurations, we investigate 20 levels based on the amount of vCPU requirements of a container between 0.25 and 5 vCPUs. We increase 0.25 vCPUs in each level incrementally. 0.25 vCPUs correspond to a Processing Rate $\mu = 8$ r/s in our experiment, i.e., $8 \leq \mu \leq 160$ r/s (see Section III-C2).

Regarding the Cost Threshold, we take $C_{th} = 1$ cent per second $cent/s$. This threshold is taken into account for each reconfiguration step and for each component separately. All in all, we performed an extensive evaluation of 9600 cases, i.e., three levels of n_{serv}, four levels of n_{rout}, twenty levels of cf and twenty levels of μ, which are all considered for routers and services separately. We define the Average Cost $\overline{C}$ and the Average Percentage Difference of Buffer Fill Rate $\overline{\Delta}$ as:

$$\overline{C} = \frac{1}{n} \sum_{c \in Cases} C(n_{scal}, n_{pro}) \tag{9}$$

$$\overline{\Delta} = \frac{100\%}{n} \sum_{c \in Cases} \frac{BFR(0,0) - BFR(n_{scal}, n_{pro})}{BFR(0,0)} \tag{10}$$



(a) $\overline{\Delta}$ for Routers

(b) $\overline{\Delta}$ for Services

(c) $\overline{C}$ for Routers

(d) $\overline{C}$ for Services

Fig. 3: Plots of Evaluation Data

in which $Cases$ is the set of n_{serv} and n_{rout} so $n = 12$.

The BFR average percentage difference $\overline{\Delta}$ for routers and services are shown in Figure 3a and Figure 3b. We can see that as the Processing Rate μ of a router increases, i.e., the number of vCPUs per container rises, we have a higher $\overline{\Delta}$ specially with a higher Call Frequency cf. That is, the ADR architecture yields the most improvements when a container with a high processing rate is under stress of a high load. Regarding average costs for routers, as shown in Figure 3c, the reconfiguration costs rise as the Processing Rate μ of router containers increases. This is expected because, e.g., scaling-out a router with five vCPUs costs higher than scaling-out a router with only one vCPU. However, for service containers with a higher processing power, there is a constant average cost as shown in Figure 3d. In these cases, the ADR architecture chooses the same reconfiguration measure, i.e., adding a Processing Rate $\mu = 40\ r/s$ to an overloading service. This is because as the processing power of a service container increases, the cost of scaling-out the service also rises. In this case, adding 40 r/s with $\overline{C} = 0.0016\ cents/s$ gives the highest Reconfiguration Ratio R and is chosen repeatedly[7].

Table I reports the statistics of $\overline{\Delta}$ and $\overline{C}$, in which σ, Q_1 and Q_3 are the standard deviation, first and third quartiles of the data. The proposed architecture can yield an BFR average percentage improvement $\overline{\Delta}$ of up to 100.0 % with $\overline{C}$ of 0.0058 and 0.0035 $cents/s$ for routers and services. For the mean of data over overloading cases, i.e., those cases with $BFR(0,0) > 0$, the ADR architecture gives an enhancement of $\overline{\Delta} = 46.7\%$ with $\overline{C} = 0.0024\ cents/s$ for routers, and $\overline{\Delta} = 61.8\%$ with $\overline{C} = 0.0016\ cents/s$ for services.

TABLE I: Statistics of the Evaluation Data

	Evaluation Metric	min	Q_1	$median$	Q_3	max	$mean$	σ
Routers	$\overline{\Delta}$	9.804	32.680	49.528	57.132	100.000	46.703	15.487
	$\overline{C}$	25e-5	15e-4	25e-4	32e-4	58e-4	24e-4	12e-4
Services	$\overline{\Delta}$	29.412	52.632	60.475	69.754	100.000	61.848	15.654
	$\overline{C}$	5e-4	15e-4	16e-4	16e-4	35e-4	16e-4	6e-4

V. Threats to Validity

In this paper, we studied reconfiguration measures of increasing the processing rate and scaling-out a component to prevent system overload. While this is a common approach in service- and cloud-based research (see Section VI), the threat remains that other measures might work better in terms of system overload prevention, for instance changing the routing technology, e.g., using a circuit breaker [17], or adding more routers and reconfiguring the routing [5].

We considered each reconfigurable component, i.e., a router or a service, of a cloud-based application separately by modeling them as queuing stations. If a reconfiguration is required, our approach performs a multi-criteria optimization analysis individually for the component. The threat remains that these

[7]The evaluation script and log (Pareto fronts) are anonymously downloadable to support reproducibility: https://doi.org/10.5281/zenodo.6566131

components can have interdependencies, e.g., preventing overload of an upstream component might stress the downstream components. In future work, we plan to model and study the proposed ADR architecture as queuing networks [14].

We designed our novel architecture with generality in mind. In spite of the fact that we systematically evaluated our approach with an extensive number of 9600 evaluation cases using the experiment infrastructure of our experiment of 1200 hours (see Section IV-A), the threat remains that evaluating the ADR architecture based on another infrastructure may lead to different results. To mitigate this thread, we performed many rounds of reviews and improvements in the author team and constantly compared with the related work.

VI. Related Work

The proposed ADR architecture is related to *self-adaptive systems*, which typically use MAPE-K loops [6], and similar approaches to realize adaptations. Our architecture extends such studies with support specific to dynamic cloud- and service-based routing architectures. Moreover, research on efficient resource provisioning, e.g., [9], [15], and cloud elasticity, e.g., [10], [11], are related to our work. Our study extends these approaches by considering the increase of the processing power of a container as a reconfiguration measure.

Multidimensional auto-scalers have been studied in the literature. AutoMAP [7] uses response time triggers to provision resources. To support cost-efficiency, AutoMAP finds optimal resources using Virtual Machine (VM) image sizes. Nguyen et al. [16] use a forecasting model to predict CPU demand and uses these predictions to start new machines before load peak to increase performance. CloudScale [19] supports scaling of CPU and memory resources when local scaling is possible. Otherwise, it migrates VMs to prevent overloaded hosts. Our work is different from all these studies in that they consider auto-scaling at VM level and configure the resources. We proposed a cost-aware multidimensional auto-scaler that works at container level adjusting the resources of each container.

In contrast to the existing related work, major contributions of our study are that we proposed a model of system overload specifically designed for dynamic routing in service- and cloud-based architectures. Having this specific view, and considering possible runtime adaptations, we defined a targeted reconfiguration algorithm to perform multi-criteria optimization analysis to find the (Pareto) optimal reconfiguration solutions. This would be hard to do in the generic case.

VII. Conclusion

In this study, we extended our previously proposed novel architecture, i.e., Adaptive Dynamic Routers (ADR) [5] that automatically reconfigures the routers and services of a cloud-based system taking the costs of reconfiguration into account. For **RQ1**, we proposed our extended ADR architecture which uses queuing theory to model routers and services of a cloud-based system and performs multidimensional auto-scaling on each component individually to prevent system overload. For **RQ2**, we systematically evaluated our approach using 9600 evaluation cases based on our extensive experiment of 1200 hours of runtime (see Section IV-A). Our results show that the ADR architecture yields up to 100% average percentage difference of buffer fill rate for routers and services. Regarding the mean of the data, over cases where an overload is predicted, our approach reduces the average overload rate by 46.7% and 61.8% for routers and services, respectively.

Acknowledgment

This work was supported by FWF (Austrian Science Fund) project API-ACE: I 4268 and Baden-Württemberg Stiftung, project ORCAS.

References

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Trans. Software Eng.*, 39(5):658–683, 2013.

[2] A. Amiri, C. Krieger, U. Zdun, and F. Leymann. Dynamic data routing decisions for compliant data handling in service- and cloud-based architectures: A performance analysis. In *IEEE International Conference on Services Computing*, 2019.

[3] A. Amiri, U. Zdun, G. Simhandl, and A. van Hoorn. Impact of service- and cloud-based dynamic routing architectures on system reliability. In *International Conference on Service Oriented Computing*, 2020.

[4] A. Amiri, U. Zdun, and A. van Hoorn. Modeling and empirical validation of reliability and performance trade-offs of dynamic routing in service- and cloud-based architectures. In *IEEE Transactions on Services Computing*, 2021.

[5] A. Amiri, U. Zdun, A. van Hoorn, and S. Dustdar. Automatic adaptation of reliability and performance tradeoffs in service- and cloud-based dynamic routing architectures. In *IEEE international Conference of Software Quality, Reliability and Security*, 2021.

[6] P. Arcaini, E. Riccobene, and P. Scandurra. Formal design and verification of self-adaptive systems with decentralized control. *ACM Transactions on Autonomous and Adaptive Systems*, 11(4):1–35, 2017.

[7] M. Beltrán. Automatic provisioning of multi-tier applications in cloud computing environments. *The Journal of Supercomputing*, 2015.

[8] D. A. Chappell. *Enterprise service bus*. O'Reilly, 2004.

[9] J. Comden, S. Yao, N. Chen, H. Xing, and Z. Liu. Online optimization in cloud resource provisioning: Predictions, regrets, and algorithms. In *Publication: Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2019.

[10] G. Galante and L. C. E. de Bona. A survey on cloud computing elasticity. In *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, pages 263–270. IEEE, 2012.

[11] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing*, pages 23–27, 2013.

[12] G. Hohpe and B. Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[13] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[14] V. V. Kalashnikov. *Mathematical Methods in Queuing Theory*. Springer, 2013.

[15] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu. Efficiency analysis of provisioning microservices. In *IEEE International Conference on Cloud Computing Technology and Science*, 2016.

[16] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *10th International Conference on Autonomic Computing*, 2013.

[17] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018.

[18] C. Richardson. Microservice architecture patterns and best practices. *http://microservices.io/index.html*, 2019.

[19] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *2nd ACM Symposium on Cloud Computing*, 2011.