Impact of Service- and Cloud-Based Dynamic Routing Architectures on System Reliability*

Amirali Amiri[1], Uwe Zdun[1], Georg Simhandl[1], and André van Hoorn[2]

[1] University of Vienna, Austria {firstname.lastname}@univie.ac.at
[2] University of Stuttgart, Germany van.hoorn@informatik.uni-stuttgart.de

Abstract. Various kinds of dynamic routing architectures are used in today's service- and cloud-based architectures, including sidecar-based routing, routing through a central entity such as an event store, or architectures with multiple dynamic routers. We propose an analytical model of request loss during router and service crashes, as well as an empirical validation of that model. The comparison of the empirical data to the predicted values by our model shows a low enough and converging error rate for using the model during system architecting. Our model predicts that, having the same crash probability, decentralized routing results in losing a higher number of requests in comparison to more centralized approaches. To the best of our knowledge, our study is the first to empirically study the reliability trade-off in such architectural decisions.

1 Introduction

Many distributed system architecture patterns [3, 10, 15] have been suggested for dynamic routing [8]. Some dynamic routing architectures require a single dynamic request routing decision, e.g., when using load balancing. More complex request routing decisions or combinations of 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.

In our prior work [1], we studied representative service- and cloud-based system architecture patterns for dynamic request routing. A typical cloud native architecture pattern is the *sidecar* pattern [10,12] in which the sidecar of each service handles incoming and outgoing traffic [6]. In contrast, a *central entity*, e.g., an API Gateway, an event streaming platform [15], or any kind of central service bus [3], can be used to process the request routing decisions. These two extremes are often combined and multiple routers are used; this is called *dynamic routers* in this paper. Consider an API Gateway, two event streaming platforms, and a number of sidecars, all making routing decisions in a cloud-based architecture.

At present, the impacts of such architectures and their different configurations on system reliability have not been studied. More is known about other

* This work was supported by FWF (Austrian Science Fund), project ADDCompliance: I 2885-N33; FFG (Austrian Research Promotion Agency), project DECO no. 846707; Baden-Württemberg Stiftung, project ORCAS.

qualities relevant for this decision. For instance, our prior work [1] has shown that more distributed approaches for dynamic data routing offer a better performance compared to more centralized solutions. As reliability is a core consideration in service and cloud architectures [14], a reasonably accurate failure prediction for the feasible architecture design options in a certain design situation would help architects to better design system architectures considering quality trade-offs.
RQ1: *What is the impact of choosing a dynamic routing architecture, in particular central entity, sidecar-based, or dynamic routers, on system reliability?*
RQ2: *How can we predict this impact when making architectural design decisions regarding system reliability?*

We model 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. To validate our analytical model, we designed an experiment in which we studied 36 representative experimental cases (i.e., different experiment configurations) for the three kinds of architectures Our results show that the error is constantly reduced with a higher number of experimental runs, converging at a prediction error of 8.1%. Given the common target prediction accuracy of up to 30% in the cloud performance domain [11] these results are more than reasonable. Our model predicts and our experiment confirms that more decentralized routing results in losing a higher number of requests than more centralized approaches.

2 Related Work and Background

2.1 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 [17]. 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., a Markov model [4].

Empirical Reliability or Resilience Assessment. Today many software organizations use large-scale experimentation in production systems to assess the reliability of their systems, which is called chaos/resilience engineering [2]. A crucial aspect in resilience assessment of software systems is efficiency [13]. 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 [18].

Service-Specific Reliability Studies. Some related works introduce service-specific reliability models, e.g., Wang et al. [19] propose a discrete time Markov chain model for analyzing system reliability based on constituent services. Grassi and Patella [7] propose an approach for reliability prediction that considers the

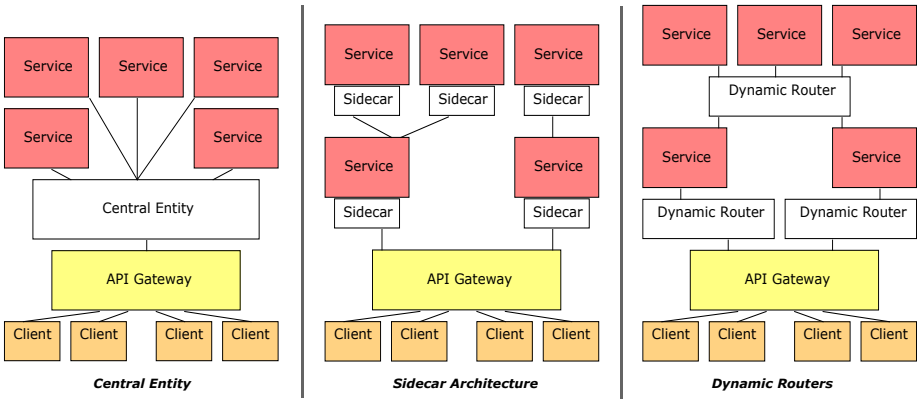

Fig. 1: Dynamic Routing Architecture Patterns (adapted from [1])

decentralized and autonomous nature of services. However, none of these approaches studies and compares major architecture patterns in service and cloud architectures; they are based on a very generic model about the notion of service.

2.2 Background: Dynamic Routing Architecture Patterns

Central Entity (CE). In a CE architecture, as shown in Figure 1, 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 flow 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. CE can be implemented utilizing an API Gateway, an event store, an event streaming platform [15], or a service bus [3].

Sidecar Architecture (SA). Figure 1 presents an SA example. Sidecars [6, 10, 12] 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; however, it is not always possible to add *sidecars*, e.g., when services are off-the-shelf products.

Dynamic Routers (DR). Figure 1 shows a specific dynamic router [8] configuration. 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 the dynamic routing; nonentheless, *dynamic routers* introduce an implementation overhead regarding control logic, deployment and so on since they are usually distributed on multiple hosts.

3 Model of Request Loss During Crashes

We use the common term *router* for all request flow control logic.

3.1 Definition of Internal and External Loss

In Figure 1 *routers* and *services* send *internal requests* amongst one another to complete the processing of one *external request* received from *clients*. In case of a crash, *external requests* will not be processed fully. We define external and internal loss as the number of lost *external* and *internal requests*, respectively.

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

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

Note that IR_T and n_c^{exec} need to be parameterized based on the application. An example of this parameterization is given in Section 4.

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

$$EL_c = d_c \cdot cf \tag{2}$$

3.2 Bernoulli Process to Model Request Loss

In this section, we model request loss based on Bernoulli processes [17]. We only model the crash of *routers* and *services* in Figure 1 because we assume an *API Gateway* is stable and reliable. Moreover, a crash of a *Client* results in *external requests* not being generated; as a result, *external requests* are not lost. Hence, from now on, we use the common term *components* for all *routers* and *services*.

Number of Crash Tests. During T, all *components* can crash with certain failure distributions. 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). 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 [9] is used for checking system health. Let n_{crash} be the number of times we check for a crash of *components* during T, i.e., the number of crash tests:

$$n_{crash} = \lfloor \frac{T}{CI} \rfloor \tag{3}$$

Expected Number of Crashes. Each crash test is a Bernoulli trial in which success is defined as "*component* crashed". Assuming $CI > d_c$ (justifiable because when a component crashes it cannot crash again) all n_{crash} crash tests of a *component* c are independent. The binomial distribution of each Bernoulli process gives us the number of successes. Let P_c be the crash probability of a *component* c every time we check for a crash and $E[C_c]$ the expected number of its crashes, i.e., the expected value of its binomial distribution during T:

$$E[C_c] = n_{crash} \cdot P_c \tag{4}$$

Total Internal and External Loss. The total *internal loss* (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*. Using Equations (1) to (4):

$$IL_T = \sum_{c \in C} E[C_c] \cdot IL_c = \lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in C} P_c \cdot d_c \cdot (IR_T - n_c^{exec}) \quad (5)$$

The total *external loss* (EL_T) is the sum of external loss per crash of each *component*. Using Equations (2) to (4):

$$EL_T = \sum_{c \in C} E[C_c] \cdot EL_c = \lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in C} P_c \cdot d_c \quad (6)$$

Total Number of Crashes. The total number of crashes (C_T) is the sum of the expected number of crashes of each *component*. Using Equations (3) and (4):

$$C_T = \sum_{c \in C} E[C_c] = \lfloor \frac{T}{CI} \rfloor \cdot \sum_{c \in C} P_c \quad (7)$$

4 Empirical Validation

4.1 Experimental Planning

Goals. We aim to empirically validate our model's accuracy with regard to the number of crashes as well as the total *external* and *internal loss* represented by Equations (5) and (6). 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.

Technical Details. We used a private cloud with three physical nodes, each having two identical Intel® Xeon® E5-2680 CPUs. On top of the cloud nodes we installed Virtual Machines (VMs) with eight CPU cores and 60 GB system memory running Ubuntu Server 18.04.01 LTS. Docker containerization is used to run the cloud services which are implemented in Node.js. We utilized five desktop computers to generate load, each hosting an Intel®Core™i3-2120T CPU @ 2.60GHz, 8GB of system memory which run Ubuntu 18.10. They generate load using Apache JMeter which sends HTTP version 1.1 requests to the cloud nodes.

Specific Model Formulae. In our example configurations each *service* receives an *internal requests*, processes it and sends it back either to a *router* or the *API gateway*, so we can calculate IR_T based on the number of services (n_{serv}):

$$IR_T = 2n_{serv} + 1 \quad (8)$$

In order to calculate n_c^{exec}, we need to differentiate between *service* and *router* crashes. In case of a *service* crash, all *internal requests* up until the last *router* will be executed. Let $s_{crashed}$ be the label number of the crashed *service*:

$$n_c^{exec} = 2s_{crashed} - 1 \quad (9)$$

In case of a *router* crash, we need to know the allocation of routers (A) which is a set indicating the number of directly linked *services* of each *router*. Let $r_{crashed}$ be the label number of the crashed *router*:

$$n_c^{exec} = 2 \sum_{r=1}^{r_{crashed}} A_{r-1} \tag{10}$$

Experimental Cases. 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, e.g., [5,16], as 10, 25, 50, and 100 requests per second. Based on our experience and a survey on existing cloud applications in the literature and industry [1], the number of cloud services which are directly dependent on each other in a call sequence is usually rather low. As a result, we chose 3, 5, and 10 as values for n_{serv}. We simulated a node crash by separately generating a random number for each cloud component. 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 $n_{crash} = 40$ (Equation (3)). Each component had a uniform crash probability of 0.5%; akin to the related works we chose a relatively high crash probability to have a high enough likelihood to observe a few crashes during T.

Data Set Preparation. For each experimental case, we instantiated the architectures and ran the experiment for exactly ten minutes (excluding setup time). We studied three architectures, three levels of n_{serv} and four levels of cf, resulting in a total of 36 experimental cases; therefore, a single run of our experiment takes exactly six hours (36×10 minutes) of runtime. Since our model revolves around expected values in a Bernoulli process, we repeated this process 200 times (1200 hours of runtime) and report the arithmetic mean of the results[3].

4.2 Results

Experimental Results Analysis. Based on Equation (5), IL_T is a model element that incorporates crashes of all *components* and it includes all model views, e.g., architecture configurations, expected average downtime, etc. Therefore, we conduct our analysis mainly based on IL_T. It can be observed from Table 1 that when we keep n_{serv} constant, increasing cf results in a rise of EL_T (predicted by Equation (6)) in all cases, which leads to a higher value of IL_T (Equation (5)).

Since in our experiment, we instantiated the DR architecture with three dynamic routers, it is interesting to consider the experimental case of $n_{serv} = 3$. In this case, SA and DR have the same number of components, i.e., routers and services. Note that SA uses a *sidecar* per each cloud 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

[3] The data of this study is published as an open access data set for supporting replicability: `https://zenodo.org/record/4008041`, doi:10.5281/zenodo.4008041

Arch.	n_{serv}	cf	C_T	EL_T	IL_T	C_T	EL_T	IL_T	$\sigma(IL_T)$
			Model			Experiment			
CE	3	10	0.800	24.000	114.000	0.760	23.395	98.960	118.552
		25	0.800	60.000	285.000	0.620	47.435	228.975	292.389
		50	0.800	120.000	570.000	0.705	106.370	480.235	608.635
		100	0.800	240.000	1140.000	0.725	218.130	1045.000	1216.765
	5	10	1.200	36.000	246.000	1.165	36.405	236.575	236.536
		25	1.200	90.000	615.000	1.110	85.400	608.040	574.267
		50	1.200	180.000	1230.000	1.115	172.085	1155.550	1173.295
		100	1.200	360.000	2460.000	1.040	317.585	2223.655	2101.272
	10	10	2.200	66.000	786.000	1.920	62.000	720.190	616.778
		25	2.200	165.000	1965.000	2.125	171.290	2063.305	1711.931
		50	2.200	330.000	3930.000	2.160	344.765	4223.665	3458.119
		100	2.200	660.000	7860.000	1.960	590.665	6853.500	6567.047
DR	3	10	1.200	36.000	162.000	1.075	32.505	153.045	175.952
		25	1.200	90.000	405.000	1.225	92.745	452.160	466.814
		50	1.200	180.000	810.000	1.225	182.595	882.695	916.540
		100	1.200	360.000	1620.000	1.130	328.925	1477.405	1470.332
	5	10	1.600	48.000	306.000	1.670	51.995	319.210	301.989
		25	1.600	120.000	765.000	1.760	135.105	816.895	686.709
		50	1.600	240.000	1530.000	1.790	270.540	1597.535	1324.199
		100	1.600	480.000	3060.000	1.635	490.990	2909.115	2353.168
	10	10	2.600	78.000	930.000	2.525	82.255	921.610	495.543
		25	2.600	195.000	2325.000	2.355	187.715	2181.590	1275.035
		50	2.600	390.000	4650.000	2.205	345.350	4043.070	2508.002
		100	2.600	780.000	9300.000	2.375	741.870	8544.700	5022.780
SA	3	10	1.200	36.000	162.000	1.140	34.910	170.265	186.911
		25	1.200	90.000	405.000	1.230	93.265	435.685	452.190
		50	1.200	180.000	810.000	1.215	181.305	883.510	911.088
		100	1.200	360.000	1620.000	1.185	345.950	1634.850	1844.829
	5	10	2.000	60.000	390.000	1.795	55.745	350.055	244.898
		25	2.000	150.000	975.000	1.795	138.910	891.525	647.402
		50	2.000	300.000	1950.000	1.715	261.740	1716.095	1284.733
		100	2.000	600.000	3900.000	1.790	528.420	3385.240	2633.592
	10	10	4.000	120.000	1380.000	3.900	127.715	1443.040	773.632
		25	4.000	300.000	3450.000	3.745	306.745	3477.305	1979.270
		50	4.000	600.000	6900.000	3.860	617.375	7140.655	4262.114
		100	4.000	1200.000	13800.000	3.870	1232.770	14072.910	8287.361

Table 1: Results of the Model and the Experiment

on a different VM than their directly linked services, but in SA *sidecars* are placed on the same VM as their corresponding cloud services. For this reason, it can be observed that the reported values for SA and DR closely resemble each other when we have different values of cf but keep n_{serv} constant at three. Considering the cases with five or ten cloud services, we almost always observe higher IL_T when we change the architecture from a CE to a DR or from a DR to

Number of Runs	$C_T(\%)$	$EL_T(\%)$	$IL_T(\%)$
50	12.919	12.307	13.946
100	9.416	8.492	9.593
150	8.326	7.426	8.731
200	8.081	7.097	8.105

Table 2: Prediction Error of Experimental Runs

an SA but keep the same configurations, i.e., constant n_{serv} and cf. It is because in our experiment, CE has only one control logic component (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 C_T, which results in losing more requests.

5 Discussion and Conclusions

Evaluation of the Prediction Error. We measure the prediction error by calculating the Mean Absolute Percentage Error (MAPE) [17]. Let $model_i$ and $empirical_i$ be the result of the model, and the measured empirical data for experimental case i, respectively. n_{case} is the number of cases (36 in this study).

$$MAPE = \frac{100\%}{n_{case}} \cdot \sum_{i=1}^{n_{case}} \left| \frac{model_i - empirical_i}{empirical_i} \right| \tag{11}$$

Table 2 reports prediction error measurements of our model for a different number of runs. 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 final prediction error regarding IL_T is 8.1%. As mentioned before, the common target prediction accuracy in the cloud performance domain is 30% [11].

Threats to Validity. While injecting crashes is a commonly taken approach (see Section 2.1), a threat remains that measuring internal and external loss based on these crashes might not measure reliability well, e.g., cascading effects of crashes [14] are not covered in our experiment. 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 experiment machines; much of the related literature takes a similar approach. To increase internal validity we decided not to run the experiment on a public cloud where, e.g., other load on the experiment machines might have had a significant impact on the results. As a consequence, there is the threat that generalization to a public cloud setting might be limited. As our private cloud setting uses very similar hardware and software stacks as many public cloud offerings, we believe this threat to be small. 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 results.

Conclusions. We investigated the impact of architectural design decisions on system reliability. Regarding **RQ1**, our study concludes that more decentralized routing results in losing a higher number of requests in comparison to more centralized approaches. Regarding **RQ2**, 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 8.1%.

References

1. A. Amiri, C. Krieger, U. Zdun, and F. Leymann. Dynamic data routing decisions for compliant data handling in service- and cloud-based architectures: A performance analysis. In *IEEE International Conference on Services Computing*, 2019.
2. A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.
3. D. A. Chappell. *Enterprise service bus.* O'Reilly, 2004.
4. R. C. Cheung. A user-oriented software reliability model. *IEEE transactions on Software Engineering*, pages 118–125, 1980.
5. 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.
6. Envoy. Service mesh. *https://www.learnenvoy.io/articles/service-mesh.html*, 2019.
7. V. Grassi and S. Patella. Reliability prediction for service-oriented computing environments. *IEEE Internet Computing*, 10(3):43–49, 2006.
8. G. Hohpe and B. Woolf. *Enterprise Integration Patterns.* Addison-Wesley, 2003.
9. A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson. *Cloud Design Patterns.* Microsoft Press, 2014.
10. 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.
11. D. A. Menascé and V. A. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods.* Prentice Hall PTR, 2001.
12. Microsoft. Sidecar pattern. `https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar`, 2010.
13. R. Natella, D. Cotroneo, and H. S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)*, 48(3):44, 2016.
14. M. Nygard. *Release It!: Design and Deploy Production-Ready Software.* Pragmatic Bookshelf, 2007.
15. C. Richardson. Microservice architecture patterns and best practices. *http://microservices.io/index.html*, 2019.
16. O. Sukwong, A. Sangpetch, and H. S. Kim. Sageshift: managing slas for highly consolidated cloud. In *2012 Proceedings IEEE INFOCOM*, pages 208–216, 2012.
17. K. S. Trivedi and A. Bobbio. *Reliability and availability engineering: modeling, analysis, and applications.* Oxford University Press, 2017.
18. A. van Hoorn, A. Aleti, T. F. Düllmann, and T. Pitakrat. Orcas: Efficient resilience benchmarking of microservice architectures. In *IEEE International Symposium on Software Reliability Engineering Workshops*, pages 146–147. IEEE, 2018.
19. L. Wang, X. Bai, L. Zhou, and Y. Chen. A hierarchical reliability model of service-based software system. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 199–208, July 2009.