Impacts of Decomposition Techniques on Performance and Latency of Microservices

Chaitanya K. Rudrabhatla
Executive Director-Solutions Architect
Media and Entertainment Domain, Los Angeles, USA

***Abstract*—Micro service architecture (MSA) has undoubtedly become the most popular modern-day architecture, often used in conjunction with the rapidly advancing public cloud platforms to reap the best benefits of salability, elasticity and agility. Though MSA is highly advantageous and comes with a huge set of benefits, it has its own set of challenges. To achieve the separation of concerns and optimal performance, defining the boundaries for the services clearly and their underlying persistent stores is quintessential. But logically segregating the services is a major challenge faced while designing the MSA. Some of the guiding principles like Single responsibility principle (SRP) and common closure principle (CCP) are put in place to drive the design and separation of microservices. With the use of these techniques the service layer can be designed either by (i) Building the services related to a business subdomain and packaging them as a microservice; (ii) or Defining the entity relationship model and then building the services based on the business capabilities which are grouped together as a microservice; (iii) or understanding the big picture of the application scope and combining both the strategies to achieve the best of both worlds. This paper explains these decomposition approaches in detail by comparing them with the real-world use cases and explains which pattern is suitable under which circumstances and at the same time examines the impacts of these approaches on the performance and latency using a research project.**

Keywords—Microservices; decomposition techniques; single responsibility principle; common closure principle; performance; latency

I. INTRODUCTION

In the recent past all the applications were built using a monolithic design pattern. This design pattern was a great advancement from the traditional client-server architecture which was prevalent before [1]. Monolithic design pattern was well suited and worked fine in the waterfall software development methodology. But the cutthroat competition and everchanging nature of the businesses demanded a software model which is more agile and nimble enough to cope up with the business needs. This thought process gave birth to the agile methodology. In the agile methodology, the software needs to be developed quickly in smaller pieces and deployed continuously to production. Monolithic systems struggled hard to find their place in the agile methodology. It was soon discovered that monoliths are more complex in nature due to their fundamental design, where the entire application code into a single deployable unit. Due to this design, the code changes need to be well planned in advance and need to be thoroughly tested before deploying to production and thus leading to longer build cycles. This was found to be anti-agile. These issues have prompted the search for the newer design patterns which involved breaking down the monolith to a more loosely coupled services. Service Oriented Architecture (SOA) is one such pattern which evolved on these lines [2]. In this pattern, backend services are isolated and distributed. But these services are handled by a layer called Enterprise Service Bus (ESB) which is an integration and guarding layer for the entire backend. Though ESB pattern has the advantages of conducting the health checks, performing the routing for the backend services, it was soon found to be a cumbersome layer and a bottle neck as the application services grow in size. To handle these shortfalls, Microservice Architecture (MSA) was introduced with the similar premise of isolation and separation of concerns as proposed by SOA [3], but with a lightweight design.

In MSA, the application services are designed to be heterogeneous, light weight, independent, isolated and highly distributed in nature [20]. The biggest advantage of micro services is that it supports the services to be built in any technology of choice and permits them to be deployed independently from each other. This greatly reduces the efforts and simplifies the development, testing, build and deployment cycles as the changes are limited to a single service rather than the entire monolith. With the proliferation of cloud technologies and advancements in containerization and their orchestration technologies like Kubernetes, Docker swarm etc., the microservice architecture became even more efficient. This has resulted in a continuous integration and continuous delivery (CI/CD) which is the most important feature of agile methodology [6]. But these benefits can only be harnessed if the backend services are carefully examined and decomposed in the most optimal way by considering the big picture of the entire application scope rather than in an ad-hoc way. If not, this design might prove counter-productive and lead to latency, complexity and inefficiency [4]. Rest of this paper is organized as follows – In section (II) various decomposition techniques and their benefits and shortfalls are explained by considering a real-world e-commerce scenario. In section (III) the research project which was conducted to examine the impacts of decomposition techniques on latency and performance [12] of the system is explained and the results are compared. In section (IV) summary, conclusions are provided and an overview of future research work is given.

II. DECOMPOSITION OF MICROSERVICES

For gaining the benefits of MSA, it is very important to strategize the decomposition of micro services [17]. As the name suggests, the services must be designed to be fine grained and independent. There are two major guiding principles which can drive the decomposition of services. (i) Single Responsibility Principle (SRP) is a guiding principle which states that a service class should exist to serve a single major responsibility and that class should only have one reason to change. Due to the isolation proposed by this theory, it is highly beneficial to apply SRP to build the microservices that implement a small set of closely related business functions. (ii) The second guiding principle states that it is not only in the initial development but also that the design should consider the future changes in such a way that they impact a single service. That is because changes that affect multiple services require more planning, coordination and testing. This slows down development and deployment cycles and would present the same issues of a monolithic application design. This constitutes the fundamental essence of Common Closure Principle (CCP), which is a second guiding principle for service decomposition. As per this principle if there are multiple micro services which serve a business functionality, a change in the business rule should only impact one single service rather than all the microservices involved. Using these two guiding principles, the services can be decomposed into granular microservices [15][16]. The below section examines the ways to implement the decomposition by taking a real-world example of a module from an e-commerce system [19].

A. Decomposition based on Domain Driven Desgin

Domain Driven Design is a decomposition technique which is based on the common closure principle (CCP). As per this principle, all the functions and classes which get impacted by a single business rule change should be packaged together as a single microservice. Some of the key considerations for this implementation are:

1) Services must be designed to be cohesive in nature. Which means that the methods, functions and services included in a microservice should strongly relate to activities related to a granular business functionality.

2) Each service should be designed to be completely autonomous. Which means if the business rule changes, it should be possible to apply a patch and deploy the service independently without impacting anything else.

To explain these designs, let us consider an e-commerce application module [5]. This module has three primary business entities: (i) User entity; (ii) Order entity; and (iii) Item Details entity. Here are some of the important business functionalities around each of these entities:

1) User Entity Responsibilities

- It stores the user details.
- It maintains the user profile settings.
- It has the user address details.
- It stores the payment details like credit card information.

2) Order Entity Responsibilities

- It maintains the Order history.
- It stores the item details of the Order.
- It stores the Order shipment details.
- It stores the payment details for each order.

3) Item Entity Responsibilities

- It contains the Item sku details.
- It maintains the item inventory details.
- It maintains the like score of the other items to give recommendations.

As it can be visualized from the Fig. 1, the business entities in the real world are not independent of each other. They need to overlap with one another for various use cases. For example – User entity relates to Order entity when there is a screen in the UI interface which displays the order history of the user. There will be a foreign key relationship between the User to Order entities. Similarly, an Order entity maintains a relationship with Item entity, such that when the item in a particular order is clicked, it takes to the details page of that particular item. In the same lines, User entity maintains a relationship with Item entity as the application might show the recommended items for purchase for that particular user based on the items purchased by the user in earlier transactions.

Now applying the Decomposition technique based on the Domain Driven Design [18], it can be visualized that the application module might be designed to have three microservices which can deal with the business functionality of each domain as shown in Fig. 2.

Using this principle there is one microservice per each business domain. Applying the Database per Service Pattern [7], there would be one database per microservice to achieve the isolation needed. This model is advantageous for the following reasons: (i) Each business functionality is encapsulated in its own micro service. Thus, it complies to Single Responsibility Principle (SRP). (ii) Changes related to a business functionality are mostly limited to its own micro service. Which means faster time to deploy the changes.

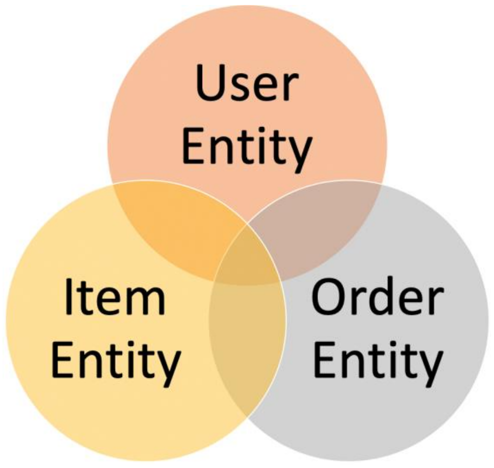

Fig. 1. Overlapping Business Entities in an E-commerce Application.

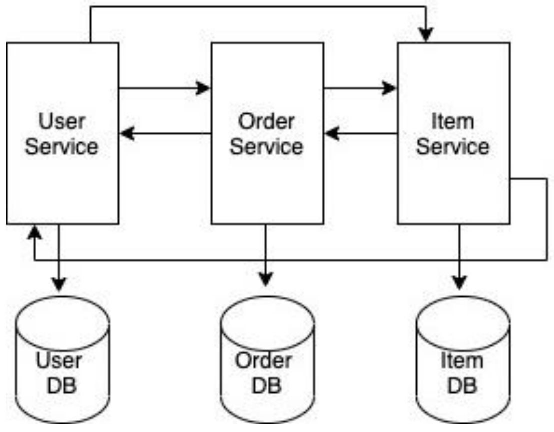

Fig. 2. Domain Driven Model for Microservice Decomposition.

However, it is not always advantageous as this design might lead to cross service events and communications which might complicate the maintenance as the application and business functionality grows. Here are some of the scenarios where the interservice dependency or communication is needed:

- Consider a scenario where the order is in progress and user updates the shipping address in the user profile. In this case, the address for the current order in progress may need to be updated as well upon the user confirmation. In this case, "update address" event on the User DB need to trigger the action for Order Service.
- Consider the other scenario where the order is in progress for shipment and there is a change in inventory because of which the shipment needs to be cancelled. In this case an "update inventory" event on the Item DB needs to trigger an action for the Order Service.
- In the same lines, when user changes the personal preferences, this event needs to trigger an action on the Item Service so that the recommended items are shown as per the new preferences.

There are multiple ways to handle this cross-service events and communication:

1) Inter-Service communication based on the Event Choreography technique [8], where each service can discover the other service using private load balancers and there by trigger the action in the destination service. This can be seen in Fig. 3.

Although this technique works for limited use cases, it has the following challenges:

a) It becomes difficult in the larger applications as the number of events [11] and triggers grow in number due to the intricacies of the business functionalities.

b) Also, the microservices no longer become independent as one team developing the Service1 may need to depend on the other team developing Service2 to call the actions in it.

c) The network traffic would also increase which might be another factor to consider.

d) It might also cause latency due to the inter service communication over network.

That gives rise to the second approach for communication.

2) Central orchestrator approach [9] where services are unaware of each other but post the events to a central orchestrator which takes care of firing the relevant actions in the destination services. This can be seen in Fig. 4.

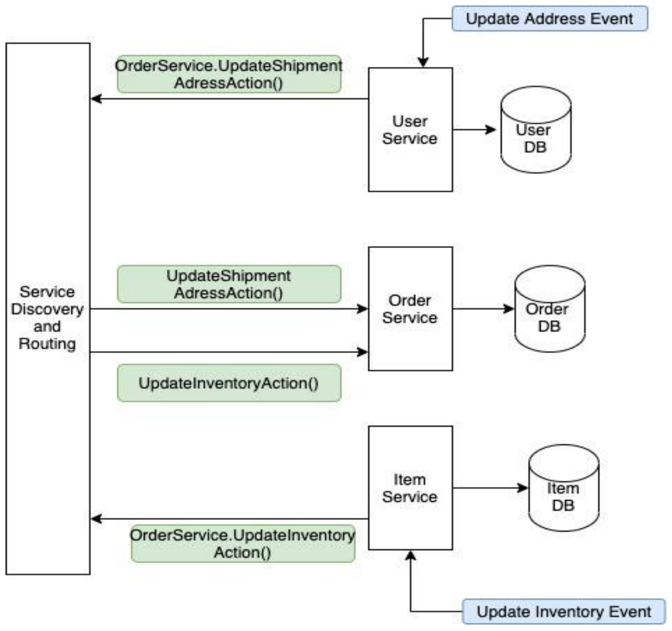
Update Address Event
OrderService.UpdateShipment AdressAction()
User Service
User DB
Service Discovery and Routing
UpdateShipment AdressAction()
Order Service
Order DB
UpdateInventoryAction()
Item Service
Item DB
OrderService.UpdateInventory Action()
Update Inventory Event

Fig. 3. Inter-Service Communication using Event Choreography and Service Discovery-Routing.

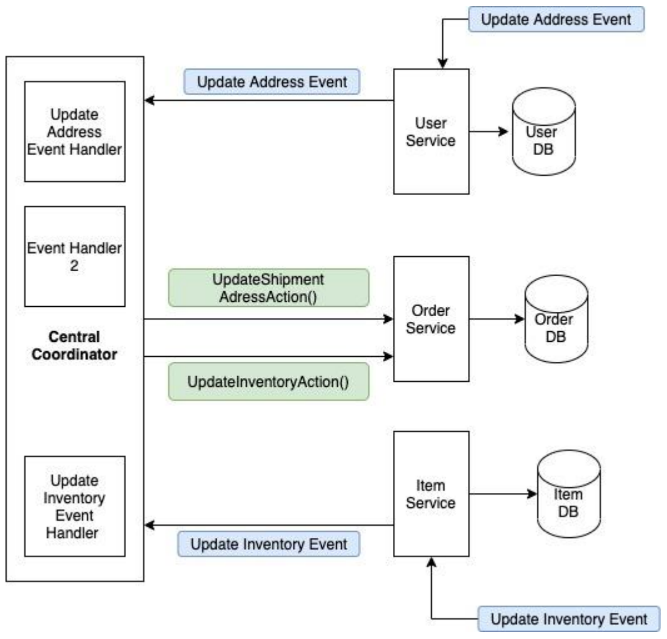

Fig. 4. Inter-Service Communication using Orchestrator or Central Coordinator.

This alternative works fine even in large and complicated applications. But it comes with its own set of challenges:

a) When the number of event handlers grow, the central coordinator becomes a fat layer. This becomes an anti-pattern for micro service architecture where each component needs to granular and light weight.

b) There is a risk that the central coordinator can act as a single point of failure. If this layer fails, entire application functionality might get impacted as it is responsible for multiple service actions.

c) Eventually this pattern might also introduce latency as the logical decisions to trigger the actions to the relevant services based on the incoming events might get tricky.

B. Decomposition based on Business capability

Decomposition based on Normalized Entity Relationship model is a design pattern where the major entities are normalized to the optimal level [14] and services are designed for the management of normalized entities involved in the business transaction. In this design, the services are designed around the activities of the entities. This can be visualized based on the Fig. 5. In this model the entities are broken down to the level where they can be considered standalone and the CRUD operations needed for managing the entities.

This approach is considered as an anti-pattern in the microservice architecture as it might pose a risk where a single transaction might have to flow through lot of management services before persisting finally. In case of a roll back this might get even more complex due to the large number of services which are granular and identifying the roll back steps might become challenging.

C. Decomposition based on the Hybrid approach

This is a third approach where the bigger picture of the application flows is analyzed and a hybrid approach which is a combination of both the approaches – domain based, and capability-based decompositions are used to come up with a hybrid model to gain the efficiencies. Fig. 6 shows the hybrid approach where a combination of subdomains like, user service, item service and order service are used and capability-based services like, address management service and inventory management service are used.

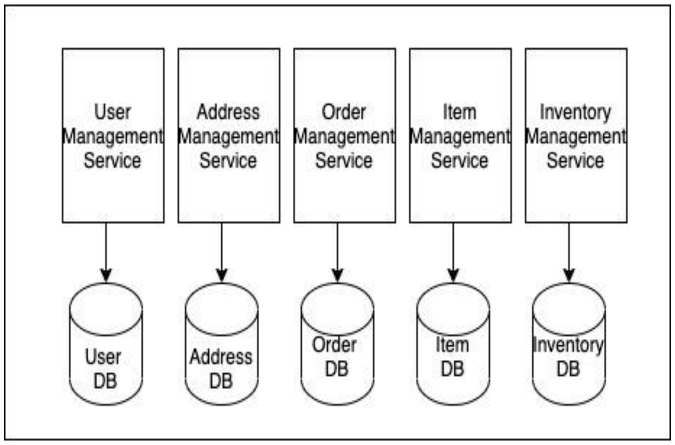

Fig. 5. Decomposition of Services based on the Business Capabilities.

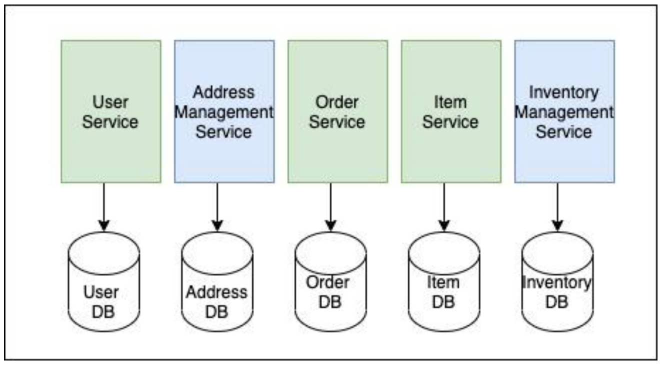

Fig. 6. Decomposition of Services based on Hybrid Approach.

This approach is beneficial because it combines the benefits of both the approaches. It packages the subdomain in the single service for the most part. But in some cases which lead to actions across multiple entities and warranting interservice communication can be isolated as business capability-based services.

III. Related Work

Related work is done to compare the performance [10] [13] and latency of decomposition techniques by building the micro services using each of the techniques mentioned in the earlier section and response times are observed when a database transaction is invoked. For this work Java based Spring boot microservices are used. MongoDB is used as the backend database.

A. Run 1 – Domain Driven Decomposition

For this simulation, four micro services MS1, MS2, MS3 and MS4 were used each of them having their individual MongoDB databases DB1, DB2, DB3 and DB4. DB1 with 2 dependent collections C11, C12 is used. Similarly, C21, C22 in DB2 C31, C32 in DB3 and C41, 42 in DB4. Eureka is used as the service discovery layer and Zuul as the routing mechanism for this experiment. First run is performed where a small message of size 50 bytes is persisted by calling one micro service MS1 and recorded the time in milli seconds. Later a test is performed where the transaction would persist in multiple collections C12 and C22. For this MS1 would make a call to a method hosted in MS2 to persist the data in C22 via discovery and routing layers. The tests are repeated couple more times where it would persist in 3, 4 and 5 collections in one transaction and time taken is recorded. Fig. 7 shows the performance of this technique.

B. Run 2 – Business capability Decomposition

In this run, the dependent collections were split as individual databases and built as six microservices MS1 to MS6. The reason for this is that in this model the services are built based on the business capability. This is to replicate the manager services for the normalized entities. In this approach a logic is added such that the transaction progresses sequentially in all steps. This is to replicate the behavior where the persistence would wait for the previous step to complete. Though this technique made the complexity of management services simple, the transactions were more time consuming as can be seen in Fig. 8.

C. Run 3 – Hybrid Decomposition

A hybrid approach is simulated where the same number of micro services as Run 1 are used, namely, MS1, MS2 MS3 and MS4. But the number of dependent collections are increased in MS1 and MS2 to 3 from 2 and reduced the collections in DB3 and DB4 to 1. This is to simulate a hybrid approach. The transaction which spans multiple collections is run and the timing is recorded as shown in Fig. 9.

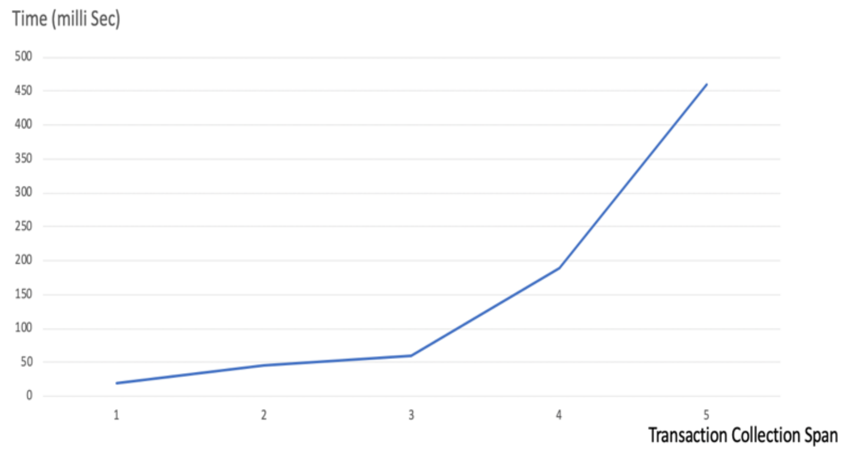

Fig. 7. Performance of Domain Driven Approach.

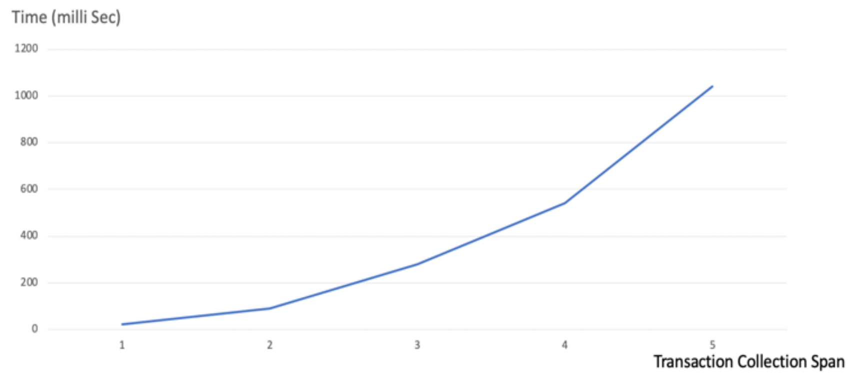

Fig. 8. Performance of Entity Driven Approach.

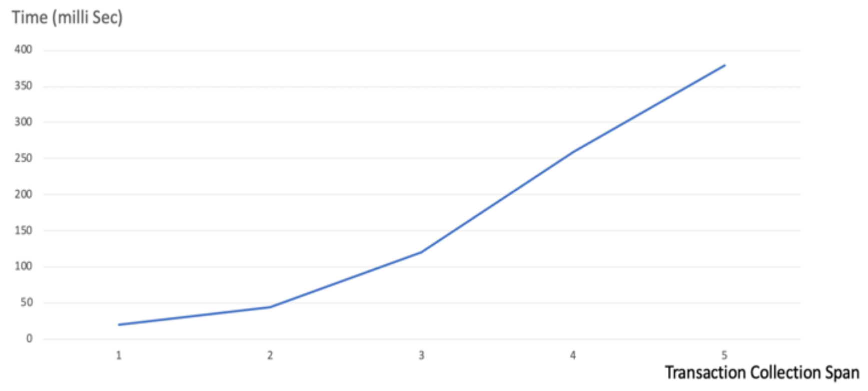

Fig. 9. Performance of Hybrid Approach.

IV. Conclusion

Though there is no standard approach for decomposing the micro services, it is found that Domain driven decomposition technique to be more superior than the entity model driven business capability-based decomposition. However, in any real time application design, choosing a hybrid approach by considering the big picture of the business functionalities and transactions involved would yield better results in the performance. Future scope of work includes simulating more complex transactions in each of the models and analyzing the resource consumption, throughput and latency.

References

[1] Salah, Tasneem & Zemerly, Jamal & Yeun, Chan & Al-Qutayri, Mahmoud & Al-Hammadi, Yousof. (2016). The evolution of distributed systems towards microservices architecture. 318-325. 10.1109/ICITST.2016.7856721.

[2] K., Chaitanya. (2018). A Systematic Study of Micro Service Architecture Evolution and their Deployment Patterns. International Journal of Computer Applications. 182. 18-24. 10.5120/ijca2018918153.

[3] S. Newman, Building Microservices. " O'Reilly Media, Inc.", 2015.

[4] Zhu, Yuhao & Richins, Daniel & Halpern, Matthew & Reddi, Vijay. (2015). Microarchitectural implications of event-driven server-side web applications. 762-774. 10.1145/2830772.2830792. R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[5] Asrowardi, Imam & Putra, S & Subyantoro, E. (2020). Designing microservice architectures for scalability and reliability in e-commerce. Journal of Physics: Conference Series. 1450. 012077. 10.1088/1742-6596/1450/1/012077.

[6] Balalaie, Armin & Heydarnoori, Abbas & Jamshidi, Pooyan. (2016). Microservices Architecture Enables DevOps: an Experience Report on Migration to a Cloud-Native Architecture. IEEE Software. 33. 1-1. 10.1109/MS.2016.64.

[7] Messina, Antonio & Rizzo, Riccardo & Storniolo, Pietro & Tripiciano, Mario & Urso, Alfonso. (2016). The Database-is-the-Service Pattern for Microservice Architectures. 9832. 223-233. 10.1007/978-3-319-43949-5_18.

[8] Rudrabhatla, Chaitanya. (2018). Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture. International Journal of Advanced Computer Science and Applications. 9. 10.14569/IJACSA.2018.090804.

[9] Malyuga, Konstantin & Perl, Olga & Slapoguzov, Alexandr & Perl, Ivan. (2020). Fault Tolerant Central Saga Orchestrator in RESTful Architecture. 278-283. 10.23919/FRUCT48808.2020.9087389.

[10] Jayasinghe, Malith & Chathurangani, Jayathma & Kuruppu, Gayal & Tennage, Pasindu & Perera, Srinath. (2020). An Analysis of Throughput and Latency Behaviours Under Microservice Decomposition. 10.1007/978-3-030-50578-3_5.

[11] Dayarathna, Miyuru & Perera, Srinath. (2018). Recent Advancements in Event Processing. ACM Computing Surveys. 51. 1-36. 10.1145/3170432.

[12] Dayarathna, Miyuru & Suzumura, Toyotaro. (2013). A Performance Analysis of System S, S4, and Esper via Two Level Benchmarking. 8054. 225-240. 10.1007/978-3-642-40196-1_19.

[13] G. Mazlami, J. Cito and P. Leitner, "Extraction of Microservices from Monolithic Software Architectures," 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, 2017, pp. 524-531, doi: 10.1109/ICWS.2017.61.

[14] Wuxia Jin, Ting Liu, Qinghua Zheng, Di Cui, Yuanfang Cai, "Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering", Web Services (ICWS) 2018 IEEE International Conference on, pp. 211-218, 2018.

[15] Justas Kazanavičius, Dalius Mažeika, "Migrating Legacy Software to Microservices Architecture", Electrical Electronic and Information Sciences (eStream) 2019 Open Conference of, pp. 1-5, 2019.

[16] Francisco Ponce, Gastón Márquez, Hernán Astudillo, "Migrating from monolithic architecture to microservices: A Rapid Review", Chilean Computer Science Society (SCCC) 2019 38th International Conference of the, pp. 1-7, 2019.

[17] Sara Hassan, Rami Bahsoon, Rick Kazman, "Microservice transition and its granularity problem: A systematic mapping study", Software: Practice and Experience, 2020.

[18] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga and D. Kröger, "Microservice Decomposition via Static and Dynamic Analysis of the Monolith," 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), Salvador, Brazil, 2020, pp. 9-16, doi: 10.1109/ICSA-C50368.2020.00011.

[19] W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability Agility and Reliability in E-Commerce", Proceedings of the IEEE International Conference on Software Architecture Workshops, pp. 243-246, 2017.

[20] Holger Knoche, Wilhelm Hasselbring, "Using Microservices for Legacy Software Modernization", Software IEEE, vol. 35, no. 3, pp. 44-49, 2018.