This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3098178

Modeling and Empirical Validation of Reliability and Performance Trade-Offs of Dynamic Routing in Service- and Cloud-Based Architectures

Amirali Amiri, Uwe Zdun and André van Hoorn

Abstract—*Context:* Various patterns of dynamic routing architectures are used in service- and cloud-based environments, including sidecar-based routing, routing through a central entity such as an event store, or architectures with multiple dynamic routers.
Objective: Choosing the wrong architecture may severely impact the reliability or performance of a software system. This article's objective is to provide models and empirical evidence to precisely estimate the reliability and performance impacts.
Method: We propose an analytical model of request loss for reliability modeling. We studied the accuracy of this model's predictions empirically and calculated the error rate in 200 experiment runs, during which we measured the round-trip time performance and created a performance model based on multiple regression analysis. Finally, we systematically analyzed the reliability and performance impacts and trade-offs.
Results and Conclusions: The comparison of the empirical data to the reliability model's predictions shows a low enough and converging error rate for using the model during system architecting. The predictions of the performance model show that distributed approaches for dynamic data routing have a better performance compared to centralized solutions. Our results provide important new insights on dynamic routing architecture decisions to precisely estimate the trade-off between system reliability and performance.

Index Terms—Cloud-Based Computing, Service-Based Applications, Dynamic Routing Architectures, System Reliability, Performance

1 Introduction

Many distributed system architecture patterns [8], [18], [33] have been suggested for dynamic routing [16], i.e., routing or blocking the incoming requests to different services based on a set of rules. Some dynamic routing architectures require a single request routing decision, e.g., when using load balancing. More complex request routing decisions, such as routing to the right branch of a company or checking for compliance to privacy regulations, often require multiple runtime checks during one sequence of requests. Consider the following example. A request might first be checked for the company branch in which it needs to be processed, then at the next cloud service, whether it contains privacy-sensitive data. Next, possible data centers in which private data can be stored are considered, then the request is routed to the appropriate services, and finally it is load balanced on the responsible cloud services. Such request flow paths are typically not pre-configured and rules for request routing can change dynamically.

Another scenario where the dynamic routing is of importance is the following. Assume a company offers services to customers based on their subscription type. Some customers may have access rights to a selected group of services; dynamic routers can route or block requests based on customer permissions. For another example in a different context, assume in a company with sensitive data of customers, a sudden system reliability degradation is monitored. The architect must, based on their experience, statically redesign and redeploy multiple dynamic routers to meet the quality criteria required for the application. Our work provides a systematic evaluation of different scenarios, in architectural level of abstraction, so that the decision can be made informedly based on reliability and performance trade-offs.

In our prior work [2], [3], we have studied representative service- and cloud-based system architecture patterns for dynamic request routing. A typical cloud-native architecture pattern is the *Sidecar* pattern [18], [24] in which the sidecar of each service handles incoming and outgoing traffic [11]. Thus, it can perform the request flow routing for that service in relation to its directly linked services. In contrast, other architectural patterns use some kind of *Central Entity* for processing the request routing decisions. For instance, an API Gateway [33], an event store or an event streaming platform [33], or any kind of central service bus [8], can be used to realize a central entity. In addition to these two extremes, multiple *Dynamic Routers* can be used in specific places of the request flow, e.g., consider an API Gateway, two event streaming platforms, and a number of sidecars, making routing decisions in one cloud-based architecture.

At present, the impacts of such architecture patterns and their different configurations on system reliability and performance have only been studied preliminarily in our own prior works [2], [3] – on which this study is based (we detail the new contributions of this study compared to our prior works at the end of this section). This makes it hard to consider reliability and performance as trade-offs in the architectural design decision for more or less centralized dynamic request routing. Both reliability and performance are core considerations in service and cloud architectures [26]. A reasonably accurate failure prediction

- *Amirali Amiri and Uwe Zdun are with the University of Vienna, Austria E-mail: firstname.lastname@univie.ac.at*
- *André van Hoorn is with the University of Stuttgart, Germany E-mail: van.hoorn@informatik.uni-stuttgart.de*

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3098178

for the feasible architecture design options in a certain design situation, as well as on the impacts of any such decision option on performance, would help architects to better design system architectures considering those quality trade-offs. Note that so far there is no study that considers the possible interdependencies of reliability and performance in our particular research context. For instance, the best design option with regard to performance might be significantly different in a routing architecture where requests might fail (e.g., because of node crashes) compared to a system where no substantial reliability issues have to be considered. We set out to answer the research questions:

RQ1: *What is the impact of choosing a dynamic routing architecture, in particular central entity, sidecar-based, or dynamic routers, on system reliability?*

RQ2: *What is the performance impact of these representative architectures for dynamic data routing considering potential reliability issues?*

RQ3: *How can we predict the impact of system reliability and performance when making architectural design decisions on dynamic routing architectures?*

RQ4: *Can we find an optimal or semi-optimal trade-off between architectures in terms of performance and reliability for a given system configuration and load?*

To address these research questions, we first modeled request loss during router and service crashes in an analytical model based on Bernoulli processes. Request loss is used as the externally visible metric indicating the severity of the crashes' impacts. The model abstracts central entities, dynamic routers, and sidecars in a common router abstraction. This makes it possible to predict request loss during router and service crashes for any configuration of a request flow sequence in service- and cloud-based system architectures.

To validate our analytical model of system reliability, we designed an experiment in which we studied 36 representative experimental cases (i.e., different experiment configurations). These cases covered three kinds of architectures with different numbers of cloud services, routers, and request call frequencies. We then computed the prediction error of our reliability model compared to our empirical results. Our results show that the error is constantly reduced with a higher number of experiment runs, converging at a prediction error of 7.8%. Overall, we performed 200 experiment runs, which ran a total of 1200 hours (excluding setup time). Given the target prediction accuracy of up to 30% commonly used in the cloud performance field [23], also considering hard to control effects like network latency, and bearing in mind that the goal of our study is architecting with a rough prediction of the impact on system reliability, these results are more than reasonable. With the same crash probability for all components, the same frequency of incoming requests, and the same number of cloud components, our model predicts and our experiment confirms that more decentralized routing results in losing a higher number of requests in comparison to more centralized approaches.

To analyze the performance in a potentially unreliable system, we measured round-trip time performance during our experiment runs. We then statistically analyzed this data using multiple regression analysis [34], to predict the performance of the representative dynamic routing architectures in terms of the time it takes for a request to be fully processed. Next, we compared the results of our prediction models with another run of our experiment to calculate the prediction accuracy of our performance models, which goes as low as 9.0% in case of the sidecar and the dynamic routers architecture patterns. The results show that distributed approaches for dynamic data routing have generally a better performance compared to centralized solutions in most cases, especially for a high number of services. In small corridors of (i.e., a low number of) services and high load combinations, it is necessary to inspect in detail which architecture performs better (analyzed in Section 8).

The contributions of our study are as follows. As mentioned above, this research is based on two prior studies. In one we studied performance in a small-scale experimental setting [2] where we instantiated our infrastructure and stressed the services under different load profiles. Here, we extended this work by completely repeating the above process in a much larger-scale experimental setting going from one experiment run each case taking 5 seconds to 200 runs each case taking 10 minutes. We provided a completely new set of statistical prediction models for performance, presented extensive performance results and analyzed the prediction error, which, to the best of our knowledge, has not been done before specifically concerning dynamic routing architectures in service- and cloud-based environments.

In [3], we extended our small-scale experimental configuration to also study reliability in a larger setting. Here, we present an extension of this work with substantially more detailed analyses on the reliability properties. We introduce a metamodel specifically designed to consider reliability and performance trade-offs in service- and cloud-based dynamic routing, which has not been presented before to the best of our knowledge. Finally, in this article, we present a new detailed analysis of reliability and performance trade-offs of the three architecture patterns based on our models.

The article first compares our study to the related work in Section 2. In Section 3, we explain the three considered service- and cloud-based architecture patterns. Section 4 presents a metamodel and our analytical model of system reliability. Next, in Section 5, we describe the empirical validation of our study. Section 6 presents our statistical model of performance. We then study the trade-off in terms of system reliability and performance in Section 7, discuss the threats to validity in Section 8, and conclude in Section 9.

2 Related Work

Architecture-based Reliability Prediction To predict the reliability of a system and to identify reliability-critical elements of its system architecture, various approaches such as fault tree analysis or methods based on a continuous-time Markov chain have been proposed [40]. Architecture-based approaches, like ours, are often based on the observation that the reliability of a system does not only depend on the reliability of each component but also on the probabilistic distribution of the utilization of its components, e.g., formulated as a Markov model [9], [21]. Other approaches allow software engineers to systematically improve the reliability of the software architecture, e.g., Brosch et al. [7] suggest an

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3098178

extension of the Palladio Component Model along with automated transformations into a discrete-time Markov chain. Pitakrat et al. [29] use architectural knowledge to predict how a failure can propagate to other components. They use Bayesian networks to represent conditional dependencies and infer probabilities of failures and their propagation. Our approach differs from these approaches in that it focuses specifically on cloud- and service-based dynamic routing architecture patterns. By focusing on these specific patterns, we can define a more precise model and reach a high level of prediction accuracy at the expense of generality that is higher in those other architecture-based approaches.

Empirical Reliability or Resilience Assessment Experiment-based resilience assessment approaches aim to assess a software system's ability to cope with failures, e.g., by injecting faults and observing their effects. Today many software organizations use large-scale experimentation in production systems to assess the reliability of their systems, which is called chaos or resilience engineering [5]. A crucial aspect in resilience assessment of software systems is efficiency [25]. To reduce the number of experiments needed, knowledge about the relationship of resilience patterns, anti-patterns, suitable fault injections, and the system's architecture can be exploited to generate experiments [41]. Our approach differs from these techniques in that our analytical model can be employed to predict the reliability of a software system, whereas key design decisions, i.e., routers in service- and cloud-based systems, are not only modeled analytically but also assessed empirically.

Service-Specific Reliability Studies Our approach, in contrast to many existing architecture-based reliability prediction methods, is focused on a specific category of architectures, namely services-based architectures for dynamic routing. From a practical point of view, reliability in those kinds of architectures has been studied in service and cloud architectures leading to observations of patterns and best practices [26]. Some works introduce service-specific reliability models. For instance, Wang et al. [43] propose a discrete-time Markov chain model for analyzing system reliability based on constituent services. Grassi and Patella [12] propose an approach for reliability prediction that considers the decentralized and autonomous nature of services. Zheng and Lyu [44] propose an approach that employs past failure data to predict a service's reliability. However, none of these approaches studies and compares major architecture patterns in service and cloud architectures; they are rather based on a very generic model with regard to the notion of service. So far none of the approaches considers reliability and performance trade-offs together.

Architecture-Based Performance Analysis and Prediction A number of approaches perform architecture model-based performance analysis or prediction. Spitznagel and Garlan [38] present a general architecture-based model for performance analysis based on queueing network theory. Sharma and Trivedi [35] 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 both performance and reliability aspects together. Petriu et al. [28] present an architecture-based performance analysis approach that builds Layered Queueing Network performance models from a UML description of the high-level architecture of a system. The Palladio component model [6], [32] allows precise component modeling with relevant factors for performance properties and contains a simulation framework for performance prediction. Like our work, those works focus on supporting architectural design or decision making. In contrast to our work, they do not focus on specific kinds of architectures or architecture patterns; those models offer more generality at the expense of the high accuracy with which we characterize the three architecture patterns analyzed in our work.

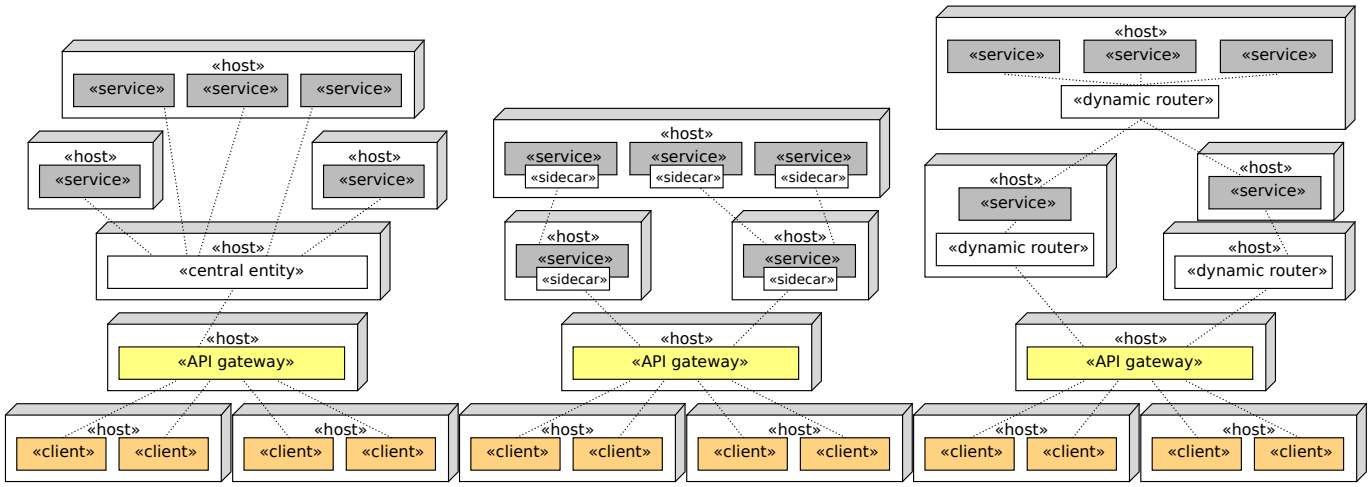
Performance Analysis: Internet of Things Vandikas et al. [42] conducted a performance analysis of their Internet of Things (IoT) framework to evaluate its behavior under heavy load produced by different amounts of producers and consumers. The main purpose of the framework is to allow producers, such as sensors, to publish data streams to which multiple interested consumers, e.g., external applications, can subscribe. This publish-subscribe functionality is realized by a central message broker implemented with RabbitMQ. In contrast to our work, dynamic data routing is not considered in this article; moreover, the performance evaluation of the framework focuses only on a single machine deployment, which may have led to results that are not easily generalizable to cloud-based deployments.

Performance Analysis: Enterprise Service Buses There are a number of existing works comparing the performance of Enterprise Service Buses (ESB). This is related to our work in the sense that ESBs provide a means for content-based routing of messages. In our experiment no ESB was used to implement the rule-based dynamic data routing, but the central entity approach is similar from a structural point of view. Sanjay et al. [1] evaluate the performance of the three open-source ESBs Mule, WSO2 ESB, and Service Mix. The performance is measured based on mean response time and throughput for proxying, content-based routing, and mediation of data. However, the test scenarios only consider communications between clients and a single web service. In contrast, our work also considers communication paths which involve the composition of multiple services and routing decisions. Shezi et al. [36] provide a performance evaluation of different ESBs in a more complex scenario in which multiple services are composed to achieve a certain business objective. As a test case, a service orchestration scenario is simulated, in which a consumer consults a number of banking services to find the best loan quote. In contrast to our work, other routing architectures are not considered.

Performance Analysis: Microservice- and Container-Based Systems Different studies evaluate the network performance of container-based applications. This is related to our work, as we analyzed the performance of containerized services. For example, Kratzke [20] evaluates the performance impact of Docker containers, software-defined networks, and encryption to network performance in distributed cloud-based systems using HTTP-based communication. The performance is measured by means of data transfer rate of m byte-long messages. A similar work is presented by Bankston et. al [4] to explore the network performance and system impact of different container networks on public clouds from Amazon Web Services, Microsoft Azure, and Google Cloud Platform.

Another kind of related studies in a wider sense com-

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3098178



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

Fig. 1: Dynamic Routing Architecture Patterns

pares different service architectures. For instance, Lloyd et al. [22] compare different states of serverless infrastructure and their influence on microservice performance. Khazaei et al. [19] study the efficiency of provisioning microservices. All these studies are related to our research as they also improve the state of (micro)service performance engineering. Our study contributes new data on three common architectures for evaluating dynamic routing rules, which has not been examined before. The literature has produced general microservice performance engineering challenges and directions (e.g., [14]). Studies like ours and the ones mentioned above address some of the microservice performance engineering challenges identified in the literature. As outlined above, our experimental setup is influenced by the named related works, broader studies on related experimental setups (e.g., [10], [15], [39]), and our own experiences in building microservice and cloud systems (see [2], [3]).

3 Background: Dynamic Routing Architecture Patterns

There are many different service- and cloud-based architectures which use or enable dynamic request routing. We study three of the widely used architecture patterns.

Central Entity (CE) Architecture In a CE architecture, as shown in Figure 1a, the central entity manages all request flow decisions. One benefit of this architecture is that it is easy to manage, understand, and change as all control logic regarding request flows is implemented in one component; however, this introduces the drawback that the design of the internals of the central entity component is a complex task. Another advantage is that in an application, which consists of stateful request flow sequences, the state does not need to be passed between various distributed components. Nonetheless, as shown in Figure 1a, services need to call back to the central entity component to fetch the saved state of prior stages in order to proceed with the next step in the request flow sequence. CE can be implemented utilizing an API Gateway [33], an event store or an event streaming platform [33], or any kind of central service bus [8]. Figure 1a shows an example configuration of CE. Note that it is not required that CE is always deployed on an exclusive host.

Sidecar-based Architecture (SA) SA is presented in Figure 1b. In contrast to the central entity architecture, the control logic is distributed and placed in so-called sidecars [11], [18], which are attached to the services. Sidecars offer a separation of concerns since the control logic regarding request flow is implemented in a different component than the service; however, they are tightly coupled with their directly linked services. Sidecars offer benefits whenever decisions need to be made structurally close to the service logic. One advantage of this architecture is that, in comparison to the central entity service, it is usually easier to implement sidecars since they require less complex logic to control the request flow of their connected services. However, it is not always possible to add sidecars, e.g., when services are off-the-shelf products. Sidecars are always implemented on the same host as their directly linked services.

Dynamic Routers (DR) Architecture Figure 1c shows a specific dynamic router [16] configuration. DR can be seen as a hybrid of the two aforementioned extremes, i.e., in between the centralized CE and the fully distributed SA. One benefit of using DR is that dynamic routers can use local information regarding request routing amongst their connected services. For instance, if a set of services are dependent on one another as steps of processing a request, DR can be used to facilitate dynamic routing. Nonetheless, dynamic routers introduce an implementation overhead regarding data structures, control logic, management, deployment, and so on since they are usually distributed on multiple hosts. We use the common term router for all request flow control logic components, i.e., the central entity in CE, sidecars in SA, and dynamic routers in DR.

4 Model of Request Loss During Router and Service Crashes

In this section we first explain central concepts of our work with a metamodel, then propose a Bernoulli process model of request loss during router and service crashes. Table 1 presents the mathematical notations used in this article.

TABLE 1: The Mathematical Notations Used in this Article

Notation	Description
T	Observed system time
n_{serv}	Number of services
n_{crash}	Number of crash tests
CI	Crash interval
cf	Incoming call frequency
A	Allocation of routers
C	Set of all components
R	Set of all routers
S	Set of all services
$r_{crashed}$	A router r when crashed
$s_{crashed}$	A service s when crashed
IR	Internal request
IR_T	Total number of internal requests per a call sequence
IL_T	Total internal loss
IL_R	Sum of the internal loss per crash of each router
IL_S	Sum of the internal loss per crash of each service
IL_c	Internal loss for a component c
IL_r	Internal loss for a router r
IL_s	Internal loss for a service s
ER	External request
EL_T	Total external loss
EL_R	Sum of the external loss per crash of each router
EL_S	Sum of the external loss per crash of each service
EL_c	External loss for a component c
EL_r	External loss for a router r
EL_s	External loss for a service s
C_T	Total number of crashes
C_R	Sum of the expected number of crashes of each router
C_S	Sum of the expected number of crashes of each service
n_c^{exec}	Number of executed internal requests for the crash of a component c
n_r^{exec}	Number of executed internal requests for the crash of a router r
n_s^{exec}	Number of executed internal requests for the crash of a service s
d_c	Expected average downtime after a component c crashes
d_r	Expected average downtime after a router r crashes
d_s	Expected average downtime after a service s crashes
P_c	Crash probability of a component c every CI
P_r	Crash probability of a router r every CI
P_s	Crash probability of a service s every CI
$E[C_c]$	Expected number of crashes of a component c during T
$E[C_r]$	Expected number of crashes of a router r during T
$E[C_s]$	Expected number of crashes of a service s during T

4.1 Metamodel

As depicted in Figure 3a, we consider various kinds of *Components* in service-based architectures: *Services*, *Clients*, *API Gateways*, and *Routers*. *Host* is an abstraction of any execution environments for these components, either physical or virtual. *Request* models the request flow, linking a source and a destination component. *External Request* is an abstraction of a request flow between a *Client* and an *API Gateway*. *Internal Request* models a request flow amongst *API Gateway*, *Router*, and *Service* components. Figure 3b extends the metamodel with specific concepts for modeling request loss. The *Profile* and *Crash* classes contain member variables which are explained below in our model.

4.2 Definition of Internal and External Loss

To illustrate our model, let us use the basic concepts of our metamodel to instantiate an example model. Figure 2 shows a configuration of a DR architecture with three routers and five services. The instantiated components send internal requests, labeled from IR_1 to IR_{11}, amongst one another to complete the processing of the one external request, labeled ER. The partially ordered set representing the call trace ER, $IR_1, \ldots, IR_{11}$ is called the call sequence. When a router or a service crashes before it has processed a pending request, external requests will not be processed fully, which results in the application not being responsive to the client. We define external loss as the number of external requests that are not processed during a crash of a component, and internal loss as the number of lost internal requests.

Internal Loss In case of a crash, per each external loss, the internal loss is the total number of internal requests per a call sequence, i.e., IR_T, minus the ones that have been successfully executed. Let IL_c, EL_c and n_c^{exec} be the internal loss, the external loss and the number of executed internal requests for the crash of a component c:

$$IL_c = EL_c \cdot (IR_T - n_c^{exec}) \tag{1}$$

Example Crash Scenario for a Router Crash To illustrate the internal loss metric, let us consider the crash of R_3 ($c = R_3$) in Figure 2. In this case, IR_1 to IR_8 are executed, i.e., $n_c^{exec} = 8$, but we lose three internal requests, namely IR_9, IR_{10} and IR_{11}. We can see that there are a total of eleven internal requests, i.e., $IR_T = 11$, then:

$$IL_c = EL_c \cdot (11 - 8) = EL_c \cdot 3 \tag{2}$$

which means per each external loss, we lose three internal requests. Note that IR_T and n_c^{exec} need to be parameterized based on the application. An example of this parameterization is given in Section 5.1 "Specific Model Formulae".

Example Crash Scenario for a Service Crash Let us consider the crash of S_5 ($c = S_5$) in Figure 2. In this case, IR_1 to IR_9 are executed, i.e., $n_c^{exec} = 9$, and we lose two internal requests, i.e., IR_{10} and IR_{11}, then:

$$IL_c = EL_c \cdot (11 - 9) = EL_c \cdot 2 \tag{3}$$

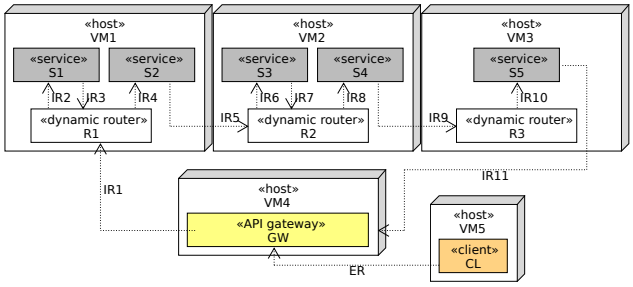


Fig. 2: Example Model Instance

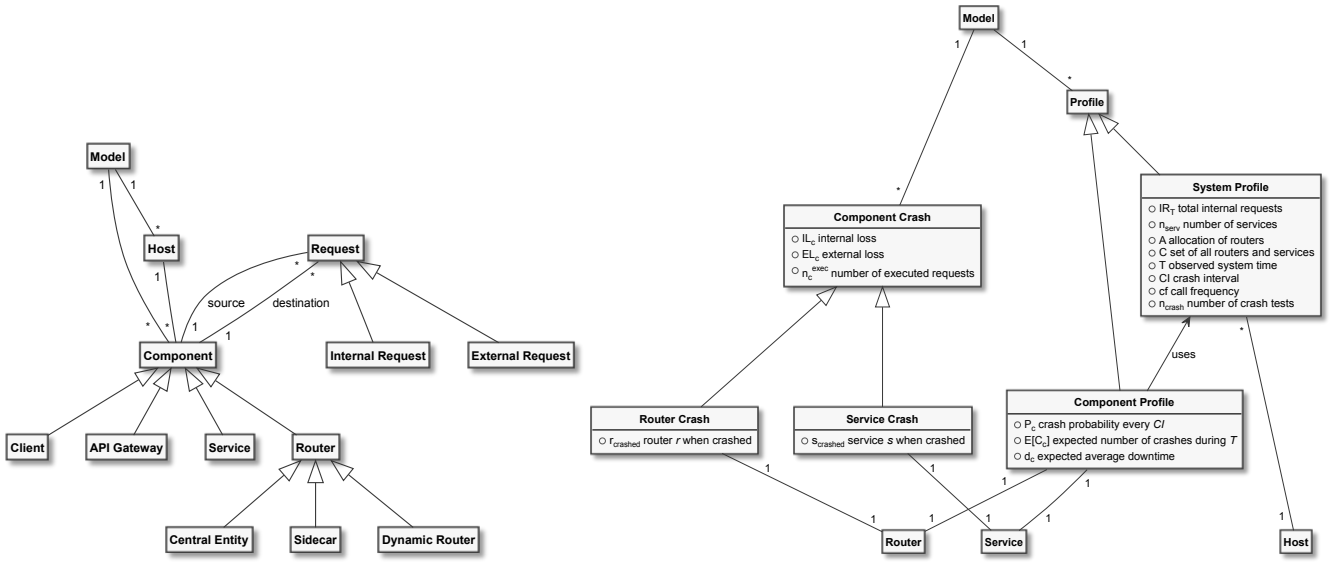
External Loss Let d_c be the expected average downtime after a component c crashes and cf the incoming call frequency, i.e., the frequency at which external requests are received. The external loss per crash of each component c is:

$$EL_c = d_c \cdot cf \tag{4}$$

4.3 Bernoulli Process to Model Request Loss During Router and Service Crashes

In this section we model request loss based on Bernoulli processes, which is a set of independent Bernoulli trials [40]. A Bernoulli trial is a random experiment with two outcomes, i.e., "success" and "failure". This fits perfectly to our modeling of the crash of a component based on a random variable. At certain intervals, we generate a random variable for each component. If this number is above the

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3098178



(a) Basic Concepts (b) Specific Concepts for Modeling Request Loss

Fig. 3: Metamodel

crash probability of the component, we manually crash the component by stopping its Docker container.

We only model the crash of the *Router* and *Service* subconcepts of the *Component* in our metamodel. This is because we assume an *API Gateway* is stable and reliable. Moreover, a crash of a *Client* results in *Requests* not being generated; as a result, *Requests* are not lost. Hence, throughout the rest of the paper, we use the common term components for all instantiated routers and services.

Number of Crash Tests During T, i.e., the observed system time, all components can crash with certain failure distributions. It is realistic to assume that these distributions are known with a certain error, as they can be estimated from the past system runs, e.g., recorded in system logs. Note that many cloud systems run without being stopped: here, T should be interpreted as the time interval in which these failure distributions are observed (e.g., failure distributions of a day or a week). A crash of each component can happen at any point of time in T. We model this behavior by checking for a crash of any of the system's components every crash interval CI. That is, our model "knows" about crashes in discrete time intervals only, as it would be the case, e.g., if the Heartbeat pattern [17] or the Health Check API pattern [31] is used for checking system health. Our model allows any possible values for T or CI and different crash probabilities for each component, e.g., based on empirical observations in a system under consideration. Let n_{crash} be the number of times we check for a crash of components, i.e., the number of crash tests:

$$n_{crash} = \lfloor \frac{T}{CI} \rfloor \tag{5}$$

Expected Number of Crashes Each crash test is a Bernoulli trial in which success is defined as "component crashed" and failure as "component did not crash". Assuming $CI > d_c$, all n_{crash} crash tests of a component c are independent from each other. This assumption is justifiable since in reality, when a component crashes and is down, it cannot crash again; another crash of the same component can happen only after the component is up and running, i.e., the component's downtime has passed. Therefore, for each component, we can create a Bernoulli process of its crash tests. Then the binomial distribution of each Bernoulli process gives us the number of successes, i.e., the number of times a component crashes during T. For each component, the expected value of the binomial distribution is the expected number of crashes of the component. Let P_c be the crash probability of a component c every time we check for a crash which is derived, dependent on the application, from the failure distributions. Note that P_c is different for each component which is specified by the *Crash Profile* concept in our metamodel (see Figure 3b). Let $E[C_c]$ be the expected number of crashes of a component c during T:

$$E[C_c] = n_{crash} \cdot P_c \tag{6}$$

Total Internal and External Loss The total internal loss, i.e., IL_T, is the sum of internal loss per crash of each component. Let C be the set of all components that can crash, i.e., routers and services.

$$IL_T = \sum_{c \in C} E[C_c] \cdot IL_c \tag{7}$$

which can be rewritten using Equations (1) and (4) to (6) as:

$$IL_T = \lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in C} P_c \cdot d_c \cdot (IR_T - n_c^{exec}) \tag{8}$$

The total external loss, i.e., EL_T, is the sum of external loss per crash of each component.

$$EL_T = \sum_{c \in C} E[C_c] \cdot EL_c \tag{9}$$

which can be rewritten using Equations (4) to (6) as:

$$EL_T = \lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in C} P_c \cdot d_c \tag{10}$$

Total Number of Crashes C_T is the sum of the expected number of crashes of each component.

$$C_T = \sum_{c \in C} E[C_c] \tag{11}$$

which we can rewrite based on Equations (5) and (6) as:

$$C_T = \lfloor \frac{T}{CI} \rfloor \cdot \sum_{c \in C} P_c \tag{12}$$

5 Empirical Validation of the Request Loss Model

In this section we describe an experiment which we designed to empirically validate the accuracy of our request loss model model. Moreover, we provide application-specific model formulae regarding our experimental setup. Finally, we present our results.

5.1 Experimental Planning

Goals We aim to empirically validate our model's accuracy with regard to the number of crashes as well as the total internal and external loss represented by Equations (7) to (10). Based on our experiences from studies of microservice-based architectures and the related literature in our prior work (see [2], [3]), we decided on a number of experimental cases which are explained below. Then we realized these architectures using a prototypical implementation, instantiated and ran them in a cloud infrastructure, measured the empirical results, and compared the results with our model. The experimental setup is based on our prior work [2], [3].

Technical Details We used a private cloud with three physical nodes, each having two identical CPUs. Two cloud nodes hosted the Intel® Xeon® E5-2680 v4 @ 2.40 GHz[1] and the other one hosted the same processor family but version v3 @ 2.50 GHz. The v4 and v3 versions had 14 and 12 cores respectively and two physical threads per core (56 and 48 threads in total). On top of the cloud nodes, we installed Virtual Machines (VMs), each of which used the VMware[2] ESXi version 6.7.0 u2 hypervisor, had eight CPU cores, 60 GB system memory, and ran Ubuntu Server 18.04.01 LTS[3]. Docker[4] containerization was used to run the services which were implemented in Node.js[5]. We utilized five desktop computers to generate load, each hosting an Intel®Core™i3-2120T CPU @ 2.60 GHz with two cores and two physical threads per core. All desktop computers had 8 GB of system memory and ran Ubuntu 18.10. They generated load using Apache JMeter[6] which sent Hypertext Transfer Protocol (HTTP) version 1.1[7] requests to the cloud nodes.

Architecture Configurations Any application that has a request flow can be modeled using our proposed meta-model. We used a few sample architecture configurations to calculate the accuracy of our model discussed later. These configurations followed the convention for the request flow shown by the example model in Figure 2. That is, all clients send external requests (ERs), to the API gateway, then per each ER, internal requests (IRs) were sent one-by-one from routers to services and vice versa. Also for the sake of simplicity, we labeled the services and the routers incrementally from 1, and made the IRs go through all of them linearly.

As in the example model, we utilized one virtual machine exclusively, with only one Docker container inside of it, to run the API gateway. Then we distributed the services, each on a separate container, among three VMs. The distribution of the services was so that all virtual machines had the same number of services (with a maximum difference of one service). However, the placement of routers on hosts were different from that of the example model. For CE, we placed the central entity service in a Docker container exclusively on one VM. For DR, we used three dynamic routers which followed the same convention as CE, i.e., three separate exclusive VMs each with only one container running the routers. Finally, for SA, we placed each sidecar in a separate container on the same VM, on which its directly linked service resides.

Experimental Cases According to Equation (8), the total internal loss (IL_T) is influenced by a number of factors: the incoming call frequency (cf), number of services (n_{serv}), downtime of components (d_c), system run time (T), crash interval (CI) and crash probability of components (P_c).

We chose different levels for cf and n_{serv} to study their effects on IL_T. We selected cf based on a study of related works as 10, 25, 50, and 100 Hr/s. In many related studies (see, e.g., [10], [39]), 100 Hr/s (or even lower numbers) are chosen; as a result, we chose this number as our highest bound and selected different portions of it to study its effects. As for n_s, based on our experience and a survey on existing cloud applications in the literature and industry [2], [3], the number of services which are directly dependent on each other in a call sequence is usually rather low; we chose 3, 5, and 10 as values for the number of services (n_{serv}).

We simulated a node crash by separately generating a random number for each cloud component, i.e., routers and services. If the generated random number for a component was below its crash probability, we stopped the component's Docker container and started it again after a time interval $d = 3$ seconds. We chose $T = 10$ minutes, during which we checked for a crash for all components simultaneously every $CI = 15$ seconds resulting in $n_{crash} = 40$ based on Equation (5). Each component had a uniform crash probability of 0.5% each time we checked for a crash as mathematically expressed by Equation (13). This crash probability is much higher than what is observed for real-life cloud applications: akin to the related works we chose a relatively high crash probability in order to have a high enough likelihood to observe a few crashes during T.

$$P_c = 0.5\% \quad \forall \ c \in C \tag{13}$$

Note that it is a common assumption to have a uniform crash probability in experiments like ours (see, e.g., [29], [30]) to increase the control over the experiment's dependent variables. However, in real-world applications, different components may have different failure rates which need to be considered. Our model in the general form considers any failure profile for components (see Equation (8)).

[1] https://www.intel.com/content/www/us/en/homepage.html
[2] https://docs.vmware.com
[3] https://www.ubuntu.com
[4] https://www.docker.com
[5] https://nodejs.org/en/
[6] https://jmeter.apache.org
[7] https://tools.ietf.org/html/rfc7230

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3098178

Specific Model Formulae As explained before, in Equation (1), IR_T and n_c^{exec} need to be parameterized based on the application. Since in our example configurations each service receives an IR, processes it and sends it back either to a router or the API gateway, we can calculate IR_T based on the number of services as:

$$IR_T = 2n_{serv} + 1 \tag{14}$$

We use two different concepts for the crash of a router and a service in the metamodel (see Figure 3b) because the number of executed requests (n_c^{exec}) is different in each case. With a service crash, all internal requests (IRs) up until the last router will be executed. Let $s_{crashed}$ be the label number of the crashed service, for our architecture configurations:

$$n_s^{exec} = 2s_{crashed} - 1 \tag{15}$$

Using Equation (8), the internal loss for all services (IL_S) is:

$$IL_S = 0.6 \cdot cf \cdot n_{serv}(n_{serv} + 1) \tag{16}$$

In case of a router crash, to calculate the number of executed requests, we need to know the allocation of routers (A) which is a set indicating the number of directly linked services of each router, e.g., the allocation of routers in the example model presented in Figure 2 is:

$$A = \{2, 2, 1\} \quad and \quad A_0 = 0 \tag{17}$$

which means there are two services allocated to router $R1$ and $R2$ and one service allocated to router $R3$. Let $r_{crashed}$ be the label number of the crashed router, then for our architecture configurations we have:

$$n_r^{exec} = 2 \sum_{r=1}^{r_{crashed}} A_{r-1} \tag{18}$$

which means, to find the number of executed requests before the crash of router r, we sum over the allocated services of all routers up until the crashed router and multiply it by two since there are an incoming and an outgoing request from a service to a router (see Figure 2).

In case of CE in our experiment, all n_{serv} services are connected to the one router, i.e., the central entity service:

$$A = \{n_{serv}\} \quad and \quad A_0 = 0 \tag{19}$$

Then we can rewrite Equation (8) for all routers (IL_R) as:

$$IL_R = 0.6 \cdot cf \cdot (2n_{serv} + 1) \tag{20}$$

We can calculate IL_T for CE using Equations (16) and (20):

$$IL_T = IL_R + IL_S \tag{21}$$

$$IL_T = 0.6 \cdot cf \cdot [(n_{serv})^2 + 3n_{serv} + 1] \tag{22}$$

In case of DR in our experiment, all n_{serv} services are equally distributed (with a maximum difference of one service) on the three dynamic routers:

$$A = \{\frac{n_{serv}}{3}, \frac{n_{serv}}{3}, \frac{n_{serv}}{3} \pm 1\} \quad and \quad A_0 = 0 \tag{23}$$

Then for DR using Equations (16) and (21) we have:

$$IL_R = 0.6 \cdot cf \cdot (4n_{serv} + 3) \tag{24}$$

$$IL_T = 0.6 \cdot cf \cdot [(n_{serv})^2 + 5n_{serv} + 3] \tag{25}$$

In case of SA in our experiment, each service is connected to one router, i.e., a sidecar. Therefore:

$$A = \{1, 1, ..., 1\} \quad and \quad A_0 = 0 \tag{26}$$

in which A has the length of n_{serv}. Then for SA we have:

$$IL_R = 0.6 \cdot cf \cdot ((n_{serv})^2 + 2n_{serv}) \tag{27}$$

$$IL_T = 0.6 \cdot cf \cdot [2(n_{serv})^2 + 3n_{serv}] \tag{28}$$

Data Set Preparation For each experimental case we instantiated the architectures and ran the experiment exactly ten minutes (excluding setup time), during which we checked for crashes and logged the output so we could later process the logs and calculate the number of external loss precisely. As outlined above we studied three architectures, three levels of n_{serv} and four levels of cf resulting in a total of 36 experimental cases; therefore, a single run of our experiment took exactly six hours (36×10 minutes) of runtime. Since our model revolves around expected values in a Bernoulli process, we repeated this process 200 times (1200 hours), and report the arithmetic mean of the results.

Methodological Principles of Reproducibility We followed the eight principles of reproducibility introduced in [27]. *Repeated experiments:* see this section. *Workload and configuration coverage:* we covered 36 experimental cases, and analytically modeled the probabilistic behavior of component crash in Section 4. *Experimental setup description:* see Section 5.1. *Open access artifact:* the data of this study is published as an open access data set for supporting replicability[8]. *Probabilistic result description of measured performance:* see Section 5.2. *Statistical evaluation:* see Section 6. *Measurement units:* we reported all units. *Cost:* we did not use a public cloud setting; see Section 5.1 for container configurations.

5.2 Experimental Results

Description Based on Equation (8), IL_T is a model element that incorporates crashes of all components. Moreover, it includes all model views, e.g., architecture configurations, expected average downtime, etc; therefore, we conduct our analysis mainly based on IL_T. Table 2 presents our experimental results; $\sigma(IL_T)$ is the standard deviation of IL_T in 200 runs. We can see that when we keep n_{serv} constant, increasing cf results in a rise of EL_T (Equation (10)) in all cases, which leads to a higher value of IL_T (Equation (8)).

Since in our experiment, we instantiated the DR architecture with three dynamic routers, it is interesting to consider the experimental case of $n_{serv} = 3$. Here, SA and DR have the same number of components, i.e., routers and services. Note that SA uses a sidecar per each service; as a result, with $n_{serv} = 3$, we will also have three sidecars. The difference between the two architectures in this experimental case is that in DR dynamic routers are placed on a different VM than their directly linked services, but in SA sidecars are placed on the same VM on which their corresponding services reside. For this reason, it can be observed that the reported values for SA and DR closely resemble each other when we different values of cf but keep the number of services (n_{serv}) constant at three. Considering the cases with

[8]https://ieee-dataport.org/documents/amiri-tsc-2021 doi:10.21227/mahp-mw44

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3098178

TABLE 2: Results of the Model and the Experiment

Arch.	n_{serv}	cf	C_T	EL_T	IL_T	C_T	EL_T	IL_T	$\sigma(IL_T)$
			Model			Experiment			
CE	3	10	0.800	24.000	114.000	0.760	23.395	98.960	118.552
		25	0.800	60.000	285.000	0.620	47.435	228.975	292.389
		50	0.800	120.000	570.000	0.705	106.370	480.235	608.635
		100	0.800	240.000	1140.000	0.725	218.130	1045.000	1216.765
	5	10	1.200	36.000	246.000	1.165	36.405	236.575	236.536
		25	1.200	90.000	615.000	1.110	85.400	608.040	574.267
		50	1.200	180.000	1230.000	1.115	172.085	1155.550	1173.295
		100	1.200	360.000	2460.000	1.040	317.585	2223.655	2101.272
	10	10	2.200	66.000	786.000	1.920	62.000	720.190	616.778
		25	2.200	165.000	1965.000	2.125	171.290	2063.305	1711.931
		50	2.200	330.000	3930.000	2.160	344.765	4223.665	3458.119
		100	2.200	660.000	7860.000	1.960	590.665	6853.500	6567.047
DR	3	10	1.200	36.000	162.000	1.075	32.505	153.045	175.952
		25	1.200	90.000	405.000	1.225	92.745	452.160	466.814
		50	1.200	180.000	810.000	1.225	182.595	882.695	916.540
		100	1.200	360.000	1620.000	1.130	328.925	1477.405	1470.332
	5	10	1.600	48.000	318.000	1.670	51.995	319.210	301.989
		25	1.600	120.000	795.000	1.760	135.105	816.895	686.709
		50	1.600	240.000	1590.000	1.790	270.540	1597.535	1324.199
		100	1.600	480.000	3180.000	1.635	490.990	2909.115	2353.168
	10	10	2.600	78.000	918.000	2.525	82.255	921.610	495.543
		25	2.600	195.000	2295.000	2.355	187.715	2181.590	1275.035
		50	2.600	390.000	4590.000	2.205	345.350	4043.070	2508.002
		100	2.600	780.000	9180.000	2.375	741.870	8544.700	5022.780
SA	3	10	1.200	36.000	162.000	1.140	34.910	170.265	186.911
		25	1.200	90.000	405.000	1.230	93.265	435.685	452.190
		50	1.200	180.000	810.000	1.215	181.305	883.510	911.088
		100	1.200	360.000	1620.000	1.185	345.950	1634.850	1844.829
	5	10	2.000	60.000	390.000	1.795	55.745	350.055	244.898
		25	2.000	150.000	975.000	1.795	138.910	891.525	647.402
		50	2.000	300.000	1950.000	1.715	261.740	1716.095	1284.733
		100	2.000	600.000	3900.000	1.790	528.420	3385.240	2633.592
	10	10	4.000	120.000	1380.000	3.900	127.715	1443.040	773.632
		25	4.000	300.000	3450.000	3.745	306.745	3477.305	1979.270
		50	4.000	600.000	6900.000	3.860	617.375	7140.655	4262.114
		100	4.000	1200.000	13800.000	3.870	1232.770	14072.910	8287.361

five or ten services, we almost always observe higher IL_T when we change the architecture from a CE to a DR or from a DR to an SA but keep the same configurations, that is, if we keep n_{serv} and cf constant. It is because in our experiment, CE has only one router (the central entity), DR has three (dynamic routers), and SA has n_{serv} (sidecars). Consequently, the number of crashes corresponding to control logic components goes up from CE to DR and then to SA. This increases the total number of crashes C_T (predicted by Equation (12)), which results in losing more requests.

Evaluation of the Prediction Error of Reliability We use the predicted results of our model, presented in Table 2, to measure the accuracy of our analytical model compared to the empirical data from our experiment. The prediction error is measured by calculating the Mean Absolute Percentage Error (MAPE) [40]. Let $model_i$ and $empirical_i$ be the result of the model, and the measured empirical data for experimental case i, respectively:

$$MAPE = \frac{100\%}{n_{case}} \cdot \sum_{i=1}^{n_{case}} \left| \frac{model_i - empirical_i}{empirical_i} \right| \qquad (29)$$

in which n_{case} is the number of cases considered, which is 36 in this experiment. By definition the expected value is the mean of a large number of repetitions [13]. As previously mentioned, a single run of our experiment takes six hours of runtime (plus more than three hours of experimental setup and post-processing of the results); in total we were able to run the experiment 200 times (1200 hours of run-time).

Table 3 reports prediction error measurements of our model for different numbers of runs. A low number of repeats is expected to increase the error since the effects of outliers on the arithmetic mean of the data is considerable. As the table shows, with a higher number of experimental runs the prediction error is reduced, which indicates a converging error rate. After 200 runs the prediction error of 7.8% regarding IL_T is already low enough to use our model for predictions during architecture decision making.

TABLE 3: Prediction Error of the Reliability Model for Different Number of Runs of the Experiment

Num. of Runs	C_T (%)	EL_T (%)	IL_T (%)
50	12.919	12.307	14.072
100	9.416	8.492	9.508
150	8.326	7.426	8.695
200	8.081	7.097	7.776

6 Statistical Model of Performance

Here we describe performance models from the data of our experiment. In the next section, the reliability and performance models are used to perform a trade-off analysis.

The Round-Trip Time In order to compare and measure the performance of the architectures, we recorded the Round-Trip Time (RTT) of requests in our experiment. The

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3098178

RTT is defined as the difference in time from the moment a request is received by the API gateway until it is routed through all cloud services involved in the processing of the request. JMeter generates an identification (ID) number for each HTTP request. Whenever the API gateway receives a request, it starts a timer with an attached ID. The request is routed through cloud services and returns to the gateway when processing is finished. Next, the gateway reads the request ID and stops the corresponding timer. The RTT is the time calculated by the timer.

Statistical Methods Multiple regression analysis is a technique used to create prediction models that estimate the value of a dependent variable based on values of two or more independent variables [34]. The following hypotheses were formulated for this experiment:

H_0 : There is no significant prediction accuracy of the RoundTrip Time (RTT) of requests by the number of services (n_{serv}) and call frequencies (cf).

H_A : There is a significant prediction accuracy of the RoundTrip Time (RTT) of requests by the number of services (n_{serv}) and call frequencies (cf).

We created two prediction models, i.e., linear and nonlinear models, per each architecture configuration to estimate the RTT, i.e., the dependent variable, based on call frequency and number of services, i.e., the independent variables.

Prediction Models Table 4 presents our prediction models for each architecture which we created based on our multiple regression analysis. All of our models result in a very low *p-value* (high statistical significance of the predicted results) which allows us to reject the null hypothesis and accept the alternative hypothesis indicating that the number of services and the call frequency affect the RTT.

$$LinearReg. = SC \cdot n_{serv} + FC \cdot cf + Int \quad (30)$$

$$NonlinearReg. = SC \cdot n_{serv} + FC \cdot cf + IC \cdot n_{serv} \cdot cf + Int \quad (31)$$

The interaction term in Equation (31), i.e., $IC \cdot n_{serv} \cdot cf$, tells us that the effect of the number of services on the predicted RTT is not constant; it changes with different values of call frequency (and vice versa). Note that regression models are calculated from all 200 runs of our experiment.

TABLE 4: Prediction Models of Performance

Arch.	Service Coefficient (SC)	Frequency Coefficient (FC)	Interaction Coefficient (IC)	Intercept (Int)	F-statistic: p-value
CE	3.384e+00	-3.042e-01	5.528e-02	1.608e+01	<2.2e-16
	7.343e+00	0.0265e+00	-	-7.599e+00	<2.2e-16
DR	4.881e+00	-1.254e-01	-1.509e-05	1.287e+01	<2.2e-16
	4.870e+00	-0.125e+00	-	12.872e+00	<2.2e-16
SA	3.360e+00	-0.034e+00	-0.011e+00	5.708e+00	<2.2e-16
	2.552e+00	-0.102e+00	-	10.540e+00	<2.2e-16

TABLE 5: Prediction Error of the Performance Models

Regression	Empirical Data	CE (%)	DR (%)	SA (%)
Linear	Mean	21.527	8.966	10.343
	Median	25.483	9.902	11.119
Nonlinear	Mean	13.654	8.959	10.158
	Median	19.270	9.915	8.958

Evaluation of the Prediction Error of Performance We compare the results of our prediction models to another run of our experiment (not used in the training set). Table 5 presents the prediction error of the regression models. The nonlinear regression compared to the arithmetic mean of the empirical data results in a lower prediction error.

Table 6 compares the empirical data with the predicted results. We report the first quartile (Q_1), the median, the third quartile (Q_3), 95th percentile, the mean and the standard deviation of the recorded round-trip times ($\sigma(RTT)$). We can observe that the predictions in case of DR and SA lie within the interquartile range of the empirical data in most cases; exceptions are the following cases with call frequency of 10 Hr/s: DR with five and ten services, and SA with $n_{serv} = 10$. In these cases, the nonlinear prediction is slightly below the first quartile of the empirical data. Moreover, the predicted RTT in case of DR with $n_{serv} = 10$ and $cf = 50$ Hr/s is above Q_3. With CE the nonlinear predicted results are closer to the arithmetic mean of the data than to the median, as also confirmed in Table 5 with the lower prediction error of 13.7% compared to 19.3%. Note the 30% target prediction accuracy in the cloud performance [23].

7 Trade-Off Analysis

So far we described two models for the qualities reliability and performance which we created for each architecture. In this section we analyze the trade-offs of the architectures with regard to the two qualities in different combinations of configurations, i.e., $1 \leq n_{serv} \leq 10$ and $1 \leq cf \leq 100$.

Reliability Comparison We use the reliability models provided in Section 5 "Specific Model Formulae." Let R_{arch} be the analytical reliability model for each architecture, then:

$$R_{CE} = 0.6 \cdot cf \cdot [(n_{serv})^2 + 3n_{serv} + 1] \quad (32)$$

$$R_{DR} = 0.6 \cdot cf \cdot [(n_{serv})^2 + 5n_{serv} + 3] \quad (33)$$

$$R_{SA} = 0.6 \cdot cf \cdot [2(n_{serv})^2 + 3n_{serv}] \quad (34)$$

which is plotted in Figure 4. CE results in an equal or higher reliability than SA and DR but there are some cases specially in the lower ranges of n_{serv} where SA gives a higher reliability than DR. We compare the architectures.

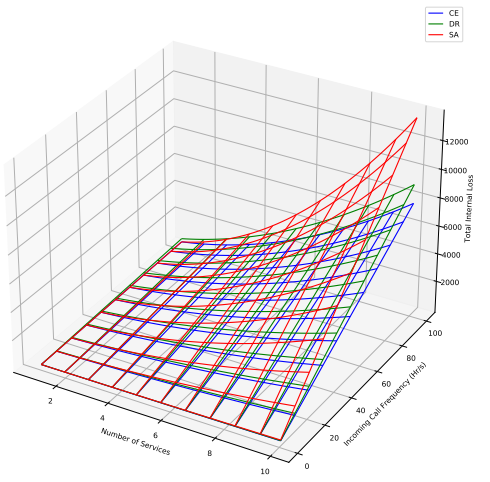


Fig. 4: Reliability Models

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3098178

TABLE 6: Comparison of the Prediction Results of the Performance Models and the Empirical Data

Arch.	n_{serv}	cf (Hr/s)	Q_1 (ms)	Median RTT (ms)	Q_3 (ms)	95th Percentile (ms)	Mean RTT (ms)	$\sigma(RTT)$	Linear Regression (ms)	Nonlinear Regression (ms)
CE	3	10	22.173	24.277	27.504	36.627	26.0169	11.899	14.695	24.848
		25	19.228	21.327	26.773	39.951	24.423	9.466	15.093	22.773
		50	16.618	18.339	23.367	35.350	21.333	9.863	15.756	19.314
		100	13.101	14.597	17.983	27.975	16.938	9.843	17.083	12.396
	5	10	36.490	40.021	44.845	55.381	42.138	15.835	29.381	32.722
		25	27.564	29.862	33.428	44.942	31.987	12.791	29.780	32.305
		50	24.185	26.618	30.752	42.250	28.948	11.276	30.442	31.610
		100	18.078	19.794	24.810	35.657	23.966	24.713	31.769	30.220
	10	10	64.488	69.357	74.901	88.528	72.344	30.946	66.096	52.406
		25	47.363	51.796	58.966	72.632	55.832	33.015	66.494	56.135
		50	39.035	43.826	50.811	63.718	48.306	37.599	67.158	62.350
		100	48.634	58.066	70.423	95.812	74.398	139.257	68.484	74.780
DR	3	10	23.371	26.374	30.955	40.017	28.322	11.521	26.257	26.259
		25	20.845	23.152	27.744	38.264	25.477	9.504	24.374	24.377
		50	18.053	19.601	22.588	35.026	21.901	9.295	21.237	21.241
		100	13.536	14.817	18.005	28.168	17.192	10.349	14.962	14.968
	5	10	37.844	42.893	49.4277	62.270	45.422	18.780	36.016	36.020
		25	30.442	34.011	39.034	51.303	36.345	14.731	34.133	34.138
		50	23.863	26.637	31.799	43.272	29.350	15.122	30.996	31.001
		100	18.242	20.235	25.503	36.201	23.584	16.343	24.721	24.727
	10	10	70.034	76.020	83.473	97.357	79.636	36.074	60.414	60.424
		25	50.677	55.427	60.877	75.861	58.545	29.661	58.532	58.541
		50	41.436	46.638	52.788	65.423	51.010	47.884	55.394	55.402
		100	40.997	47.254	55.167	70.112	54.562	75.960	49.119	49.125
SA	3	10	13.500	15.938	20.042	26.399	17.427	6.483	17.176	15.106
		25	11.747	13.381	16.782	22.975	14.881	5.155	15.648	14.083
		50	10.449	11.875	16.258	25.607	14.188	6.349	13.102	12.377
		100	6.923	7.898	9.975	18.061	9.456	5.196	8.010	8.965
	5	10	21.554	25.185	29.860	37.137	26.561	10.007	22.279	21.601
		25	17.330	20.227	24.383	33.295	21.881	7.671	20.751	20.239
		50	13.573	15.174	18.158	27.103	16.831	6.913	18.205	17.968
		100	11.456	13.896	17.857	27.665	15.726	7.908	13.113	13.427
	10	10	44.875	48.860	53.678	63.075	50.705	18.464	35.037	37.838
		25	32.633	36.5545	41.214	53.287	38.577	16.120	33.509	35.628
		50	26.433	29.718	34.265	45.468	32.117	18.422	30.963	31.946
		100	19.509	22.221	26.482	37.321	25.646	27.174	25.871	24.582

Reliability Trade-Off Between CE and DR Trying to find the intersecting line where $R_{CE} = R_{DR}$, we find that there is no combination of cf and n_{serv}, where the two curves collide; therefore, CE always results in a higher reliability than DR in our focused context.

Reliability Trade-Off Between CE and SA We find the intersecting line where $P_{CE} = P_{SA}$ in our focused context is $n_{serv} = 1$. That is when we have only one service, since we use the same implementation for all architectures, SA and CE become the same application. Therefore, they result in the same value of reliability. In any other case, CE results in a higher reliability than SA.

Reliability Trade-Off Between DR and SA We find the intersecting line where $P_{DR} = P_{SA}$ in our focused context is $n_{serv} = 3$. That is when there are three services, DR and SA are the same application in our implementation since they both have the same number of routers; therefore, they result in the same value of reliability. Note that in our experiment we instantiated DR with three and SA with n_{serv} routers. When $n_{serv} < 3$, SA has fewer routers than DR; consequently, SA results in a lower number of request loss, i.e., higher reliability, than DR. When $n_{serv} > 3$, DR has fewer routers and results in a higher reliability than SA.

Summary of the Reliability Trade-Offs When $n_{serv} \leq 3$ we have $R_{CE} \leq R_{SA} \leq R_{DR}$ and when $n_{serv} > 3$ we have $R_{CE} < R_{DR} < R_{SA}$ for all studied call frequencies.

Performance Comparison For the performance models we used the nonlinear regression, i.e., Equation (31), in which the coefficients are taken from Table 4. Let P_{arch} be the performance prediction model for each architecture:

$$P_{CE} = 3.384 \cdot n_{serv} - 0.3042 \cdot cf + 16.08 + 0.05528 \cdot n_{serv} \cdot cf \quad (35)$$

$$P_{DR} = 4.881 \cdot n_{serv} - 0.1254 \cdot cf + 12.87 - 0.00001509 \cdot n_{serv} \cdot cf \quad (36)$$

$$P_{SA} = 3.360 \cdot n_{serv} - 0.0340 \cdot cf + 5.708 - 0.011 \cdot n_{serv} \cdot cf \quad (37)$$

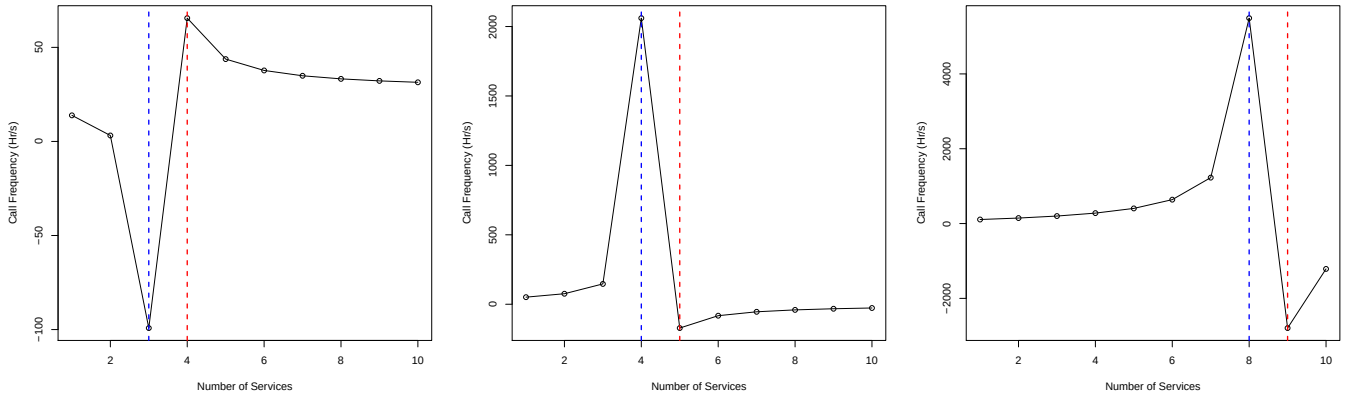
which are plotted in Figure 6. In most cases, SA results in a lower RTT than the other architectures. However, there are some cases that CE outperforms DR and SA. We compare the architectures to find the exact range of n_{serv} and cf, in which each architecture performs the highest.

Performance Trade-Off Between CE and DR To characterize the trade-off more precisely, we have to study the intersecting line where $P_{CE} = P_{DR}$, i.e., the line where the curves of the architectures collide:

$$cf = \frac{1.497 \cdot n_{serv} - 3.21}{0.0552951 \cdot n_{serv} - 0.1788} \quad (38)$$

which is plotted in Figure 5a. Note that the blue dashed line, i.e., $n_{serv} = 3$, and the red dashed line, i.e., $n_{serv} = 4$,

This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3098178



(a) CE vs. DR (b) CE vs. SA (c) DR vs. SA

Fig. 5: Plot of All Intersecting Lines

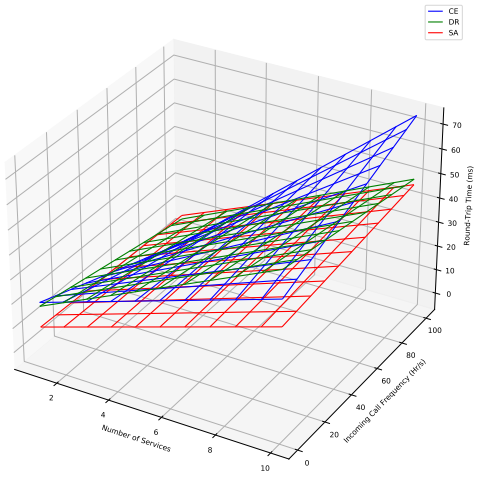


Fig. 6: Performance Models

indicate the extrema of the intersecting line; therefore, CE outperforms DR in the area above the intersecting line when $n_{serv} \leq 3$, and below the intersecting line when $n_{serv} > 4$.

Table 7 summarizes the regions of cf and n_{serv}, in which CE outperforms DR. It can be confirmed by the results of our model for the experimental cases reported in Table 6, in which under nonlinear regression, we can observe that in case of $n_{serv} = 3$, CE outperforms DR for all values of cf. However, when we have five or ten services, only in the lower range of incoming call frequency, i.e., 10 and 25, CE results in a lower performance value.

Performance Trade-Off Between CE and SA We find the intersecting line where $P_{CE} = P_{SA}$ in our focused context:

$$cf = \frac{-0.024 \cdot n_{serv} - 10.372}{0.06628 \cdot n_{serv} - 0.2702} \tag{39}$$

plotted in Figure 5b. In our focused context, CE outperforms SA only with the following conditions:

$$n_{serv} = 1 \quad and \quad cf \geq 50.98 \tag{40}$$
$$n_{serv} = 2 \quad and \quad cf \geq 75.71 \tag{41}$$

Performance Trade-Off Between DR and SA The intersecting line where $P_{DR} = P_{SA}$ is plotted in Figure 5c:

$$cf = \frac{-1.521 \cdot n_{serv} - 7.162}{0.010985 \cdot n_{serv} - 0.091} \tag{42}$$

Summary of the Performance Trade-Offs In Table 8, lower P_{arch} means lower RTT, i.e., better performance.

TABLE 8: Comparison of the Performance of the Architectures

n_{serv}	cf (Hr/s)	Performance
1	until 13.87	$P_{SA} \leq P_{DR} \leq P_{CE}$
	between 13.87 and 50.98	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 50.98	$P_{CE} \leq P_{SA} \leq P_{DR}$
2	until 3.17	$P_{SA} \leq P_{DR} \leq P_{CE}$
	between 3.17 and 75.71	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 75.71	$P_{CE} \leq P_{SA} \leq P_{DR}$
3	all	$P_{SA} \leq P_{CE} \leq P_{DR}$
4	until 65.55	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 65.55	$P_{SA} \leq P_{DR} \leq P_{CE}$
5	until 43.77	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 43.77	$P_{SA} \leq P_{DR} \leq P_{CE}$
6	until 37.73	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 37.73	$P_{SA} \leq P_{DR} \leq P_{CE}$
7	until 34.90	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 34.90	$P_{SA} \leq P_{DR} \leq P_{CE}$
8	until 33.26	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 33.26	$P_{SA} \leq P_{DR} \leq P_{CE}$
9	until 32.19	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 32.19	$P_{SA} \leq P_{DR} \leq P_{CE}$
10	until 31.43	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 31.43	$P_{SA} \leq P_{DR} \leq P_{CE}$

TABLE 7: The Region Where CE outperforms DR

n_{serv}	1	2	3	4	5	6	7	8	9	10
cf (Hr/s)	≥ 13.87	≥ 3.17	≥ 1.00	≤ 65.55	≤ 43.77	≤ 37.73	≤ 34.90	≤ 33.26	≤ 32.19	≤ 31.43

8 Threats to Validity

Construct Validity In our study, we injected crashes to simulate real world crash behavior at a given probability. While this is a commonly taken approach (see Section 2), a threat remains that measuring internal and external loss based on these crashes might not measure reliability well. For example, system reliability is also influenced by cascading effects of crashes beyond a single call sequence [26] which are not covered in our experiment. More research, probably with real-world systems, is needed to exclude this threat.

Internal Validity We collected an extensive amount of data to validate our model. However, we did so in limited experiment time and with injected crashes, simulated by stopping Docker containers. We avoided factors such as other load on the machines where the experiment ran and much of the related literature takes a similar approach (see Section 2), but research observing real-world cloud-based systems with real crashes would be needed to confirm that there are no other factors influencing the measurements.

External Validity The results might not be generalizable beyond the given experimental cases of 10-100 requests per second and call sequences of length 3-10. However, this covers a wide variety of loads and call sequences in cloud-based applications. Moreover, in our experiments we considered a uniform crash probability for all components. This is a common assumption made in such experiments (see, e.g., [29], [30]) to increase the control over the experiment's dependent variables and thus the internal validity of the experiment. At the same time, this might decrease the external validity, if the crash profiles observed in a real-world application are substantially different (see [37] for the trade-off between the internal and external validity in empirical software engineering). To mitigate this threat, our model, in the general form, does not assume a uniform crash probability for all components.

Conclusion Validity As the statistical method to compare our model's predictions to the empirical data, we used the MAPE metric as it is widely used and offers good interpretability in our research context. To mitigate the threat that this statistical method might have issues, we double-checked three other error measures, which led to similar converging results. We reported MAPE; the other measurements are included in the online appendix.

9 Conclusions and Future Work

In this article, we investigated three representative service and cloud architecture patterns for dynamic routing regarding their impact and trade-offs on reliability and performance. Regarding **RQ1**, our study concludes that more decentralized routing results in losing a higher number of requests, i.e., lower reliability, in comparison to more centralized approaches; however, regarding **RQ2**, our results show that distributed settings indicate better performance, specially under high load, because of using more routers.

Regarding **RQ3**, we derived an analytical model for predicting request loss in the studied architectures and empirically validated this model using 36 representative experimental cases. Our results indicate that, with a higher number of experimental runs, the prediction error is constantly reduced, converging at a prediction error of 7.8%. Furthermore, we have created prediction models providing an estimation on the performance impact of the investigated architectures. The found models show high statistical significance; in addition, we cross-validated the estimated RTTs with measurements from an additional experiment run.

Regarding **RQ4**, we precisely calculated the range of the incoming call frequency and the number of services, where each architecture gives better results. With regard to system reliability, CE always results in a lower request loss; however, SA results in a better performance specially in higher number of services. DR can be seen as a middle ground specially in a higher number of services; this introduces an interesting future work focus in which we abstract all three architectures under DR with reconfigurable routers. Then we set out to find the optimal configuration under certain constraints, e.g., the cost of cloud deployment.

The major impact of our work is on architectural design decisions for dynamic routing in service- and cloud-based architectures. Prior to our work, for system reliability and performance trade-offs, architects had to rely on their experiences as no empirical evidence was available. To the best of our knowledge, our work is the first to provide such evidence. Our work's main contributions are models and an empirical study of widely used architectures, about which little was known before our study. Such empirical works enable building new algorithms and architectures which are based on a solid and well-founded understanding of the existing architectures. For instance, this enables exploring more sophisticated prediction models, such as machine learning-based approaches and evaluation theories of reliability, which are possible studies based on our research. To be successful, such works require careful empirical studies laying the foundation for understanding the existing state of the art and its limitations, providing ground truths, and offering data sets for further studies (such as the open access data set provided in our article[8]). For our future work, we plan to use the empirical data and model from this study to design a novel adaptive routing software architecture, which chooses among the architectural options dynamically.

Acknowledgment

This work was supported by FWF (Austrian Science Fund), project ADDCompliance: I 2885-N33; FFG (Austrian Research Promotion Agency), project DECO no. 864707; Baden-Württemberg Stiftung, project ORCAS.

References

[1] S. P. Ahuja and A. Patel. Enterprise service bus: A performance evaluation. *Communications and Network*, 3(03):133, 2011.

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

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

[4] R. Bankston and J. Guo. Performance of container network technologies in cloud environments. In *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE, 2018.

[5] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

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

[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] 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.

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

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

[13] G. Grimmett and D. Welsh. *Probability: An Introduction*. Cambridge University Press, 1986.

[14] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger. Performance engineering for microservices: research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 223–226. ACM, 2017.

[15] N. R. Herbst, S. Kounev, A. Weber, and H. Groenda. Bungee: An elasticity benchmark for self-adaptive iaas cloud environments. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, pages 46–56, Piscataway, NJ, USA, 2015. IEEE Press.

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

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

[18] 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.

[19] 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.

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

[21] G. Kumar, M. Kaushik, and R. Purohit. Reliability analysis of software with three types of errors and imperfect debugging using markov model. *International Journal of Computer Applications in Technology (IJCAT)*, 2018.

[22] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *Cloud Engineering (IC2E), 2018 IEEE International Conference on*, pages 159–169. IEEE, 2018.

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

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

[25] R. Natella, D. Cotroneo, and H. S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)*, 48(3):44, 2016.

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

[27] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tuma, and A. Iosup. Methodological principles for reproducible performance evaluation in cloud computing. In *IEEE Transactions on Software Engineering*. IEEE, 2019.

[28] 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.

[29] T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske. Hora: Architecture-aware online failure prediction. *Journal of Systems and Software*, 137, 2017.

[30] T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske. An architecture-aware approach to hierarchical online failure prediction. In *12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, 2016.

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

[32] 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.

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

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

[35] 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.

[36] T. Shezi, E. Jembere, and M. Adigun. Performance evaluation of enterprise service buses towards support of service orchestration. In *Proc. of International Conference on Computer Engineering and Network Security (ICCENS'2012)*, 2012.

[37] J. Siegmund, N. Siegmund, and S. Apel. Views on internal and external validity in empirical software engineering. In *37th International Conference on Software Engineering (ICSE)*, 2015.

[38] 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.

[39] O. Sukwong, A. Sangpetch, and H. S. Kim. Sageshift: managing slas for highly consolidated cloud. In *2012 Proceedings IEEE INFOCOM*, pages 208–216, 2012.

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

[41] A. Van Hoorn, A. Aleti, T. F. Düllmann, and T. Pitakrat. Orcas: Efficient resilience benchmarking of microservice architectures. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 146–147. IEEE, 2018.

[42] K. Vandikas and V. Tsiatsis. Performance evaluation of an iot platform. In *Next Generation Mobile Apps, Services and Technologies, 8th International Conference on*, pages 141–146. IEEE, 2014.

[43] 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.

[44] 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, 2010.

Amirali Amiri Amirali Amiri M.Sc. is a doctoral candidate at the Software Architecture Group of Faculty of Computer Science, University of Vienna. Amirali received his Master's degree in Informatics from Technical University of Munich, Germany and his Bachelor's degree in Computer Engineering from Ferdowsi University of Mashhad, Iran. His research focuses on software design and architecture, also fault injection, detection and prediction in service-based systems.

Uwe Zdun Prof. Dr. Uwe Zdun is a full professor for software architecture at the Faculty of Computer Science, University of Vienna. His research focuses on software design and architecture, distributed systems engineering (service-based, cloud, mobile, IoT, and microservices systems), DevOps and continuous delivery, and empirical software engineering. Uwe has published more than 240 peer-reviewed articles, and is co-author of 3 professional books.

André van Hoorn André van Hoorn is a senior researcher with the Institute of Software Technology at the University of Stuttgart, Germany. He received his Ph.D. degree (with distinction) from Kiel University, Germany and his Master's degree from the University of Oldenburg, Germany. His research focuses on designing, operating, and evolving trustworthy distributed software systems, focusing on quality attributes such as performance, reliability, and resilience.