Dynamic Data Routing Decisions for Compliant Data Handling in Service- and Cloud-Based Architectures: A Performance Analysis

Amirali Amiri*, Christoph Krieger†, Uwe Zdun*, Frank Leymann†

*University of Vienna, Faculty of Computer Science, Research Group Software Architecture, Vienna, Austria,
Email: {firstname.lastname}@univie.ac.at
†University of Stuttgart, Institute of Architecture of Application Systems, Stuttgart, Germany,
Email: {firstname.lastname}@iaas.uni-stuttgart.de

***Abstract*—In many service-based applications, decisions about data routing need to be made at runtime, for instance to ensure compliant data handling. Different service- and cloud-based architectures to make dynamic data routing decisions exist including central entities, multiple dedicated dynamic router services, or using a sidecar for each involved service. These architectures differ in various quality attributes including complexity, understandability, and changeability of the decision logic. Choosing the wrong architecture for decision-making at runtime may severely impact the performance of the software system. In this paper, we have evaluated the performance of three representative approaches for processing compliance rules concerned with data routing in service- and cloud-based architectures. The results show that distributed approaches for dynamic data routing have a better performance compared to centralized solutions. On the other hand, centralized solutions are easier to understand and change, but this strongly depends on the domain problem.**

***Index Terms*—Service- and Cloud-Based Architectures, Performance Analysis, Dynamic Data Routing, Compliance**

I. Introduction

In service- and cloud-based architectures, data flow paths are typically not pre-configured, which means decisions about data routing are made at runtime, e.g., based on a set of rules. A very simple example is a load balancer which follows just a single rule for round robin load balancing. Compliance rules for data routing are a typical example of more complex rule sets for data flows. In general, compliance, in the context of software systems, means ensuring that the software and systems of an organization act in accordance with established laws, regulations, and business policies [13]. For instance, a compliance rule might state that data originating in the EU must be processed and stored on cloud resources located in the EU. It is obvious that a combination of different such rules in service-based systems can quickly lead to a complex web of decision logic that is difficult to engineer well with regard to quality attributes such as performance, scalability, and elasticity.

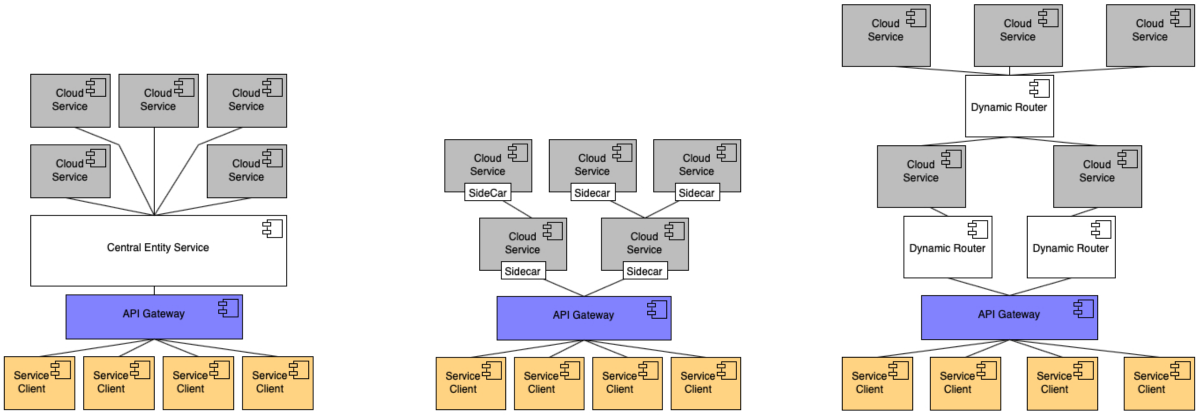
For modern service- and cloud-based systems, a number of architectures have been proposed that could be used to process such dynamic data flow routing rules. One essential architecture proposed in different technologies is using a *central entity* for processing the rules. For example, an API gateway [10] or any kind of central service bus [3] can play this role. Another typical architecture is a *sidecar architecture* [8] in which a sidecar for each service handles inbound and outbound traffic [5] and can thus perform the data flow routing for that service. These two architectures are two extremes: one is centrally managed, the other is completely decentralized. Finally, another option is a compromise between the two extremes, which uses specific services as *dynamic router services* on which routing decisions are made, exactly at those points in the data flow where a data routing decision is needed.

Unfortunately, at present the effects of these architectures in terms of performance of service- and cloud-based applications have not been sufficiently analyzed. In this paper, we aim to study the performance of different representative cloud/service architectures using the case of processing compliance rules concerning privacy (as for instance implied by the General Data Protection Regulation, GDPR). It is our goal that our study results are transferable to other scenarios of dynamic data flow routing rules in cloud services. We investigate the following *Research Question: What is the performance impact of different representative service- and cloud-based architectures for dynamic data flow routing?*

This problem is important as the architectural options have different impacts regarding many important qualities, such as understandability, testability, changeability, complexity, etc., of the dynamic data flow routing decision logic. For instance, a central decision logic is usually easier to understand and change than a distributed decision logic, but this depends on the dependencies between decisions. As the performance under load is crucial for many cloud applications, it would be helpful to be able to understand the performance impact of different design options well.

II. Background: Architectures for Runtime Checking of Data-Flow Compliance Rules

There are many different service- and cloud-based software architectures that can check compliance of data-flow rules



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

Fig. 1: Architectures for Dynamic Data Routing

at runtime. In this paper, three of the most widely used architectures are investigated which – as explained above – can be seen as representatives for a variety of similar architectures.

a) Central Entity: A Central Entity (CE) is one central service that manages all communication and data control, as shown in Fig. 1a. Although CE is easy to manage, understand, and change because all control logic is in one place, it is hard to design the internals of the central entity service. If compliance rules require the state of the processing steps, one disadvantage of CE is that subsequent processing steps need to call back into the central entity service in order to proceed. An obvious advantage is that all needed states for decisions from prior stages and all decision logic can be kept in a central place, not requiring the state to be passed along with invocations. CE can be implemented for instance using an API gateway [10] or any kind of central service bus [3].

b) Sidecars: The Sidecar Architecture (SA) shown in Fig. 1b places data control logic in so-called sidecars [8], [5] that are attached to services. Sidecars offer the same level of decentralization as if each service would make data flow decisions in its implementation, but at the same time they offer separation of concerns, i.e., the data flow logic concern is placed outside of the service. Sidecars offer benefits whenever decisions need to be made structurally close to the service logic. One advantage of this architecture is that it is usually easier to implement the internals of sidecars than those of central entities as they need to check only those few rules specific to their services. In contrast, the central entity manages all rules regarding all services under its control, which results in more complex control logic and data structures. One disadvantage is that adding sidecars is not always possible, since some (cloud) services are off-the-shelf or third-party products. Another disadvantage is that data needs to be sent to services in order for sidecars to check the rules. If the user has not agreed that the corresponding service is allowed to process the data, the sidecar will need to discard it but there is a risk that this results in a privacy breach since the data has already been sent.

c) Dynamic Routers: Using specific Dynamic Routers (DR) [7] for data control decisions in the web of services is shown in Fig. 1c. This can be seen as a hybrid of CE and SA, as this introduces more levels of data control logic. One advantage of this architecture is that (as in SA) it is easier to implement comparing to CE architecture. Dynamic router services can check a reduced set of rules regarding their connected services, contain simpler data structures and data flow control logic, and can use local information about placement in the web of services (e.g., they might know about pre-processing steps that have happened). A disadvantage compared to CE is the management and deployment overhead introduced by dynamic router services since they are distributed and placed on different hosts.

III. Experimental Planning

a) Goals: The experiment's goal is to measure the performance of the three approaches for processing compliance rules concerned with data routing in cloud service architectures, namely *Central Entity*, *Dynamic Router*, and *Sidecar Architecture*, outlined in the previous section.

b) Technical Details: We have used a private cloud with 4 nodes, each having 2 identical CPUs. 2 cloud nodes host Intel®Xeon®E5-2680 v4 @ 2.40GHz[1] and the other 2 host the same processor family but version v3 @ 2.50GHz. The v4 and v3 versions have 14 and 12 cores respectively and 2 physical threads per core (56 and 48 threads in total). All cloud nodes have 256GB of system memory and run Ubuntu Server 18.04.01 LTS[2]. On top of the operating system, Docker[3] containerization is used to run the cloud services which are implemented using Node.js[4]. We have utilized 5 desktop computers to simulate load generation, each hosting an Intel®Core™i3-2120T CPU @ 2.60GHz with 2 cores and 2 physical threads per core (4 threads in total). All desktop

[1]https://www.intel.com/content/www/us/en/homepage.html
[2]https://www.ubuntu.com
[3]https://www.docker.com
[4]https://nodejs.org/en/

TABLE I: Experimental Results of All Architectures

Arch.	Call Freq. (Hr/s)	Num. Cloud Serv.	Min. RTT (ms)	Q_1 (ms)	Median RTT (ms)	Q_3 (ms)	95th Percentile (ms)	Max. RTT (ms)	Mean RTT (ms)	STD	WAvg (ms)
CE	100	5	25.598	31.377	37.884	77.283	466.155	631.252	98.059	136.076	37.131
		10	62.059	107.9975	136.734	323.3727	1179.303	1292.048	325.288	367.279	
		25	351.298	1191.604	1438.716	1849.310	3131.327	3325.957	1651.817	732.885	
		50	1558.662	3198.889	3486.238	3888.954	8763.811	8813.965	4174.527	1938.576	
	500	5	14.965	951.199	1114.719	1274.4817	1413.479	1998.354	1033.383	336.828	295.815
		10	199.221	2485.889	2669.604	3080.838	3403.013	3959.199	2664.332	546.876	
		25	1848.500	7482.008	8077.590	8519.928	9132.945	9987.151	7843.596	1089.819	
		50	8800.803	17552.022	18510.818	19030.105	19701.113	20501.393	18043.396	1569.938	
	1000	5	10.021	2116.356	2770.525	3229.294	3914.324	4909.937	2673.797	842.366	727.784
		10	213.977	5622.522	6741.707	7574.174	8658.320	15762.526	6484.570	1584.818	
		25	4928.274	16971.840	19620.402	21115.877	22539.220	35547.201	18650.974	3394.748	
		50	12779.84	39582.05	44902.12	47242.38	49364.520	75751.217	42879.598	7349.506	
DR_3	100	5	12.407	17.962	20.614	45.938	219.418	358.606	49.327	67.591	8.979
		10	24.475	31.143	42.555	117.595	622.318	786.069	131.759	184.488	
		25	58.763	110.093	178.123	619.097	1948.941	2145.527	534.422	649.009	
		50	207.219	517.992	1020.674	1554.446	3530.587	3681.506	1381.369	1080.652	
	500	5	9.1600	20.973	54.699	142.444	510.137	663.389	122.964	155.557	111.427
		10	36.760	378.067	497.419	590.276	1146.401	1484.203	506.953	265.814	
		25	57.249	2505.792	3965.498	4866.302	5527.488	6279.350	3676.808	1444.207	
		50	570.627	8326.662	11320.266	12351.667	13032.551	13774.296	10092.286	2917.993	
	1000	5	7.070	329.4105	595.3050	893.1605	1499.987	2576.772	653.093	437.785	241.631
		10	15.125	1185.353	1946.702	2856.923	3497.138	4297.438	1994.626	1035.537	
		25	22.686	4440.487	6943.560	9629.477	10971.649	12175.990	6937.950	2862.820	
		50	4085.752	11233.909	18752.584	24458.039	26567.286	28846.745	17818.120	6870.081	
DR_6	100	5	13.130	18.081	21.482	53.835	292.091	439.211	66.442	92.213	5.345
		10	24.523	31.956	41.547	154.331	588.913	692.254	140.782	187.354	
		25	59.725	88.546	143.640	698.237	1612.439	1669.844	478.032	550.105	
		50	129.424	249.395	359.192	1872.233	3896.844	4033.159	1273.733	1377.067	
	500	5	6.053	56.572	83.383	130.114	456.810	587.507	126.677	123.246	47.934
		10	15.222	133.602	226.448	354.651	695.318	860.288	278.271	194.367	
		25	51.818	883.619	1432.079	1801.626	2371.261	5103.557	1450.629	852.522	
		50	202.680	2709.620	4746.551	6618.730	7636.278	9139.602	4639.890	2152.850	
	1000	5	6.0190	362.8822	563.8395	722.0797	1279.516	1679.144	568.558	326.043	125.817
		10	6.369	575.932	829.978	1170.606	2386.818	3425.810	938.111	614.172	
		25	28.540	2183.593	3450.883	4693.387	5855.607	6720.997	3367.412	1595.319	
		50	71.612	5548.778	8473.436	11046.606	13570.286	15530.022	8232.277	3465.464	
SA	100	5	13.939	18.297	21.818	53.860	174.662	465.173	48.827	96.112	5.218
		10	27.784	34.110	42.278	175.466	601.827	731.133	142.529	188.345	
		25	66.538	93.071	130.398	639.999	1411.885	1570.485	426.281	465.568	
		50	144.760	201.866	353.252	1280.374	2977.373	3112.574	981.886	1031.220	
	500	5	7.019	25.249	49.632	93.276	308.843	462.864	80.743	89.948	14.494
		10	16.027	99.895	142.365	183.956	1523.994	1685.055	308.145	448.920	
		25	44.503	316.710	426.120	771.040	1702.167	2350.455	625.794	490.145	
		50	162.088	658.954	838.437	1052.560	1882.903	2297.493	893.149	431.288	
	1000	5	5.155	169.817	346.633	547.109	1148.907	1695.961	412.302	328.463	55.265
		10	3.111	385.790	674.881	1259.548	2086.450	3670.457	852.858	643.786	
		25	12.249	864.723	1152.719	1539.998	2848.677	4187.7180	1320.540	761.473	
		50	45.632	1341.721	1906.893	2643.657	3641.671	4805.411	2011.770	942.661	

computers have 8GB of system memory and run Ubuntu 18.10. They generate load using Apache JMeter[5] which sends HTTP/1.1 requests to cloud nodes.

c) Architecture Configurations: We have used one cloud node with 56 threads to run the *API Gateway* and distributed the cloud services among the remaining three nodes. The distribution of services is so that all nodes have the same number of cloud services (with maximum a difference of one service). In case of CE, the central entity service is also placed on the *API Gateway* node, to minimize network communication. For DR, we have placed a dynamic router service on each of three nodes that host cloud services. Each router controls data communication regarding services on their corresponding node. We call this configuration 3 Dynamic Router services (DR_3). We have added another configuration for DR in which we put two routers on each cloud node and let each router control data flow for half of the cloud services on the corresponding node. We call this configuration 6 Dynamic Routers services (DR_6). SA places one sidecar per each cloud service on the corresponding node. We have chosen to implement all three architecture options from scratch in Node.js and did not use existing implementations of these options, such as Envoy[6] for sidecar architectures. The reason is that we wanted comparable implementations to avoid measuring the impact of a particular technology implementation rather than the impact of the canonical architecture.

d) RTT Calculation: To measure the performance of the different prototypical architectures, we have calculated the Round-Trip Time (RTT) of requests which is defined as the difference in time from the moment a request enters the application through the *API Gateway* until it is routed through all cloud services involved in the processing of the request.

e) Experimental Cases: Many factors can influence RTT, out of which we have chosen two, call frequency and number of cloud services, to study their effects. Call frequency is defined as number of requests per second coming from service clients, which affects RTT since higher frequency of calls requires either more processing power or buffering. A higher number of cloud services increases RTT because there are more rules to be checked by controlling services.

In this experiment, we have chosen call frequencies of 100, 500 and 1000 HTTP requests per second (Hr/s). We have selected these numbers based on a study of related works. In many related studies, 100 requests per second (or even lower numbers) are chosen (see e.g. [4], [12]). As we focus on higher loads, we have chosen 100 Hr/s as the *lowest* call frequency. A recent benchmark for self-adaptive IaaS cloud environments [6] uses 339 requests per second as its upper limit. We have thus chosen 500 Hr/s as a close, but slightly higher number (again to focus rather on high load scenarios). Finally, to study even higher load conditions, we have also taken 1000 Hr/s into consideration. In case of 100 Hr/s, one desktop computer is used to generate the load. For call frequencies of 500 and 1000 Hr/s, we have used two and five computers respectively.

We have chosen the experimental cases of 5, 10, 25, 50 cloud services, which we believe are representative of most applications. Note that today many real-world microservice architectures use a much larger number of microservices, but in our experience the number of microservices that have close interactions (like a common compliance rule base) is usually in the range [5-50]. In our point of view, early performance analysis in early architecture design should be focused on such interacting clusters of microservices, rather than considering microservices which have little impact on the performance aspects in focus.

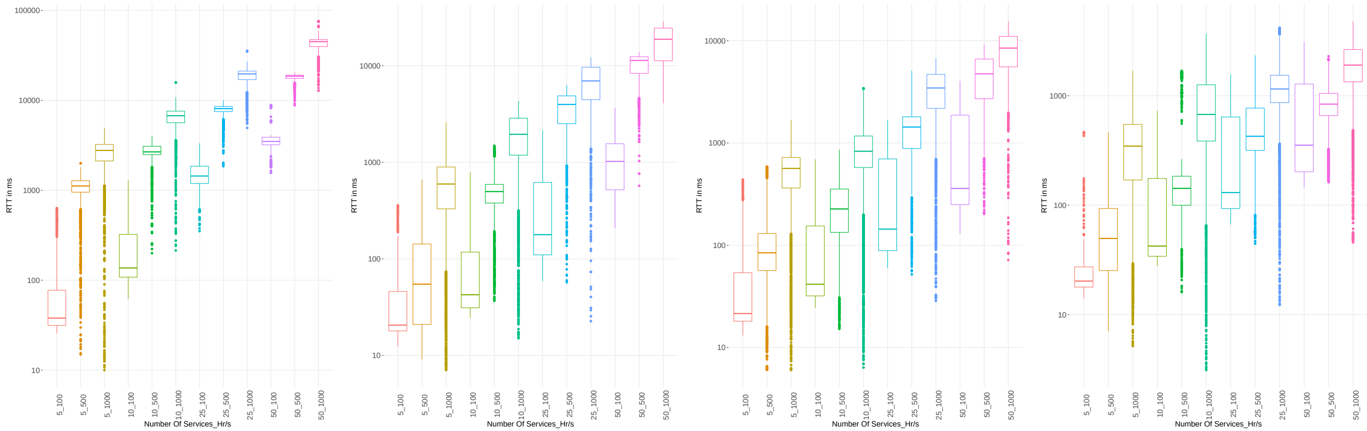
f) Data Set Preparation: We have executed each experimental case 5 times and report minimum, first quartile (Q_1), median, third quartile (Q_3), 95th percentile, maximum, mean and standard deviation (STD) of recorded RTTs. Additionally, a weighted average of median RTTs is calculated over number of cloud services. The formula for weighted average is: $WAvg = (RTT_5/5 + RTT_{10}/10 + RTT_{25}/25 + RTT_{50}/50)/4$ in which RTT_n is median RTT for number of cloud services. Weighted average corresponds to the average RTT per cloud service which is used to normalize the result data and make them comparable across the different studied architectures.

IV. Experimental Results

Table I presents the experimental results of all architectures. We can see that for CE, when taking the same number of cloud services, increasing call frequency from 100 to 500 Hr/s results in a nonlinear rise of median RTT of more than 5 times. However, when we double the call frequency from 500 to 1000 Hr/s, the median increases almost linearly. We observe the same trend with weighted average of RTTs.

[5]https://jmeter.apache.org

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



(a) Central Entity (b) Dynamic Router_3 (c) Dynamic Router_6 (d) Sidecar Architecture

Fig. 2: Distribution of Experimental Results for All Architectures

Standard deviations are highest in CE compared to the other architectures. This is explainable since there is only one service which receives all requests and checks compliance, i.e., the central entity service. At the beginning of a run, lower RTTs are observed; however, as more requests arrive and this service becomes overloaded, delays become larger resulting in higher RTTs. For DR_3, as expected, we achieve lower mean RTTs and weighted averages compared to CE since we have three dynamic routers that can process requests simultaneously. We can see that choosing higher number of cloud services results in an almost linear rise of median RTTs when increasing call frequency from 500 to 1000 Hr/s. We observe lower STDs for DR_3 compared to CE, most likely because three dynamic routers become less overloaded than only one central entity service.

For DR_6 and SA, when having 5 or 10 cloud services, we see almost identical numbers. This is because in our implementation, which aims to implement the architectures in a comparable way, these architectures are identical when having 5 cloud services and only slightly different when having 10. With the increase of cloud services to 25 and 50, we also increase the number of sidecars in SA but still have only 6 dynamic routers in DR_6, resulting in higher numbers in median RTTs, weighted averages and STDs in DR_6 compared to SA. In both of these architecture configurations, we can see an almost linear increase of median RTTs when having 25 and 50 cloud services and doubling call frequency from 500 to 1000 Hr/s. Furthermore, in SA, we observe an almost linear rise of median RTTs when we increase the number of cloud services but keep the call frequency constant. SA results in lower STDs compared to the other architectures in most cases, most probably because we have more controlling services, i.e., sidecars, which process incoming requests simultaneously.

Fig. 2 shows the distribution of RTTs for each experimental case for all architecture configurations. We can clearly see the decrease of STDs when moving from CE to DR and SA architectures. In CE, we observe a rather low interquartile range. By adding more controlling services, i.e. dynamic routers and sidecars, we get a higher interquartile range in DR and SA. An interesting observation is that in all architectures, the outliers mostly lie between minimum RTT and Q_1 except for the call frequency of 100 Hr/s. As explained before, at the beginning of a run, RTTs are very low and as more requests arrive, RTTs increase. In case of 100 Hr/s, since frequency of calls are not so high that they can overload cloud nodes, the majority of the RTTs stay in the lower range and only some calls are delayed, resulting in outliers being plotted above the interquartile range.

V. Threats to Validity

Concerning *internal validity* threats, we have made sure that all three groups of the experiment are deployed on the same infrastructure with the same distribution of cloud services, and tried to avoid any possible implementation differences between the architectures. Nonetheless, such internal validity threats cannot be completely excluded. In particular, despite our careful implementation and deployment work, some aspects may have been slightly distinct in the different implementations and deployments. We have tried to mitigate this threat by carefully double-checking all technical aspects by all researchers in the author team. We have made sure the machines we have run our study on were idle, but possibly other services, e.g., of the operating systems, may have influenced our measurements. We have tried to mitigate this threat by running the experiment multiple times.

The *external validity* refers to the degree to which results are generalizable outside the scope of our study. One external validity threat is that potentially our experimental setup for cloud environments is not chosen well; therefore, it cannot be compared to real-world setups. A related threat is that we have chosen to implement all three architecture options from scratch in Node.js and did not use existing implementations of these options. We have chosen to do so in order to make the implementations comparable in an experiment; however, this entails the threat that our implementations might not represent the existing off-the-shelf tools like Envoy for sidecars or enterprise service buses for central entities well. Moreover, the cloud services are deployed using container technology Docker, which is commonly used in cloud-based architectures. Real-world cloud applications are often composed of different computing and storage services offered by multiple cloud providers. Such scenarios may have additional effects on the performance of the evaluated architectures. These threats are at odds with internal validity; we have tried to model, implement, and deploy the tested architectures in a similar way as much as possible to ensure comparability. From our experience, they are close to existing architectures in the cloud, but the external validity threats cannot be excluded.

VI. Related Work

Vandikas et al. [14] conducted a performance analysis of their IoT framework to evaluate its behavior under heavy load produced by different amounts of producers and consumers. In contrast to our work, dynamic data routing or compliance rules are not considered in this paper. 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.

There is 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. Sanjay et al. [1] evaluate the performance of the three open source ESBs Mule, WSO2 ESB, and Service Mix. The performance is measured based on mean response time and throughput for proxying, content-based routing, and mediation of data. However, the test scenarios only consider communications between clients and a single web service. In contrast, our work also considers communication paths which involve the composition of multiple services and routing decisions. Shezi et al. [11] provide a performance evaluation of different ESBs in a more complex scenario in which multiple services are composed to achieve a certain business objective. None of these works consider compliance decisions, e.g., for privacy, which is unique in the sense that the routing decisions sometimes need to be made outside of the services and might require stopping the ongoing communication due to a compliance violation.

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 [9] evaluates the performance impact of Docker containers, software-defined networks, and encryption to network performance in distributed cloud-based systems using HTTP-based communication. A similar work is presented by Bankston et. al [2] 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. Our experimental setup is influenced by the named related work, a broader study of related experimental setups (e.g. [4], [12], [6]), and our own experiences in building microservice and cloud systems as outlined above.

VII. Conclusion and Future Work

In this paper, we have investigated three representative service- and cloud-based architectures for making and enacting dynamic data flow routing decisions (here, scenarios in compliant data handling) with regard to their performance. For a set of representative application sizes in terms of cloud nodes and across various call frequencies, we were able to provide precise estimates of performance impacts of the three architectures. This can help in (early) architectural decision making. A limitation of our research is that we have only tested a typical range of call frequencies for a smaller number of cloud nodes. For very large cloud setups or very low or high call frequencies, more studies are needed to improve data set. Moreover, we have only focused on a limited number of server resources, and designed architecture configurations accordingly. For our future work, we plan to extend our studies in such directions.

Acknowledgment

This work was supported by Austrian Science Fund (FWF) project ADDCompliance (no. I 2885-N33), Austrian Research Promotion Agency (FFG) project DECO (no. 864707) and DFG project ADDCompliance (636503).

The authors would like to thank Konstantinos Plakidas.

References

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

[2] R. Bankston and J. Guo. Performance of container network technologies in cloud environments. In *2018 IEEE International Conference on Electro/Information Technology (EIT)*, pages 0277–0283. IEEE, 2018.

[3] D. A. Chappell. *Enterprise service bus.* " O'Reilly Media, Inc.", 2004.

[4] 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.

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

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

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

[8] 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.

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

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

[11] 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.

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

[13] A. Tarantino. *Governance, Risk, and Compliance Handbook: Technology, Finance, Environmental, and International Guidance and Best Practices.* Wiley, 2008.

[14] 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.