Analytical Modeling and Empirical Validation of Performability of Service- and Cloud-Based Dynamic Routing Architecture Patterns

1st Amirali Amiri, 2nd Uwe Zdun
University of Vienna, Faculty of Computer Science, Research Group Software Architecture
Vienna, Austria
firstname.lastname@univie.ac.at

3rd André van Hoorn
University of Hamburg, Department of Informatics, Software Engineering and Construction Methods
Hamburg, Germany
andre.van.hoorn@uni-hamburg.de

***Abstract*—Many dynamic routing architectural patterns are available, including distributed routing, e.g., using the sidecar pattern, or centralized routing, e.g., using event stores or service buses. Different Quality-of-Service (QoS) factors influence routing schemas and technology selection, such as performance, reliability, scalability, and control properties offered by the patterns. An analytical model can formalize the QoS factors and facilitate the architectural decision-making when changing the routing scheme, i.e., to more distributed or centralized routing. So far, the impact of these architectural patterns on performability, i.e., the overall performance of a system with impeded reliability, has not been extensively studied. This is important because deciding to increase performance, e.g., by parallel processing of requests, may lead to decreased reliability because of the added points of a crash. We propose an analytical performability model during component crashes. For the empirical validation of our proposed model, we ran an extensive experiment of 2412 hours of runtime on a private cloud infrastructure and Google Cloud Platform. The low prediction error of 1.75% indicates the high accuracy of our performability model. These results provide important insights when making architectural decisions regarding service- and cloud-based dynamic routing.**

***Index Terms*—Service- and Cloud-based, Dynamic Routing, Architectural Patterns, Performability, Analytical Model, Empirical Validation, Private and Public Clouds**

I. Introduction

DYNAMIC routing [9] is an important part of today's internet applications. Some dynamic routing decisions are simple, e.g., when using load balancing. However, some decisions are more complex, e.g., when checking for compliance with privacy regulations such as General Data Protection Regulation (GDPR)[1]. GDPR requires that companies comply with a regulation that European customer data must be stored on European servers. Dynamic routers can update the data flow path at runtime to ensure compliant data handling.

A typical cloud-native and distributed dynamic-routing architectural pattern is the sidecar pattern [11], [14]. The sidecar of each service handles the incoming and outgoing traffic [8] and performs the request flow routing for the services. In contrast, some architectural patterns use centralized routing to process the routing decisions. For instance, an API gateway, an event streaming platform [21], or any kind of central service bus [5] can be used. In addition, these two extremes are often combined, and multiple routers are used. For instance, consider an API gateway, event streaming platforms, and sidecars, all making routing decisions in a larger cloud-based application.

These architectural patterns are based on vastly different implementations; however, they all route or block requests. There is a possibility to change between these dynamic-routing patterns by changing the number of routers in a service- and cloud-based system. To do so automatically, we should monitor the QoS measures and make architectural decisions. So far, the impacts of dynamic routing architectural patterns on *performability* have not been specifically and extensively studied in the literature. Performability considers the effects of structural changes in a system, e.g., when a component crashes (impeded reliability) on the overall performance of the system [24]. This is an essential factor that needs to be considered because changing the routing architecture to improve performance, e.g., adding more routers for parallel processing of requests, may decrease system reliability as more points of a crash are introduced to a system.

To the best of our knowledge, there is a lack of an analytical model of performability specific to dynamic-routing patterns for architectural decision-making. This makes it hard to consider performability as a trade-off in design decisions for centralized or distributed dynamic routing, i.e., choosing the number of routers. We set out to answer the research questions:

RQ1: *What is the impact of service- and cloud-based dynamic-routing architectural patterns on performability, i.e., performance in the presence of component crashes?*

RQ2: *How well can we predict this impact when making architectural design decisions regarding performability?*

RQ3: *Are the model predictions generalizable to other cloud infrastructures regarding the performability of dynamic-routing applications?*

Supported by FWF (Austrian Science Fund) project API-ACE: I 4268.

[1] https://gdpr.eu

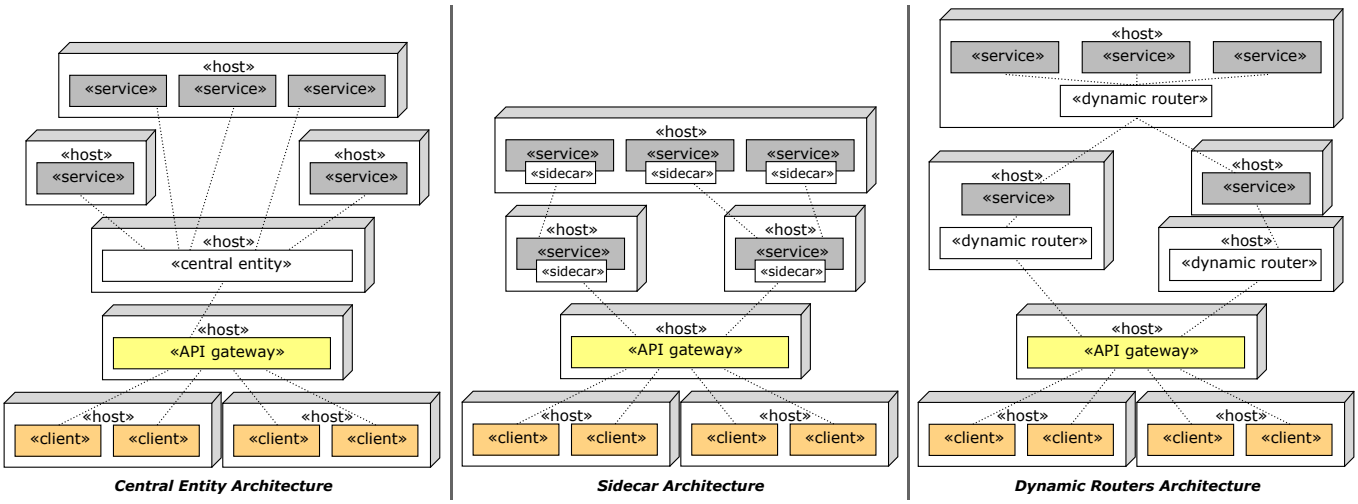


Fig. 1: Dynamic Routing Architecture Patterns

A major contribution of this research is a novel analytical model of performability concerning dynamic-routing architectural patterns. We model performability analytically when component crashes occur (i.e., impeded reliability). This analytical model makes our predictions generalizable and applicable to public cloud platforms. We use the average request processing time per router as a metric of performability, e.g., when parallel processing the incoming requests. This model considers the impact of dynamic routing when changing from one architectural pattern to another, i.e., changing the number of routers and their connected services to address **RQ1**.

Another contribution of our study is an extensive experiment of 2412 hours of runtime to evaluate our performability model. We studied 72 cases for different architectural patterns, i.e., centralized and distributed, with different architecture configurations, multiple numbers of cloud services, and request call frequencies. We performed 2400 hours of empirical measurements on our private cloud infrastructure and did a validation experiment of 12 hours on Google Cloud Platform (GCP)[2].

We calculated the prediction error of our model compared to our empirical measurements. Our results show that our prediction model of performability is highly accurate with a prediction error of 1.75% averaged over private and public clouds, which addresses **RQ2**. The validation experiment on GCP with a very low prediction error (0.66%) confirms that our model is generalizable and applicable to other dynamic-routing applications addressing **RQ3**.

The structure of the paper is as follows. Section II presents our analytical performability model. Section III explains the empirical validation of our model, and Section IV discusses the threats to the validity of our study. Section V compares to the related work. Finally, Section VI concludes the paper.

II. Analytical Model of Performability

To answer **RQ1**, we propose an analytical model of average request processing time per router. This model allows us to quantify the impact of dynamic-routing architectural patterns on performability in the presence of component crashes (impeded reliability). Table I presents the mathematical notations.

A. Dynamic-Routing Architectural Patterns

We studied three architectural patterns as shown in Figure 1. In the *central entity* architecture, a central router manages all request flow decisions. This pattern can be implemented, e.g., using an API gateway, an event store, an event streaming platform [21], or a service bus [5]. One benefit of this architectural pattern is that it is easy to manage, understand, and change as all control logic regarding request flow is implemented in one component. On the other hand, *Sidecars* [8], [11], [14] offer benefits whenever decisions need to be made structurally close to the service logic. One advantage of this pattern is that, compared to the *central entity* service, it is usually easier to

TABLE I: The Mathematical Notations Used in this Study

Notation	Description
T	Observed system time
n_{rout}	Number of routers
n_{serv}	Number of services
n_{crash}	Number of crash tests
P	Performability
$E[C_c]$	The expected number of crashes of a component c during T
CI	Crash interval
cf	Incoming call frequency
Com	Set of all components, i.e., routers and services
d_c	Expected average downtime after a component c crashes
CP_c	Crash probability of a component c every CI
Req	Number of client requests
R_{loss}	Total number of request losses
R^c_{loss}	Number of request losses per crash of component c
r/s	Requests per second
$MAPE$	Mean absolute percentage error
MAE	Mean absolute error
MSE	Mean squared error
MSE	Root mean squared error
$model_c$	Result of the model for the experiment case c
$empirical_c$	Measured empirical data for the experiment case c
$Cases$	Set of experiment cases
n_c	Length of $Cases$

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

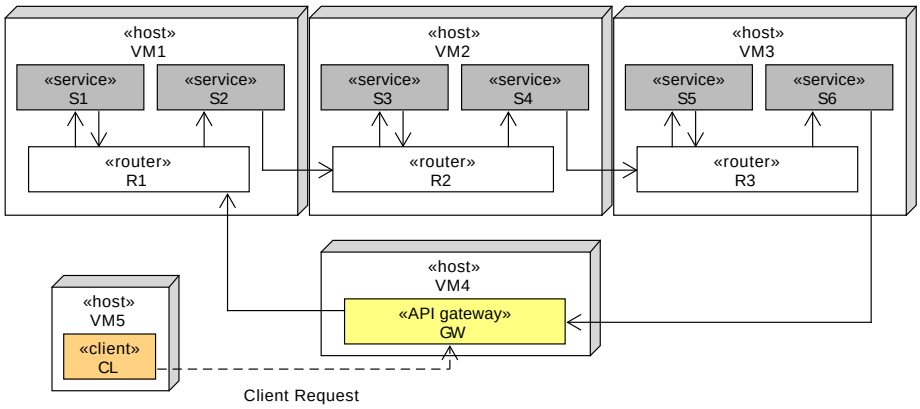


Fig. 2: Example Model Instance

implement sidecars since they require less complex logic to control the request flow. *Dynamic Routers* [9] can use local information regarding request routing amongst their connected services. For instance, if multiple services depend on one another as steps of processing a request, this pattern can facilitate dynamic routing.

B. Definition of Request Loss

Figure 2 shows an example model with three routers and six services. We define *client requests* as those that clients send to the system. When any component in the system crashes, client requests will not be processed fully. This results in the application not being responsive. We define *request loss* as the number of client requests not processed during a component crash. Let d_c be the expected average downtime after a component c crashes, and cf the incoming call frequency, i.e., the frequency with which a system receives requests. We define R^c_{loss} as the request loss per crash of a component c, which is given as the number of lost requests during downtime of component c.

$$R^c_{loss} = cf \cdot d_c \tag{1}$$

C. Bernoulli Process to Model Request Loss During Crashes

In this section, we model request loss based on Bernoulli processes [28]. We use this model to have an expected number of total request losses. This helps us to calculate the average request processing per router as a metric of performability. Note that we only model the crash of the routers and services in Figure 2. This is because we assume an API gateway is stable and reliable. Since we consider crashes at the container level, the services and routers have the same properties (e.g., probability of container crash), and can be abstracted into one concept. Thus, throughout the rest of the paper, we use the common term *components* for all routers and services.

Number of Crash Tests During an observed system time T, 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 past system runs, e.g., recorded in system logs. Note that many cloud systems run without being stopped. T should be interpreted as the interval during 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 during 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. This happens in real-world systems, e.g., when the Heartbeat pattern [10] or the Health Check API pattern [19] 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 of a system. Let n_{crash} be the number of times we check for a crash of components during T, i.e., the number of crash tests:

$$n_{crash} = \lfloor \frac{T}{CI} \rfloor \tag{2}$$

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 $CI > d_c$ (crash interval is greater than the downtime of a component c), all n_{crash} crash tests of a component c are independent of each other. This assumption is justifiable: 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. Thus, we can create a Bernoulli process of its crash tests for each component. The binomial distribution of each Bernoulli process gives us the number of successes, i.e., the number of times a component crashes during T. Hence, for each component, the expected value of the binomial distribution is the expected number of crashes of the component. Let CP_c be the crash probability of a component c every time we check for a crash derived from the failure distributions. Let $E[C_c]$ be the expected number of crashes of a component c during T:

$$E[C_c] = n_{crash} \cdot CP_c \tag{3}$$

Total Request Loss The total request loss R_{loss} is the sum of each component's request loss per crash. Let Com be the set of components, i.e., routers and services:

$$R_{loss} = \sum_{c \in Com} E[C_c] \cdot R^c_{loss} \tag{4}$$

that can be rewritten using Equations (1) to (3) as:

$$R_{loss} = \lfloor \frac{T}{CI} \rfloor \cdot cf \sum_{c \in Com} CP_c \cdot d_c \tag{5}$$

D. Performability Model

We model the average processing time of requests per router as a performability metric. This metric is important as it allows us to study the QoS factors, such as the efficiency of architecture configurations. Let Req be the total number of client requests, which is the call frequency cf multiplied by the observed time T:

$$Req = cf \cdot T \tag{6}$$

The number of processed requests is the total number of client requests minus the request loss. Let P be performability and n_{rout} the number of routers in a dynamic-routing application. The average processing time of requests per router is calculated so that we divide the total system time T over the processed requests and the number of routers:

$$P = \frac{T}{n_{rout} \cdot (Req - R_{loss})} \tag{7}$$

Using Equations (5) to (7), the average processing time is:

$$P = \frac{T}{n_{rout} \cdot cf \cdot \left(T - \lfloor \frac{T}{CI} \rfloor \cdot \sum_{c \in Com} CP_c \cdot d_c\right)} \tag{8}$$

Note that our analytical model has some assumptions outlined in the threats to validity (see Section IV).

III. Empirical Validation

In this section, we introduce an experiment to empirically validate the accuracy of our model to address **RQ2** and **RQ3**.

A. Experiment Planning

Goals We aim to empirically validate the accuracy of our performability model represented by Equation (8) in the presence of component crashes (impeded reliability). Based on our experiences from studies of microservice-based cloud architectures in our prior work (see, e.g., [3]) and the related literature, we decided on several experiment cases that are explained below with the rationale behind choosing them. We realized these architectures using a prototypical implementation, ran them on private and public cloud infrastructures, measured the empirical results, and compared them with our model. We follow the convention for the request flow shown by the example model in Figure 2. Client requests are sent to the API gateway, and internal requests are routed between routers and services. Also, for the sake of simplicity, we label the services and the routers incrementally from 1 and make the requests go through all of them linearly.

Technical Details We used private and public cloud infrastructures to validate the accuracy of our model.

Private Cloud Infrastructure: We used a private cloud with three physical nodes, each having two identical CPUs. Two cloud nodes host Intel® Xeon® E5-2680 v4 @ 2.40GHz[3] and the other one hosts the same processor family but version v3 @ 2.50GHz. The v4 and v3 versions have 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 uses VMware ESXi version 6.7.0 u2 hypervisor, has eight CPU cores, 60 GB system memory, and runs Ubuntu Server 18.04.01 LTS[4]. Docker[5] containerization is used to run the cloud services that are implemented in Node.js[6].

Validation Experiment on a Public Cloud: We used our private cloud to have control over the infrastructure. On a public cloud, other factors can influence the results, such as the parallel workload of other applications. To show that our approach can also be used on other cloud platforms, we empirically validated the analysis of our proposed model on Google Cloud Platform (GCP)[2]. On GCP, we used 7 instances of a compute-optimized C2 machine type[7], each with 4 vCPUs and 16 GB of memory. We duplicated our private cloud infrastructure on this machine and repeated the experiment for a validation run.

Load Generation: We utilized five desktop computers to generate load, each hosting an Intel®Core™i3-2120T CPU @ 2.60GHz with two cores and two physical threads per core. All desktop computers have 8 GB of system memory and run Ubuntu 18.10. They generate load using Apache JMeter[8] that sends Hypertext Transfer Protocol (HTTP) version 1.1[9] requests to the cloud nodes.

Experiment Cases According to Equation (8), the performability of a system can be influenced by several factors, such as the incoming call frequency (cf), downtime of each component (d_c), system observation time (T), crash interval (CI) and crash probability of each component (CP_c). Additionally, a system's number of components, i.e., routers and services, can influence the performability. We chose different levels for cf, CP_c, and the number of components to study their effects. We selected cf based on a study of related work as 10, 25, 50, and 100 requests per second (r/s). In many related studies (see, e.g., [6], [26]), 100 r/s (or even lower numbers) are chosen. Consequently, we selected this number as our highest bound and selected multiple proportions:

$$cf \in \{10, 25, 50, 100\} \tag{9}$$

Let n_{serv} and n_{rout} be the number of services and routers in a system, respectively. We chose the following values for

[3]https://www.intel.com/content/www/us/en/homepage.html
[4]https://www.ubuntu.com
[5]https://www.docker.com
[6]https://nodejs.org/en
[7]https://cloud.google.com/compute/docs/compute-optimized-machines
[8]https://jmeter.apache.org
[9]https://tools.ietf.org/html/rfc7230

these model elements, and the rationale is provided for each separately. The number of cloud services directly dependent on each other in a call sequence is usually rather low. As a result, we selected 3, 5, and 10 as values for the number of services in a call sequence.

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

We studied three representative architecture configurations (see Section II-A). Centralized routing, e.g., any kind of enterprise service bus [5], where there is only one router processing all incoming requests, i.e., $n_{rout} = 1$. Completely distributed routing, e.g., the sidecar pattern [11] using Envoy proxy[10], where there is one router per each service, i.e., $n_{rout} = n_{serv}$. Additionally, we studied a combination of these two extremes, where several routers are processing the requests, e.g., one router per each geographical region. We chose three routers, i.e., $n_{rout} = 3$, each deployed on a separate VM that processes the requests of services on the same machine.

$$n_{rout} \in \{1, 3, n_{serv}\} \tag{11}$$

We simulated a node crash by generating a random number for each cloud component separately, i.e., the services and routers. 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 downtime interval of three seconds. To study different crash profiles, we selected two levels of crash probabilities for each component, i.e., a low 0.1% and a high 0.5%, each time we checked for a crash. These profiles are based on our experiments in previous work [3] and are akin to the related scientific and industrial studies. Note that the crash probability was uniform for all components, and we performed the experiment twice, once with the low and once with the high crash probability.

$$CP_c \in \{0.1\%, 0.5\%\} \quad \forall \ c \in C \tag{12}$$

Data Set Preparation We instantiated the architectures for each empirical case. We ran the experiment for exactly ten minutes, i.e., 600 seconds (excluding setup time), during which we checked for a crash for all routers and services simultaneously every 15 seconds, resulting in 40 crash tests for each component during our experiment based on Equation (2).

$$T = 600\,s \tag{13}$$

$$CI = 15 \tag{14}$$

$$n_{crash} = 40 \tag{15}$$

We logged the number of processed requests and calculated the performability based on Equation (8).

As outlined in the previous section, we studied four levels of cf, three levels of n_{serv}, three levels of n_{rout}, and two levels of CP_c, resulting in 72 experiment cases. A single run of our experiment takes exactly 12 hours (72×10 minutes) of runtime. Since our model revolves around expected values in a Bernoulli process, we repeated this process 200 times on our private cloud infrastructure, i.e., 2400 hours of runtime. We reported the arithmetic mean of the results. Moreover, for public cloud validation, we performed an experiment run of 12 hours on GCP. Overall, we had an extensive empirical validation of 2412 hours of runtime (excluding setup time).

Architecture Configurations As in the example model in Figure 2, we utilized one virtual machine exclusively, with only one Docker container inside, to run the API gateway. We distributed the cloud services on a separate container amongst three VMs. The services were distributed so that all virtual machines have the same number of cloud services (with a maximum difference of one service). However, the placement of the routers on hosts differs from the example model. We placed the router in a Docker container exclusively on one VM for centralized routing ($n_{rout} = 1$). For distributed routing ($n_{rout} = 3$), we used three exclusive VMs, each with only one router container. Finally, for the sidecar architectural pattern [11] ($n_{rout} = n_{serv}$), we placed each sidecar in a separate container on the same VM, on which its directly linked service resides.

Specific Models (Illustrative Sample Case) As we had a uniform crash probability and downtime of each component in our experiment, we can rewrite the performability given by Equation (8) in ms as follows:

$$P = \frac{1000}{n_{rout} \cdot cf \left(1 - (n_{rout} + n_{serv}) \cdot \lfloor \frac{T}{CI} \rfloor \cdot \frac{1}{T} \cdot CP_c \cdot d_c\right)} \tag{16}$$

Using our experiment values reported in this section, we have:

$$P = \frac{1000}{n_{rout} \cdot cf(1 - 0.2 \cdot CP_c(n_{serv} + n_{rout}))} \tag{17}$$

To clarify, we study an illustrative sample case presented in Figure 2. In this example configuration, we have six services, i.e., $n_{serv} = 6$, and three routers, i.e., $n_{rout} = 3$. We consider a case where each component *c* has a uniform crash probability of $CP_c = 0.5\%$ and is under stress with an incoming call frequency of $cf = 100\ r/s$. The performability of such a system, i.e., the average request processing time per router, can be predicted using Equation (17) as:

$$P = 3.36\ ms \tag{18}$$

Methodological Principles of Reproducibility We followed the eight principles of reproducibility introduced in [17]:

- *Repeated experiments:* Each experiment case is repeated with precise values (see this section).
- *Workload and configuration coverage:* We covered 72 empirical cases and modeled the probabilistic behavior of performability in Section II.
- *Experimental setup description:* Our experimental setup is reported in Section III-A.
- *Open access artifact:* Our code and data are published as an open access data set to support replicability[11].

[10]https://www.envoyproxy.io

[11]https://zenodo.org/record/10022346, doi:10.5281/zenodo.10022346

- *Result description of measured performance:* We described our results in Section III-B.
- *Statistical evaluation:* The prediction error of our approach is reported in Section III-C.
- *Measurement units:* All units are reported.
- *Cost:* We had a private cloud setting. For the public cloud validation, we used the free trial offered by GCP[12].

B. Experiment Results

We present the predicted results of our analytical performability model and the empirical measurements. In Table II, we grouped our results for the crash probability of $CP_c = 0.1\%$ and $CP_c = 0.5\%$. We report the mean performability on the GCP public cloud and our private cloud over 200 experiment runs. As predicted by our model (see Equation (17)) and confirmed by our empirical measurements, the call frequency cf and the number of routers affect the performance conversely. When taking the same configuration, i.e., keeping n_{serv} and n_{rout} constant, increasing cf results in a lower average processing time per router in all cases. Moreover, the more routers in a dynamic-routing application, the lower the performability. This is expected as performability is defined in our study as the average request processing time per router.

Our performability model predictions compared to the empirical measurements of the GCP public cloud and the private cloud infrastructure are shown in Figure 3 (only $CP_c = 0.5\%$ for space reasons). As can be seen, all of the empirical measurements are very close to the predictions. When we investigate the cases with $CP_c = 0.1\%$ reported in Table II, the same trend of experiment values being very close to the model predictions are observed. However, we see a slight decrease in performability with the central routing ($n_{rout} = 1$) only on our private cloud. This is likely because the virtual machine hosting the one router is overloaded with processing buffered requests. As can be read in Table II, having a higher number of routers in DR ($n_{rout} = 3$) and SA ($n_{rout} = n_{serv}$) solves this performability decrease. So we further investigate the prediction error of our model to ensure its high accuracy.

C. Evaluation of the Prediction Error

We measure the accuracy of our model predictions. The prediction error is calculated using the four error measurements commonly used in the cloud quality-of-service research [28], i.e., Mean Absolute Percentage Error (MAPE), Mean Absolute Error (MAE), Mean Squared Error (MSE) and Root Mean Squared Error (RMSE). The error measurements are calculated in terms of performability P. Let $model_c$ and $empirical_c$ be the result of the model and the measured empirical data for an experiment case c, $Cases$ is the set of experiment cases, and n_c is the length of $Cases$. In the error measurements, we average over the $n_c = 72$ experiment cases (see Section III-A). We use the following formulae for our error measurements:

$$MAPE = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \left| \frac{model_c - empirical_c}{empirical_c} \right| \quad (19)$$

$$MAE = \frac{1}{n_c} \cdot \sum_{c \in Cases} |model_c - empirical_c| \quad (20)$$

$$MSE = \frac{1}{n_c} \cdot \sum_{c \in Cases} (model_c - empirical_c)^2 \quad (21)$$

$$RMSE = \sqrt{MSE} \quad (22)$$

Table III presents the prediction error of our performability model (using the values reported in Table II). Our model has a MAPE prediction error of 0.66% on GCP and 2.85% on our private infrastructure over 200 experiment runs. Averaged over public and private clouds (2412 hours of experiment), we have a very low error rate of 1.75%. Other low error measurements also confirm the high accuracy of our prediction model that answers **RQ2**. Given the 30.0% target prediction accuracy commonly used in the cloud performance research [13], the prediction error of our approach on GCP is more than reasonable to answer **RQ3**. That is, our model is generalizable and applicable to other cloud platforms.

IV. Threats to Validity

As in all empirical research, there are several threats to the validity and limitations of our study that we discuss in this section based on the four threat types by Wohlin et al. [30].

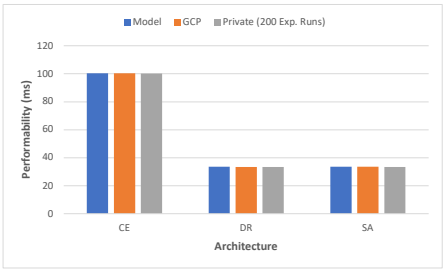
A. Consttruct validity

This threat concerns a measurement's accurate representation of the intended construct. We injected crashes to simulate real-world crash behavior at a given probability to measure performability. While this is a commonly taken approach (see Section V), a threat remains that measuring request loss based on these crashes might not be well representative of real-world systems. For example, request loss is also influenced by cascading effects of crashes beyond a single call sequence [16] that are not covered in our experiment. More research is needed to exclude this threat, probably with real-world systems and crashes. Moreover, component overload is also not considered, which can influence the system's performability, e.g., when an overloaded component is non-responsive to incoming requests. As the self-adaptivity of cloud-based systems is mature in the industrial tools, e.g., using Google Kubernetes[13] autoscaling, we believe this threat to be small.

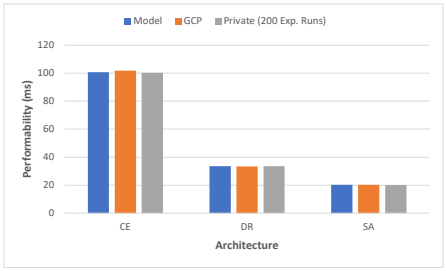
For the sake of simplicity, we made some assumptions when designing our analytical model of performability, which is common when modeling a real-world phenomenon. We did not consider the requests that were in the system already at the time of a crash of a component as we restarted all containers for each experiment run to increase internal validity (see the next section). We assumed the crash probabilities were known

[12]https://cloud.google.com/free/docs/free-cloud-features

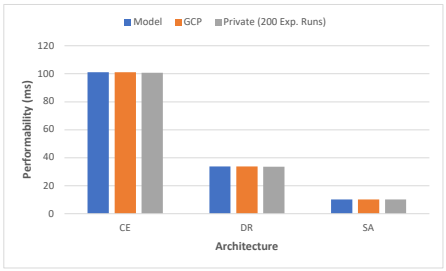
[13]https://kubernetes.io



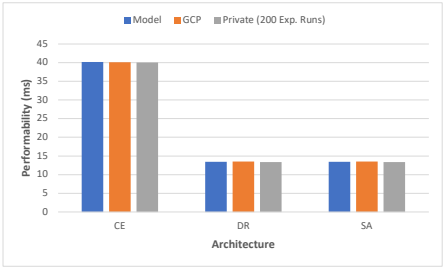
(a) $n_{serv} = 3$
$cf = 10$ r/s



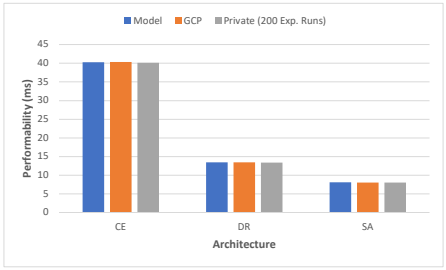
(b) $n_{serv} = 5$
$cf = 10$ r/s



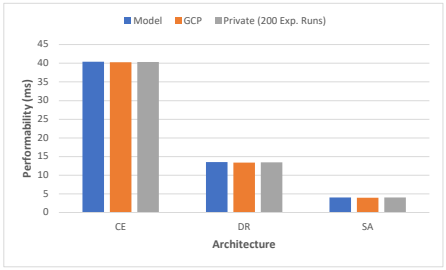
(c) $n_{serv} = 10$
$cf = 10$ r/s



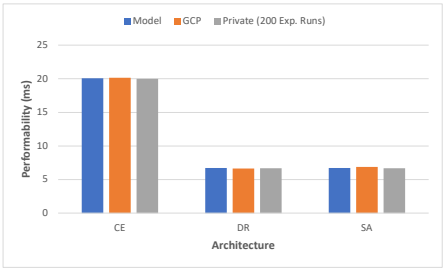
(d) $n_{serv} = 3$
$cf = 25$ r/s



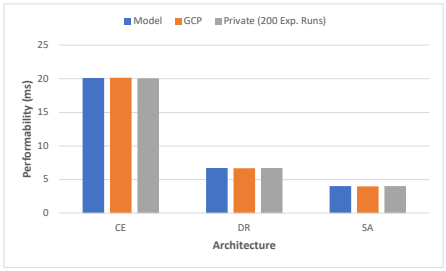
(e) $n_{serv} = 5$
$cf = 25$ r/s



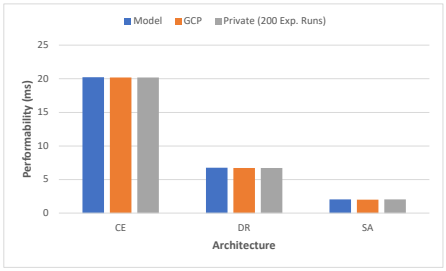
(f) $n_{serv} = 10$
$cf = 25$ r/s



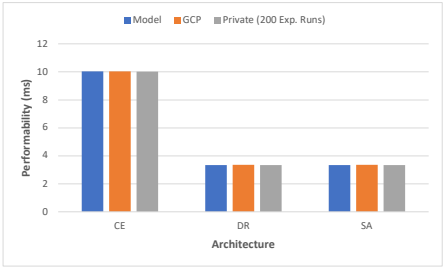
(g) $n_{serv} = 3$
$cf = 50$ r/s



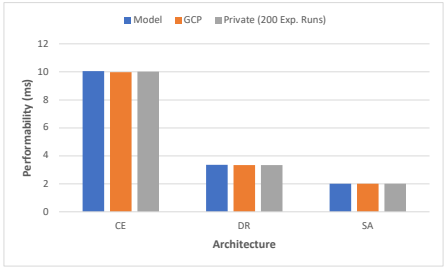
(h) $n_{serv} = 5$
$cf = 50$ r/s



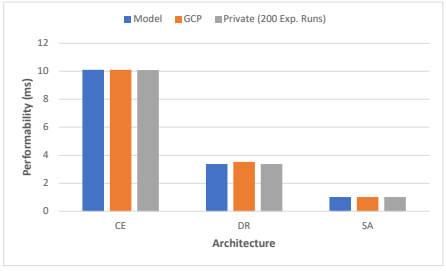
(i) $n_{serv} = 10$
$cf = 50$ r/s



(j) $n_{serv} = 3$
$cf = 100$ r/s



(k) $n_{serv} = 5$
$cf = 100$ r/s



(l) $n_{serv} = 10$
$cf = 100$ r/s

Fig. 3: Plots of Model Predictions and Empirical Measurements ($CP_c = 0.5\%$)

TABLE II: Model Predictions and Empirical Measurements of Performability (ms)

n_{rout}	n_{serv}	cf	$CP_c = 0.1\%$	$CP_c = 0.5\%$	$CP_c = 0.1\%$	$CP_c = 0.5\%$	$CP_c = 0.1\%$	$CP_c = 0.5\%$
			Model		**GCP**		**Private (200 Exp. Runs)**	
1	**3**	**10**	100.08	100.40	102.02	100.33	117.27	100.13
		25	40.03	40.16	40.14	40.13	45.75	40.04
		50	20.02	20.08	20.04	20.15	19.95	20.01
		100	10.01	10.04	10.80	9.98	9.98	10.02
	5	**10**	100.12	100.60	99.73	101.83	116.60	100.33
		25	40.05	40.24	39.89	40.36	45.66	40.11
		50	20.02	20.12	19.99	20.14	19.96	20.06
		100	10.01	10.06	10.69	9.98	9.98	10.02
	10	**10**	100.22	101.11	100.23	101.01	116.80	100.80
		25	40.09	40.44	39.89	40.25	44.37	40.37
		50	20.04	20.22	19.95	20.18	24.69	20.19
		100	10.02	10.11	10.04	10.10	12.55	10.08
3	**3**	**10**	33.37	33.53	33.24	33.43	33.29	33.43
		25	13.35	13.41	13.40	13.53	15.20	13.37
		50	6.67	6.71	6.69	6.65	6.65	6.68
		100	3.34	3.35	3.34	3.36	3.33	3.34
	5	**10**	33.39	33.60	33.24	33.43	33.31	33.53
		25	13.35	13.44	13.30	13.45	13.33	13.42
		50	6.68	6.72	6.68	6.68	6.66	6.71
		100	3.34	3.36	3.34	3.34	3.33	3.35
	10	**10**	33.63	33.77	33.63	33.80	33.32	33.68
		25	13.37	13.51	13.38	13.38	13.33	13.47
		50	6.68	6.75	6.68	6.72	6.67	6.72
		100	3.34	3.38	3.35	3.53	3.33	3.37
n_{serv}	**3**	**10**	33.37	33.53	33.44	33.58	33.28	33.42
		25	13.35	13.41	13.41	13.48	13.32	13.38
		50	6.67	6.71	6.65	6.87	6.66	6.68
		100	3.34	3.35	3.34	3.36	3.33	3.34
	5	**10**	20.04	20.20	20.05	20.16	19.99	20.13
		25	8.02	8.08	8.03	8.07	8.85	8.05
		50	4.01	4.04	3.99	4.01	3.99	4.02
		100	2.00	2.02	1.99	2.02	2.00	2.01
	10	**10**	10.04	10.20	10.02	10.31	10.97	10.21
		25	4.02	4.08	3.99	3.99	5.00	4.08
		50	2.01	2.04	2.01	2.03	2.00	2.04
		100	1.00	1.02	1.00	1.01	1.30	1.02

TABLE III: Predictions Errors of the Performability Model

	GCP	**Private (200 Exp. Runs)**	**overall**
$MAPE\,(\%)$	0.66	2.85	1.75
$MASE$	0.13	1.11	0.61
MSE	0.11	12.87	6.49
$RMSE$	0.33	3.59	2.55

based on the observed system logs in the past and checked for crashes. This is a common practice in real-world systems, e.g., when the Heartbeat pattern [10] or the Health Check API pattern [19] is used for checking system health. Moreover, we considered a generic downtime of the components and did not study metrics such as mean time to failure and recovery. This paper laid the foundation for our future work, where we model more real-world aspects.

B. Internal validity

Internal validity concerns factors that affect the independent variables concerning causality. We collected extensive data to validate our model on public and private cloud infrastructures. Still, we did so in limited experiment time using simulated crashes by stopping Docker containers. However, research observing real-world cloud-based systems for a longer period would be needed to confirm that no other factors influence the measurements. One such factor is other workloads being processed simultaneously on the same infrastructure. We studied this factor by running a validation experiment on Google Cloud Platform[2] and showed that our model is applicable (see Section III-C). As we used the standard technology stack offered by most cloud providers (see Section III-A), we believe our results represent the service- and cloud-based applications.

C. External validity

External validity concerns threats that limit the ability to generalize the results beyond the experiment. We designed our approach with generality in mind and explained how architects can specify our model to their needs (see Section III). Although we evaluated our approach by designing a representative experiment and measuring empirical data, the threat remains that evaluating based on another infrastructure may lead to different results. To mitigate this thread, we validated our measurements on GCP infrastructure and showed that our results are applicable and generalizable (see Section III-C).

Moreover, the results might not be generalizable beyond the given empirical cases of 10-100 r/s and call sequences of length 3-10. As this covers a wide variety of loads and call sequences in cloud-based applications, the impact of this threat should be limited. Also, the load was constant and not time-varying. We plan to study bursty load profiles in future work. A related threat is that we implemented our model instances with Node.js, not using off-the-shelf implementations, e.g., the Envoy proxy[14]. We did so to have a comparable infrastructure and to avoid technological impacts on our results.

D. Conclusion Validity

This threat concerns factors that affect the ability to conclude the relations between treatments and study outcomes. 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 interpretability in our research context. To mitigate the threat that this statistical method might have issues, we double-checked three other error measures, i.e., MAE, MSE, and RMSE. These measures similarly confirmed the high accuracy of our prediction model (see Section III-C).

V. Related Work

In this section, we present the related work of our research.

A. Studies on Performability of Systems

Ahamad and Ratneshwer [1] provide a review on the performability of Safety-Critical Systems (SCS). They study the available approaches and the metrics to evaluate the performability of SCS. Moreover, they define performance and reliability challenges in studying the SCS. This study is related to our work presenting the state of the art in performability studies. However, in contrast to our work, it does not provide an analytical model of performability and its empirical validation to improve the state of the art. Mo et al. [15] study the performability analysis of multi-state sliding window systems. Like our study, they propose an analytical approach based on multivalued decision diagrams. They analyze this analytical approach in multiple case studies. Lisnianski et al. [12] present a Markov multi-state model for large-scale, highly responsive distributed systems. Similar to our work, they provide an analytical performability model and present a short-term analysis to prevent performance and reliability decreases. Unlike our research, none of the above works provide extensive empirical data supporting the accuracy of their proposed models.

A particular related work is [27], in which Torquato et al. study the migration of virtual machines of a cloud-based system in the presence of workload. Like our work, they provide a modeling framework to support virtualized environment management decisions regarding the performability of a system. Particularly recent work is [7], in which Di Mauro et al. study containerized network applications. They use queuing network theory and consider each container as a queuing node. Like our research, they performed a cloud-based experiment to evaluate their performability model. In contrast to the above studies, our research is specific to dynamic routing and provides an analytical performability model with extensive empirical data. To the best of our knowledge, this has not been considered in the literature concerning dynamic-routing architectural patterns.

[14]https://www.envoyproxy.io/

B. Architecture-Based Performance Analysis and Prediction

Several approaches perform architecture model-based performance analysis or prediction. Spitznagel and Garlan [25] present a general architecture-based model for performance analysis based on queueing network theory. Sharma and Trivedi [22] present an architecture-based unified hierarchical model for software reliability, performance, security, and cache behavior prediction. Petriu et al. [18] 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 [4], [20] allows precise component modeling with relevant factors for performance properties and contains a simulation framework for performance prediction. Like our study, those works focus on supporting architectural design or decision-making.

C. Performance Analysis: Internet of Things

Vandikas et al. [29] 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. The authors evaluated the system using two sets of tests: The first one creates a total of 105 HTTP POST requests that simulate a different number of users (producers). Second tests were done using a simple Java client to generate various consumers to see the impact on the system. In contrast to our work, dynamic data routing is not considered in this article. 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.

D. Performance Analysis: Enterprise Service Buses

Several existing works compare the performance of Enterprise Service Buses (ESB). This is related to our work because ESBs provide a means for the 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. [2] evaluate the performance of the three open-source ESBs, i.e., Mule, WSO2 ESB, and Service Mix. The performance is measured based on mean response time and throughput for proxying, content-based routing, and data mediation. However, the test scenarios only consider client communications and

a single web service. In contrast, our work also considers communication paths that involve the composition of multiple services and routing decisions. Shezi et al. [23] 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 several banking services to find the best loan quote. In contrast to our work, other routing architectures are not considered.

VI. Conclusions and Future Work

In this paper, we studied the performability, i.e., performance in the presence of component crashes (impeded reliability) of service- and cloud-based systems. For **RQ1**, we proposed an analytical performability model. This model considers the average request processing time per router in the presence of component crashes. We used Bernoulli processes [28] to predict the number of request losses during crashes. Having this information, we calculated the number of processed requests during the system observation time. For **RQ2**, we designed an extensive experiment of 2412 hours of runtime (excluding setup time) to validate our analytical model empirically. The prediction error is 1.75%, indicating our performability model's high accuracy. For **RQ3**, we ran a validation experiment on Google Cloud Platform[2] and showed that our predictions are applicable and generalizable (see Section III-C). We double-checked the accuracy with three other error measurements that confirmed the results.

This paper's proposed model represents service- and cloud-based systems. Our performability model can be used in other environments and applications to give insight to architects when making architectural design decisions regarding dynamic routing. For our future work, we plan to use this model in a self-adaptive architecture that automates this decision-making process based on an optimization analysis.

References

[1] S. Ahamad and Ratneshwer. Some studies on performability analysis of safety critical systems. *Computer Science Review*, 39:100319, 2021.

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

[3] A. Amiri, U. Zdun, and A. van Hoorn. Modeling and empirical validation of reliability and performance trade-offs of dynamic routing in service- and cloud-based architectures. In *IEEE Transactions on Services Computing (TSC)*, 2021.

[4] 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.

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

[6] 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.

[7] M. Di Mauro, G. Galatro, M. Longo, F. Postiglione, and M. Tambasco. Performability analysis of containerized ims through queueing networks and stochastic models. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–8, 2022.

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

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

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

[11] 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.

[12] A. Lisnianski, E. Levit, and L. Teper. Short-term availability and performability analysis for a large-scale multi-state system based on robotic sensors. *Reliability Engineering and System Safety*, 2021.

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

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

[15] Y. Mo, L. Xing, L. Zhang, and S. Cai. Performability analysis of multi-state sliding window systems. *Reliability Engineering and System Safety*, 202:107003, 2020.

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

[17] 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*, 2019.

[18] 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.

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

[20] 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.

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

[22] 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. Association for Computing Machinery, 2005.

[23] 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.

[24] R. Smith, K. Trivedi, and A. Ramesh. Performability analysis: measures, an algorithm, and a case study. *IEEE Transactions on Computers*, 37(4):406–417, 1988.

[25] 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.

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

[27] M. Torquato, P. Maciel, and M. Vieira. Model-based performability and dependability evaluation of a system with vm migration as rejuvenation in the presence of bursty workloads. *Journal of Network and Systems Management*, 30(1):3, 2021.

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

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

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