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 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 percent. Overall, we performed 200 experiment runs, which ran a total of 1,200 hours (excluding setup time). Given the target prediction accuracy of up to 30 percent 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 percent 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 7).

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 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, WS02 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 n 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 compares 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 Fig. 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, 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]. Fig. 1a shows an example configuration of CE. Note that it is not required that the central entity component is always deployed on an exclusive host.

Sidecar-Based Architecture (SA). SA is presented in Fig. 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. Fig. 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.

| Notation | Description |
|----------|-------------|
| T        | Observed system time |
| n_{cr}   | Number of services |
| n_{cr}   | Number of crash tests |
| C\(_{T} \) | Crash interval |
| c\(_{f} \) | 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 |
| IR_{L}   | Total internal loss |
| IR_{L,c} | Sum of the internal loss per crash of each router |
| IR_{L,s} | Sum of the internal loss per crash of each service |
| IR_{L,c} | Internal loss for a component c |
| IR_{L,r} | Internal loss for a router r |
| IR_{L,s} | Internal 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_{exe}  | Number of executed internal requests for the crash of a component c |
| n_{exe}  | Number of executed internal requests for the crash of a router r |
| n_{exe}  | Number of executed internal requests for the crash of a service s |
| d_{c}    | Expected average downtime after a component c crashes |
| d_{c}    | 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 C\(_{T} \) |
| P_{r}    | Crash probability of a router r every C\(_{T} \) |
| P_{s}    | Crash probability of a service s every C\(_{T} \) |
| E\(_{C} \) | Expected number of crashes of a component c during T |
| E\(_{R} \) | Expected number of crashes of a router r during T |
| E\(_{S} \) | Expected number of crashes of a service s during T |

4.1 Metamodel

As depicted in Fig. 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. Fig. 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.

Table 1: The Mathematical Notations Used in This Article

Fig. 2. Example model instance.

The partially ordered set representing the call trace ER, IR\(_{R_{1}}\), ..., IR\(_{R_{11}}\) 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\(_{exe}\) be the internal loss, the external loss and the number of executed internal requests for the crash of a component \(c\).

\[
IL_{c} = EL_{c} \cdot (IR_{T} - n_{exe}).
\]

**Example Crash Scenario for a Router Crash.** Let us consider the crash of R\(_{3}\) in Fig. 2. In this case, IR\(_{1}\) to IR\(_{8}\) are executed, i.e., n\(_{exe}\) = 8, but we lose three internal requests, namely IR\(_{9}\), IR\(_{10}\) and IR\(_{11}\), then

\[
IL_{c} = EL_{c} \cdot (11 - 8) = EL_{c} \cdot 3.
\]

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. Fig. 2 shows a configuration of a DR architecture with three routers and five services. The instantiated components send internal requests, labeled from IR\(_{R_{1}}\) to IR\(_{R_{11}}\), amongst one another to complete the processing of the one external request, labeled ER. The partially ordered set representing the call trace ER, IR\(_{R_{1}}\), ..., IR\(_{R_{11}}\) 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.

**Example Crash Scenario for a Router Crash.** Let us consider the crash of R\(_{3}\) in Fig. 2. In this case, IR\(_{1}\) to IR\(_{8}\) are executed, i.e., n\(_{exe}\) = 8, but we lose three internal requests, namely IR\(_{9}\), IR\(_{10}\) and IR\(_{11}\), then

\[
IL_{c} = EL_{c} \cdot (11 - 8) = EL_{c} \cdot 3.
\]

**Example Crash Scenario for a Service Crash.** Let us consider the crash of S\(_{5}\) in Fig. 2. In this case, IR\(_{1}\) to IR\(_{9}\) are executed, i.e., n\(_{exe}\) = 9, and we lose two internal requests, i.e., IR\(_{R_{10}}\) and IR\(_{R_{11}}\), then

\[
IL_{c} = EL_{c} \cdot (11 - 9) = EL_{c} \cdot 2.
\]

**External Loss.** Let \(d_{c}\) be the expected average downtime after a component \(c\) crashes and c\(_{f}\) the incoming call frequency, i.e., the frequency at which external requests are received. The external loss per crash of each component \(c\) is

\[
EL_{c} = d_{c} \cdot c_{f}.
\]

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 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 $n_{\text{crash}}$ be the number of times we check for a crash of components, i.e., the number of crash tests

$$n_{\text{crash}} = \left\lfloor \frac{T}{CI} \right\rfloor.$$  

**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 $n_{\text{crash}}$ 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 Fig. 3b). Let $E[C_c]$ be the expected number of crashes of a component $c$ during $T$

$$E[C_c] = n_{\text{crash}} \cdot P_c.$$  

**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.

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

which can be rewritten using Equations (1), (4), (5), and (6) as

$$IL_T = \left\lfloor \frac{T}{CI} \right\rfloor \cdot cf \cdot \sum_{c \in C} P_c \cdot d_c \cdot (IR_T - n_{\text{crash}}^{\text{exec}}).$$  

The total external loss, i.e., $EL_T$, is the sum of external loss per crash of each component.
\[ EL_T = \sum_{c \in C} E[C_c] \cdot EL_c, \]  
which can be rewritten using Equations (4), (5), and (6) as

\[ EL_T = \left[ \frac{T}{CI} \right] \cdot cf \cdot \sum_{c \in C} P_c \cdot d_c. \]

**Total Number of Crashes.** \( C_T \) is the sum of the expected number of crashes of each component.

\[ C_T = \sum_{c \in C} E[C_c], \]

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

\[ C_T = \left[ \frac{T}{CI} \right] \cdot \sum_{c \in C} P_c. \]

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. 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), (8), (9), and (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 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 total). On top of the cloud nodes, we installed Virtual Machines (VMs), each of which used the VMware® ESXi version 6.7.0 u2 hypervisor, had eight CPU cores, 60 GB system memory, and ran Ubuntu Server 18.04.01 LTS.³ Docker⁴ 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 JMeter⁶ which sent Hypertext Transfer Protocol (HTTP) version 1.1⁷ requests to the cloud nodes.

**Architecture Configurations.** Any application that has a request flow can be modeled using our proposed metamodel. 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 Fig. 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 in Fig. 2. 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 the number of services \( (n_{serv}) \), 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 services.

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 percent 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 $T$.

$$P_c = 0.5\% \quad \forall \ c \in C.$$  

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)).

Specific Model Formulae. As explained before, in Equation (1), $IR_T$ and $n^{\text{exec}}$ need to be parameterized based on the application. Since in our example configurations each service receives an internal request (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

$$IR_T = 2n_{\text{serv}} + 1.$$  

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

$$n^{\text{exec}}_s = 2s^{\text{crashed}} - 1.$$  

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

$$IL_S = 0.6 \cdot cf \cdot n_{\text{serv}}(n_{\text{serv}} + 1).$$  

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 Fig. 2 is

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

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

$$n^{\text{exec}}_r = 2 \sum_{i=1}^{r^{\text{crashed}}} A_{r-1},$$  

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 Fig. 2).

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

$$A = \{n_{\text{serv}}\} \quad \text{and} \quad A_0 = 0.$$  

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

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

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

$$IL_T = IL_R + IL_S$$  

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

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

$$A = \left\{ \frac{n_{\text{serv}}}{3}, \frac{n_{\text{serv}}}{3}, \frac{n_{\text{serv}}}{3} \pm 1 \right\} \quad \text{and} \quad A_0 = 0.$$  

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

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

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

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

$$A = \{1, 1, \ldots, 1\} \quad \text{and} \quad A_0 = 0,$$  

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

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

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

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 $n_{\text{serv}}$ and four levels of $cf$ resulting in a total of 36 experimental cases; therefore, a single run of our experiment took exactly six hours ($36 \times 10$ minutes) of runtime. Since our model revolves around expected values in a Bernoulli process, we repeated this process 200 times (1,200 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. 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 use different values of $cf$ but keep the number of services ($n_{serv}$) constant at three. Considering the cases with 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

$$MAPE = \frac{100\%}{n_{case}} \sum_{i=1}^{n_{case}} \left| \frac{model_i - empirical_i}{empirical_i} \right|,$$

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 (1,200 hours of run-time).

| Arch. | $n_{serv}$ | $cf$ |
|-------|------------|------|
| CE    | 3          | 10   |
|       | 100        | 0.800 24.000 114.000 |
|       | 25         | 0.800 60.000 285.000 |
|       | 50         | 0.800 120.000 570.000 |
|       | 100        | 0.800 210.000 1140.000 |
| DR    | 5          | 10   |
|       | 100        | 1.200 36.000 246.000 |
|       | 25         | 1.200 90.000 615.000 |
|       | 50         | 1.200 180.000 1230.000 |
|       | 100        | 1.200 360.000 2460.000 |
|       | 10         | 1.200 66.000 786.0390 |
|       | 25         | 1.200 165.000 1965.000 |
|       | 50         | 1.200 330.000 3930.000 |
|       | 100        | 1.200 660.000 7860.000 |
| SA    | 5          | 10   |
|       | 100        | 1.200 36.000 162.000 |
|       | 25         | 1.200 90.000 405.000 |
|       | 50         | 1.200 180.000 810.000 |
|       | 100        | 1.200 360.000 1620.000 |
|       | 10         | 1.600 48.000 318.000 |
|       | 25         | 1.600 120.000 795.000 |
|       | 50         | 1.600 240.000 1590.000 |
|       | 100        | 1.600 480.000 3180.000 |
|       | 10         | 2.600 78.000 918.000 |
|       | 25         | 2.600 195.000 2295.000 |
|       | 50         | 2.600 390.000 4590.000 |
|       | 100        | 2.600 780.000 9180.000 |
|       | 10         | 1.200 36.000 162.000 |
|       | 25         | 1.200 90.000 405.000 |
|       | 50         | 1.200 180.000 810.000 |
|       | 100        | 1.200 360.000 1620.000 |
|       | 10         | 2.000 60.000 390.000 |
|       | 25         | 2.000 150.000 975.000 |
|       | 50         | 2.000 300.000 1950.000 |
|       | 100        | 2.000 600.000 3900.000 |
|       | 10         | 4.000 120.000 3900.000 |
|       | 25         | 4.000 300.000 3450.000 |
|       | 50         | 4.000 600.000 6900.000 |
|       | 100        | 4.000 1200.000 13800.000 |
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 percent regarding \( ILC_T \) is already low enough to use our model for predictions during architecture decision making.

### 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 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.

### 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 $R_{\text{arch}}$ be the analytical reliability model for each architecture, then

$$R_{\text{CE}} = 0.6 \cdot c_f \cdot \left( \left[ n_{\text{serv}} \right]^2 + 3n_{\text{serv}} + 1 \right)$$ (32)

$$R_{\text{DR}} = 0.6 \cdot c_f \cdot \left[ \left( n_{\text{serv}} \right)^2 + 5n_{\text{serv}} + 3 \right]$$ (33)

$$R_{\text{SA}} = 0.6 \cdot c_f \cdot \left[ 2 \left( n_{\text{serv}} \right)^2 + 3n_{\text{serv}} \right],$$ (34)

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

Reliability Trade-Off Between CE and DR. Trying to find the intersecting line where $P_{\text{CE}} = P_{\text{DR}}$, we find that there is no combination of $c_f$ and $n_{\text{serv}}$ 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 $P_{\text{DR}} = P_{\text{SA}}$ in our focused context is $n_{\text{serv}} = 3$. 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 \( n_{\text{serv}} \) routers. When \( n_{\text{serv}} < 3 \), SA has fewer routers than DR; consequently, SA results in a lower number of request loss, i.e., higher reliability, than DR. When \( n_{\text{serv}} > 3 \), DR has fewer routers and results in a higher reliability than SA.

**Summary of the Reliability Trade-Offs.** When \( n_{\text{serv}}/C20 \), we have 
\[
R_{\text{CE}} < R_{\text{SA}} < R_{\text{DR}}
\]
and when \( n_{\text{serv}} > 3 \), we have 
\[
R_{\text{CE}} < R_{\text{DR}} < R_{\text{SA}}
\]
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 \( P_{\text{arch}} \) be the performance prediction model for each architecture

\[
P_{\text{CE}} = 3.384 \cdot n_{\text{serv}} - 0.3042 \cdot cf + 16.08 + 0.05528 \cdot n_{\text{serv}} \cdot cf
\]
(35)

\[
P_{\text{DR}} = 4.881 \cdot n_{\text{serv}} - 0.1254 \cdot cf + 12.87 - 0.00001509 \cdot n_{\text{serv}} \cdot cf
\]
(36)

\[
P_{\text{SA}} = 3.360 \cdot n_{\text{serv}} - 0.0340 \cdot cf + 5.708 - 0.011 \cdot n_{\text{serv}} \cdot cf.
\]
(37)

which are plotted in Fig. 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 \( n_{\text{serv}} \) 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 \( P_{\text{CE}} = P_{\text{DR}} \), i.e., the line where the curves of the architectures collide

\[
cf = \frac{1.497 \cdot n_{\text{serv}} - 3.21}{0.0552951 \cdot n_{\text{serv}} - 0.1788}
\]
(38)

which is plotted in Fig. 5a. Note that the blue dashed line, i.e., \( n_{\text{serv}} = 3 \), and the red dashed line, i.e., \( n_{\text{serv}} = 4 \), indicate the extrema of the intersecting line; therefore, CE outperforms DR in the area above the intersecting line when \( n_{\text{serv}} /C20 \), and below the intersecting line when \( n_{\text{serv}} > 4 \).

Table 7 summarizes the regions of \( cf \) and \( n_{\text{serv}} \) 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 \( n_{\text{serv}} = 3 \), 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 \( P_{\text{CE}} = P_{\text{SA}} \) in our focused context

\[
cf = \frac{-0.024 \cdot n_{\text{serv}} - 10.372}{0.06628 \cdot n_{\text{serv}} - 0.2702}
\]
(39)

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

\[
n_{\text{serv}} = 1 \quad \text{and} \quad cf \geq 50.98
\]
(40)

\[
n_{\text{serv}} = 2 \quad \text{and} \quad cf \geq 75.71.
\]
(41)

**Performance Trade-Off Between DR and SA.** The intersecting line where \( P_{\text{DR}} = P_{\text{SA}} \) is plotted in Fig. 5c

\[
cf = \frac{-1.521 \cdot n_{\text{serv}} - 7.162}{0.010985 \cdot n_{\text{serv}} - 0.091}
\]
(42)
Summary of the Performance Trade-Offs. In Table 8, lower $P_{\text{crash}}$ means lower RTT, i.e., better performance.

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. 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, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSC.2021.3098178.

9 CONCLUSION 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 percent. 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\(^\text{8}^\)). 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.

ACKNOWLEDGMENTS

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,” Commun. Netw., vol. 3, no. 03, 2011, Art. no. 133.
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 Proc. IEEE Int. Conf. Services Comput., 2019, pp. 215–219.
3. A. Amiri, U. Zdun, G. Simhandl, and A. van Hoorn, “Impact of service- and cloud-based dynamic routing architectures on system reliability,” in Proc. Int. Conf. Serv. Oriented Comput., 2020, pp. 165–174.
4. R. Bankston and J. Guo, “Performance of container network technologies in cloud environments,” in Proc. IEEE Int. Conf. Elector Info. Technol., 2018, pp. 0277–0283.
5. A. Basiri et al., “Chaos engineering,” IEEE Softw., vol. 33, no. 3, pp. 35–41, May/Jun. 2016.
6. S. Becker, H. Koziolek, and R. Reussner, “Model-based performance prediction with the palladio component model,” in Proc. 6th Int. Workshop Softw. Perform., 2007, pp. 54–65.
7. F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, “Architecture-based reliability prediction with the palladio component model,” IEEE Trans. Softw. Eng., vol. 38, no. 6, pp. 1319–1339, Nov./Dec. 2012.
8. D. A. Chappell, Enterprise Service Bus. Sebastopol, CA, USA: O’Reilly, 2004.
9. R. C. Cheung, “A user-oriented software reliability model,” IEEE Trans. Softw. Eng., vol. SE-6, no. 2, pp. 118–125, Mar. 1980.
10. D. J. Dean, H. Nguyen, P. Wang, and X. Gu, “PeriCompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds,” in Proc. 6th USENIX Workshop Hot Topics Cloud Comput., 2014, Art. no. 16.
11. Envoy, “Service mesh,” 2019. [Online]. Available: https://www.learnenvoy.io/articles/service-mesh.html
12. V. Grassi and S. Patella, “Reliability prediction for service-oriented computing environments,” IEEE Internet Comput., vol. 10, no. 3, pp. 43–49, May/Jun. 2006.
13. G. Grimmett and D. Welsh, Probability: An Introduction. Cambridge, U.K.: Cambridge Univ. Press, 1986.
14. R. Heinrich et al., “Performance engineering for microservices: Research challenges and directions,” in Proc. 8th ACM/IEEE Int. Conf. Perform. Eng. Companion, 2017, pp. 223–226.
15. N. R. Herbst, S. Kounев, A. Weber, and H. Grauena, “BUNGEE: An elasticity benchmark for self-adaptive IaaS cloud environments,” in Proc. 10th Int. Symp. Softw. Eng. Adaptive Self-Manag. Syst., 2015, pp. 46–56.
16. G. Hohpe and B. Woolf, Enterprise Integration Patterns. Boston, MA, USA: Addison-Wesley, 2003.
17. A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson, Cloud Design Patterns. Redmond, WA, USA: Microsoft Press, 2014.
18. P. J. Iatridi, C. Paleologou, S. Mavromat, I. Lewis, and S. Kivok, “Microservices: The journey so far and challenges ahead,” IEEE Softw., vol. 35, no. 3, pp. 24–35, May/Jun. 2018.
19. H. Khazaee, C. Barma, N. Beigi-Mohammadi, and M. Litoui, “Efficiency analysis of provisioning microservices,” in Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci., 2016, pp. 261–268.
20. N. Kratzke, “About microservices, containers and their underestimation impact on production,” 2017, arXiv:1710.04049. doi: 10.13140/RG.2.1.2039.3046.
21. G. Kumar, M. Kaushik, and R. Purohit, “Reliability analysis of software with three types of errors and imperfect debugging using Markov model,” Int. J. Comput. Appl. Technol., vol. 58, pp. 241–249, 2018.
22. W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallikkara, “Serviceless computing: An investigation of factors influencing microservice performance,” in Proc. IEEE Int. Conf. Cloud Eng., 2018, pp. 159–169.
23. D. A. Menascé and V. A. Almeida, Capacity Planning for Web Services: Metrics, Models, and Methods. Hoboken, NJ, USA: Prentice Hall PTR, 2001.
24. Microsoft, “Sidebar pattern,” 2010. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidebar
25. R. Natella, D. Cotroneo, and H. S. Madeira, “Assessing dependency with software fault injection: A survey,” ACM Comput. Surv., vol. 48, no. 3, 2016, Art. no. 44.
26. M. Nygard, Release It!: Design and Deploy Production-Ready Software. Raleigh, NC, USA: Pragmatic Bookshelf, 2007.
27. A. V. Papadopoulos et al., “Methodological principles for reproducible performance evaluation in cloud computing,” IEEE Trans. Softw. Eng., to be published, doi: 10.1109/TSE.2019.2927908.
28. D. Petriu, C. Shousha, and A. Jalnapurkar, “Architecture-based performance analysis applied to a telecommunication system,” IEEE Trans. Softw. Eng., vol. 26, no. 11, pp. 1049–1065, Nov. 2000.
29. T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske, “Hora: Architecture-aware online failure prediction,” J. Syst. Softw., vol. 133, pp. 669–685, 2018.
30. T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske, “An architecture-aware approach to hierarchical online failure prediction,” in Proc. 12th Int. ACM SIGSOFT Conf. Quality Softw. Archit., 2016, pp. 60–69.
31. P. Raj, A. Raman, and H. Subramanian, Architectural Patterns: Uncover Essential Patterns in the Most Indispensable System. Birming- ham, U.K.: Packt Publishing, Dec. 2017.
32. R. H. Reussner et al., Modeling and Simulating Software Architectures: The Palladio Approach. Cambridge, MA, USA: The MIT Press, 2016.
33. C. Richardson, “Microservice architecture patterns and best practices,” 2019. [Online]. Available: http://microservices.io/index.html
34. J. S. Rubinfeld, “Reference guide on multiple regression,” National Research Council, 3rd ed., Washington, DC, USA: The Nat. Academies Press, 2011, pp. 303–358, doi:10.17226/13163.
35. V. S. Sharma and K. S. Trivedi, “Architecture based analysis of performance, reliability and security of software systems,” in Proc. 5th Int. Workshop Softw. Perform., 2005, pp. 217–227.
36. T. Shozu, E. Jembere, and M. Adigun, “Performance evaluation of enterprise service buses towards support of cloud orchestration,” in Proc. Int. Conf. Comput. Eng. Netw. Secur., 2012, pp. 260–265.
37. J. Siegmund, N. Siegmund, and S. Apel, “Views on internal and external validity in empirical software engineering,” in Proc. 37th Int. Conf. Softw. Eng., 2015, pp. 9–19.
38. B. Spitznagel and D. Garlan, “Architecture-based performance analysis,” in Proc. Conf. Softw. Eng. Knowl. Eng., 1998, pp. 146–151.
O. Sukwong, A. Sangpetch, and H. S. Kim, “SageShift: Managing SLAs for highly consolidated cloud,” in Proc. IEEE Int. Conf. Comput. Commun., 2012, pp. 208–216.

K. S. Trivedi and A. Bobbio, Reliability and Availability Engineering: Modeling, Analysis, and Applications. Oxford, U.K.: Oxford Univ. Press, 2017.

A. Van Hoorn, A. Aleti, T. F. Düllmann, and T. Pitakrat, “ORCAS: Efficient resilience benchmarking of microservice architectures,” in Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops, 2018, pp. 146–147.

K. Vandikas and V. Tsiatsis, “Performance evaluation of an IoT platform,” in Proc. 8th Int. Conf. Next Gener. Mobile Apps, Services Technol., 2014, pp. 141–146.

L. Wang, X. Bai, L. Zhou, and Y. Chen, “A hierarchical reliability model of service-based software system,” in Proc. 33rd Annu. IEEE Int. Comput. Softw. Appl. Conf., 2009, vol. 1, pp. 199–208.

Z. Zheng and M. R. Lyu, “Collaborative reliability prediction of service-oriented systems,” in Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng., 2010, vol. 1, pp. 35–44.

Amirali Amiri received the bachelor’s degree in computer engineering from the Ferdowsi University of Mashhad, Iran, and the master’s degree in informatics from the Technical University of Munich, Germany. He is currently working toward the doctoral degree in the Software Architecture Group of Faculty of Computer Science, University of Vienna, Austria. His research focuses on software design and architecture, also fault injection, detection and prediction in service-based systems.

Uwe Zdun is currently a full professor for software architecture with the Faculty of Computer Science, University of Vienna, Austria. 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. He has published more than 240 peer-reviewed articles, and is co-author of three professional books.

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

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdll.