Microservices Data Consistency Approaches: A tailored approach towards current limitations

Asim Riaz

1732102

PhD CS

Dr. Husnain Mansoor Ali

SZABIST Karachi Campus

Contents

INTRODUCTION

The most crucial factor in the popularity of the microservices architectural style is its autonomous modularity. Any monolithic architecture can be easily transformed into microservices architecture by functional decomposition. This process attains strong isolation and provides scalability and variation of database systems, which helps deploy microservices in a broader spectrum. The fast-tracked adaptation of microservice style is the flexibility of designing service into any platform and vendor-neutral database system, which may help to increase its performance.

The most important aspect of microservices architecture is usually deployed in a cloud environment. It allows new practices and cutting-edge technologies like lightweight containers. All services related to any domain can be easily maintained and deployed using container orchestration technologies (e.g., [2] Docker Swarm, [3] Kubernetes, and[4] Moses. It also encourages a DevOps[1] environment which is industry top proven practices Many small and medium-sized organizations are adopting microservices architecture because it is independent of the domain and size of the organization even system. It shows significant differences in benefits and adaptations of better practices and new technologies that implicitly cure legacy issues of monolithic architecture. These new practices and technologies involve uncertainties, risks, and massive investment in process and infrastructure. To overcome the uncertainty and risk, the modularity of this architecture allows an organization's selective and gradual adaptation.

The gradual and selective transition allows an organization to incur more sophisticated solutions and infrastructure according to modern business demands. For example, an organization might consider adopting microservice architecture for fast delivery rather than flexible scalability because the system is running on a required and stable workload but unable to deliver fast delivery of new features. Therefore, an organization will focus more on the corresponding feature, and investment will be directed towards those advanced microservices infrastructure. There is a significant difference between the supported advantages and practices available in the literature and the industry implementation and achieved benefits. It is necessary to discover the difference in industry adaptation for microservices architecture [14]

Microservice Architecture

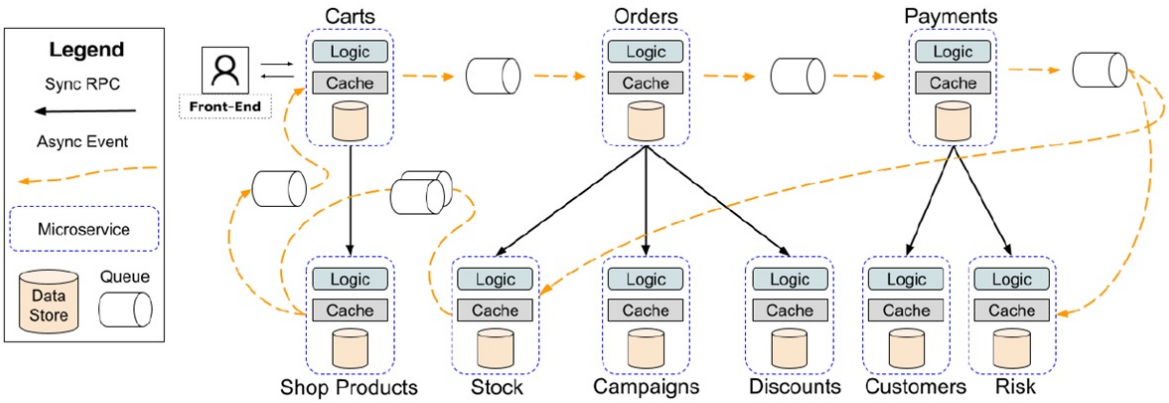

Figure 1: Microservices Architecture.

The shift in architectural design of data-driven applications is due to the beginning of large-scale online services. It initiates the requirement of distributed applications with both software development and computational requirement concerning team organization[**?**]. This situation promotes microservice architecture to take over traditional monolithic architecture because microservice architecture supports small-scale isolated services built and deployed independently. In contrast to this scenario, the conventional environment system is designed in a modular-based architecture, maintained centralized. We can analyze an e-commerce application to evaluate both architectures to depict the current scenario. In Figure 1, traditional architecture, a module is called which interns the next module or submodules to complete its functionality. e.g., a Cart module is responsible for to call Order module, which will call Stock, Campaign, and Discount modules sequentially. A transaction control protocol monitors all these modules.

In contrast to the direct function call, a microservices architecture communicates through remote calls such as asynchronous messages and HTTP-based protocol. In case of new functionality or a bug fix, only redeployment of a microservice unit is required, whereas, in the case of monolithic architecture, a new build or patch involves the deployment of the whole application; in a similar context, microservices manages failures, each service is isolated in nature which limits the failure propagation to other services or building blocks. Moreover, every microservice handles its repository requirement in handling data concerning its format to fulfill the workload of the microservice. This adaptability is associated with the diligence

of database architecture, and different services are maintained for the relational database management system and loosely structures NoSQL For instance, microservices may use loosely structured databases like NoSQL for variable structure data management, whereas another microservice may utilize a relational model to implement constraints on the data incurred. This results in substantial modification of microservice architecture transaction processing systems compared to monolithic architecture. Microservice architecture has an isolated environment that demands the decomposition of transactions into its service level. Otherwise, monolithic architecture transaction processing has well-defined steps for execution across modules.

1 Problem Statement

Most organizations are shifting their monolithic architectures to microservices architecture. The autonomous feature tends to implement data consistency model at application level according to their domain model. Software industry heavily relies on custom data consistency implementations while using microservice architecture. Even having multiple microservices data consistency frameworks and approaches available at their disposal, however, their adaption is limited. The research will:

- Pursue to identify the limitation of existing microservices data consistency approaches
- Segregate the limitations that hinder the adaptation of microservices data consistency approaches by industry
- Propose potential solutions to overcome selected limitations to increase industry adaptation of microservices data consistency approaches

2 Research Objective

1. Evaluate limitations of microservices data consistency approaches for industry adaptation
2. **Design and proposed Adaptation Model.**
3. Testing and evaluation of proposed adaptation model.

DESIGN AND PROPOSED ADAPTATION MODEL

3 SAGA Orchestration Vs Choreography

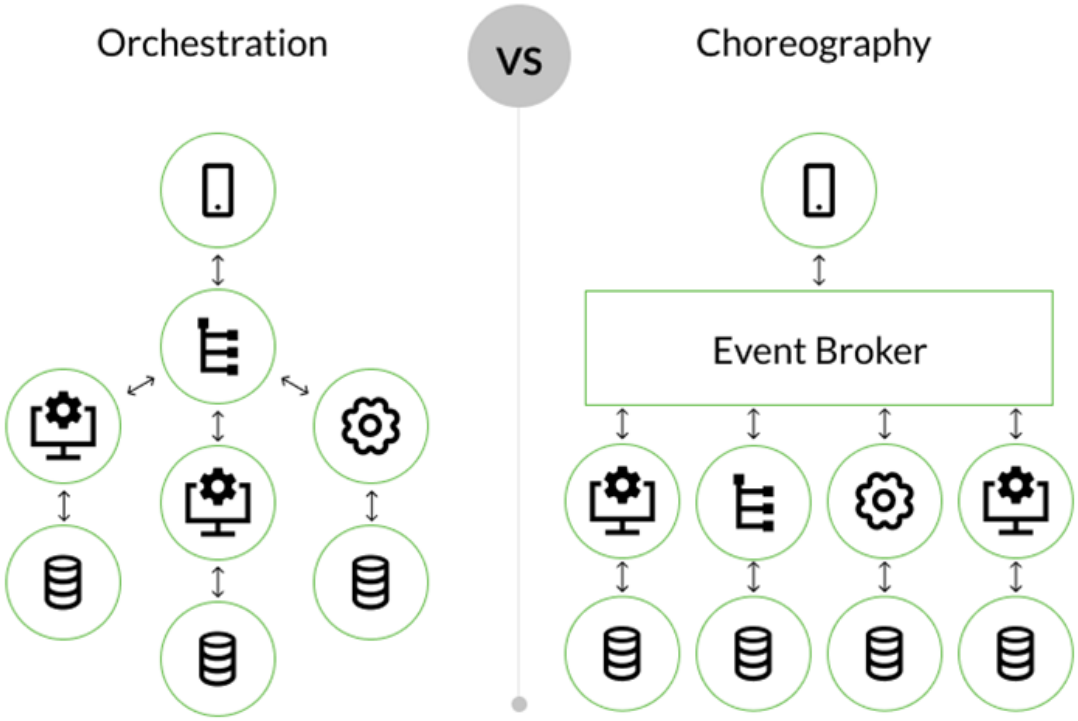

Figure 2: SAGA Orchestration Vs Choreography

The most important distinction between microservice choreography and orchestration is that there is no core focus in the case of choreography. The acceptance of any one of them is contingent on the use case that you have in consideration. If you require highly strong interaction among the service, then you have no choice but to go with orchestration; otherwise, choreography is the wiser option.[26]

In the case of orchestration, In Figure 2, we need to have good logging since, absent it, it would be quite challenging to determine which service was unsuccessful in correctly providing the data. The solution to any problem is simply to log the error. When designing the final version of the architecture, most architects combine the two different methods into a single strategy. This helps the whole system maintain its resilience. Because of the nature of some of the activities, which are synchronous, and the nature of other operations, which are asynchronous, both of these techniques are necessary to make your product fill proof.[5]

A choreographed microservices architecture makes the process of adding and delet-

ing services considerably less complicated. Because loose service coupling allows for the installation and removal of microservices without breaking current functionality, this approach results in less churn and flux in the development process. Due to the fact that choreography separates microservices, business services that are not reliant on a particular application are able to continue operating normally while the problem is being resolved. Choreography offers loose service coupling for agility and fault tolerance, which enables quicker, more agile development and yields applications that are more consistent and efficient.[28, 29]

4 Evaluation of Data Consistency in Microservices

When many instances of the same microservice are simultaneously attempting to access the same data set, Controlling the degree of concurrency an application is prepared to accept enables administrators to isolate the consequences of one transaction from those of other transactions.

4.1 Dirty Reads

It is possible for a transaction to do a dirty read if it reads data that was written by another transaction but has not yet been committed by the transaction that wrote the data. [8, 9]

4.2 Non-repeatable read

A non-repeatable read occurs when a transaction reads data but receives a different response when it rereads the same data inside the same transaction. This indicates that the read cannot be repeated. [8, 27]

4.3 Lost Updates

Attempts by two separate transactions to synchronize updates to the same data source fail. Unfortunately, while attempting to update one of them, the new value is not recognized. [7]

RELATED WORK

The shift from service-oriented monolithic systems to the more modern and flexible microservice architecture is becoming more common. On the other hand, this necessitates the segmentation of data that was before stored in a single database into many databases that are each specialised to a particular industry. So now developers have to deal with a new problem: functions like transaction processing, coordination, and consistency preservation that were enabled by a centralized database need to be handled in a decentralized, asynchronously communicating, distributed structure. Most previous research have shown that these problems are not effectively handled, which leads to inconsistencies in the system states, which may result in negative repercussions.

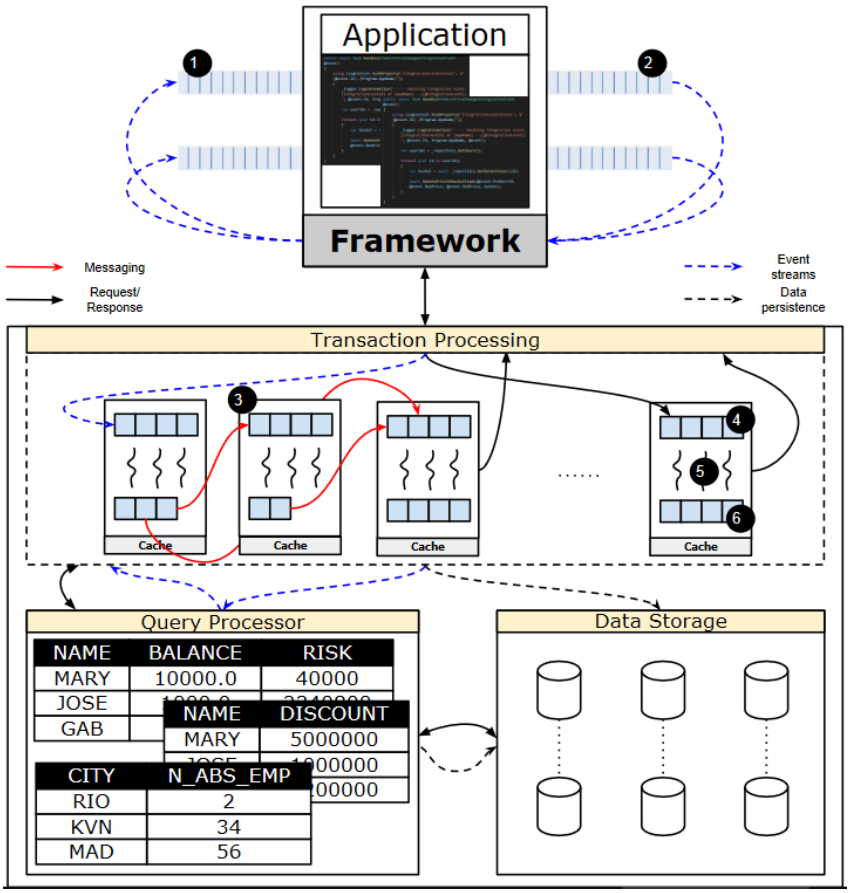

Figure 3: Virtual Microservices

Laigner et al. [19] suggests the potential for research and development for database management system which works in distributed environment which emphasizes on event-driven microservices. In addition, In Figure 3, they recommend that database systems be changed so

that they can better accept microservices and free programmers from the cumbersome data management duties that they now have to do [18]. The definition of microservice activities is accurately identified by a valid abstraction. contains information that is required to delegate data management duties to the system that controls the database.[13]

They represent the idea of using microservices that are virtualized as the fundamental structural components of a database system [16]. A virtual microservice duplex, also known as an abstract representation of a microservice application, should exist in the database for each microservice application. Moreover, a virtual instance of a microservice is a concept that separates incoming and outgoing event flows while conceptually encapsulating a specific microservice's current operational status, together with its constraints and data dependencies [21]. The idea is to introduce a virtual instance of the microservices, which enables to track the dataflow activities of the microservices but also about the operations that occurs across the microservices.[6, 15]

By using microservices, practitioners hope to get benefits like strong separation, ownership of data, and independence. To get these benefits, virtualized versions of microservices can be pushed down into the database. This means that all of these benefits can be built right into the database system. This is done by giving practitioners access to advanced data management functions that they don't have access to in modern database systems. Also, the database management system can support all of the benefits that practitioners hope to get from using microservices.[19, 13]

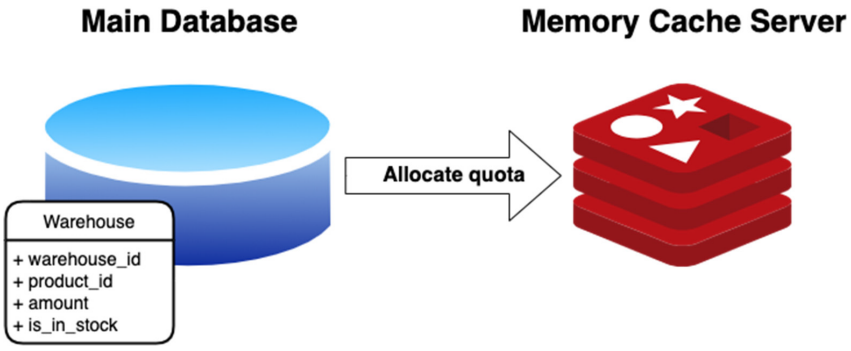

Figure 4: Quota Cache Example.

Daraghmi et al. [10] focuses on the enhancement of saga pattern by confining to stop the transaction failure or minimizing the chance of executing compensating transaction. Data consistency is assured in a distributed manner based on local transactions sequentially by

publishing message to next service.[22] The enhancement works on absence of isolation by restricting data inflow and outflow from an incomplete transaction. The resolution provides better transaction isolation by transferring few transactions to memory layer from data layer through quota cache and a service called commit-syncservice.[30] In Figure 4, A cache is a component that may be either hardware or software in a computer system that is used to store data. Usually used for high-speed data operations in memory. A cache makes it faster to get data by reducing the number of times the main memory has to be accessed. In this research the concept of quota cache has been utilized which is a resource specified by main database.[20]

Commit-sync service works synchronously manner by pending all commits until the commit completes successfully or an unrecoverable failure occurs (in which case it follows the saga pattern, it is given back to the service who called it.) The key responsibility of this service is to handle all commits to the main database because all CRUD operations are performed in memory cache server. The proposed solution handles all data related operation in memory cache server rather than a data source which is later synchronized with a service called commit-sync service. [17]

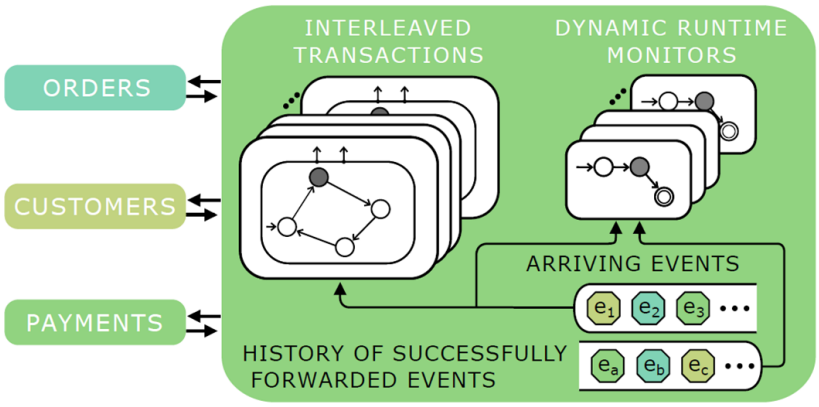

Figure 5: Architecture Connecting Microservices through a Coordinator.

Zuckmantel at al. [31] design a service for coordinating several microservices, as shown in Figure 5, by defining the formal semantics that describe how microservices communicate with one another and how data is shared between them. Furthermore, they formalize consistency properties and explain how they can be utilized to support both dynamic monitoring and enforcement of alignment features, making microservice systems that are reliable. The architecture in mind can make it a lot easier for developers to keep consistency across decentralized databases without having to implement complicated distributed algorithms.[11]

So, they come up with a way to record cross-microservice transactions in a way that makes sense. This will let us see the computational tasks of many microservice transactions all over the system and make decisions about how consistent each of those transactions is. So, it can plan for and control which events, when sent, will lead to secure transaction phases in the microservices that are affected. We need a system that can look at what happens to microservice databases when a cross-microservice transaction fails[8].

Frameworks that make an effort to overcome the difficulties of cross-microservice transactions often include centralized components. In such case the coordination is handled by a central event-bus component. A coordinator is in charge of coordinating the sequence of events in relation to the states of the microservices for this reason.[24]

An enhanced interface automaton may be used to represent state shifting. This provides a formal definition for concurrent execution of transaction that is focused on the data interchange and event exchange between each microservice.[12] Events are sent to the central coordinator by microservices that wish to start operations in other microservices. Based on the current state of the functionality mechanism encapsulating the distributed transactions, the central coordinator may assess if a state change does not jeopardies transaction guarantees.[23, 25]

5 Adaptation Model

Saga Pattern employ two modes of collaboration. First one is Orchestration, which works on request/response pattern and second one is Choreography which uses event driven architecture.

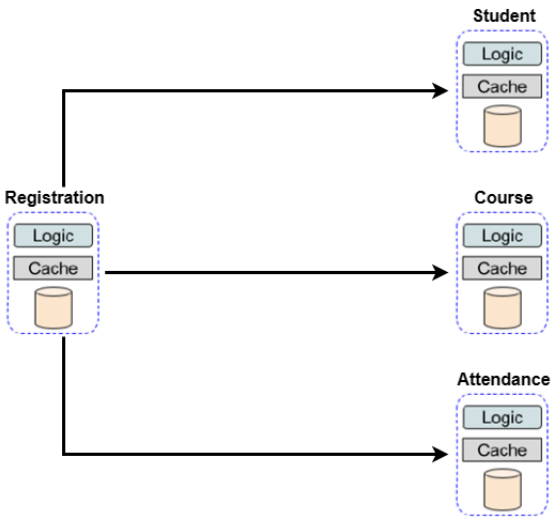

Figure 6: Orchestration

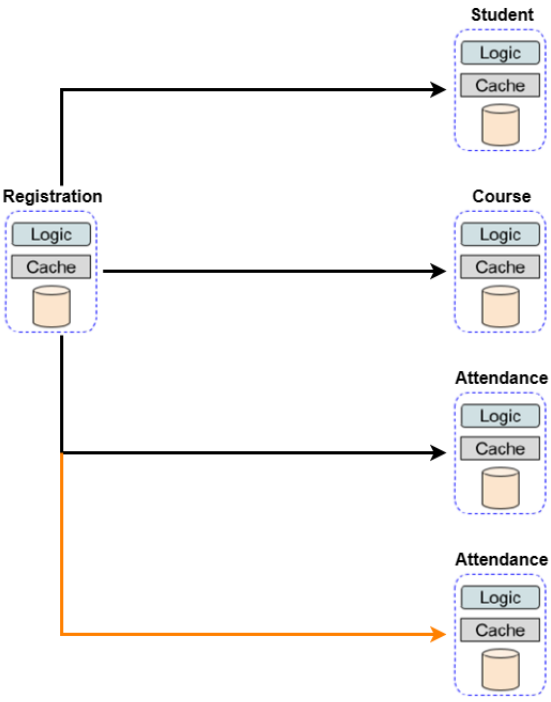

Figure 7: Addition in Orchestration

In orchestration as shown in Figure 6 everything is directed in the right direction by a centralized service. It communicates with other downstream services in order to ensure that everything goes according to plan, and those downstream services then reply to the requests that it sends their way.

In choreography, In the initial service, an event is only published (Figure 8). These event channels are available for subscription by different services, and when they see a certain event published, they may reply anyway they see fit. The registration service would function as our core service if we had a procedure that was well planned. Following the submission of the request by the student, the registration service will then send three API requests to the student service, the course service, and the attendance service respectively. The registration service notifies the student service to send out an email, the course service registers the student for the course, and the attendance service keeps track of the student's attendance for a specific event.

In this way, the registration service is like a conductor, telling everyone what to do and when to do it.With a choreographed or event-driven architecture, our services handle the

same business process in a slightly different way. When the process of signing up starts, the service for signing up would post an event. This event would have a name like ”RegistrationRequested” and would have information about the student. The event channel would be subscribed to, and our three services (student, course, and attendance) would be listening for events. Our three services would detect the ”RegistrationRequested” event when it is sent out and react appropriately by looking up students, enrolling them in courses, and keeping track of attendance. The main difference is that choreography, or event-driven architecture, allows us more flexibility when it comes to system adaptation.

What if, for instance, a fourth service is developed down the future and is likewise required to respond to course registration in some way? In an orchestrated approach (Figure 7), we would have to alter our registration service to send a fourth API call to this new service in order to upgrade our app. Changes of this kind would once again need us to arrange the registration service change. Unintentionally, we've linked our microservices together!

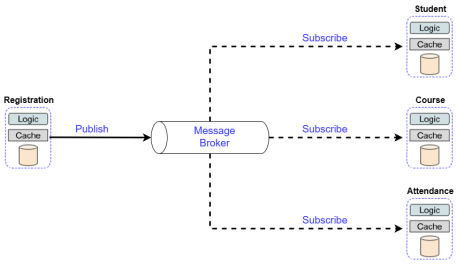

Figure 8: Choreography

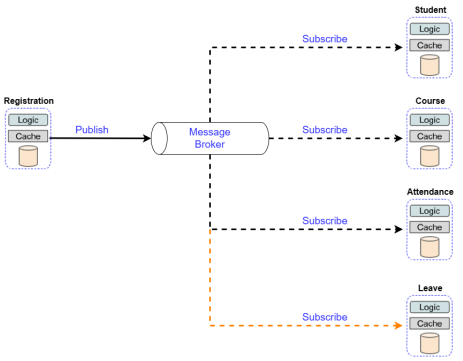

Figure 9: Addition in Choreography

5.1 Benefits of Choreography

Now, let's consider about the architecture that was choreographed. The registration service will only send out an event as shown in Figure 9, and all of the services that come after it will just react to the event. Do you really need the addition of a fourth service to the mix? No worries, just make sure that it subscribes to the event in question. Do you need to alter how an existing service reacts to the situation? No issue. We have the ability to alter the behavior of a downstream service without impacting any other services, such as the registration service. We have more flexibility because to choreography, which also keeps our services separate.

5.2 Publish / Subscribe Pattern

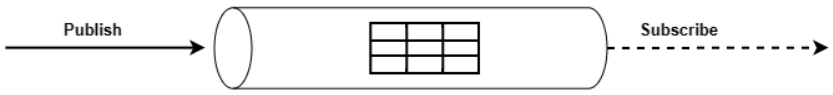

Figure 10: Publish/Subscribe Pattern.

With the saga design pattern, you can make sure that data is the same across microservices in cases where transactions are spread out. Basically, saga patterns let you make a set of transactions that update microservices in order and publish events that trigger the next transaction for the subsequent microservices. Saga patterns start rolling back transactions when one of the steps fails, which is essentially doing the opposite of what was done by posting rollback events to earlier microservices. Distributed Transactions between microservices are managed in this manner. As you are aware, the Saga design makes use of concepts like the publish/subscribe pattern with brokers and the patterns for API Composition.

The Adaptation Model will use the Saga Choreography Publish / Subscribe Pattern as shown in Figure 10. In which whenever a service publishes an event to a message broker. A pre-publish middleware event handles its functionalities before a service want to publish an event. Usually in choreography when a service wants to initiate a task which leads to subscribe multiple services it generates an event. All services which are willing to subscribe will listen to an event and perform certain task.

5.3 Pre-Publish Event

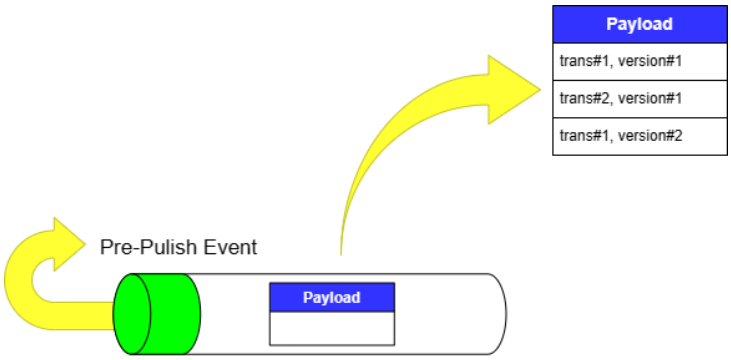

Figure 11: Pre-Publish Event.

Pre-publish event (Figure 14) will occur whenever a service want to publish an event for a particular task. The concept of pre-publish task was inherited from Debezium. Debezium is a platform for the open and distributed capturing of change data that is open source. After turning it on and directing it to your databases, your applications will be able to begin reacting to all of the insertions, updates, and deletions that are committed to your databases by other applications. Debezium works on multi model database environment in which multiple databases are required to synchronized. Pre-publish event can access the all the events previously available for execution in a any specific message broker. In choreography different services interact with different message broker depending upon the task and nature of functionality which is determined by business model. Let's consider an example in which two multiple services have been reading from third service data source. If another service wants to update the third service, pre-publish event can easily access the desired message broker and find that this data is already in read mode by other services so pre-publish event can be used to handle data accordingly.

5.4 Message Broker

Message broker is the core part of this adaptation model. In this solution message broker events must be persistent in nature so that when a service pre-publish event is captured it can access the events payload to find the availability of a service or a resource. In order to facilitate pre-publish event some information must be included along with the payload of the message. Following information are as follows:

5.4.1 Transaction Number

This unique id is required to track a transaction in multiple message brokers.

5.4.2 Version Number

A transaction can update any data multiple times which can be track by similar truncation number with different versions.

5.4.3 Status

This information is use to handle the status of any transaction that is pending due to any reason. in Saga compensation semantic lock are use to mark a transaction as PENDING for a particular event. When that event occurs of pending transactions are rollback according to its execution sequence.

5.4.4 Payload

it contains all the data and type of operation required to the subscribed services.

5.4.5 Data

This includes any other data required to maintain any information about distributed transaction.

5.4.6 Post-Event Handler

its is required for developer to handle any such which is require to execute after message processing.

6 Validation of Adaptation Model

6.1 Data validation across microservices

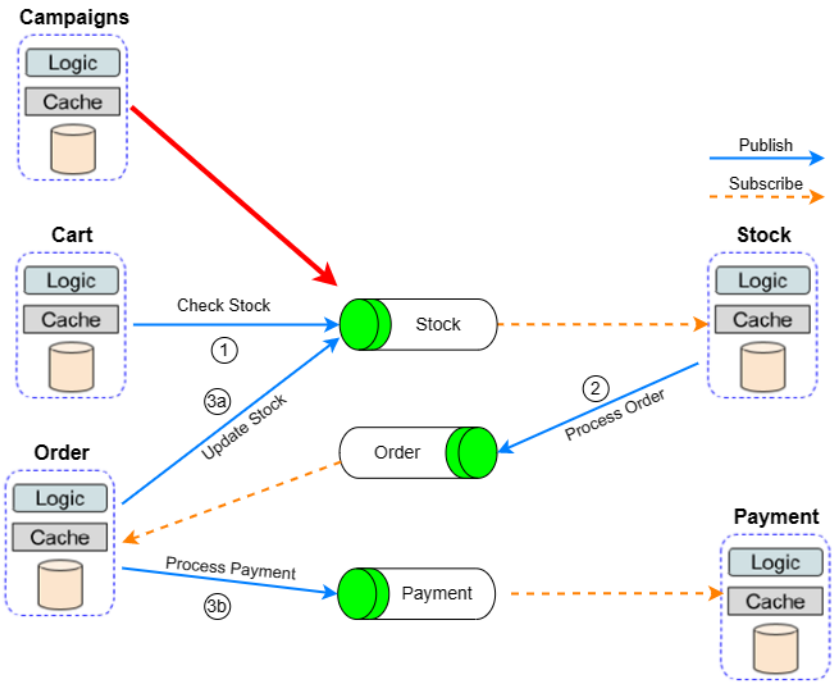

Figure 12: Data validation across microservices with Adaptation Model.

Across microservice data validation problem exist when any service can access call Sync RPC (synchronous Remote Procedure Call) and update the data which can create inconsistency between service that when any service should access any data. This problem can be resolve by saga choreography but with additional functionality it can be ensure that transaction number and status of the event in message broker it can be easily controlled by tagging a transaction status pending. When any service try to access a particular record in pre-publish event it can identity that another service is using the required data.

6.2 Foreign Key Constraint

Functional decomposition is the fundamental microservices dependent method. Decomposition isolates each service's database. This results in foreign key constraints or implicit data connections between different microservices. A problem arises when numerous services request the stock service asynchronously. If a service does an operation that can cascade delete or update

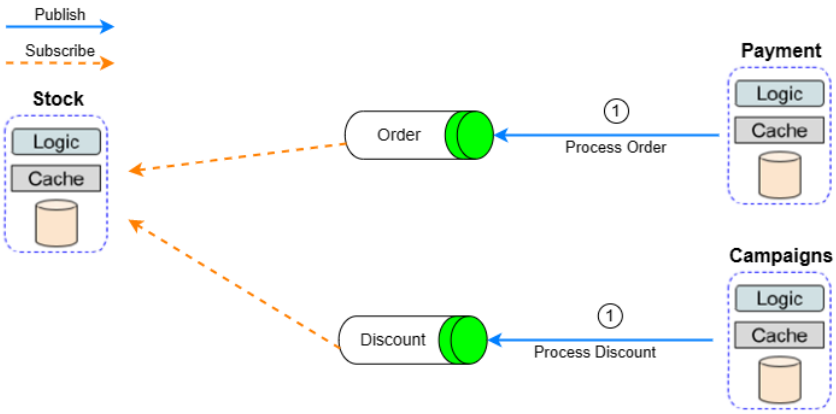

Figure 13: Foreign Key Constraint with Adaptation Model.

a master record, that operation can't be reflected in other transactions or services, especially in the delete operation. To keep data consistent, developers must impose foreign key constraints at the application level, making sure that the system is in a consistent state. This situation can be easily handled by version number in payload of a transaction. If a specific payload exists with a non-zero version number than that event can be postponed to a certain criterion.

6.3 Feral Ordering

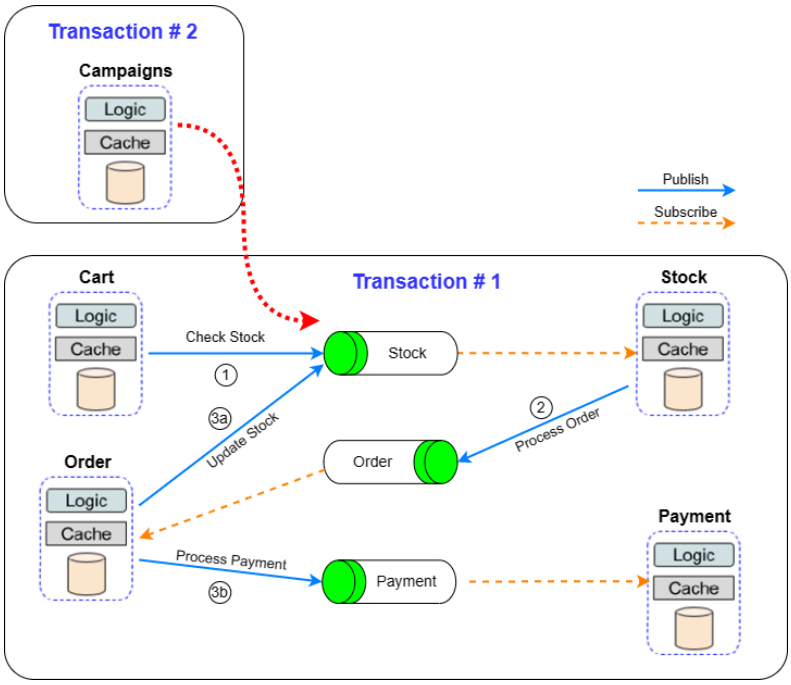

Figure 14: Feral Ordering with Adaptation Model.

This constraint is related to execution sequence of microservices. When a transac-

tion is in a process in which user has selected multiple items to purchase. Meanwhile in that period some other update the price of any product which is in stock. Updated price will not reflected in under process transaction they will be updated when services will re-acquire the price of that product. This situation can easily handle in a pre-publish event. Whenever any product amount is updated the status of under process event can be updated to the new price of at least that they can flag to be updated price in the custom post event handler

7 Research Roadmap

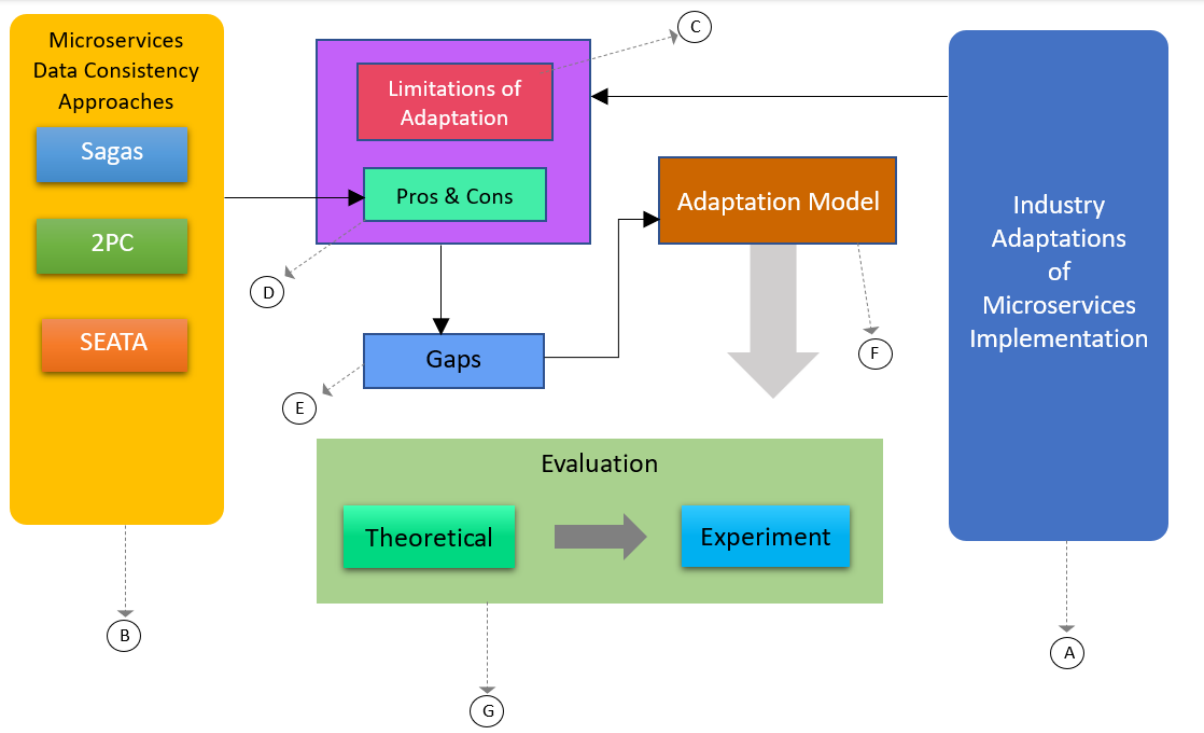

Figure 15: Research Roadmap.

A. **Industry adaptation microservices Implementations:** This is the core of this study in which adaptations of microservices are evaluated:

   - Why are organizations adopting microservices very rapidly?
   - Factor involves in migration from monolithic to microservices architecture?
   - What data consistency issues were faced by practitioners / developers?

B. **Microservice Data Consistency Approaches:** This area will explore all approaches available currently in industry for maintaining data consistency in a distributed environment.

C. **Limitation of adaptations:** What is the limitation of adopting microservices over monolithic architecture. Why almost every implementation is different from any other implementation in between organization

D. **Pros and Corns:** It will explore pros and corns of each Framework, Approach and practice used to maintain data consistency in microservices architecture

E. **Gaps:**The limitations of industry implementation and pros and cons of frameworks, approaches and practices will identify the reasons why almost all implementations of microservices differ from each other. The most important part is why developers have different approaches in maintaining data consistency at application-level. Some organizations have fully or partially implemented these models, but they must customize to some extent.

F. **Adaptation Model:** This will be the contribution towards the problem of data consistency. It will focus the limitation of industry implementation. It also investigates those area due to which organization are not able to completely implement the present solution. The most important aspect of this model it would focuses open architecture of microservices architecture. A microservice do not need to be rewritten to adapt a particular model, rather than it would an improvisation to the current implementation.

G. **Evaluation:** This phase is used to test the desired adaptation model. There will be two stages to test this model. First stage will be a theoretical model which can define the steps of using or integrating this model. The next stage will be the simulation part in which a model will be experimented on its adaptation

References

[1] System adminstrarion, Jan 2023. URL `https://devops.com/`.

[2] Accelerated, containerized application development, Jan 2023. URL `https://www.docker.com/`.

[3] Production-grade container orchestration., Jan 2023. URL `https://kubernetes.io/`.

[4] Apache mesos.., Jan 2023. URL `https://mesos.apache.org/`.

[5] S. Aydin and C. B. Çebi. Comparison of choreography vs orchestration based saga patterns in microservices. In *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, pages 1–6. IEEE, 2022.

[6] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[7] B. Christudas. *Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud*. Apress, 2019.

[8] B. Christudas. *Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud*. Apress, 2019.

[9] R. Cuomo, D. D'Agostino, and M. Ianulardo. Mobile forensics: Repeatable and non-repeatable technical assessments. *Sensors*, 22(18):7096, 2022.

[10] E. Daraghmi, C.-P. Zhang, and S.-M. Yuan. Enhancing saga pattern for distributed transactions within a microservices architecture. *Applied Sciences*, 12(12):6242, 2022.

[11] M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos. Distributed transactions on serverless stateful functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, pages 31–42, 2021.

[12] S. Debois, T. T. Hildebrandt, and T. Slaats. Replication, refinement & reachability: complexity in dynamic condition-response graphs. *Acta Informatica*, 55(6):489–520, 2018.

[13] P. Di Francesco, I. Malavolta, and P. Lago. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *2017 IEEE International conference on software architecture (ICSA)*, pages 21–30. IEEE, 2017.

[14] L. Frank and T. U. Zahle. Semantic acid properties in multidatabases using remote procedure calls and update propagations. *Software: Practice and Experience*, 28(1):77–98, 1998.

[15] M. O. Gökalp, A. Koçyiğit, and P. E. Eren. A visual programming framework for distributed internet of things centric complex event processing. *Computers & Electrical Engineering*, 74:581–604, 2019.

[16] J.-P. Gouigoux and D. Tamzalit. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE international conference on software architecture workshops (ICSAW)*, pages 62–65. IEEE, 2017.

[17] W. Hasselbring and G. Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246. IEEE, 2017.

[18] R. Laigner, M. Kalinowski, P. Diniz, L. Barros, C. Cassino, M. Lemos, D. Arruda, S. Lifschitz, and Y. Zhou. From a monolithic big data system to a microservices event-driven architecture. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 213–220. IEEE, 2020.

[19] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski. Data management in microservices: state of the practice, challenges, and research directions. *Proceedings of the VLDB Endowment*, 14(13):3348–3361, 2021.

[20] A. Luckow, L. Lacinski, and S. Jha. Saga bigjob: An extensible and interoperable pilot-job abstraction for distributed applications and systems. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 135–144. IEEE, 2010.

[21] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S. T. Larsen, and S. Dustdar. Microservices: Migration of a mission critical system. *IEEE Transactions on Services Computing*, 14(5):1464–1477, 2018.

[22] A. Messina, R. Rizzo, P. Storniolo, M. Tripiciano, and A. Urso. The database-is-the-service pattern for microservice architectures. In *Information Technology in Bio-and Medical Informatics: 7th International Conference, ITBAM 2016, Porto, Portugal, September 5-8, 2016, Proceedings 7*, pages 223–233. Springer, 2016.

[23] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4): 378–396, 1986.

[24] D. Monteiro, P. H. M. Maia, L. S. Rocha, and N. C. Mendonça. Building orchestrated microservice systems using declarative business processes. *Service Oriented Computing and Applications*, 14:243–268, 2020.

[25] R. Petrasch. Model-based engineering for microservice architectures using enterprise integration patterns for inter-service communication. In *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–4. IEEE, 2017.

[26] C. K. Rudrabhatla. Comparison of event choreography and orchestration techniques in microservice architecture. *International Journal of Advanced Computer Science and Applications*, 9(8), 2018.

[27] N. Santos and A. R. Silva. A complexity metric for microservices architecture migration. In *2020 IEEE international conference on software architecture (ICSA)*, pages 169–178. IEEE, 2020.

[28] S. Speth, S. Stieß, and S. Becker. A saga pattern microservice reference architecture for an elastic slo violation analysis. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 116–119. IEEE, 2022.

[29] S. Speth, S. Stieß, and S. Becker. A saga pattern microservice reference architecture for an elastic slo violation analysis. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 116–119. IEEE, 2022.

[30] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pages 1–12, 2012.

[31] T. Zuckmantel, Y. Zhou, B. Düdder, and T. Hildebrandt. Event-based data-centric semantics for consistent data management in microservices. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, pages 97–102, 2022.