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
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 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 compared 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 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. On top of the operating system, Docker3 containerization is used to run the cloud services which are implemented using Node.js4. 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

2https://www.ubuntu.com
3https://www.docker.com
4https://nodejs.org/en/
computers have 8GB of system memory and run Ubuntu 18.10. They generate load using Apache JMeter\(^3\) 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\(^4\) 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: \( W\text{Avg} = (\text{RTT}_5/5 + \text{RTT}_{10}/10 + \text{RTT}_{25}/25 + \text{RTT}_{50}/50)/4 \) in which \( \text{RTT}_n \) 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.

---

\(^3\)https://jmeter.apache.org

\(^4\)https://www.envoyproxy.io/

<table>
<thead>
<tr>
<th>Arch</th>
<th>Test Hrs</th>
<th>Cloud Servs</th>
<th>RTT (min)</th>
<th>Q(_1) (sec)</th>
<th>Q(_3) (sec)</th>
<th>95th Pctl (sec)</th>
<th>Mean (sec)</th>
<th>STD (sec)</th>
<th>Weighted Avg of Median RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CE</td>
<td>500</td>
<td>10</td>
<td>104.9</td>
<td>31.1</td>
<td>41.1</td>
<td>3.8</td>
<td>2.9</td>
<td>0.8</td>
<td>2.9</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>25</td>
<td>251.8</td>
<td>75.5</td>
<td>81.3</td>
<td>12.4</td>
<td>8.4</td>
<td>3.5</td>
<td>8.4</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>50</td>
<td>502.6</td>
<td>125.3</td>
<td>138.8</td>
<td>19.2</td>
<td>16.4</td>
<td>6.8</td>
<td>16.4</td>
</tr>
<tr>
<td>SA</td>
<td>100</td>
<td>10</td>
<td>104.9</td>
<td>31.1</td>
<td>41.1</td>
<td>3.8</td>
<td>2.9</td>
<td>0.8</td>
<td>2.9</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>25</td>
<td>251.8</td>
<td>75.5</td>
<td>81.3</td>
<td>12.4</td>
<td>8.4</td>
<td>3.5</td>
<td>8.4</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>50</td>
<td>502.6</td>
<td>125.3</td>
<td>138.8</td>
<td>19.2</td>
<td>16.4</td>
<td>6.8</td>
<td>16.4</td>
</tr>
</tbody>
</table>

**Table I: Experimental Results of 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, 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 compared to CE, most likely because three dynamic routers become less overloaded than only one central entity service.

For DR 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, resulting in higher numbers in median RTTs, weighted averages and STDs in DR 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 Q1 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. [11] 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