Tool Support for the Adaptation of Quality of Service Trade-Offs in Service- and Cloud-Based Dynamic Routing Architectures*

Amirali Amiri[1,2] and Uwe Zdun[1]

[1] University of Vienna, Software Architecture Group, Vienna, Austria
[2] University of Vienna, Doctoral School Computer Science, Vienna, Austria
{firstname.lastname}@univie.ac.at

Abstract. Dynamic routing is an essential part of service- and cloud-based applications. Routing architectures are based on vastly different implementation concepts, such as API Gateways, Enterprise Service Buses, Message Brokers, or Service Proxies. However, their basic operation is that these technologies dynamically route or block incoming requests. This paper proposes a new approach that abstracts all these routing patterns using one adaptive architecture. We hypothesize that a self-adaptation of the dynamic routing is beneficial over any fixed architecture selections concerning reliability and performance trade-offs. Our approach dynamically self-adapts between more central or distributed routing to optimize system reliability and performance. This adaptation is calculated based on a multi-criteria optimization analysis. We evaluate our approach by analyzing our previously-measured data during an experiment of 1200 hours of runtime. Our extensive systematic evaluation of 4356 cases confirms that our hypothesis holds and our approach is beneficial regarding reliability and performance. Even on average, where right and wrong architecture choices are analyzed together, our novel architecture offers 9.82% reliability and 47.86% performance gains.

Keywords: Self-Adaptive Systems · Dynamic Routing Architectures · Service- and Cloud-Based Applications · Reliability and Performance Trade-Offs · Prototypical Tool Support

1 Introduction

Dynamic routing is common in service- and cloud-based applications, for which different techniques are available. These techniques range from simple strategies, e.g., request routing based on load balancing, to more complex routing, such as checking for compliance with regulations. Assume a company has to comply with a regulation that the data of European customers have to be stored and processed on European servers based on the General Data Protection Regulation[3].

*This work was supported by FWF (Austrian Science Fund), projects IAC^2: I 4731-N, API-ACE: I 4268.

[3] https://gdpr.eu

In such a case, Dynamic Routers [17] can update the data-flow paths at runtime to ensure compliant data handling. Multiple dynamic-routing architectural patterns are provided for service- and cloud-based environments. These patterns include centralized routing, e.g., using an API Gateway [27] or an Enterprise Service Bus [10], and distributed routing using multiple Dynamic Routers [17] or Sidecars [20, 27] to make local routing decisions.

The dynamic-routing architectures are based on vastly different implementations. However, they all route or block requests essentially. There is a possibility to change these patterns, e.g., from centralized to distributed routing, by adjusting the number of routers in a service- and cloud-based system. To do so, we should monitor the quality-of-service measures and make architectural decisions. So far, the trade-offs of reliability and performance measures in cloud-based dynamic routing have not been specifically and extensively studied. Reliability and performance in relation are essential for designing routing architectures. This factor must be considered because changing the routing schema to improve performance, e.g., by adding more routers for parallel processing of requests, may lead to a decrease in system reliability as more points of a crash are introduced to a system (empirically validated in [3]).

Our study is motivated by example scenarios, such as assuming a sudden reliability decrease is observed in a software system by adding services to the system for the parallel processing of requests (increasing the performance). In such a situation, time is important 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 trade-offs. Thus, we study the 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: *What is the architecture of a supporting tool that analyses the system at runtime and facilitates the reconfiguration of a dynamic routing application using the optimal configuration solution?*

RQ3: *How do the reliability and performance predictions of the chosen optimal solution compare with the case where one architecture runs statically?*

The contributions of this paper are three-fold. Firstly, we propose an adaptive-routing architecture that automatically adjusts the quality-of-service trade-offs. Secondly, we introduce an analytical model of performance that is generalizable to dynamic-routing applications and analyze the trade-offs of reliability and performance. Finally, we provide a prototypical tool that generates deployment artifacts for reconfiguring a dynamic-routing application. Additionally, our tool provides a visualization environment for users to study different configurations without generating additional artifacts.

The structure of the paper is as follows: Section 2 presents the overview of our approach. Section 3 explains the proposed architecture in detail, presenting our performance model and the trade-off analysis. Section 4 provides the tool

that supports our architectural concepts. Section 5 presents the evaluation of the presented approach, and Section 6 discusses the threats to the validity of our research. We study the related work in Section 7 and conclude in Section 8.

2 Approach Overview

The proposed architecture in this paper is based on Monitor, Analyze, Plan, Execute, Knowledge (MAPE-K) loops [4, 5, 19]. Our adaptive architecture automatically changes between different dynamic-routing patterns by reconfiguring service- and cloud-based applications according to an optimization analysis [2]. We define a *router* as an abstraction for any controller component that makes routing decisions, e.g., an API Gateway [27], an Enterprise Service Bus [10], or Sidecars [20, 27]. Our approach changes the number of routers, i.e., changes between different configurations 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.

Metamodel Figure 1 presents the metamodel of our architecture. A *Model* describes multiple elements. *Host* is any execution environment, either physical or virtual. Each *Component* is deployed on (up to) one *Host* at each point in time. *Request* models the request flow, linking a source and a destination component. There are several different component types. *Clients* send *Client Requests* to *API Gateways*. The gateways send *Internal Requests* to *Routers* and *Services*.

Configurator Components perform the reconfiguration, and *Reconfigurable Components* are the adaptation targets of our architecture. *Monitor* observes reconfigurable components and the requests that pass the gateways. *Manager* manages the control flow of the reconfiguration by calling *Infrastructure as Code (IaC)* to update the infrastructure, or *Scheduler* to reschedule the containers. *Visualizer* provides visualizations of the architectural configurations.

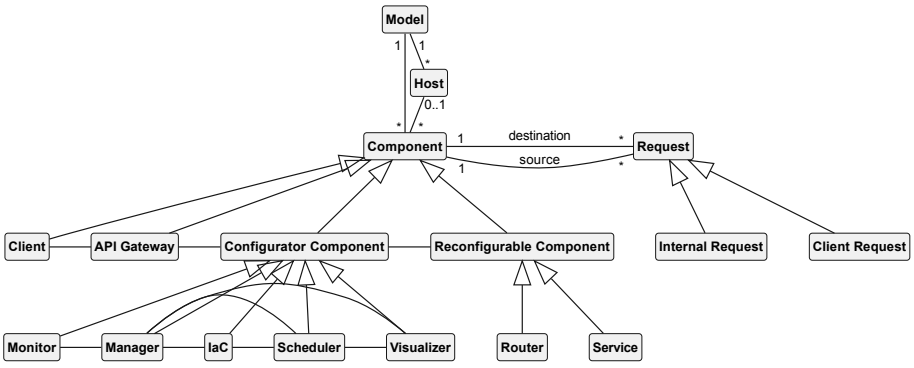


Fig. 1: Metamodel of the Adaptive Architecture

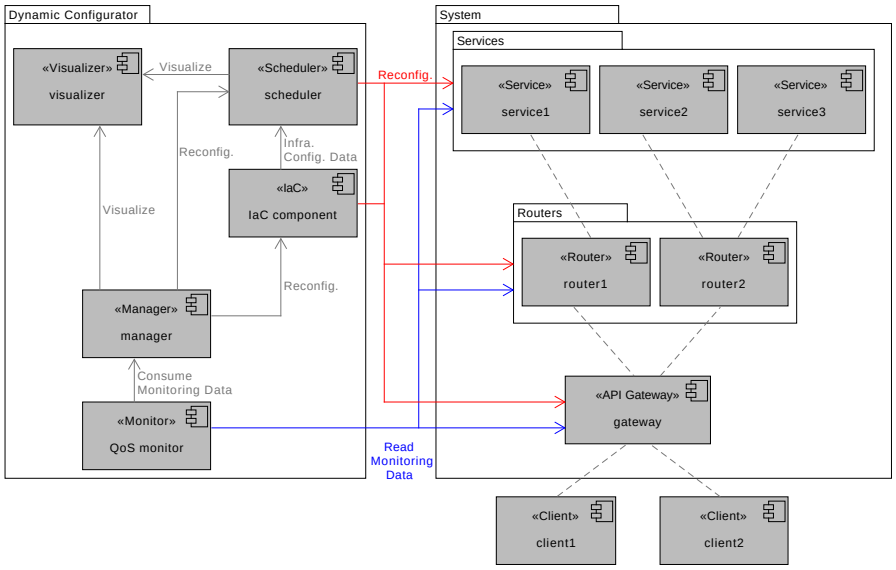


Fig. 2: Component Diagram of an Example Configuration (dashed lines represent the data flow and solid lines the reconfiguration control flow.)

Example of a Routing Configuration Figure 2 presents a component diagram of a sample configuration, in which dashed lines represent the data flow and solid lines the reconfiguration control flow of an application. As shown, clients access the system via a gateway that publishes monitoring data to the Quality-of-Service (QoS) monitor component. The configuration manager observes the monitoring data and triggers a reconfiguration. Moreover, the manager can communicate with the visualizer component to visualize the current architecture configuration. The manager calls the IaC component if infrastructure changes are needed. IaC reconfigures the infrastructure and triggers the scheduler to reschedule the containers. Alternatively, if there is no need for infrastructure reconfiguration, the manager directly triggers the scheduler. After a reconfiguration, the scheduler can call the visualizer.

3 Approach Details

This section introduces the details of our proposed architecture.

3.1 Reconfiguration Activities of the Dynamic Configurator

Figure 3 shows the reconfiguration activities of the dynamic configurator. The QoS monitor reads monitoring data and checks for reconfiguration, e.g., when degradation of reliability and performance metrics are observed. Moreover, the

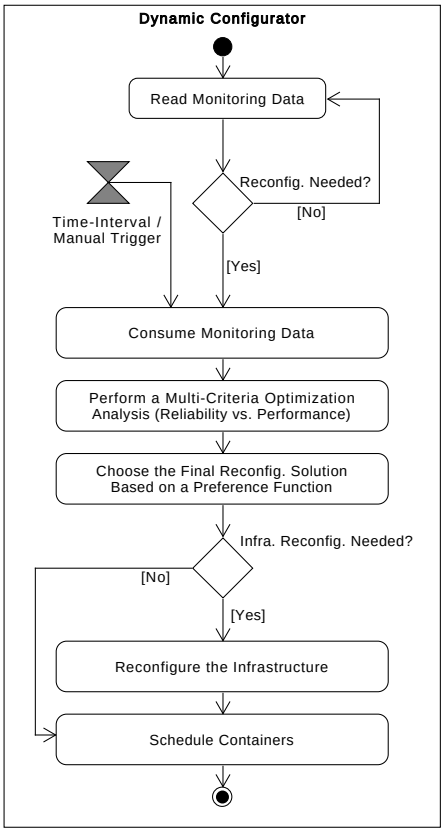


Fig. 3: Reconfiguration Activities of the Dynamic Configurator

reconfiguration can be triggered periodically or manually by an architect. When a reconfiguration is triggered, the reconfiguration manager consumes the monitoring data, performs a multi-criteria optimization analysis [2], and chooses a final reconfiguration solution. Either the IaC component is triggered to reconfigure the infrastructure or the scheduler reschedules the containers. Our architecture is based on MAPE-K loops [4, 5, 19]. The QoS monitor implements the *monitor* and *analyze* stages, the manager develops the *plan* step, and the IaC component and the scheduler realize the *execute* step. We use our models as *knowledge*.

3.2 Analytical Models

Reliability Model Based on Bernoulli processes [31], request loss during router and service crashes can be modeled as follows [3]:

$$R = \frac{\lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in Com} CP_c \cdot d_c}{T} \tag{1}$$

In this formula, request loss is defined as the number of client requests not processed due to a failure, such as a component crash. Equation (1) gives the request loss per second as a metric of reliability by calculating the expected value of the number of crashes. Having this information, we sum all the requests received by a system during the downtime of a component and divide them by the observed system time T. We model the crash interval as CI that is the interval during which we check for a crash of a component. To clarify, CI is the time between two consecutive health checks when the heartbeat pattern [18] or the health check API pattern [26] are used. cf is the incoming call frequency based on requests per second (r/s). Com is the set of components, i.e., routers and services. CP_c is the crash probability of each component, and d_c is the average downtime of a component after it crashes.

Performance Model We model the average processing time of requests per router as a performance metric. This metric is important as it allows us to study the quality of service factors, such as the efficiency of architecture configurations. The total number of client requests, i.e., Req, is the call frequency cf multiplied by the observed time T:

$$Req = cf \cdot T \tag{2}$$

The number of processed requests is the total number of client requests minus the request loss. Let P be performance. The average processing time of requests per router is given as follows:

$$P = \frac{T}{n_{rout}(Req - R)} \tag{3}$$

Using Equations (1) to (3), the average processing time is the following:

$$P = \frac{T}{n_{rout} \cdot cf \left(T - \lfloor \frac{T}{CI} \rfloor \cdot \sum_{c \in Com} CP_c \cdot d_c\right)} \tag{4}$$

Model Validations To empirically validate our models, we ran an experiment of 200 runs with a total of 1200 hours of runtime (excluding setup time) [3]. We had a private cloud setting with three physical nodes and installed virtual machines with eight cores and 60 GB of system memory. Each router or service was containerized in a Docker[4] container. Moreover, to ensure generalizability, we duplicated our experiment on Google Cloud Platform[5] and empirically validated our results (see below for experiment cases). We compared our analytical reliability and performance model with our empirical results using the mean absolute percentage error [31]. With more experiment runs, we observed an ever-decreasing error, converging at 7.1%. Our analytical performance model yielded a low error rate of 0.5%, indicating the very high accuracy of our model. We also evaluated our models using the mean absolute error, the mean square error, and the root mean square error, which confirmed our results.

[4] `https://www.docker.com`

[5] `https://cloud.google.com/`

Parameterization of Model to Experiment Values In our experiment, we defined n_{serv} and n_{rout} as the number of services and routers to study their effects. We had three levels for the number of services, i.e., $n_{serv} \in \{ 3, 5, 10 \}$. Based on our experience and a survey of existing cloud applications in the literature and industry, the number of cloud services directly dependent on each other in a call sequence is usually rather low. Moreover, we had four levels for incoming call frequencies, i.e., $cf(r/s) \in \{ 10, 25, 50, 100 \}$. The call frequency of $cf = 100\, r/s$, or even lower numbers, is chosen in many studies (see, e.g., [13,30]). Therefore, we chose different portions between 10 to 100 r/s. We studied three architecture configurations, i.e., centralized routing ($n_{rout} = 1$), completely distributed routing with one router per each service ($n_{rout} = n_{serv}$), and a middle ground with three routers ($n_{rout} = 3$). Therefore, we have $n_{rout} \in \{ 1, 3, n_{serv} \}$. Overall, we evaluated our model in 36 experiment cases.

We also defined some constants as follows: We observed the system for $T = 600\ s$ in each experiment case, had a crash interval of $CI = 15\ s$, and studied uniform crash probabilities and downtimes for all components as $CP_c = 0.5\%$ and $d_c = 3\ s$, respectively. These values are system-specific and can be updated based on different infrastructures. Considering these experiment cases, we can parameterize our general reliability model in r/s (presented by Equation (1)) and performance model in ms (given by Equation (4)) as follows:

$$R = cf \cdot 0.001(n_{serv} + n_{rout}) \tag{5}$$

$$P = \frac{1000}{n_{rout} \cdot cf(1 - 0.001(n_{serv} + n_{rout}))} \tag{6}$$

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 proposed architecture has n_{serv} services and is under stress for a period of time with the call frequency of cf. To optimize reliability and performance, the system can change between different architecture configurations dynamically by adjusting n_{rout}, ranging from a centralized routing ($n_{rout} = 1$) and up to the extreme of one router per service ($n_{rout} = n_{serv}$).

We use the notations $R_{n_{rout}}$ and $P_{n_{rout}}$ to specify the reliability and performance of an architecture configuration by its number of routers. For instance, only configuring one router R_1 indicates the reliability model of centralized routing, and configuring n_{serv} routers (i.e., $R_1, \ldots, R_{n_{serv}}$) indicates completely distributed routing. Let R_{th} and P_{th} be the reliability and performance thresholds. The MCO question is: Given a cf and n_{serv}, what is the optimal number of routers that minimizes request loss and average processing time for requests per router without the predicted values violating the respective thresholds?

$$Minimize$$

$$R_{n_{rout}} \tag{7}$$

$$P_{n_{rout}} \tag{8}$$

Subject to

$$R_{n_{rout}} \leq R_{th} \tag{9}$$

$$P_{n_{rout}} \leq P_{th} \tag{10}$$

$$1 \leq n_{rout} \leq n_{serv} \tag{11}$$

Typically, there is no single answer to an MCO problem. Using the above MCO analysis, we find a range of n_{rout} configurations that all meet the constraints. One end of this range optimizes reliability and the other performance. We need a preference function so our approach can automatically select a final n_{rout} value.

Preference Function An architect defines an importance vector that gives weights to reliability and performance. The preference function instructs the proposed architecture to choose a final n_{rout} value in the range found by the MCO analysis based on this importance vector. Let us consider an example: When performance is of the highest importance to an application, an architect gives the highest weight, i.e., 1.0, to performance and the lowest weight, i.e., 0.0, to reliability. Thus, the preference function chooses the highest value on the n_{rout} range to choose more distributed routing. This reconfiguration results in processing client requests in parallel, giving a higher performance.

Algorithm 1: Reconfiguration Algorithm

Input: R_{th}, P_{th}, performanceWeight

$R_{n_{rout}}, P_{n_{rout}}, cf, n_{serv} \leftarrow$ **readMonitoringData()**

routersRange $\leftarrow$ **MCO(**$cf, n_{serv}, R_{n_{rout}}, P_{n_{rout}}, R_{th}, P_{th}$**)**

reconfigSolution $\leftarrow$ **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
```

Automatic Reconfiguration As shown in Figure 2, the QoS monitor reads the monitoring data from the API Gateway and feeds this information to the reconfiguration manager. This manager reconfigures the infrastructure or reschedules the containers. Algorithm 1 presents our reconfiguration algorithm. The QoS monitor triggers the reconfiguration algorithm, e.g., whenever reliability or performance metrics degrade. Time intervals, manual triggering or change in the incoming load can also be used to trigger the algorithm if more appropriate than metrics degradation. Note that `reconfigureRouters(reconfigSolution)` performs the final reconfiguration based on the chosen solution by either reconfiguring the infrastructure using the IaC component or rescheduling the containers using the container scheduler. Our supporting tool provides a simple implementation.

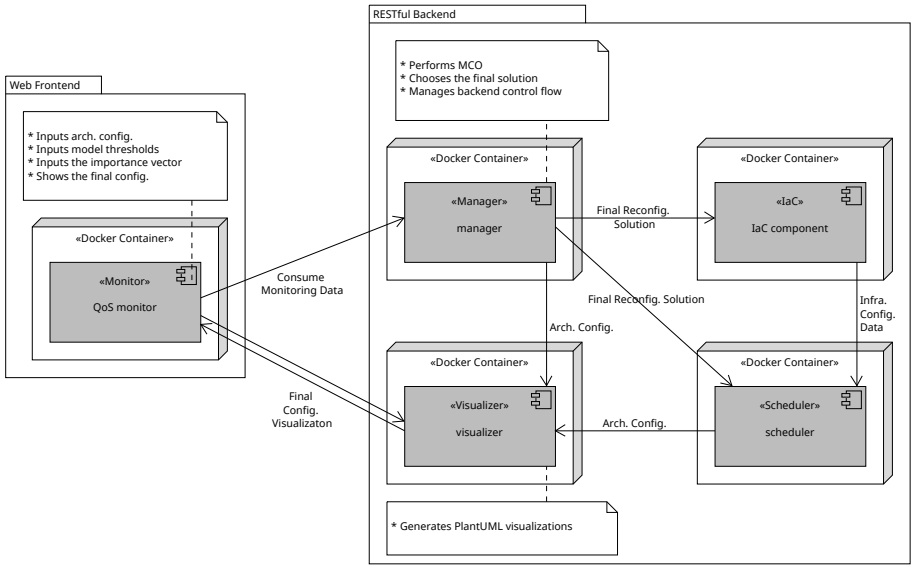


Fig. 4: Tool Architecture Diagram

4 Tool Overview

We developed a prototypical tool to demonstrate our adaptive architecture, which is available in our online artifact[6]. Figure 4 shows the tool architecture. We provide two modes, i.e., deployment and visualization. In the case of deployment, our tool generates artifacts in the form of Bash[7] scripts and configuration

[6]The online artifact of our study can be anonymously downloaded from `https://zenodo.org/record/7944823`

[7]`https://www.gnu.org/software/bash/`

files, e.g., infrastructure configuration data to be used by an IaC tool. These scripts can schedule containers using the Docker technology[8]. We also provide a visualization environment that only generates diagrams using PlantUML[9].

The frontend of our application provides the functionalities of the QoS monitor, i.e., to specify architecture configurations as well as model elements such as reliability and performance thresholds. This information is sent to the manager component in the backend that finds the final reconfiguration solution (see Algorithm 1). The manager sends this solution to the IaC component and the scheduler to generate deployment artifacts. A visualization is then created in the backend and shown in the frontend. The frontend is implemented in React[10] and the backend is developed in Node.js[11] as a RESTful application.

The tool flow of our application is as follows: An architect gives the architecture configuration by entering the number of services and routers. Users also specify model thresholds, call frequency of client requests, and performance weight. A reconfiguration is triggered when metrics degradation is observed, according to timers or manually. When reconfiguration is triggered, the backend performs an MCO analysis and chooses a final reconfiguration solution. If the deployment mode is chosen, deployment artifacts will be generated. The reconfiguration visualization is then created and shown.

5 Evaluation

In this section, we evaluate our architecture by comparing the reliability and performance predictions to the empirical results of our experiment (see Section 3.2). The proposed architecture is neither specific to our infrastructure nor our cases. We use our empirical data set in our online artifact[6] to evaluate our approach.

5.1 Evaluation Cases

We systematically evaluate our proposed architecture through various thresholds and importance weights for reliability and performance. We compare our model predictions with our 36 experiment cases (see Section 3.2 for the rationale behind choosing them). That is, we compare with three fixed architecture configurations, i.e., $n_{rout} \in \{ 1, 3, n_{serv} \}$ and three levels of services, i.e., $n_{serv} \in \{ 3, 5, 10 \}$. We consider four levels of call frequencies, i.e., $cf \in \{ 10, 25, 50, 100 \}\ r/s$. Regarding reliability and performance thresholds, we start with very tight reliability and very loose performance thresholds so that only centralized routing is acceptable. We increase the reliability and decrease the performance thresholds by 10% in each step so that distributed routing becomes applicable.

To find the starting points, we consider the worst-case scenario of our empirical data. Equation (1) informs that a higher n_{serv} results in a higher expected

[8] `https://www.docker.com/`
[9] `https://plantuml.com/`
[10] `https://reactjs.org/`
[11] `https://nodejs.org/`

request loss. In our experiment, the highest number of services is ten. With $n_{serv} = 10$, the worst-case reliability for centralized routing and completely distributed routing ($n_{rout} = 10$) is 1.1 and 2.0 r/s, respectively. Regarding performance, for the case of $n_{serv} = 10$, we investigate our predictions to find a range where a reconfiguration is possible. The lowest possible performance prediction is 33.7 ms, and the highest is 101.1 ms. We adjust these values slightly and take our boundary thresholds as follows. We analyze step-by-step by increasing the reliability threshold and decreasing the performance threshold by 10% as before.

$$1.1 \leq R_{th} \leq 2.0 \ r/s \tag{12}$$

$$35 \leq P_{th} \leq 100 \ ms \tag{13}$$

We start with an importance weight of 1.0 for reliability and 0.0 for performance. We decrease the reliability importance and increase the performance weight by 10% in each iteration. Overall we evaluate 4356 systematic evaluation cases: 36 experiment cases, 11 importance weight levels, and 11 thresholds. To support reproducibility, the evaluation script and the evaluation log detailing information about each case are provided in the online artifact of our study[6].

5.2 Results Analysis

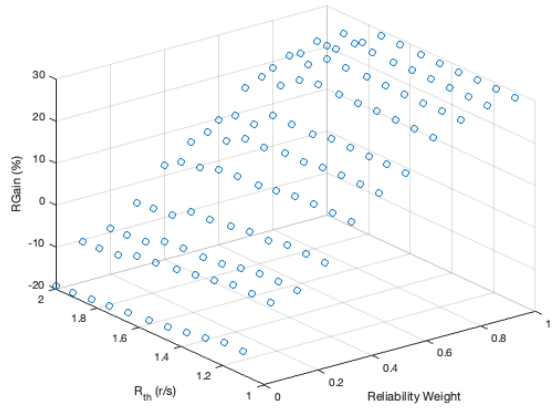
We define reliability gain, i.e., $RGain$, and performance gain, i.e., $PGain$, as the average percentage differences of our predictions compared to those of fixed architectures, i.e., $n_{rout} \in \{\ 1, 3, n_{serv}\ \}$. These formulas are based on the Mean Absolute Percentage Error (MAPE), widely used in the cloud QoS research [31].

$$RGain = \frac{100\%}{n} \cdot \sum_{c \in Cases} \frac{R_c - R_{n_{rout}}}{R_{n_{rout}}} \tag{14}$$

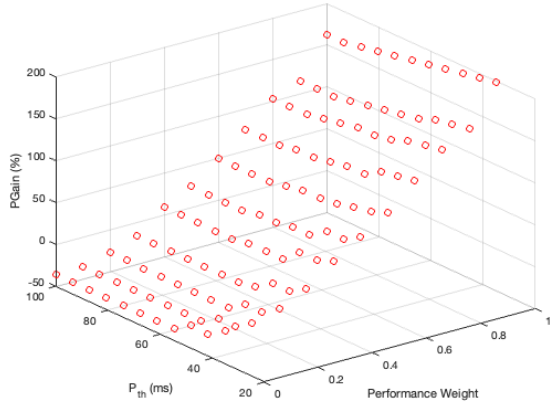
$$PGain = \frac{100\%}{n} \cdot \sum_{c \in Cases} \frac{P_c - P_{n_{rout}}}{P_{n_{rout}}} \tag{15}$$

Remember $R_{n_{rout}}$ and $P_{n_{rout}}$ are reliability and performance predictions (see the MCO analysis in Section 3.2). $Cases$ are our experiment cases, so $n = 36$.

Figure 5 shows the reliability and performance gains compared to the predictions of fixed architecture configurations, i.e., without adaptations. Our adaptive architecture provides improvements in both reliability and performance gains. As more importance is given to the reliability of a system, i.e., reliability weight increases, our architecture reconfigures the routers so that the gain in reliability rises, as shown by Figure 5a. Regarding performance, the same trend can be seen in Figure 5b. A higher performance weight results in a higher performance gain. On average, when cases with correct and incorrect architectural choices are analyzed together, our adaptive architecture provides 9.82% and 47.86% reliability and performance gains, respectively. A higher gain for performance compared to reliability is expected. To clarify, studying Equations (5) and (6) informs that changing the number of routers has a higher effect on the performance than



(a) Reliability Gain



(b) Performance Gain

Fig. 5: Reliability and Performance Gains of our Adaptive Architecture Compared to Fixed Architecture Configurations ($n_{rout} \in \{ 1, 3, n_{serv} \}$)

a system's reliability. We define performance as the average processing time of requests per router. Having a higher number of routers to process the requests in parallel divides the average processing time by more routers. However, only the sum of the number of services and routers affects the reliability.

6 Threats to Validity

Regarding *construct validity*, we used request loss and the average processing time of requests per router as reliability and performance metrics, respectively.

While this is a common approach in service- and cloud-based research (see Section 7), the threat remains that other metrics might model these quality attributes better, e.g., a cascade of calls beyond a single call sequence for reliability [22], or data transfer rates of messages which are m byte-long for performance [21]. More research, probably with real-world systems, is required for this threat to be excluded.

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

Regarding *External validity*, we designed our novel architecture with generality in mind. However, the threat remains that evaluating our approach based on another infrastructure may lead to different results. To mitigate this thread, we systematically evaluated the proposed architecture with 4356 evaluation cases (see Section 5 for details). Moreover, the results might not be generalizable beyond the given experiment cases of 10-100 requests per second and call sequences of length 3-10. As this covers a wide variety of loads and call sequences in cloud-based applications, the impact of this threat should be limited.

Regarding *Conclusion validity*, as the statistical method to evaluate the accuracy of our model's predictions, we defined reliability and performance gains based on the Mean Absolute Percentage Error (MAPE) metric [31] as it is widely used and offers good interpretability in our context.

7 Related Work

Architecture-based approaches [12, 31] employ probabilistic analytical models such as discrete-time Markov chains (DTMCs) [11] and Queueing Networks (QNs) [29]. Some papers use high-level architectural models such as profile-extended UML [23] or Palladio [7, 8] models that are simulated or transformed into analytical models. These works are based on the observation that a system's reliability and performance depend on those of each component, along with the interplay between them. Pitakrat et al. [24] use architectural knowledge to predict how a failure propagates to other components based on Bayesian networks.

Other studies introduce service- and cloud-specific reliability models. For instance, Wang et al. [32] propose a DTMC model for analyzing system reliability based on constituent services. Grassi and Patella [15] propose an approach for reliability prediction that considers the decentralized and autonomous nature of services. Zheng and Lyu [34] propose an approach that employs past failure data to predict a service's reliability. However, none of these approaches focuses on major routing architectural patterns in service- and cloud-based ar-

chitectures; they are rather based on a very generic model concerning the notion of service. Moreover, numerous approaches have been proposed that study architecture-based performance prediction. Spitznagel and Garlan [29] present a general architecture-based model for performance analysis based on QNs.

Architecture-based MCO [2] builds on top of these prediction approaches and the application of architectural tactics to search for optimal architectural candidates. Example MCO approaches supporting reliability and performance is ArcheOpterix [1], PerOpteryx [9], and SQuAT [25]. Sharma and Trivedi [28] present an architecture-based unified hierarchical model for software reliability, performance, security, and cache behavior prediction. This is one of the few studies that consider performance and reliability. 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 architecture or architectural patterns. Our approach focuses on service- and cloud-based dynamic routing.

Finally, our approach is related to self-adaptive systems, which typically use MAPE-K loops [4, 5, 19] and similar approaches to realize adaptations. Our approach is based on the MAPE-K loop structure and extends such approaches with support specific to the cloud- and service-based dynamic routing architectures. Similarly, auto-scalers for the cloud [6, 33], which promise stable QoS and cost minimization when facing changing workload intensity, and in general research on cloud elasticity [14,16] 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 architectural patterns for dynamic routing in service- and cloud-based architectures. By focusing on runtime adaptations, we defined a targeted model and a reconfiguration algorithm, which is hard to consider in the generic case.

8 Conclusions

In this paper, 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**), what the architecture of a supporting tool that analyses the system at runtime and facilitates the reconfiguration of a dynamic routing application using the optimal configuration solution is (**RQ2**), and how the reliability and performance predictions of the chosen optimal solution compare with the case where one architecture runs statically (**RQ3**).

For **RQ1**, we proposed a routing architecture that dynamically self-adapts between different routing patterns based on the need of an application. For **RQ2**, we provided a prototypical tool that analyzes different inputs and creates deployment artifacts. This tool also provides visualizations to study different architecture configurations. For **RQ3**, we systematically evaluated our approach using 4356 evaluation cases based on the empirical data of our extensive experiment of 1200 hours of runtime (see Section 5). The results confirms that the

proposed architecture can adapt the routing pattern in a running system to optimize reliability and performance. Even on average, where cases with the right and the wrong architecture choices are analyzed together, our approach offers a 9.82% reliability gain and a 47.86% performance gain. For our future work, we plan to apply our novel architecture to real-world applications.

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, 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.
4. P. Arcaini, E. Riccobene, and P. Scandurra. Modeling and analyzing mape-k feedback loops for self-adaptation. In *IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 13–23, 2015.
5. 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.
6. 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.
7. 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. ACM, 2007.
8. 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.
9. 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.
10. D. A. Chappell. *Enterprise service bus*. O'Reilly, 2004.
11. R. C. Cheung. A user-oriented software reliability model. *IEEE transactions on Software Engineering*, pages 118–125, 1980.
12. V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-based software performance analysis*. Springer, 2011.
13. D. J. Dean, H. Nguyen, P. Wang, and X. Gu. Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, 2014.
14. 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.
15. V. Grassi and S. Patella. Reliability prediction for service-oriented computing environments. *IEEE Internet Computing*, 10(3):43–49, 2006.

16. 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.
17. G. Hohpe and B. Woolf. *Enterprise Integration Patterns.* Addison-Wesley, 2003.
18. A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson. *Cloud Design Patterns.* Microsoft Press, 2014.
19. 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.
20. 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.
21. N. Kratzke. About microservices, containers and their underestimated impact on network performance. *arXiv preprint arXiv:1710.04049*, 2017.
22. M. Nygard. *Release It!: Design and Deploy Production-Ready Software.* Pragmatic Bookshelf, 2007.
23. 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.
24. T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske. Hora: Architecture-aware online failure prediction. *Journal of Systems and Software*, 137:669–685, 2018.
25. 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.
26. P. Raj, A. Raman, and H. Subramanian. *Architectural Patterns: Uncover essential patterns in the most indispensable realm.* Packt Publishing, December 2017.
27. C. Richardson. Microservice architecture patterns and best practices. *http://microservices.io/index.html*, 2019.
28. V. S. Sharma and K. S. Trivedi. Architecture based analysis of performance, reliability and security of software systems. In *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, page 217–227, New York, NY, USA, 2005. Association for Computing Machinery.
29. 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.
30. O. Sukwong, A. Sangpetch, and H. S. Kim. Sageshift: managing slas for highly consolidated cloud. In *2012 Proceedings IEEE INFOCOM*, pages 208–216, 2012.
31. K. S. Trivedi and A. Bobbio. *Reliability and availability engineering: modeling, analysis, and applications.* Oxford University Press, 2017.
32. 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.
33. 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.
34. 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.