Cost-Aware Multifaceted Reconfiguration of Service- and Cloud-Based Dynamic Routing Applications

Amirali Amiri
University of Vienna, Software Architecture Group
University of Vienna, Doctoral School Computer Science
Vienna, Austria
amirali.amiri@univie.ac.at

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

***Abstract*—Dynamic reconfiguration is commonly used in service- and cloud-based applications. In combination with autoscalers, dynamic routers can adapt the system to the resource demands, e.g., in an e-commerce application offering discounts for services in a specific location. Without such measures, the quality-of-service measures are affected negatively, and a system overload can lead to an application being non-responsive. However, the cost of cloud resource usage must be considered when performing these reconfiguration steps to avoid adding high additional costs. This paper proposes a cost-aware multifaceted reconfiguration of dynamic routing applications. We study the depletion and rescheduling of idle components and use an infrastructure-as-code module to apply changes to the infrastructure. Moreover, when system components are in a steady state, our approach dynamically self-adapts between more central or distributed routing to optimize reliability and performance. This adaptation is calculated based on a system-wide optimization analysis. When components are overloaded, we perform a per-component optimization to autoscale components multidimensionally. Our extensive systematic evaluation shows significant improvements in quality trade-off adaptations and system overload prevention. We provide prototypical tool support to demonstrate our concepts with illustrative sample cases.**

***Index Terms*—Self-Adaptive Systems, Dynamic Routing, Reliability and Performance Trade-Offs, Prototypical Tool Support, System Overload, Cost-Awareness, Multidimensional Autoscaling**

I. Introduction

Cloud-based systems require dynamic routing for efficient performance. Due to the constantly changing nature of modern applications, dynamic routers such as API Gateways [27], Enterprise Service Buses [8], Message Brokers [14], or Sidecars [16] are typically utilized. These routing patterns may differ in implementation, but all serve the purpose of routing or blocking requests. To switch between these dynamic routing patterns, the number of routers in a service- and cloud-based system can be adjusted. However, monitoring the quality of service measures and making architectural decisions automatically is essential. Designing routing architectures requires careful consideration of both reliability and performance. Adding more routers to improve performance may increase the risk of system crashes due to introducing additional points of failure.

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

Moreover, when adapting the routing architecture pattern from distributed to centralized routing (and vice versa), we should ensure that the components are not overloaded. Cloud computing provides an elastic infrastructure to manage this dynamic behaviour. Horizontal autoscaling, i.e., adding or removing replicas, and vertical autoscaling, i.e., adding or removing resources, are commonly used in practice. A newer concept is multidimensional autoscaling[1] that combines the two previous methods in one decision-making step. However, the concept is not fully developed and has limitations, such as not considering the incoming load as an input.

Consider, for instance, an e-commerce shop that offers discounted products for a specific location. The application must cope with a sudden incoming load increase that needs to be routed to these services. Dynamic routers and autoscalers can accommodate the increased demand. However, when adding components for the parallel processing of requests to increase performance, a reliability decrease is observed as there are more points of failure [2] in a system. Without such measures, a system overload can lead to an application being non-responsive. Nevertheless, if cloud resource costs are not considered, a business may lose profit by inducing high costs when dealing with sudden load spikes. To address such scenarios, we set out to answer the research questions:

RQ1: *Can we find a cost-aware multifaceted reconfiguration approach for dynamic routing applications to adapt quality-of-service trade-offs and prevent component overloads?*

RQ2: *What is the architecture of a supporting tool that facilitates the reconfiguration of a dynamic routing application using the optimal configuration solution?*

RQ3: *How well does this multifaceted reconfiguration perform compared with the case where one architecture runs statically?*

The contributions of this paper are as follows. Firstly, we model components as queuing stations [17] and consider different scenarios, i.e., when components are idle, steady, and transient. We introduce a cost-aware multifaceted reconfiguration of dynamic routing applications using an *Infrastructure as Code (IaC)* module to apply changes to the infrastructure.

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

Moreover, we consider a *system-wide* Multi-Criteria Optimization analysis (MCO) [1] to optimize system reliability and performance, as well as a *per-component* MCO to autoscale components multidimensionally. Secondly, we provide a prototypical tool that facilitates the reconfiguration of dynamic routing applications. Our application provides artifacts to be used by IaC tools and a visualization environment to study different configurations and demonstrate our concepts.

To evaluate our approach, we consider multiple levels of call frequencies, component configurations, and routing profiles that we studied in an already-published empirical study [3]. Our extensive systematic evaluation shows significant improvements in quality trade-off adaptations and system overload prevention. Our approach yields up to 16.60% reliability gain and an average performance gain of 74.22%.

The structure of the paper is as follows: Section II presents an approach overview. Section III explains our approach in detail, and Section IV gives illustrative sample cases. Section V provides our prototypical tool support. Section VI presents the evaluation of the presented approach, and 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. Approach Overview

In this paper, we study a cost-aware multifaceted reconfiguration of dynamic routing applications. A *router* is defined as an abstraction for any controller component that makes routing decisions, e.g., an API Gateway [27], an enterprise service bus [8], or Sidecars [16]. We model the system *components*, i.e., services and routers, as queuing stations [17] having two subcomponents, namely a buffer and a processor as shown in Figure 1. Let λ be the arrival rate and μ the processing rate of a component based on the number of requests per second r/s. Incoming requests are buffered in a queue by a rate of λ and processed by a rate of μ.

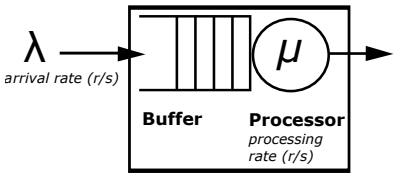


Fig. 1: Components as Queuing Stations

A component is in a *steady state* when its processing rate is greater than or equal to its arrival rate:

$$\mu \geq \lambda \tag{1}$$

In the steady state, a component is not overloaded and can process incoming requests without delay because of buffering. On the other hand, the *transient state* refers to when a component is overloaded because its processing rate is lower than the arrival rate of the requests:

$$\mu < \lambda \tag{2}$$

We study three interrelated scenarios for components:

TABLE I: Table of Mathematical Notations

Notation	Description
n_{rout}	Number of routers
n_{serv}	Number of services
n_{req}	Number of request
n_{scal}	Number of scaling replicas
n_{pro}	Number of processing rate improvements
r/s	requests per second
R	Reliability Request loss model in r/s
R_{th}	Reliability threshold in r/s
$RGain$	Reliability gain in %
$R(n_{rout})$	Reliability prediction in r/s
P	Performance model in ms
P_{th}	Performance threshold in ms
$PGain$	Performance gain in %
$P(n_{rout})$	Performance prediction in ms
T	Observed system time in s
CI	Crash interval in s
cf	Incoming call frequency in r/s
Com	Set of all components
$Rout$	Set of all routers
d_c	Average downtime of a component c in r/s
P_c	Crash probability of a component c in %
BFR	Buffer fill rate in r/s
BFR_r	Buffer fill rate of a router r in r/s
$BFR(n_{scal}, n_{pro})$	Buffer fill rate for autoscaling in r/s
μ	Processing rate of a component in r/s
μ_r	Processing rate of a router r in r/s
λ	Arrival rate of a component in r/s
λ_r	Arrival rate of a router r in r/s
$\overline{RR}$	Average reconfiguration ratio
$\overline{C}$	Average reconfiguration costs in $cents/s$
C_{th}	Cost Threshold in $cents/s$
$C(n_{rout})$	Reconfiguration costs in $cents/s$
$C(n_{scal} = 1)$	Cost of scaling out in $cents/s$
$C(n_{pro} = 1)$	Cost of increasing the processing rate in $cents/s$
$C(n_{scal}, n_{pro})$	Cost of multidimensional autoscaling in $cents/s$

- when components are idle and can be depleted.
- when components are active and steady.
- when components are overloaded.

The first scenario considers the infrastructure changes when a reconfiguration occurs. The second scenario studies a *per system* reconfiguration, which means we monitor the state of a system as a whole and reconfigure the components. The third scenario is a *per component* reconfiguration, i.e., our approach monitors and reconfigures each component separately.

We study our empirical data set already reported in [3] to present illustrative examples (see Section IV) and to evaluate our approach. In our prior work, we performed an extensive experiment of 1200 hours and measured the quality-of-service metrics of dynamic routing applications. Our data set can be downloaded in the online artifact of this paper to support reproducibility[2]. Table I presents our mathematical notations.

III. Approach Details

This section presents the details of our approach.

[2] Published as an open-access artifact: https://zenodo.org/record/7771328

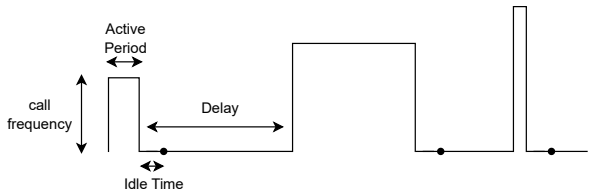


Fig. 2: The Sporadical Load Profile of System Components (Dots represent depletion.)

A. Depletion of Idle Components

Some system components process requests sporadically and are idle between active periods. We characterize the *sporadical load profile* with a frequency of incoming requests for an active period followed by a delay of no incoming requests. As shown in Figure 2, we deplete components when idle (represented by dots). However, the depleted components can become active again and must be rescheduled. So we must consider the infrastructure changes, e.g., not overloading cloud nodes. We use an IaC module to automatically create and free cloud nodes to efficiently use resources and reduce costs. On the one hand, when depleting idle components, an efficient rescheduling of other active components might free a node. On the other hand, when a depleted component receives a request and needs to be rescheduled, all nodes might be occupied. The IaC module can create a node and schedule the component.

Assume the capacity of each node is known based on the number of scheduled containers. Algorithm 1 provides the steps to reconfigure the infrastructure. The IaC Module calculates the total capacity of nodes. If the number of system components exceeds the total capacity, a new node is created, and components are scheduled. Otherwise, the IaC module checks if the containers can be rescheduled efficiently on fewer nodes. Having defined the infrastructure reconfiguration steps, we check if a component is idle and deplete it (see Figure 2). When a request is received for a depleted component, Algorithm 1 schedules it either on existing nodes or a new one.

Algorithm 1: Infrastructure Reconfiguration Algorithm (reconfigure)

```
Input: n_serv, n_rout
totalCapacity ← 0
foreach node : nodes do
    totalCapacity ← totalCapacity + capacity(node)
end
scheduleContainers()
if (n_serv + n_rout) > totalCapacity then
    createNode()
    scheduleContainers()
else
    foreach node : nodes do
        restCapacity ← totalCapacity − capacity(node)
        if (n_serv + n_rout) ≤ restCapacity then
            scheduleContainers()
            deleteNode(node)
        end
    end
end
```

B. Reconfiguration of Steady Components

When all components are active and steady according to Equation (1), we consider a system-wide MCO [1] optimizing reliability and performance trade-offs of the system as a whole.

1) Definitions: We define the following model elements in our reliability and performance models. n_{rout} and n_{serv} are the number of routers and services, and CI is the crash interval, i.e., the interval during which we check for a crash of a component. Assuming the heartbeat pattern [15] or the health check API pattern [24] are used, CI is the time between two consecutive health checks. cf is the call frequency (r/s), Com is the set of components, i.e., routers and services, P_c is the crash probability of each component, and d_c is the average downtime of a component after it crashes.

2) Reliability Model: Based on Bernoulli processes [31], request loss R during component crashes is modeled [2] as:

$$R = \frac{\lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in Com} P_c \cdot d_c}{T} \tag{3}$$

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 (3) 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.

3) Performance Model: We model the average processing time of requests per router as a performance metric P. This metric is important as it allows us to study the quality of service factors, e.g., the efficiency of architecture configurations.

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

We count the processed requests in this formula by subtracting the request loss from the total requests. We divide the observed time by the processed requests and the number of routers. Section IV-A presents an illustrative sample case.

4) System-Wide MCO: We perform a multi-criteria optimization analysis to reconfigure an application by adjusting n_{rout}. We use the notations $R(n_{rout})$ and $P(n_{rout})$ to specify the reliability and performance predictions of an architecture configuration by their number of routers. Let R_{th} and P_{th} be the reliability and P_{th} performance thresholds. We aim to minimize request loss and average processing time of requests per router without the prediction values violating R_{th} and P_{th}.

Additionally, we must ensure that the reconfiguration costs do not exceed a cost threshold. We define $C(n_{rout})$ as the reconfiguration costs for an architecture configuration by its number of routers and C_{th} as the cost threshold. Moreover, when choosing a lower n_{rout}, we must ensure that reducing

Algorithm 2: System-Wide Optimization Analysis (systemWideMCO)

Input: cf, n_{serv}, R_{th}, P_{th}, C_{th}

highBound $\leftarrow (R = R_{th})$ and $(1 \leq \text{highBound} \leq n_{serv})$
lowBound $\leftarrow (P = P_{th})$ and $(1 \leq \text{lowBound} \leq \text{highBound})$
$routersRange \leftarrow \{\}$
foreach *solution : [lowBound, highBound]* **do**
  $solutionInRange \leftarrow$ true
  if $C(n_{rout}) > C_{th}$ **then**
    $solutionInRange \leftarrow$ false
  end
  foreach *r : Rout* **do**
    if $\lambda_r > \mu_r$ **then**
      $solutionInRange \leftarrow$ false
  end
  if $solutionInRange$ **then**
    $routersRange \leftarrow routersRange \cup \{solution\}$
  end
end
return routersRange

the number of routers does not overload the routers. Let $Rout$ be the set of routers of a system:

$$Minimize$$
$$R(n_{rout}) \tag{5}$$
$$P(n_{rout}) \tag{6}$$
$$Subject\ to$$
$$R(n_{rout}) \leq R_{th} \tag{7}$$
$$P(n_{rout}) \leq P_{th} \tag{8}$$
$$C(n_{rout}) \leq C_{th} \tag{9}$$
$$1 \leq n_{rout} \leq n_{serv} \tag{10}$$
$$\mu_r \geq \lambda_r \quad \forall\ r \in Rout \tag{11}$$

Typically, there is no single answer to an MCO problem but a set of acceptable points in the solution space [1]. Algorithm 2 provides a simple solution to find a range of acceptable n_{rout}. The lower end of this range represents more centralized routing, so we find the lowest acceptable n_{rout} that does not violate the performance threshold. Conversely, the highest possible n_{rout} is bound by the reliability threshold. Having found the lower and upper values, we exclude the solutions that violate the cost threshold or result in overloading a router.

5) Preference Function: We must choose a final reconfiguration solution on the n_{rout} range returned from the above analysis. An architect assigns weights to reliability and performance, so a preference function can automatically choose a final solution. For example, when performance is highly important, the preference function selects a higher n_{rout} to choose more distributed routing. This reconfiguration processes requests in parallel, giving a higher performance.

6) Reconfiguration Algorithm: Algorithm 3 presents our reconfiguration steps triggered, for instance, whenever reliability or performance metrics degrade. Time intervals, manual triggering, or changes in the incoming load can also trigger the algorithm if more appropriate than metrics degradation.

C. Autoscaling of Overloaded Components

In this paper, we study a multifaceted reconfiguration of dynamic routing applications. When a system component is in a transient state (see Equation (2)), request processing is delayed because of buffering in an overloading component. In this case, we use multidimensional autoscaling[3] to bring the transient component to a steady state. To clarify, we consider two reconfiguration measures in a per-component MCO analysis: horizontal autoscaling, i.e., scaling out the component, and vertical autoscaling, i.e., adding resources.

1) Buffer Fill Rate: We define the Buffer Fill Rate (BFR) as the difference between the arrival and processing rates.

$$BFR = \lambda - \mu \tag{12}$$

BFR is an indicator that a component is in a transient state. In this case, we reconfigure an overloaded component. We define n_{scal} as the number of scaling replicas, n_{pro} as the number of processing rate improvements, and $BFR(n_{scal}, n_{pro})$ as buffer fill rate predictions for multidimensional autoscaling.

$$BFR(n_{scal}, n_{pro}) = \frac{n_{req} \cdot cf}{n_{scal} + 1} - (\mu + n_{pro}) \tag{13}$$

In this formula, n_{req} is the number of incoming requests for a component, and cf is the call frequency of requests.

Algorithm 3: System-Wide Reconfiguration Steps (systemWideReconfig)

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

$cf, n_{serv} \leftarrow$ **consumeMonitoringData()**
routersRange $\leftarrow$ **systemWideMCO**($cf, n_{serv}, R_{th}, P_{th}, C_{th}$)
reconfigSolution $\leftarrow$ **preferenceFunction**(routersRange, performanceWeight)
reconfigure(n_{serv}, reconfigSolution)

function preferenceFunction(range, PW)
begin
  length $\leftarrow$ max(range) - min(range) +1
  floor $\leftarrow \lfloor$ PW * length $\rfloor$
  if *floor == max(range)* **then**
    return max(range)
  else if *floor == 0* **then**
    return min(range)
  else
    return floor + min(range) -1
  end
end

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

Equation (13) comes from the fact that scaling out an overloading component divides its arrival rate by the total number of replicas, i.e., $n_{scal}+1$. The BFR is also affected by the added processing rate, i.e., $\mu + n_{pro}$.

2) Reconfiguration Cost: The cost of reconfiguration must be considered. Let $C(n_{scal}, n_{pro})$ be the cost of multidimensional autoscaling, $C(n_{scal}=1)$ the cost of scaling out a component by one replica, and $C(n_{pro}=1)$ the cost of increasing the processing rate of an overloading component by one r/s. The cost depends on the n_{scal} and n_{pro} improvements.

$$C(n_{scal}, n_{pro}) = n_{scal} \cdot C(n_{scal}=1) + n_{pro} \cdot C(n_{pro}=1) \quad (14)$$

Section IV-B presents a parameterization and a sample case.

3) Per-Component MCO: We adjust the buffer fill rate of an overloading component to bring it to a steady state. This reconfiguration is based on a second multi-criteria optimization analysis performed for each component separately. We aim to minimize BFR but with a minimum reconfiguration cost.

$$Minimize$$
$$BFR(n_{scal}, n_{pro}) \quad (15)$$
$$C(n_{scal}, n_{pro}) \quad (16)$$
$$Subject\ to$$
$$\lambda \leq \mu \quad (17)$$
$$C(n_{scal}, n_{pro}) \leq C_{th} \quad (18)$$

Remember that there is typically no single answer to an MCO problem but a set of acceptable points called the Pareto front [1]. Using a preference function, we choose a final solution that brings the component to a steady state according to Equation (1) with a minimum cost. Having done this analysis, all the components are in a steady state. We must perform a system-wide analysis as described in Section III-B. Algorithm 4 presents the reconfiguration steps.

Algorithm 4: Reconfiguration Algorithm for an Overloading Component

```
Input: R_th, P_th, C_th, performanceWeight
cf, n_serv, n_req, μ ← consumeMonitoringData()
paretoFront ← perComponentMCO(cf, n_req, μ, C_th)
reconfigSolution ← preferenceFunction(paretoFront)
reconfigure(n_serv, reconfigSolution)
systemWideReconfig(R_th, P_th, C_th, performanceWeight)

function preferenceFunction(paretoFront)
begin
    C ← C_th
    reconfigSolution ← (0, 0)
    foreach solution : paretoFront do
        C(n_scal, n_pro) ←
                n_scal · C(n_scal = 1) + n_pro · C(n_pro = 1)
        if C(n_scal, n_pro) ≤ C then
            C ← C(n_scal, n_pro)
            reconfigSolution ← solution
        end
    end
    return reconfigSolution
end
```

IV. Illustrative Sample Cases

A. Reconfiguration of Steady Components

We study an example from the data set of our experiment[2] (see Section II for details) to parameterize our models and give sample cases. An example configuration is shown in Figure 3, where clients send requests to an API gateway that forwards them to the services. We observed the system for $T = 600\,s$, had a crash interval of $CI = 15\,s$ and studied uniform crash probabilities and downtimes for all components as $P_c = 0.5\%$ and $d_c = 3\,s$. We can parameterize our reliability model (r/s) and performance model (ms) in Equations (3) and (4) as:

$$R = cf \cdot 0.001(n_{serv} + n_{rout}) \quad (19)$$
$$P = \frac{1000}{n_{rout} \cdot cf(1 - 0.001(n_{serv} + n_{rout}))} \quad (20)$$

In the example configuration, we have $n_{rout} = 3$ routers and $n_{serv} = 6$ services. Let us consider that this sample case has an expected call frequency of $cf = 25\,r/s$, and all routers have a processing rate of $\mu = 64\,r/s$. We parameterize the arrival rates and the number of incoming requests of routers (solid arrows in Figure 3) to check if they are overloaded. In our experiment, we allocated services equally to routers:

$$n_{req} = \frac{n_{serv}}{n_{rout}} \quad (21)$$
$$\lambda_r = cf \cdot n_{req} = \frac{cf \cdot n_{serv}}{n_{rout}} \quad \forall \quad r \in Rout \quad (22)$$

In our sample case, $n_{req} = 2$ and $\lambda_r = 50\,r/s$. Therefore, all routers are steady according to Equation (1).

To parameterize the cost functions, we use the Google Autopilot pricing[4]. Autopilot allows increments of 0.25 vCPUs per container (same is offered by Amazon Fargate[5]) that corresponds to 8 r/s in our experiment:

$$C(n_{pro} = 8) = 5 \cdot 10^{-4}\ cents\,/s \quad (23)$$

The scaling cost of our routers with $\mu = 64\ r/s$ accounts to:

$$C(n_{scal} = 1) = 4 \cdot 10^{-3}\ cents\,/s \quad (24)$$

We consider a reliability threshold of $1.2\ r/s$, a performance threshold of $35\ ms$, and a cost threshold of $1\ cent/s$. We study a case with a weight of 1.0 for performance and 0.0 for reliability. We perform the system-wide MCO analysis in Section III-B4 by rewriting Equations (19) and (20):

$$Minimize$$
$$R_{n_{rout}} = 0.075 + 0.025 \cdot n_{rout} \quad (25)$$
$$P_{n_{rout}} = \frac{1000}{n_{rout} \cdot (24.925 - 0.025 \cdot n_{rout})} \quad (26)$$

[4]https://cloud.google.com/kubernetes-engine/pricing
[5]https://aws.amazon.com/fargate/pricing/

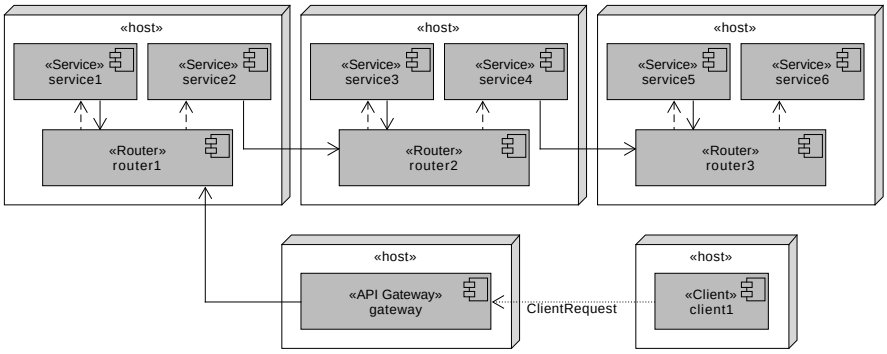


Fig. 3: Example Configuration of Dynamic Routing Applications
(Solid arrows show the incoming requests of routers.)

$$Subject\ to$$

$$R_{n_{rout}} \leq 1.2 \tag{27}$$

$$P_{n_{rout}} \leq 35\ ms \tag{28}$$

$$C(n_{rout}) \leq 1\ cent/s \tag{29}$$

$$1 \leq n_{rout} \leq 6 \tag{30}$$

$$\mu_r \geq \lambda_r \quad \forall\ r \in Rout \tag{31}$$

Equation (25) informs that the reliability predictions in the $1 \leq n_{rout} \leq 6$ always satisfy the reliability threshold. In Equation (26), the constraint on the performance threshold of $P_{n_{rout}} \leq 35\ ms$ gives the lowest value for the number of routers as $n_{rout} = 2$. Therefore, the range for n_{rout} is:

$$2 \leq n_{rout} \leq 6 \tag{32}$$

Following Algorithm2, we see that the cost threshold of 1 $cent/s$ is always satisfied in this range. We check if any solution results in overloading the routers in this range. On the lowest bound, i.e., $n_{rout} = 2$, we have the following according to Equations (21) and (22):

$$n_{req} = 3 \tag{33}$$

$$\lambda_r = 75\ r/s \tag{34}$$

Since $\mu_r = 64\, r/s$, this overloads the routers according to Equation (1). So we exclude this solution, and all the other points on the range are acceptable. The acceptable range is:

$$3 \leq n_{rout} \leq 6 \tag{35}$$

The performance weight is 1.0, so the preference function chooses the highest possible value for n_{rout} according to Algorithm 3. Therefore, the final solution is a configuration with six routers, i.e., $n_{rout} = 6$. We use this analysis also when illustrating our other scenario, i.e., autoscaling transient components multidimensionally to prevent system overload.

B. Autoscaling of Overloaded Components

Let us consider the studied example in Figure 3. Assume this application is stressed with a call frequency of $cf = 100\, r/s$. According to Equations (13), (21) and (22), we have:

$$n_{req} = 2 \tag{36}$$

$$\lambda_r = 200\ r/s \quad \forall\ r \in Rout \tag{37}$$

$$\mu_r = 64\ \ r/s \quad \forall\ r \in Rout \tag{38}$$

$$BFR_r = 136\ r/s \quad \forall\ r \in Rout \tag{39}$$

Having the same cost threshold of $C_{th} = 1\ cents/s$, we can rewrite the per-component MCO analysis in Section III-C3:

$$Minimize$$

$$\frac{200}{n_{scal} + 1} - (n_{pro} + 80) \tag{40}$$

$$8 \cdot n_{pro} \cdot 5 \cdot 10^{-4} + n_{scal} \cdot 4 \cdot 10^{-3} \tag{41}$$

$$Subject\ to$$

$$\lambda \leq \mu \tag{42}$$

$$C(n_{scal}, n_{pro}) \leq 1\ cents/s \tag{43}$$

As mentioned in Section III-C3, we choose a final solution that brings the component to a steady state with a minimum cost. Following Algorithm 4, this reconfiguration solution is:

$$(n_{scal}, n_{pro}) = (1, 40) \tag{44}$$

that gives the buffer fill rate of $BFR(1, 40) = -4$. This solution results in scaling out each router and increasing n_{rout} from three to six routers. Therefore, we must check that the system-wide MCO does not violate the thresholds. As we calculated before in Equation (35), the acceptable range of routers is $3 \leq n_{rout} \leq 6$. So the solution is acceptable.

V. Tool Support

We provide a prototypical tool in our online artifact[2]. Figure 4 shows the high-level tool architecture. The *Web Frontend* of our application provides the functionalities to specify architecture configurations and model elements, such as thresholds and cost functions. This information is sent to the *RESTful API* in the backend that invokes the *Optimizer* to perform MCO analyses and find the final reconfiguration solution. The *IaC Module* generates artifacts in the form

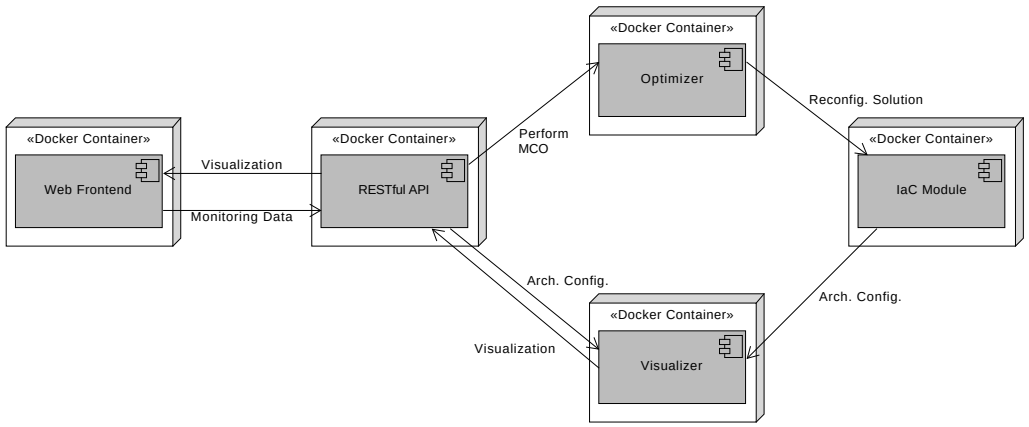


Fig. 4: Tool Architecture Diagram

of Bash[6] scripts and configuration files, e.g., infrastructure configuration data to be used by an IaC tool. These scripts can be used to schedule containers using the Docker technology[7]. The *Visualizer* creates diagrams of the configurations using PlantUML[8] that are shown in the *Web Frontend*. The frontend is implemented in React[9] and the backend in Node.js[10].

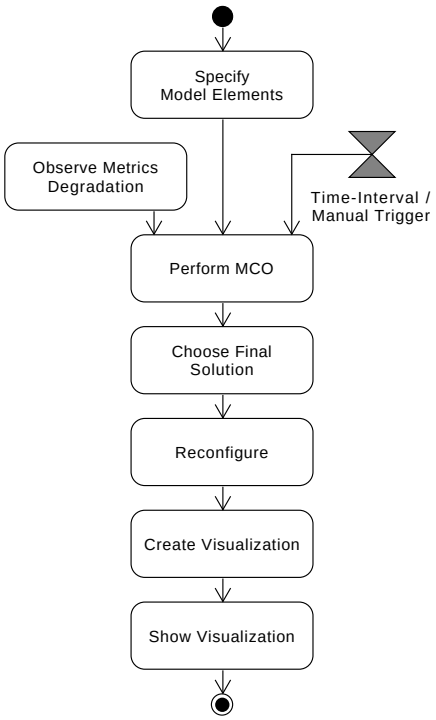


Fig. 5: Model Reconfiguration Toolflow

[6]https://www.gnu.org/software/bash/
[7]https://www.docker.com/
[8]https://plantuml.com/
[9]https://reactjs.org/
[10]https://nodejs.org/

Figure 5 shows the flow regarding the model reconfiguration. An architect specifies various model elements, i.e., the number of routers and services, thresholds, incoming call frequencies, performance weight, processing rates of components, and cost functions. A reconfiguration is triggered when metrics degradation is observed, according to timers or manually. When reconfiguration is triggered, the backend performs an MCO analysis, chooses a final reconfiguration solution, and generates IaC artifacts. The reconfiguration visualization is then created using PlantUML and shown in the frontend.

VI. Evaluation

This section evaluates our approach in both scenarios illustrated in Section IV systematically. We compare the model values to our empirical data set (see Section II). Note that our study is neither specific to our experiment infrastructure nor our cases. We use our empirical data set to evaluate our approach using measured data from an extensive experiment.

A. Reconfiguration of Steady Components

We present our evaluation when components are steady.

1) Evaluation Cases: We systematically evaluate our method through various thresholds and importance weights for reliability and performance. We compare our model predictions with 9 experiment cases: three levels of routers and three levels of services, each operational for four levels of cf.

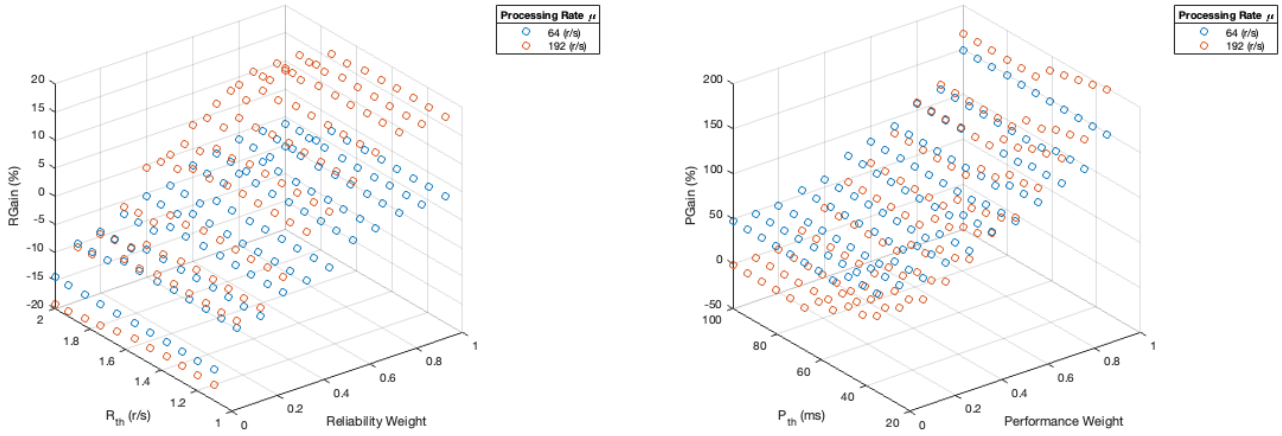
$$n_{serv} \in \{\ 3, 5, 10\ \} \tag{45}$$

$$n_{rout} \in \{\ 1, 3, n_{serv}\ \} \tag{46}$$

$$cf \in \{\ 10, 25, 50, 100\ \}\ r/s \tag{47}$$

Regarding the Cost Threshold, we take $C_{th} = 1\,cent/s$ as in our illustrative sample cases. For the processing rates, we investigate 9 levels as follows. In Section IV-A, we mentioned that a component with one vCPU has a processing rate of roughly $32\,r/s$ in our experiment. We start with components having two vCPUs up to six in increments of 0.5 vCPUs.

$$64 \leq \mu \leq 192\ r/s \tag{48}$$



(a) Reliability Gain

(b) Performance Gain

Fig. 6: Reliability and Performance Gains with Processing Rates of $\mu = 64$ and $192\ r/s$

Regarding reliability and performance thresholds, we start with tight reliability and loose performance thresholds so that more centralized routing is acceptable (lower value of n_{rout}). 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 (3) informs that a higher n_{serv} results in a higher expected request loss as the number of components increases. In our experiment, the highest number of services is ten. With $n_{serv} = 10$, the worst-case reliability for centralized routing and fully 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.

$$1.1 \leq R_{th} \leq 2.0\ r/s \tag{49}$$

$$35 \leq P_{th} \leq 100\ ms \tag{50}$$

2) Results Analysis: We evaluate 9801 systematic evaluation cases: 9 experiment cases, 9 processing rate levels, 11 importance weights, and 11 thresholds. To support reproducibility, the evaluation script and the evaluation log are provided in the online artifact of this study[2]. 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. These formulas are based on the Mean Absolute Percentage Error (MAPE) widely used in the cloud quality-of-service research [31].

$$RGain = \frac{100\%}{n} \cdot \sum_{c \in Cases} \frac{R_c - R_{n_{rout}}}{R_{n_{rout}}} \tag{51}$$

$$PGain = \frac{100\%}{n} \cdot \sum_{c \in Cases} \frac{P_c - P_{n_{rout}}}{P_{n_{rout}}} \tag{52}$$

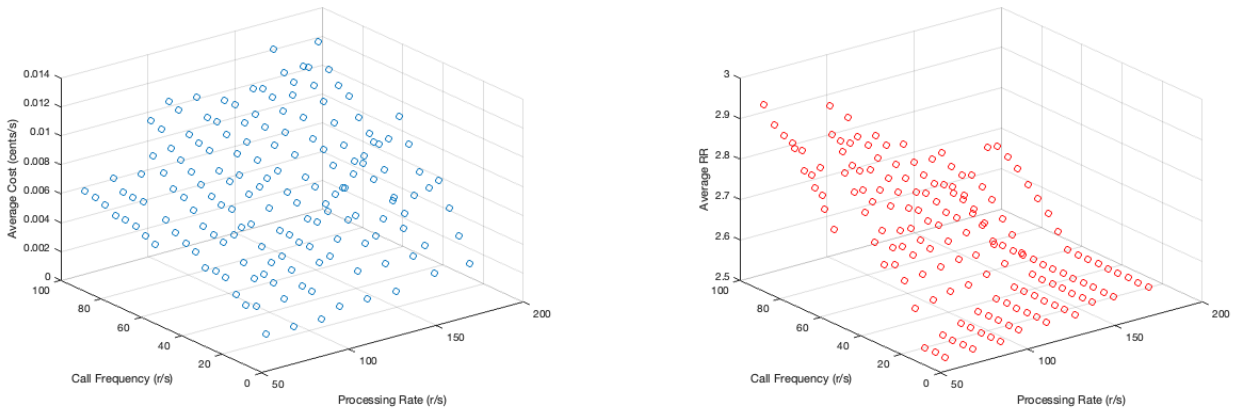
Remember $R_{n_{rout}}$ and $P_{n_{rout}}$ are reliability and performance predictions. The gains are averaged over 9 experiment cases.

Figure 6 shows the reliability and performance gains. Moreover, each figure shows the plots for our lowest studied processing rate of $\mu = 64\,r/s$ and the highest bound in our research, i.e., $\mu = 192\,r/s$. Regarding reliability, we can see in Figure 6a that with a higher reliability weight, we have an increase in reliability gain with $\mu = 192\,r/s$. Remember in Algorithm 2, we check that the components are not overloaded when choosing a more centralized routing to increase reliability. Having a higher processing rate results in a component processing higher call frequencies without being overloaded. However, as a result of choosing a less centralized routing, the gain in reliability is at most 16.60%.

Our approach provides significant improvements in performance gains. As more importance is given to the performance of a system, i.e., performance weight increases, our approach reconfigures an application by choosing more distributed routing. This reconfiguration results in a rise of a performance gain as shown by Figure 6b. On average, when cases with correct and incorrect architecture choices are analyzed together, our adaptive method provides 74.22% performance gain. A higher gain for performance compared to reliability is expected. To clarify, Equations (19) and (20) inform that changing the number of routers has a higher effect on the performance than a system's reliability. Our paper defines performance as the average processing time of requests per router. Having a higher number of routers to process the requests in parallel results in dividing the average processing time by more routers.

B. Autoscaling of Overloaded Components

This section systematically evaluates our approach regarding component overload prevention.



(a) Average Cost

(b) Average Reconfiguration Ratio

Fig. 7: Plots of Evaluation Data for the Autoscaling of Transient Components

1) Evaluation Cases: We go through the same range of values as in the previous scenario:

$$n_{serv} \in \{\ 3, 5, 10\ \} \tag{53}$$

$$n_{rout} \in \{\ 1, 3, n_{serv}\ \} \tag{54}$$

$$10 \leq cf \leq 100\ \ r/s \tag{55}$$

$$64 \leq \mu \leq 196\ \ r/s \tag{56}$$

$$C_{th} = 1\ cent/s \tag{57}$$

As before, we study increments of 0.5 vCPUs resulting in 9 μ levels. We consider the same range of call frequencies. However, since we are studying component overloads, we evaluate increments of 5 r/s, resulting in 19 cf levels.

2) Results Analysis: We evaluated 1539 cases for this scenario, i.e., three levels of n_{serv}, three levels of n_{rout}, 9 levels of μ, and 19 levels of cf. We define the average cost $\overline{C}$ and the average reconfiguration ratio $\overline{RR}$, that is, the amount of BFR improvements per cost spent as:

$$\overline{C} = \frac{1}{n} \cdot \sum_{c \in Cases} C(n_{scal}, n_{pro}) \tag{58}$$

$$\overline{RR} = \frac{1}{n} \cdot \sum_{c \in Cases} \frac{BFR(0,0) - BFR(n_{scal}, n_{pro})}{C(n_{scal}, n_{pro})} \tag{59}$$

$BFR(0,0)$ is the buffer fill rate without reconfiguration. We average over three levels of services and three levels of routers.

In Figure 7a, we can see that the reconfiguration costs increase as the processing rate and the call frequency increase. This is expected as reconfiguring a component with a higher processing rate is more expensive, especially when scaling out the overloaded component. Moreover, a higher incoming call frequency results in more overloaded components and, consequently, higher reconfiguration costs. However, as seen in Figure 7b, a higher cf results in a higher reconfiguration ratio. Our approach balances the costs with a bigger buffer fill rate improvement converging $\overline{RR}$ to an average of 2.62. The average reconfiguration cost over all cases is $0.0065\,cents/s$ bringing all overloading system components to a steady state.

VII. Threats to Validity

We discuss the four threat types by Wohlin et al. [32]. Regarding *construct validity*, we used request loss and the average processing time of requests per router as reliability and performance metrics, respectively. 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 [19]. Moreover, we studied reconfiguration measures of increasing the processing rate and scaling out a component to prevent system overload. While this is a common approach in service- and cloud-based research (see Section VIII), other measures might work better in terms of system overload prevention, for instance, changing the routing technology, e.g., using a circuit breaker [26]. More research with real-world systems is required to study.

Regarding *internal validity*, 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 due to 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. Moreover, we only considered constant load when modeling the stress of components using queuing theory. In reality, cloud-based systems are met with different load profiles, e.g., sudden load spikes. In future work, we plan to study more aspects of the proposed novel approach.

Concerning *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 evaluated our proposed approach with an extensive systematic evaluation using the data of our experiment of 1200 hours (see

Section VI). 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. Moreover, there must be a balance between the applicability of the proposed approach and the level of abstraction of the presented ideas, as in all research presenting models of a real-world phenomenon. To mitigate this thread, we performed many rounds of reviews and improvements in the author team and constantly compared them with the related work.

Concerning *conclusion validity*, as the statistical method to evaluate the accuracy of our prediction models, we used the Mean Absolute Percentage Error (MAPE) metric [31] metric. We defined reliability and performance gains, as well as the average percentage difference of buffer fill rate based on MAPE, as it is widely used and offers good interpretability in our research context.

VIII. Related Work

The proposed approach is related to *self-adaptive systems*, which typically use MAPE-K loops [4], and similar methods to realize adaptations. We extend such studies with support specific to the service- and cloud-based dynamic routing applications. Moreover, research on efficient resource provisioning, e.g., [10], [18], and cloud elasticity, e.g., [12], [13], are related to our work. Our study extends these approaches by considering the increase in the processing power of a component as a reconfiguration measure. Moreover, we consider a multifaceted reconfiguration of components taking into account system-wide and per-component optimizations.

Architecture-based reliability and performance prediction approaches [11], [31] employ (i) probabilistic analytical models such as discrete-time Markov chains (DTMCs) [9] and (layered) queueing networks (QNs) [30], or (ii) high-level architectural models such as profile-extended UML [23] or Palladio [7], [25] models, which are simulated or transformed into analytical models. Concerning *architecture-based performance prediction*, numerous approaches have been proposed. Spitznagel and Garlan [30] present a general architecture-based model for performance analysis based on queueing network theory. Probabilistic modeling is often applied, e.g. based on discrete-time Markov chains (DTMCs) [9] .

Architecture-based decision making [1], [28] uses architectural tactics to search for (Pareto) optimal architectural candidates. Architecture-based analysis approaches based on queueing theory have been studied by, e.g., [23], [30]. Like our study, those works focus on supporting architectural design and decision-making. In contrast to our work, they do not focus on specific kinds of architectures or architecture patterns in dynamic routing to prevent system overload. Our approach differs from these in focusing specifically on service- and cloud-based dynamic routing architectures.

We extend research on *auto-scalers for the cloud*, e.g., [5], [33], by adding specific cost studies. A particular related work is [20], which introduces a cost-aware component that can be added to auto-scalers. Lesch et al. use workload prediction to decide on reconfiguration costs. Our approach differs from this work in that they study the reconfiguration of cloud-based Virtual Machines (VMs). Therefore, they consider time-based cost functions to rent these VMs. Nevertheless, we study components and consider cost functions related to the number of resources an application uses. Moreover, we consider the transient analysis, i.e., when components are overloading and when they are in a steady state.

Multidimensional auto-scalers have been studied in the literature. AutoMAP [6] uses response time triggers to provision resources. AutoMAP finds optimal resources using Virtual Machine (VM) image sizes to support cost efficiency. Nguyen et al. [21] provide a forecasting model to predict CPU demand and use these predictions to start new machines before load peak to increase performance. CloudScale [29] supports scaling of CPU and memory resources when local scaling is possible. Otherwise, it migrates VMs to prevent overloaded hosts. Our work differs from all these studies because they consider auto-scaling at the VM level and configure the resources. We proposed a cost-aware multidimensional auto-scaler that works at the component level, adjusting the resources of each component.

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

IX. Conclusions

In this paper, we proposed a multifaceted reconfiguration approach that self-adapts between different routing patterns considering the component overloads and idleness (**RQ1**). Moreover, we provided a prototypical tool that provides visualizations to study different architecture configurations (**RQ2**). We systematically evaluated our approach based on our empirical data (see Section II for details). Our extensive systematic evaluation shows significant improvements in quality trade-off adaptations and system overload prevention. In our experiment cases, our approach can yield up to 16.60% reliability gain. On average, where cases with the right and the wrong architecture choices are analyzed together, our approach offers a 74.22% performance gain (**RQ3**). Our architecture adapts, based on triggers, e.g., change of incoming load frequency or degradation of monitoring data, to an optimal configuration to adapt quality trade-offs and prevent system overload. Before our work, architects needed to redesign and redeploy architecture configurations manually. For our future work, we plan to apply our novel approach to real-world applications and evaluate the results. Moreover, we plan to cover different load profiles when autoscaling components multidimensionally.

References

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

[2] A. Amiri, 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.

[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. Formal design and verification of self-adaptive systems with decentralized control. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 11(4):1–35, 2017.

[5] 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.

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

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu. Efficiency analysis of provisioning microservices. In *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*, pages 261–268. IEEE, 2016.

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

[20] V. Lesch, A. Bauer, N. Herbst, and S. Kounev. Fox: Cost-awareness for autonomic resource management in public clouds. In *Publication: Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2019.

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

[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] P. Raj, A. Raman, and H. Subramanian. *Architectural Patterns: Uncover essential patterns in the most indispensable realm*. Packt Publishing, December 2017.

[25] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann. *Modeling and Simulating Software Architectures: The Palladio Approach*. The MIT Press, 2016.

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

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

[30] 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.

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

[32] C. Wohlin, P. Runeson, M. Hoest, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering*. Springer, 2012.

[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.