Automatic Adaptation of Reliability and Performance Trade-Offs in Service- and Cloud-Based Dynamic Routing Architectures

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

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

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

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

***Abstract*—Many different dynamic routing architectures are available, including sidecar-based routing, routing through a central entity such as an event store or gateway, or architectures with multiple routers. These architectures are currently based on vastly different implementation concepts, such as API Gateways, Message Brokers, or Service Proxies. We propose a new approach that abstracts all these architecture patterns using one Adaptive Dynamic Routers architecture. We hypothesize that a dynamic self-adaptation of the routing architecture is beneficial over any fixed architecture selections for reliability and performance trade-offs. That is, if encountered with traffic and load changes, our approach dynamically self-adapts between more central or distributed routing to optimize system reliability and performance. We evaluate our approach by analyzing our previously-measured data during an experiment of 1200 hours of runtime. Our extensive systematic evaluation with 1089 cases confirms that our hypothesis holds and our approach is beneficial in terms of reliability and performance. Moreover, we empirically validate our results on Google Cloud Platform infrastructure.**

***Index Terms*—Self-Adaptive Systems, Dynamic Routing Architectures, Service-Based Computing, Cloud-Based Applications**

I. Introduction

DYNAMIC routing, i.e., routing or blocking the incoming requests to different services based on a set of rules, is very common in service- and cloud-based applications. Different techniques are available ranging from very simple, e.g., load balancing, to more complex ones such as routing to the right branch of a company or checking for compliance to regulations. Multiple dynamic routing architecture patterns are provided for service- and cloud-based environments including centralized routing that uses a *Central Entity* (e.g., an API Gateway [29] or any kind of enterprise service bus [12]), using multiple *Dynamic Routers* [19], and an extreme of completely distributed routing, for instace following the *Sidecar* pattern [22], [29]. In our prior work [3]–[5], we empirically studied these architecture patterns in terms of system reliability and performance. Our studies show that more centralized routing results in a higher reliability; however, decentralized routing offers a higher performance.

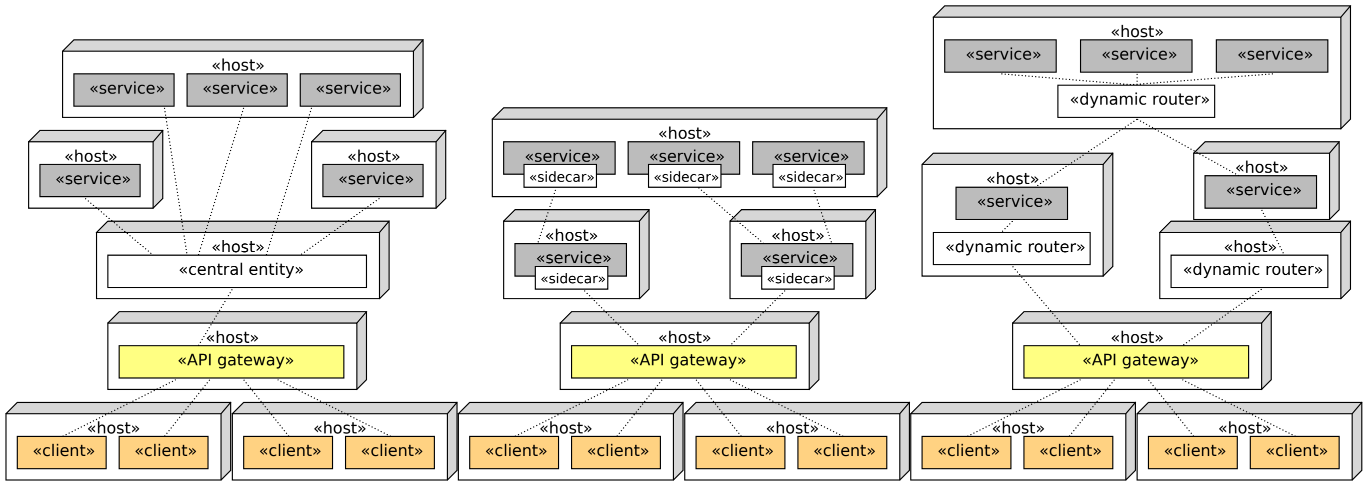
At present, there is no architecture that automatically adapts to a routing scheme, from centralized to distributed or vice versa, based on reliability and performance trade-offs. An architect must statically redesign and redeploy multiple routers if a degradation of metrics is observed. This is hard to manage, and sometimes it results in significant issues in systems: Assume a sudden system reliability decrease is observed in a company with sensitive data of customers. In such a situation, time is of the essence to reconfigure the system to meet the quality criteria required for the application. An automatic adaptation can yield benefits not only in time and effort overheads for the management of the system, but also in reliability and performance gains by adjusting the trade-offs. Thus, we set out to answer the following research questions:

RQ1: *Can we find an optimal configuration of routers that automatically adapts the reliability and performance trade-offs in dynamic routing architectures based on monitored system data at runtime?*

RQ2: *How does the reliability and performance predictions of the chosen optimal solution compare with the case where one architecture runs statically?*

The main contribution of this paper is a self-adaptive architecture in dynamic routing called *Adaptive Dynamic Routers (ADR) Architecture* that is based on Monitor, Analyse, Plan, Execute, Knowledge (MAPE-K) loops [6], [7], [21]. The proposed architecture uses the reliability and performance trade-offs analysis of representative dynamic routing architecture patterns from our prior work [4], [5]. The ADR architecture performs a multi-criteria optimization analysis [2] to find a

Supported by FWF (Austrian Science Fund), projects ADDCompliance: I 2885-N33, API-ACE: I 4268; FFG (Austrian Research Promotion Agency), project DECO no. 864707; Baden-Württemberg Stiftung, project ORCAS.



(a) Central Entity Architecture (b) Sidecar Architecture (c) Dynamic Routers Architecture

Fig. 1: Dynamic Routing Architecture Patterns

range for the number of dynamic routers. From this range, we choose a final reconfiguration solution based on importance vectors for reliability and performance.

To evaluate our proposed architecture, we used the empirical data set from our prior work [5], where we designed an extensive experiment of 200 runs with multiple experimental cases running for 1200 hours to empirically validate our reliability and performance models. Reliability was modelled based on request loss in service- and cloud-based dynamic routing. For reliability, our results showed a constantly reducing prediction error which converged at 7.8%. We validated our results with three other error measurements which all yielded the same trend of prediction error always decreasing. During our experiment, we recorded round-trip times of each request that we used as a metric of performance.

For performance predictions, we used a statistical regression analysis [30] based on the recorded round-trip times. In this paper, we created analogous regression models for our proposed novel architecture. Our performance model had a prediction error of 14.7%. Given the target prediction accuracy of 30.0% commonly used in the cloud performance field [24], and the fact that the focus of our study is to have a rough prediction of the reliability and performance impacts when architecting a system, these results are more than reasonable.

To evaluate our ADR architecture, we use these reliability and performance models and go through all experimental cases, systematically assign thresholds and importance weights to reliability and performance, and evaluate ADR in comparison to its fixed counterparts, i.e., a fixed central entity, a fixed dynamic routers, and a fixed sidecar-based architecture. Our extensive systematic evaluation with 1089 evaluation cases shows that ADR leads to significant increases in reliability or performance in cases where the wrong architecture is chosen. Even on average, when cases with correct and incorrect architecture choices are analyzed together, ADR provides improvements. On average it provides 14.0% more reliability than performance loss compared to fully distributed routing when reliability is a higher priority, and 7.5% more performance gain than reliability decrease compared to centralized routing when performance is a higher priority.

We evaluated our approach using our private cloud to have repeatable experiment runs. In a public cloud other factors might affect the results, e.g., other workloads running in parallel or the distance between different nodes. To show that our approach is applicable outside of our private infrastructure, we empirically validate our results on a public cloud. That is, we validate an illustrative sample case on Google Cloud Platform[1]. The prediction error of 14.0% confirms that our approach can be used on other infrastructures.

The structure of the article is as follows: Section II presents the background. Section III gives an approach overview of our study. In Section IV, we explain the proposed adaptive dynamic routers architecture in detail, and in Section V provide the parameterization of our models. Section VI presents the evaluation of the presented approach. Section VII discusses the threats to the validity of our research. We study the related work in Section VIII, and conclude in Section IX.

II. Background: Dynamic Routing Architecture Patterns

In our prior work [3]–[5], we have studied three representative service- and cloud-based dynamic routing architecture patterns: Central Entity based architectures (CE), as shown in Figure 1a, e.g., an API Gateway [29] or any kind of central service bus [12]; Sidecar Architectures (SA), as presented in Figure 1b, which follows the sidecar pattern [15], [22], [25], and Dynamic Routers architectures (DR) [19] in which multiple routers perform the routing for groups of services as shown in Figure 1c. DR can be seen as a hybrid of the central entity and the sidecar architectures; SA is a completely distributed approach since there is a sidecar per each service. A

[1]https://cloud.google.com

controlling logic component can be used, e.g., for data routing purposes or access level controls.

The perspective used in our background works is the one of patterns, i.e., best practice architectures observed in practice. In this work, we take a different perspective on these three patterns: DR is seen as a high-level abstraction that can be used to model CE and SA. That is, CE and SA are special cases of the DR architecture; CE is DR with only one router, and SA is DR with one router per each service, which is deployed on the same host. Any combination in between these two extremes can be modeled by the dynamic routers architecture. The CE, DR, and SA architecture patterns are implemented based on very different concepts, including API Gateways [29], such as NGINX[2] or Kong[3], Enterprise Service Buses [12], Message Brokers [19], or sidecars [15], [22], [25] such as Envoy[4]. Essentially they all route the incoming requests dynamically.

In this paper, we propose a new approach that realizes all three architecture patterns based on a common router abstraction. We hypothesize that a dynamic self-adaptation between the three architectures is beneficial over any fixed architecture selections. That is, if a traffic and load change occurs, our approach can dynamically self-adapt the degrees to which more or less central routing is used, to optimize its impact on performance and reliability trade-offs. We suggest that this adaptation can be automated using a multi-criteria optimization analysis [2]. Previously, we ran an extensive experiment (see Section V-A), where we collected substantial amounts of data and analyzed them by creating reliability and performance models. In this paper, we use these models as the basis for the multi-criteria optimization analysis. This allows us to engineer our novel self-adaptive architecture approach that dynamically adapts between the architecture patterns on-the-fly to adjust the reliability and performance trade-offs.

III. Approach Overview

In this section, we present the overview of our approach, which is further explained in detail in the next section.

Adaptive Dynamic Routers Architecture

We define a concept called *router* and abstract all the controlling logic components, i.e., the central entity service, the dynamic routers and the sidecars, under router. Now, we can use this high-level router to dynamically reconfigure the routing architecture. That is, we can change between the three architectures moving from a centralized approach with one router to a distributed system with more routers (or vice versa) to adapt based on the need of an application. We call this high-level abstraction the Adaptive Dynamic Routers (ADR) architecture.

Let us clarify the difference between the DR and the newly-introduced ADR architectures. DR is a fixed architecture that typically does not change while a system runs. If it changes in reality, a redeployment has to happen, manually designed and often manually deployed by system engineers. In our ADR approach, the system dynamically switches based on the results of a multi-criteria optimization analysis that can be triggered, for instance, in certain time-intervals or whenever certain metrics change, e.g, instance round-trip performance degrade, failed requests, change of an incoming load, or when a different route containing more services is used.

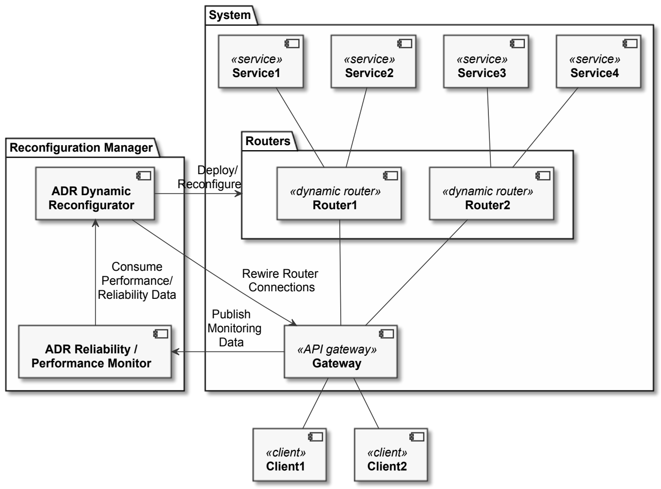


Fig. 2: ADR Component Architecture

In contrast to all three fixed architecture patterns, ADR is adaptive and changes at runtime. Figure 2 presents a UML[5] component diagram of the ADR architecture with a sample configuration of a system. As shown, clients access the system via an API Gateway, which publishes monitoring data to the *ADR Reliability/Performance Monitor* component. This way, e.g., incoming load, round-trip performance or failed requests data, are observed. The API Gateway forwards requests to the current routing architecture, and the routers forward them to the services they shield. The *Reconfiguration Manager* observes the monitoring data, and performs the multi-criteria optimization analysis. If reconfiguration is required, the architecture changes to better cope with the current load profile. This is done by automatically redeploying or reconfiguring the dynamic routers.

Figure 3 shows a UML activity diagram of the *Reconfiguration Manager*. The *ADR Reliability/Performance Monitor* triggers the activities of *ADR Dynamic Reconfigurator* further explained in Section IV-B. Note that based on MAPE-K the monitor implements the *Monitor* and *Analyse* stages and the reconfigurator develops the *Plan* and *Execute* steps. We use our models as the *Knowledge* part.

IV. Approach Details

In this section, we introduce the ADR reliability and performance models, and present our reconfiguration algorithm. Table I presents the mathematical notation used in this paper.

[2]https://www.nginx.com
[3]https://konghq.com/kong/
[4]https://www.envoyproxy.io/
[5]https://www.uml.org

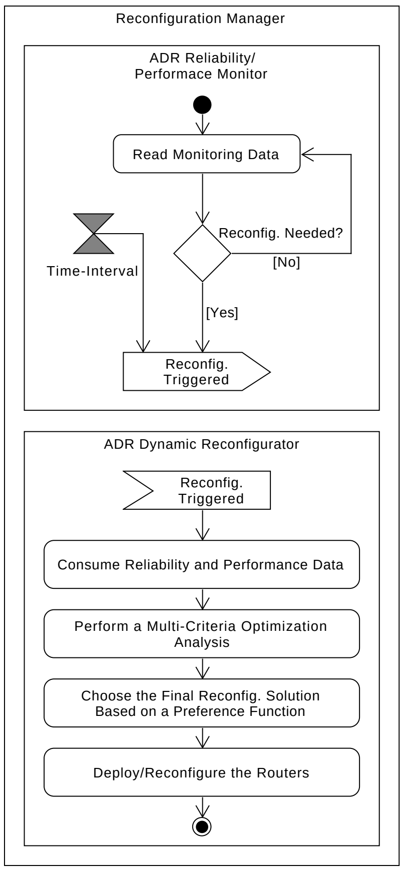

Reconfiguration Manager
ADR Reliability/ Performace Monitor
Read Monitoring Data
Reconfig. Needed?
[No]
Time-Interval
[Yes]
Reconfig. Triggered
ADR Dynamic Reconfigurator
Reconfig. Triggered
Consume Reliability and Performance Data
Perform a Multi-Criteria Optimization Analysis
Choose the Final Reconfig. Solution Based on a Preference Function
Deploy/Reconfigure the Routers

Fig. 3: Activities of ADR Configuration Manager

TABLE I: The Mathematical Notations Used in this Paper

Notation	Description
R	Reliability model
$R_{service}$	Reliability model for service crashes
R_{router}	Reliability model for router crashes
P	Performance model
T	Observed system time
CI	Crash interval
cf	Incoming call frequency
Com	Set of all components
P_c	Crash probability of a component c every CI
d_c	Expected average downtime after a component c crashes
IR_T	Total number of requests exchanged between components
n_c^{exec}	Number of successfully executed requests before the crash of a component c
n_{serv}	Number of services in an ADR instance
n_{rout}	Number of routers in an ADR instance
$s_{crashed}$	A service s when crashed
$r_{crashed}$	A router r when crashed
A	Allocation of routers
Sc	Service coefficient
Rc	Router coefficient
Fc	Frequency coefficient
SFc	Service vs. frequency coefficient
RFc	Router vs. frequency coefficient
SRc	Service vs. router coefficient
$SRFc$	Service vs. router vs. frequency coefficient
$R_{n_{rout}}$	Reliability of an ADR architecture configuration by the number of routers
$P_{n_{rout}}$	Performance of an ADR architecture configuration by the number of routers
R_{th}	Reliability threshold
P_{th}	Performance threshold
Err	Prediction error
$ErrR$	Prediction error regarding reliability
$ErrP$	Prediction error regarding performance
$\bar{\Delta R}$	Reliability average percentage difference
$\bar{\Delta P}$	Performance average percentage difference
$RGain$	Reliability gain
$PGain$	Performance gain
$RWeight$	Reliability weight
$PWeight$	Performance weight
Q_1	First quartile
Q_3	Third quartile
σ	Standard deviation

A. ADR Models

1) Reliability Model: In [4], [5], we used Bernoulli processes [32] to model request loss during router and service crashes. Request loss was defined as the number of incoming requests that were not processed due to a failure such as a crash of a component. We calculated the total request loss during an observed system time T as a metric of reliability:

$$R = \lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in Com} P_c \cdot d_c \cdot (IR_T - n_c^{exec}) \qquad (1)$$

CI is the crash interval, i.e., the interval in which we check for a crash of a component. Assume the Heartbeat pattern [20] is used to check the system health, CI is the time between two consecutive health checks. cf is the incoming call frequency based on Http requests per second (Hr/s), C is the set of components, i.e., routers and services, P_c is the crash probability of each component, d_c is the average downtime of a component after it crashes, IR_T is the number of requests exchanged between components, and n_c^{exec} is the number of successfully executed requests before the crash of a component. Note that IR_T and n_c^{exec} need to be parameterized based on the application (see Section V).

To empirically validate our model, we ran an extensive experiment (see Section V-A for details). Then we compared our analytical model of reliability with the empirical results of our experiment by using the mean absolute percentage error [32]. With more experiment runs, we observed an ever-decreasing error, converging at 7.8%. We also double-checked the accuracy of our models with three other error metrics, i.e., Mean Absolute Error (MAE), Mean Square Error (MSE) and Root Mean Square Error (RMSE), which yielded the same trend of the prediction error being constantly reduced.

2) Performance Model: During our experiment [5], we recorded the Round-Trip Time (RTT) of each request, which was defined as the difference in time from the moment a request was received until it was routed through all cloud services involved in the processing of the request. The RTTs were an indicator of the performance impact of the studied architectures, i.e., CE, DR and SA presented in Figure 1.

ADR is a reconfigurable architecture which can be configured as any of the above-mentioned architectures by changing the router configurations (see Figure 2). We introduce a variable n_{rout} which defines the number of routers in an ADR instance. To illustrate, the central entity architecture has one router which processes requests centrally; therefore, CE is ADR with $n_{rout} = 1$. The sidecar architecture, on the other hand, is completely distributed and has one router, i.e., sidecar, per service. In other words, the number of routers in SA is the same as the number of services (n_{serv}); consequently, SA

TABLE II: Performance Prediction Model

Coefficient (Related Variable)	**Sc** (n_{serv})	**Rc** (n_{rout})	**Fc** (cf)	**SFc** ($n_{serv} : cf$)	**RFc** ($n_{rout} : cf$)	**SRc** ($n_{serv} : n_{rout}$)	**SRFc** ($n_{serv} : n_{rout} : cf$)	**Int** ($Intercept$)
F-statistic p-value	<2.2e-16							
Value	4.128 e+00	-3.032 e+00	-2.249 e-01	3.682 e-02	1.228 e-02	1.308 e-01	-4.530 e-03	1.714 e+01

is ADR with $n_{rout} = n_{serv}$. The dynamic routers architecture was a middle ground and had three routers, i.e., $n_{rout} = 3$, among which we distributed services equally.

We did a multiple regression analysis [30] on the recorded RTTs, and created a prediction model of performance for ADR based on the independent variables n_{rout}, n_{serv}, and cf (see Section V-A for experiment details). Note that the nonlinear regression in Equation (2) is system-specific and needs to be performed for each application separately.

$$\begin{aligned} P =& Int + (Sc \cdot n_{serv}) + (Rc \cdot n_{rout}) + (Fc \cdot cf) + \\ & (SFc \cdot n_{serv} \cdot cf) + (RFc \cdot n_{rout} \cdot cf) + \\ & (SRc \cdot n_{serv} \cdot n_{rout}) + (SRFc \cdot n_{serv} \cdot n_{rout} \cdot cf) \quad (2) \end{aligned}$$

Table II reports the coefficients of our performance model. As it can be seen, all of the calculated coefficients of our regression model resulted in a very low *p-value* which indicates a high statistical significance of the prediction results.

As before, we used mean absolute percentage error [32] to compare the ADR performance prediction model with the recorded RTTs. The prediction error converged at 14.7% with more experiment runs. We double-checked the error with MAE, MSE and RMSE. Note that the regression analysis needs to be performed for each application separately. Since in the cloud performance field, 30.0% is commonly used as the target prediction accuracy [24], and the fact that our focus is to have a rough prediction of the impact of performance when architecting a system, the prediction accuracy is more than reasonable.

B. ADR Reconfiguration Algorithm

1) Multi-Criteria Optimization (MCO) Analysis: In our approach, the reconfiguration between the architecture configurations is performed automatically based on an MCO analysis [2]. Consider the following optimization problem: An application using the ADR architecture has n_{serv} services and is under stress for a period of time with the call frequency of cf. In order to optimize reliability and performance, the system can change between different ADR architecture configurations dynamically by adjusting the number of routers (n_{rout}), ranging from the extreme of a centralized routing (only one router), over any dynamic router configurations, up to the extreme of one router per service (n_{serv} routers), which is the SA configuration.

We use the notations $R_{n_{rout}}$ and $P_{n_{rout}}$ to specify the reliability and performance of the respective architecture configurations by their number of routers. For instance, only configuring one router R_1 indicates the reliability model of an ADR with only one router (i.e., the CE architecture), and configuring n_{serv} routers (i.e., $R_1, \ldots, R_{n_{serv}}$) indicates SA. R_{th} and P_{th} are the reliability and performance thresholds.

$$\begin{aligned} & Minimize \\ & \quad R_{n_{rout}} && (3) \\ & \quad P_{n_{rout}} && (4) \\ & Subject\ to \\ & \quad R_{n_{rout}} \leq R_{th} && (5) \\ & \quad P_{n_{rout}} \leq P_{th} && (6) \\ & \quad 1 \leq n_{rout} \leq n_{serv} && (7) \end{aligned}$$

The MCO question is: Given a cf and n_{serv}, what is the optimal number of routers which minimizes request loss and average RTTs per each requests without the predicted ADR reliability and performance going beyond a certain threshold. Typically, there is no single answer to an MCO problem; using the above MCO analysis, we find a range of n_{rout} configurations which all meet the constraints (see Section V-C for an example). One end of this range optimizes reliability and the other performance, thus we need a preference function so ADR can automatically select an n_{rout} value.

2) Preference Function: We can define different criteria to choose the final ADR router configuration. Let us consider a scenario: In our prior work [4], [5], we empirically validated that the centralized routing offers a higher reliability compared to distributed approaches; on the other hand, decentralized routing improves performance by processing the incoming requests in parallel. An architect can define operational profiles (see Section V-C2 for a sample case) for low and high levels of cf, based on which ADR adapts to more centralized routing (a lower n_{rout}) to improve reliability, or more distributed approaches (a higher n_{rout}) to improve performance. Note that the *ADR Reliability/Performance Monitor* observes the incoming call frequency and triggers the *ADR Dynamic Reconfigurator* to reconfigure the routers as soon as there is a change in these defined cf levels (see Figures 2 and 3).

Based on these operational profiles, the preference function instructs ADR to choose a final n_{rout} value in the range found by the MCO analysis based on an importance vector which gives weights to reliability and performance (see Algorithm 1). Let us consider an example: When reliability is of highest importance to an application, an architect gives the highest weight, i.e., 1.0, to reliability and the lowest weight, i.e., 0.0, to performance. Thus, the preference function chooses the lowest value on the n_{rout} range to choose more centralized routing which results in a higher reliability (see Section V-C for an illustrative example).

3) Automatic Reconfiguration: As shown in Figure 2, the component *ADR Reliability/Performance Monitor* reads the monitoring data from the API Gateway and feeds this information to the *ADR Dynamic Reconfigurator*, which deploys new routers or reconfigures the existing ones. Algorithm 1 presents our reconfiguration algorithm, which is used by the *ADR Dynamic Reconfigurator*. The *ADR Reliability/Performance Monitor* triggers the reconfiguration algorithm, for instance, whenever reliability or performance metrics degrade, e.g., observed as increase of request loss or decrease of RTTs. Time intervals, change in incoming load or a different route with more or less services can also be used to trigger the algorithm, if more appropriate than metrics degradation (see Figure 3).

Algorithm 1: ADR Reconfiguration Algorithm

```
Input: $R_{th}$, $P_{th}$, performanceWeight
$R_{n_{rout}}, P_{n_{rout}}, cf, n_{serv}$ ← consumeRPData()
routersRange ← MCO($cf, n_{serv}, R_{n_{rout}}, P_{n_{rout}}, R_{th}, P_{th}$)
reconfigSolution ← preferenceFunction(routersRange,
                                      performanceWeight)
reconfigureRouters(reconfigSolution)

function preferenceFunction(range, PW)
begin
    length ← max(range) - min(range) +1
    floor ← ⌊ PW * length ⌋
    if floor == max(range) then
        return max(range)
    else if floor == 0 then
        return min(range)
    else
        return floor + min(range) -1
    end
end
```

`reconfigureRouters(reconfigSolution)` in Algorithm 1 performs the final reconfiguration based on the chosen solution. This step differs based on an application and needs to be specified. In this paper, we assume a very simple but cost-ineffective strategy of starting the new configuration in parallel with the running setup. Afterwards, when the infrastructure is ready to process the incoming requests, we change to the new configuration and tear down the old setup. Note that the purpose of this paper is to provide a scientific proof-of-concept and not a fully functioning code base. The simple strategy fits this purpose and does not result in reliability or performance degradation because of the reconfiguration.

V. Parameterization of Model to Experiment Parameter Values

Our analytical reliability model is general for routing architectures. Therefore, it needs to be parameterized based on the application. Note that the performance model is based on a regression analysis that has to be performed for each application separately. We parameterize Equation (1) for our experiment by specifying request loss for the ADR with the introduction of the number of routers (n_{rout}) variable.

A. Experiment Details

1) Experiment on Private Cloud: We ran an experiment of 200 runs with a total of 1200 hours of runtime (excluding setup time). We had a private cloud setting with three physical nodes, each having two identical Intel® Xeon® E5-2680 CPUs. We installed Virtual Machines (VMs) with eight cores and 60 GB system memory. Each router or service was containerized in a Docker[6] container. We deployed one router exclusively on one VM for CE, three routers each deployed on one VM for DR, and one router per each service deployed on the same VM for SA.

We had three levels for the number of services (n_{serv}), i.e., 3, 5, and 10 services, and four levels for incoming call frequency (cf), i.e., 10, 25, 50, and 100 Hr/s, totaling twelve experimental cases for each architecture (36 cases overall). We utilized five desktop computers for load generation, each hosting an Intel® Core™ i3-2120T CPU with 8 GB of system memory which used Apache JMeter[7] to send Hypertext Transfer Protocol (HTTP) version 1.1[8] requests to the VMs. The routing of requests was as follows: For the sake of simplicity, we labeled the services incrementally from 1 and let the incoming requests go through all services one-by-one. An example of an experiment configuration is shown in Figure 4.

2) Validation Experiment on Public and Private Clouds: We use our private cloud to have control over the infrastructure and have repeatable experiment runs. On a public cloud other factors can influence the results such as parallel workload of other applications, or the physical distance of the node. To show that our approach can be used on other infrastructures as well, we empirically validate the analysis of an illustrative sample case (see Section V-C) once on our private cloud infrastructure and once on Google Cloud Platform (GCP)[9]. That is, we run our experiment a second time with 10 services,

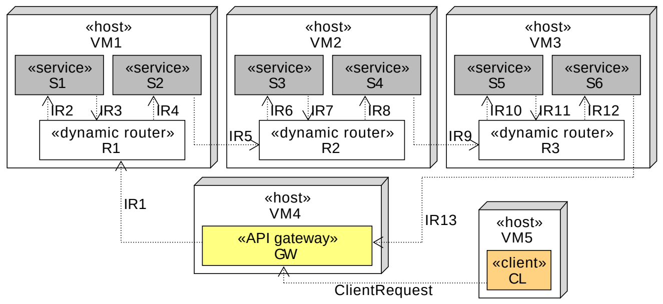


Fig. 4: Example ADR/Experiment Configuration

[6]https://www.docker.com
[7]https://jmeter.apache.org
[8]https://tools.ietf.org/html/rfc7230
[9]https://cloud.google.com

i.e., $n_{serv} = 10$, and stress the system with three considered call frequencies, i.e., 25, 50 and 100 Hr/s. In this scenario, ADR automatically changes the architecture, i.e., the value of n_{rout}, and reconfigures the routing. On GCP, we started E2 machine instances[10] with 2 vCPUs and 8 GB of memory and duplicated our experiment infrastructure.

B. Parameterization

Figure 4 shows an example ADR (and thus experiment) configuration with three routers and six services. When a router or a service crashes, some requests are not processed. In order to know how many of these requests are lost, we use the total number of requests (IR_T), from which we subtract the number of already executed ones (n_c^{exec}). Let us consider an example: Assume $S5$ crashes in the example configuration. In this case, $IR1$ to $IR9$ are processed ($n_c^{exec} = 9$) but $IR10$ to $IR13$ (four requests) are lost. In this example, we can see that there are 13 requests ($IR_T = 13$). Therefore, the number of lost requests in this example is:

$$IR_T - n_c^{exec} = 13 - 9 = 4 \tag{8}$$

We calculate IR_T based on the number of services as:

$$IR_T = 2n_{serv} + 1 \tag{9}$$

In the example ADR configuration, we have $n_{serv} = 6$ so $IR_T = 13$. In order to calculate n_c^{exec} we need to differentiate between service and router crashes. We define $s_{crashed}$ as the label number of the crashed service. For our experiment, we calculate for service crashes:

$$n_c^{exec} = 2s_{crashed} - 1 \tag{10}$$

Note that in our example case, we considered the crash of S5 ($s_{crashed} = 5$) and calculated $n_c^{exec} = 9$. We observed the system for 10 minutes (600 seconds) for each experimental case and checked for a crash every 15 seconds with a uniform crash probability of 0.5% for all components:

$$T = 600\,s \tag{11}$$

$$CI = 15\,s \tag{12}$$

$$P_c = 0.5\% \tag{13}$$

Therefore, we can rewrite Equation (1) for service crashes using Equations (9) to (13) as:

$$R_{service} = 0.6 \cdot cf \cdot n_{serv}(n_{serv} + 1) \tag{14}$$

In case of a router crash, we define the allocation of routers (A) as a set which indicates the number of directly linked services of each router. For instance, the allocation of routers in the example ADR configuration (see Figure 4) is:

$$A \ = \{2, 2, 2\} \tag{15}$$

In our experiment, services were equally allocated to routers:

$$A \ = \{\frac{n_{serv}}{n_{rout}}, \frac{n_{serv}}{n_{rout}}, ..., (\frac{n_{serv}}{n_{rout}} \pm 1)\} \tag{16}$$

[10]https://cloud.google.com/compute/docs/general-purpose-machines

in which A has the length of n_{rout}. In Figure 4, there are six services, i.e., $n_{serv} = 6$, and three routers, i.e., $n_{rout} = 3$; therefore, we have the allocation presented in Equation (15).

Let $r_{crashed}$ be the label number of the crashed router, then for router crashes we have:

$$n_c^{exec} = 2 \sum_{r=1}^{r_{crashed}} A_{r-1} \tag{17}$$

Therefore, we can rewrite Equation (1) for router crashes as:

$$R_{router} = 0.6 \cdot cf \cdot [n_{serv} + n_{rout}(n_{serv} + 1)] \tag{18}$$

Finally, we can rewrite Equation (1) by adding Equations (14) and (18) as:

$$R = 0.6 \cdot cf \cdot [(n_{serv})^2 + (n_{rout} + 2)n_{serv} + n_{rout}] \tag{19}$$

C. Illustrative Sample Case

We provide an illustrative example to explain our concepts.

1) Multi-Criteria Optimization: We customize the MCO analysis presented in Section IV-B1 with our experiment models' setup, thus we rewrite Equations (3) and (4) using Equations (2) and (19):

$$Minimize$$

$$R_{n_{rout}} = 0.6 \cdot cf \cdot [(n_{serv})^2 + (n_{rout} + 2)n_{serv} + n_{rout}] \tag{20}$$

$$\begin{aligned} P_{n_{rout}} &= 17.14 + \\ &4.128 \cdot n_{serv} - 3.032 \cdot n_{rout} - 0.2249 \cdot cf + \\ &0.1308 \cdot n_{serv} \cdot n_{rout} + 0.03682 \cdot n_{serv} \cdot cf + \\ &0.01228 \cdot n_{rout} \cdot cf - 0.00453 \cdot n_{serv} \cdot n_{rout} \cdot cf \end{aligned} \tag{21}$$

$$Subject\ to$$

$$R_{n_{rout}} \le R_{th} \tag{22}$$

$$P_{n_{rout}} \le P_{th} \tag{23}$$

$$1 \le n_{rout} \le n_{serv} \tag{24}$$

2) Operational Profiles: Let us consider an example ADR application having ten services, i.e., $n_{serv} = 10$, which is operational for the expected input call frequency of $10 \le cf \le$ 100 Hr/s with a reliability threshold of $R_{th} = 10000$ request loss per 10 minutes of experiment (on average a very high rate of 16.667 requests per second), and a performance threshold of $P_{th} = 60$ ms average RTT per each request. We do the MCO analysis for different chunks of call frequency starting with the lower bound, i.e., $cf = 10$ Hr/s:

$$Minimize$$

$$R_{n_{rout}} = 720 + 66 \cdot n_{rout} \tag{25}$$

$$P_{n_{rout}} = 59.853 - 2.0542 \cdot n_{rout} \tag{26}$$

$$Subject\ to$$

$$R_{n_{rout}} \le 10000 \tag{27}$$

$$P_{n_{rout}} \le 60\ ms \tag{28}$$

$$1 \le n_{rout} \le 10 \tag{29}$$

TABLE III: Operational Profiles of Incoming Call Frequency for the Illustrative Example

n_{serv}	cf **Operational Profile** (Hr/s)	n_{rout}
10	$10.000 \leq cf \leq 29.960$	1
	$29.961 \leq cf \leq 65.079$	2
	$65.080 \leq cf \leq 100.00$	3

TABLE IV: ADR Reliability and Performance Predictions for the Illustrative Example based on its Operational Profiles

n_{serv}	cf (Hr/s)	n_{rout}	R	P (ms)
10	**25**	**1**	1965.000	59.453
	50	**2**	4260.000	58.835
	100	**3**	9180.000	57.672

In Equations (25) and (26), the performance and reliability thresholds are always satisfied in the range of $1 \leq n_{rout} \leq 10$. In Section IV-B2, we mentioned an example preference function with an importance weight of 1.0 for reliability and 0.0 for performance giving the highest priority to reliability. This preference function chooses the lowest possible value for n_{rout}, i.e., it favors more centralized routing to improve reliability. Using the example function, we can select $n_{rout} = 1$ and decide for a central routing configuration (such as the CE architecture) on the lower bound of the expected frequency range, i.e., $cf = 10$ Hr/s. We now find the highest possible cf where central routing is still applicable, in other words the reliability and performance predictions are below the thresholds. Using Equations (20) and (21) when $n_{rout} = 1$, $n_{serv} = 10$:

$$R_1 = 78.6 \cdot cf \leq 10000 \tag{30}$$

$$P_1 = 56.696 + 0.11028 \cdot cf \leq 60\ ms \tag{31}$$

Within the expected frequency range, i.e., $10 \leq cf \leq 100$ Hr/s, the predictions for reliability are always below the reliability threshold. However, when solving the performance model (Equation (31)), the highest acceptable frequency with which central routing stays within the defined thresholds is $P = 29.960$ Hr/s. Remember in Section IV-B2, we mentioned that an architect can define operational profiles for the incoming frequency; therefore, we can define a low level for cf in which central routing is reasonable as:

$$10.000 \leq cf \leq 29.960\ Hr/s \tag{32}$$

Note that as soon as there is a call frequency outside of this range, the *ADR Reliability/Performance Monitor* triggers the *ADR Dynamic Reconfigurator* to reconfigure the routers (see Figures 2 and 3). We take the higher bound of the operational profile in Equation (32), i.e., $cf = 29.960$ Hr/s, and repeat the process in Equations 25 to 31 to find all operational profiles of the expected incoming frequency, i.e., $10 \leq cf \leq 100$ and the respective final reconfiguration choice based on the preference function, which are reported in Table III. We consider a frequency level from our experiment (see Section V-A) in each of these operational profiles, i.e., $cf \in \{25, 50, 100\}$ Hr/s. Table IV presents the ADR model predictions.

D. Empirical Validation

Table V presents the empirical measurements of the adaptation using the ADR architecture on our private cloud as well as GCP infrastructure (see Section V-A). We calculate the prediction error of our models reported in Table IV using the mean absolute percentage error [32]. Let $model_i$ and $empirical_i$ be the result of the model, and the measured empirical data for experimental case i, respectively:

$$Err = \frac{100\%}{n} \cdot \sum_{c \in Cases} \left| \frac{model_i - empirical_i}{empirical_i} \right| \tag{33}$$

$Cases$ is the set of all incoming call frequencies and the number of services and n is the length of $Cases$, i.e., $n = 3$ call frequencies in this example. The prediction errors of our models are reported in Table V. On our experiment infrastructure, the prediction error regarding reliability is $ErrR = 8.9\%$ and regarding performance is $ErrP = 17.0\%$. As mentioned, the performance model is a system-specific regression analysis that needs to be performed for each application separately. Therefore, on GCP we only empirically validate the reliability model. The prediction error regarding reliability on GCP is $ErrR = 14.0\%$. Given the commonly used target prediction accuracy of 30.0% in the cloud quality of service research [24], these results are more than reasonable.

VI. Evaluation

In this section, we evaluate the ADR architecture by comparing the ADR performance and reliability predictions to the empirical results of our experiment, already reported in [4], [5]. Note that the ADR architecture is not specific to our experiment infrastructure, nor to our experimental cases. Architects can freely use the proposed architecture and adjust it to their needs as we explained in the last section. That is, we use our empirical data set[11] for the evaluation of ADR using measured data of an extensive experiment.

Systematic Analysis

In the last section, we studied one sample case with specific reliability and performance thresholds, plus an example preference function. In order to systematically evaluate our proposed ADR architecture, we go through a range of thresholds and importance weights given for reliability and performance. Then, we compare ADR model predictions (see Table IV for an example) with our experimental cases reported in Section V-A. That is, we compare ADR with its fixed counterparts, i.e., CE ($n_{rout} = 1$), DR ($n_{rout} = 3$) and SA ($n_{rout} = n_{serv}$). As mentioned, in our experiment (see Section V-A), we had three levels for n_{serv}, i.e., 3, 5 and 10, for each of which we consider the expected incoming call frequency of $10 \leq cf \leq 100$ Hr/s (see Section V-C for an illustrative example). In this range,

[11]The data is published as an open access data set to support replicability. https://ieee-dataport.org/documents/amiri-tsc-2021 doi:10.21227/mahp-mw44

TABLE V: ADR Empirical Measurements for the Illustrative Example Case

$\boldsymbol{n_{serv}}$	$\boldsymbol{cf}$ (Hr/s)	$\boldsymbol{n_{rout}}$		$\boldsymbol{R}$	$\boldsymbol{P}$ (ms)	$\boldsymbol{ErrR}$ (%)	$\boldsymbol{ErrP}$ (%)		$\boldsymbol{R}$	$\boldsymbol{ErrR}$ (%)
				Experiment Infrastructure					Google Cloud Platform	
10	**25**	**1**		2450.000	54.116	8.852	16.973		1792.000	14.015
	50	**2**		4434.000	44.100				6014.000	
	100	**3**		9448.000	53.577				9486.000	

we studied four levels of cf, i.e., 10, 25, 50, 100 Hr/s, in our experiment. Therefore, we have nine experimental cases: Three architectures each configured with three n_{serv} values (which are operational for four levels of call frequencies.)

Regarding reliability and performance thresholds, we start with very tight reliability and very loose performance thresholds so that only centralized routing is acceptable. Then, we slightly increase the reliability and decrease the performance thresholds by 10% in each step so that distributed routing becomes applicable. In order to find the starting points, we take the worst-case scenario of our empirical data into consideration. In Equation (19), a higher n_{serv} results in a higher expected request loss (empirically validated in [4], [5]); in our experimental cases, the highest value for n_{serv} is 10 services. As mentioned before, CE is the most reliable and SA gives the best performance. With $n_{serv} = 10$, the worst-case reliability and performance predictions for CE are 7860 request lost per 10 minutes of system time, i.e., 13.1 requests per second, and 67.724 ms average RTT per each request. On the other hand, the worst-case predictions for SA with ten services are 13800 request per 10 minutes of experiment, i.e., 23.0 requests per second, and 39.311 ms average RTT. We adjust these values slightly and take our boundary thresholds as follows:

$$8000 \leq R_{th} \leq 14000 \tag{34}$$

$$40 \leq P_{th} \leq 70\ ms \tag{35}$$

Regarding importance weights, we start with an importance weight of 1.0 for reliability and 0.0 for performance giving the highest priority to reliability, and decrease the reliability importance (consequently increase the performance weight) by 10% in each iteration. In total, we evaluate 1089 systematic evaluation cases: 11 threshold levels and 11 importance weight levels, which are each evaluated for 9 experimental cases[12]. Then, we calculate the average percentage differences as:

$$\overline{\Delta R} = \frac{100\%}{n} \cdot \sum_{c \in Cases} \frac{R_c - R_{adr}}{R_{adr}} \tag{36}$$

$$\overline{\Delta P} = \frac{100\%}{n} \cdot \sum_{c \in Cases} \frac{P_c - P_{adr}}{P_{adr}} \tag{37}$$

Here, $Cases$ is the set of all incoming call frequencies and the number of services, i.e., $cf \in \{10, 25, 50, 100\}$ and $n_{serv} \in \{3, 5, 10\}$; therefore, $n = 12$, i.e., the length of $Cases$.

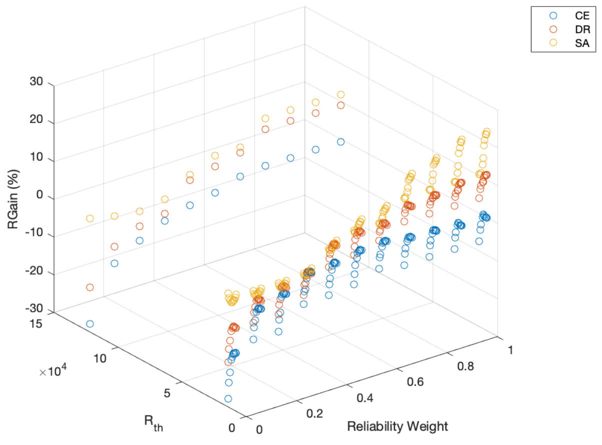
[12]To support reproducibility, the evaluation script as well as the evaluation log containing detailed information of each systematic evaluation case are published as an open access artifact: https://zenodo.org/record/5655383, doi:10.5281/zenodo.5655383

We define reliability gain as the percentage of reliability improvement in contrast to the performance worsening. Similarly, performance gain is defined as the percentage of performance increase comparing to reliability degradation:

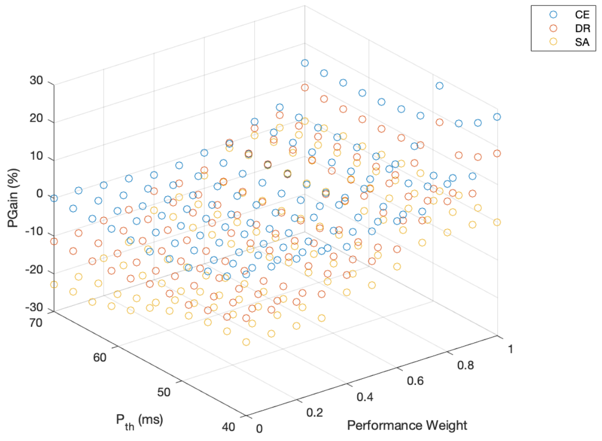
$$RGain = \left|\overline{\Delta R}\right| - \left|\overline{\Delta P}\right| \tag{38}$$

$$PGain = -RGain \tag{39}$$

Figure 5 shows the reliability and performance gains of ADR compared to the CE, DR and SA architectures. As it can



(a) Reliability Gain



(b) Performance Gain

Fig. 5: ADR Reliability and Performance Gains Compared to CE, DR and SA Architectures

TABLE VI: Statistics of the Gain Percentages

		Reliability				Performance		
Arch.		**CE**	**DR**	**SA**		**CE**	**DR**	**SA**
$\boldsymbol{Min}$		-27.923	-18.245	-2.344		-2.240	-11.390	-22.848
$\boldsymbol{Q_1}$		-5.854	-2.1640	-0.658		-0.324	-8.798	-11.809
$\boldsymbol{Median}$		-1.845	4.762	6.125		1.845	-4.762	-6.125
$\boldsymbol{Q_3}$		0.324	8.798	11.810		5.854	2.164	0.658
$\boldsymbol{Max}$		2.240	11.390	22.848		27.923	18.245	2.344
$\boldsymbol{\sigma}$		6.204	7.000	7.607		6.204	7.000	7.607
$\boldsymbol{Mean}$		-3.981	2.535	7.089		3.981	-2.535	-7.089
$\boldsymbol{Mean}$ RWeight > 0.5		-0.963	8.145	13.997		0.963	-8.145	-13.997
$\boldsymbol{Mean}$ RWeight < 0.5		-7.454	-3.550	0.073		7.454	3.550	-0.073

be seen in Figure 5a, ADR has almost always the highest reliability gain compared to SA. This is expected because SA is a completely distributed routing extreme resulting in the lowest reliability compared to the other architectures: A higher number of routers results in a higher request loss according to Equation (19). As the reliability weight increases and the performance importance lowers, the reliability gain improves.

The DR architecture, i.e., a static configuration with three routers, was considered a middle ground of the three studied architectures in our experiment. Here, the same holds true, i.e., the reliability and performance gains of DR are between those of CE and SA architectures. Figure 5b shows that ADR has the highest performance gain compared to CE. However comparing to the architectures, the ADR performance gain does not differ greatly. We investigate the statistics of the data reported in Table VI, in which Q_1, Q_3, σ and $RWeight$ are the first and the third quartiles, the standard deviation and the reliability weight, respectively. Note that according to Equation (39), the statistics regarding the mean of performance gains are the negative of those of reliability gains as shown in Table VI. Mean of data is calculated over 121 cases, i.e., 11 threshold levels and 11 weights.

This investigation should illustrate that in cases, where the wrong architecture choice is made, significant increases in reliability or performance are offered by ADR, i.e., 22.8% reliability gain compared to SA, and 27.9% performance gain compared to CE architecture. Let us now investigate the mean of data, i.e. where cases with correct and incorrect architecture choices are analyzed together, to show that even here ADR provides improvements. The mean performance gain compared to CE is 4.0%, that is on average over all cases, ADR gains more performance than it loses reliability compared to the centralized routing. When taking those cases into account, in which performance has a higher importance weight than reliability, i.e., $RWeight < 0.5$, the mean percentage gain for CE is 7.5%. On the other hand, the mean reliability gain for SA is 7.1%, i.e., ADR offers a higher reliability gain than performance loss compared to a completely distributed routing averaged over all experimental cases. Taking only those cases where reliability is of higher importance than performance, i.e., $RWeight > 0.5$, the mean reliability gain is 14.0%.

VII. Threats to Validity

In this section, we discuss the threats to the validity, along with some limitations of our study.

A. *Construct Validity*

We used request loss and the round-trip times of requests as metrics of reliability and performance, respectively. While this is a common approach in service- and cloud-based research (see Section VIII), the threat remains that other metrics might model these quality attributes better, e.g., cascade of calls beyond a single call sequence for reliability [26], or data transfer rates of messages which are *m* byte-long for performance [23]. More research, probably with real-world systems, is required for this threat to be excluded.

B. *Internal Validity*

As mentioned in Section III, the dynamic routing architectures are based on many different technologies. Our ADR architecture abstracts the controlling logic component in dynamic routing under a concept called router to allow interoperability between these architectures. In a real-world system, changing between these technologies is not an easy task, but it is not impossible either. In this paper, we provided a scientific proof-of-concept based on an experiment with prototypical implementation of these technologies. The threat remains that, in a real-world application, changing between these technologies might have other impacts on reliability and performance, e.g., network latency increasing processing time.

Moreover, we considered a simple reconfiguration strategy to start the new setup in parallel with the running configuration to avoid impacts on reliability, e.g., request loss as a result of reconfiguration, and performance, e.g., increased processing time while reconfiguring. In a real-world system, this solution is cost-ineffective that introduces additional resource demands. The architects must specify a reconfiguration strategy based on their application needs to mitigate this threat.

C. External Validity

We designed our novel architecture with generality in mind, and explained in detail how architects can specify ADR to their needs (see Section V). In spite of the fact that we systematically evaluated ADR using the data of our extensive experiment of 1200 hours with 1089 evaluation cases, the threat remains that evaluating ADR based on another infrastructure may lead to different results. To mitigate this thread, we empirically validated our measurements on Google Cloud Platform infrastructure and showed that our results are applicable (see Section V-D).

Moreover, as in all research presenting an abstract model of a real-world phenomenon, there needs to be a balance between the level of abstraction and applicability of the proposed ideas. We mitigated this threat by many rounds of review and improvement in the author team, as well as constant comparison to existing related studies.

VIII. Related Work

Related works come from the areas of architecture-based reliability and performance prediction and MCO, as well as self-adaptive software systems.

Architecture-based reliability and performance prediction approaches [14], [32] employ (i) probabilistic analytical models such as discrete-time Markov chains (DTMCs) [13] and (layered) queueing networks (QNs) [31], or (ii) high-level architectural models such as profile-extended UML [27] or Palladio [9], [10] models, which are simulated or transformed into analytical models. The approaches are based on the observation that the reliability and performance of a system depend on the reliability and performance of each component, along with the interplay between them. Probabilistic modeling is often applied, e.g. based on DTMCs. Brosch et al. [10] suggest an extension of the Palladio Component Model along with automated transformations into a DTMC.

Some other works introduce *service- and cloud-specific reliability models*. For instance, Wang et al. [33] propose a DTMC model for analyzing system reliability based on constituent services. Grassi and Patella [17] propose an approach for reliability prediction that considers the decentralized and autonomous nature of services. Zheng and Lyu [35] propose an approach that employs past failure data to predict a service's reliability. However, none of these approaches focuses on major routing architecture patterns in service- and cloud-based architectures; they are rather based on a very generic model with regard to the notion of a service. With regard to *architecture-based performance prediction*, numerous approaches have been proposed. Spitznagel and Garlan [31] present a general architecture-based model for performance analysis based on queueing network theory. Petriu et al. [27] present an architecture-based performance analysis that builds Layered Queueing Network performance models from a UML description of the high-level architecture of a system.

Architecture-based MCO [2] builds on top of these prediction approaches and the application of architectural tactics to search for (pareto) optimal architectural candidates. Example MCO approaches supporting reliability and performance are ArcheOpterix [1], PerOpteryx [11], and SQuAT [28]. Like our study, those works focus on supporting architectural design or decision making. In contrast to our work, they do not focus on specific kinds of architectures or architecture patterns. Our approach differs from these methods in that it focuses specifically on cloud- and service-based dynamic routing patterns.

Finally, our approach is related to *self-adaptive systems*, which typically use MAPE-K loops [6], [7], [21] and similar approaches to realize adaptations. Our approach is based on the MAPE-K loop structure and extends such approaches with support specific to cloud- and service-based dynamic routing architectures. In a similar way, *auto-scalers for the cloud* [8], [34], which promise stable quality of service and cost minimization when facing changing workload intensity, and in general research on cloud elasticity [16], [18] are related to our work. Our approach is similar to auto-scaling, but performs the adaptation only for the dynamic routers.

Major contributions of our approach are that, in contrast to the existing related work, it considers reliability and performance trade-offs together and focuses on specific architecture patterns for dynamic routing in service- and cloud-based architectures. By focusing on these specific patterns and possible runtime self-adaptations, we can define a very targeted model along with a specific reconfiguration algorithm and preference functions to perform MCO analysis, which would be hard in the generic case.

IX. Conclusion and Future Work

In our prior work [3]–[5], we studied three representative dynamic routing architectures (see Figure 1). We created an analytical model of request loss as a metric of system reliability, and performed a multiple regression analysis to statistically model performance using round-trip times. In this paper, we hypothesized that a self-adaptation between the dynamic routing architectures is beneficial over any fixed architecture selections. We set out to answer whether we can find an optimal configuration of routers that automatically adapts the reliability and performance trade-offs in dynamic routing architectures based on monitored system data at runtime (**RQ1**), and how the reliability and performance predictions of the chosen optimal solution compare with the case where one architecture runs statically (**RQ2**).

For **RQ1**, we proposed a novel architecture, i.e., Adaptive Dynamic Routers (ADR), which automatically adjusts performance and reliability trade-offs based on a multi-criteria optimization analysis. For **RQ2**, we systematically evaluated ADR using 1089 evaluation cases based on the empirical data of our extensive experiment of 1200 hours of runtime (see Section V-A). Our results show that our hypothesis holds, and ADR can indeed adapt the architectures in a running system to optimize between reliability and performance. If the wrong architectural choice has been made, ADR can lead to substantial gains in reliability or performance.

Even on average, where cases with the right and the wrong architecture choice are analyzed together, ADR offers good results. For example, on average it offers 14.0% higher reliability than performance loss compared to completely distributed routing when reliability is of a higher importance, and 7.5% more performance gains than reliability decrease compared to centralized routing, when performance is of a higher priority. Moreover, we empirically validated our model predictions on our private experiment infrastructure and on Google Cloud Platform (GCP) public cloud. The empirical validation had a prediction error of 14.0% on GCP which indicate that our approach is applicable outside of our private infrastructure.

To the best of our knowledge, there has not been any architecture presented in the literature which automatically adjusts reliability and performance trade-offs specifically in service- and cloud-based dynamic routing. Our proposed ADR architecture adapts, based on triggers, e.g., change of incoming load frequency or degradation of monitoring data, to an optimal configuration to prevent request loss or increase of round-trip times. Prior to our work, architects needed to manually redesign and redeploy architecture configurations. For our future work, we plan to apply our novel architecture to real-world applications and evaluate the results.

References

[1] A. Aleti, S. Björnander, L. Grunske, and I. Meedeniya. Archeopterix: An extendable tool for architecture optimization of AADL models. In *ICSE 2009 Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES 2009*, pages 61–71. IEEE, 2009.

[2] 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.

[3] 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 (SCC)*, 2019.

[4] 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 (ICSOC)*, 2020.

[5] 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 (TSC)*, 2021.

[6] P. Arcaini, E. Riccobene, and P. Scandurra. Modeling and analyzing mape-k feedback loops for self-adaptation. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 13–23. IEEE, 2015.

[7] 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 (TAAS)*, 11(4):1–35, 2017.

[8] A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev. Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):800–813, 2018.

[9] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th International Workshop on Software and Performance*, WOSP '07, page 54–65, New York, NY, USA, 2007. ACM.

[10] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner. Architecture-based reliability prediction with the palladio component model. *IEEE Transactions on Software Engineering*, 38(6):1319–1339, 2011.

[11] A. Busch, D. Fuchss, and A. Koziolek. Peropteryx: Automated improvement of software architectures. In *IEEE International Conference on Software Architecture ICSA Companion 2019*, pages 162–165. IEEE, 2019.

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

[13] R. C. Cheung. A user-oriented software reliability model. *IEEE transactions on Software Engineering*, pages 118–125, 1980.

[14] V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-based software performance analysis*. Springer, 2011.

[15] Envoy. Service mesh. *https://www.learnenvoy.io/articles/service-mesh.html*, 2019.

[16] 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.

[17] V. Grassi and S. Patella. Reliability prediction for service-oriented computing environments. *IEEE Internet Computing*, 10(3):43–49, 2006.

[18] 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 ({ICAC} 13)*, pages 23–27, 2013.

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

[20] A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson. *Cloud Design Patterns*. Microsoft Press, 2014.

[21] D. G. D. L. Iglesia and D. Weyns. Mape-k formal templates to rigorously design behaviors for self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(3):1–31, 2015.

[22] 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.

[23] N. Kratzke. About microservices, containers and their underestimated impact on network performance. *arXiv preprint arXiv:1710.04049*, 2017.

[24] D. A. Menascé and V. A. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, 2001.

[25] Microsoft. Sidecar pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar, 2010.

[26] M. Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[27] D. Petriu, C. Shousha, and A. Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. *IEEE Transactions on Software Engineering*, 26(11):1049–1065, 2000.

[28] A. Rago, S. A. Vidal, J. A. Diaz-Pace, S. Frank, and A. van Hoorn. Distributed quality-attribute optimization of software architectures. In *Proceedings of the 11th Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS 2017*, pages 7:1–7:10. ACM, 2017.

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

[30] D. L. Rubinfeld. Reference guide on multiple regression. *Federal Judicial Center, 2nd edition*, 2000.

[31] B. Spitznagel and D. Garlan. Architecture-based performance analysis. In *Proc. the 1998 Conference on Software Engineering and Knowledge Engineering*. Carnegie Mellon University, June 1998.

[32] K. S. Trivedi and A. Bobbio. *Reliability and availability engineering: modeling, analysis, and applications*. Oxford University Press, 2017.

[33] L. Wang, X. Bai, L. Zhou, and Y. Chen. A hierarchical reliability model of service-based software system. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 199–208, July 2009.

[34] F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li. Quantifying cloud elasticity with container-based autoscaling. *Future Generation Computer Systems*, 98:672–681, 2019.

[35] Z. Zheng and M. R. Lyu. Collaborative reliability prediction of service-oriented systems. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 35–44, May 2010.