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.[1, 15, 29] 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., [25]Docker Swarm, [26] Kubernetes, and [27]Moses. It also encourages a [28]DevOps 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. [16] 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.[30] 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 [9, 11, 5]

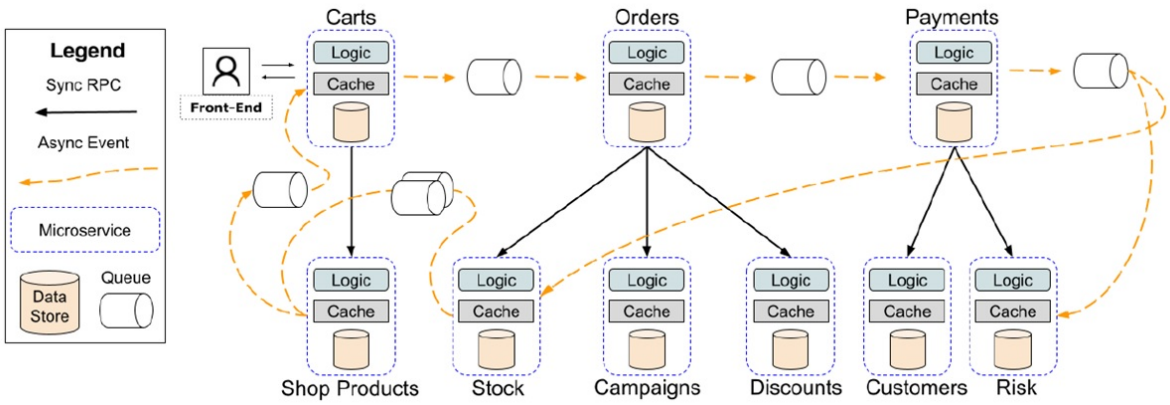

Figure 1: Microservices Architecture.[1]

Microservice 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.[2] 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.[40, 35] 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. [3, 4]

In contrast to the direct function call, a microservices architecture communicates through remote calls such as asynchronous messages and HTTP-based protocol. [31] 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. [13, 15] 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. [32] 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.[4, 9, 10]

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.

Objective - Breakdown

1. Limitations of Industry Adaptations for Microservices Implementations

2. Pros and Cons of Microservices Data Consistency Approaches / Frameworks

Limitations of Industry Adaptations for Microservices Implementations

3 Core Design Principles

A group of clearly laid out standard-based APIs reveal the capabilities of microservices. They contain a primary business component and are therefore assets to the organization. The reported standard based APIs are just described in terms of business, fully hiding the architecture of the Microservice, which may require interfaces with diverse data sources.[12]

3.1 Loose Coupling

Microservices should always be independent of one another and ought to be unaffected by changes to one service. To ensure total transparency of any modifications made to the Microservice's implementation, this is accomplished by only exposing the standard-based APIs. This enables the adjustment of the microservice's implementation without affecting downstream processes, such as by enhancing performance or switching data sources. [39]

3.2 Autonomy

During implementation, a microservice's autonomy is the degree of control it has over the database schema and runtime environment. This guarantees a better level of service quality and improves the microservice's performance and dependability. Autonomy helps the microservice's overall scalability and availability when combined with statelessness. microservices have the ability to cease communication with a failing microservice with whom they are working since they are autonomous from one another. The enterprise solution is prevented from being affected by a single microservice failure using this strategy.

3.3 Event Driven Architecture

A microservice in an event-driven architecture broadcasts an event whenever something significant occurs. When a microservice gets an event, it can respond appropriately, which may cause other events to be broadcast. Other microservices listen to those events.[39, 6]

4 Monolithic to Microservices Migration

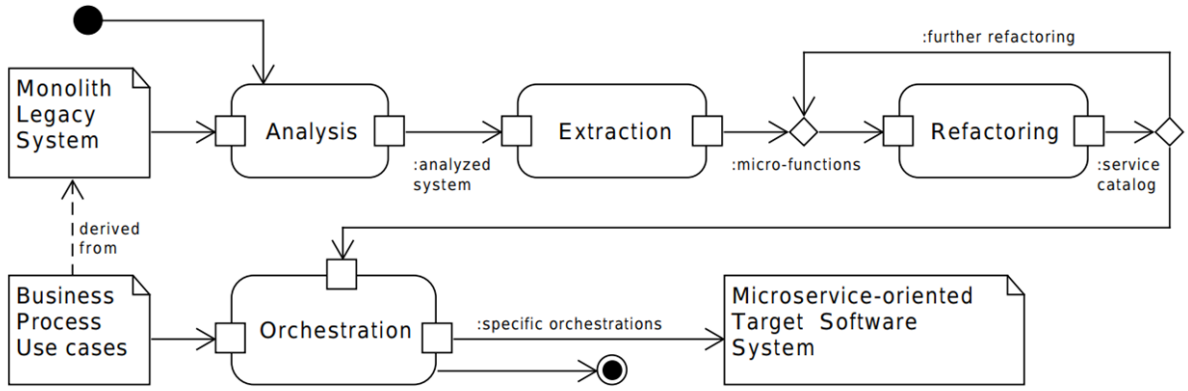

Figure 2: Migration Workflow.[6]

The IBM Market Development and Insights team had carried out a number of surveys in 2021 that captured the opinions and practical experiences of microservices users and those who were thinking about adopting them. More than 1,200 IT top management, developer executives, and developers from large and midmarket businesses that are currently utilizing a microservices approach were among them, in addition to nonusers who are presently investigating or intend to do so in the near future. The findings shed a lot of light on the opportunities and difficulties associated with applying a microservices development approach in the real world.

The research reveals that currently very few applications are design on core design principles of microservices from scratch. Majority of development in Microservices Architectures were migrated from traditional Monolithic Architecture. Which does not follow the overall core design principle. [8]

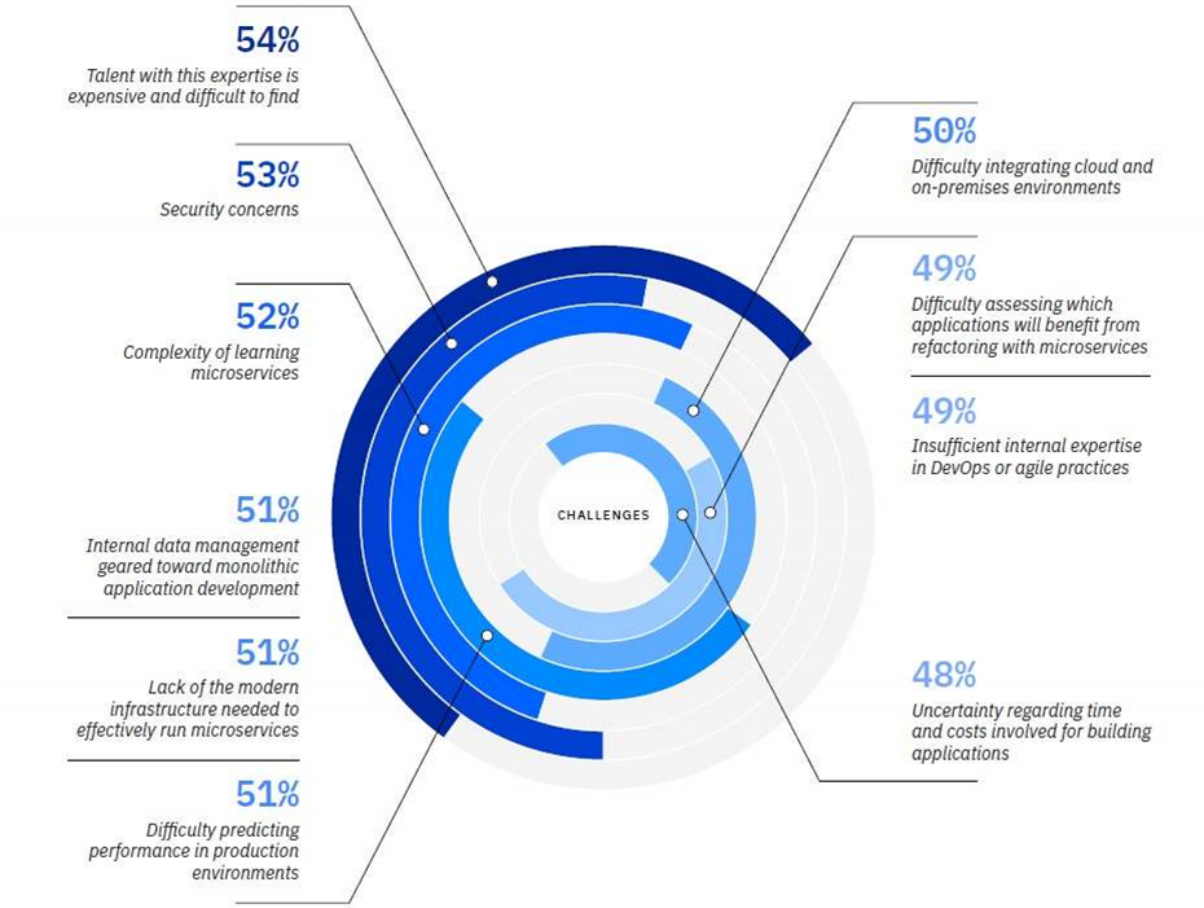

Figure 3: Data Management geared toward Monolith Application.[8]

Mostly in this type of migration data layer is not separated into each microservice, all services used a single data layer, which constrain the microservice architecture capabilities. The appropriate decomposition of microservices from monolith architecture with the required granularity might be considered as the key issue in the architectural migration. When microservices are not designed properly, tightly connected microservices are produced, sometimes known as distributed monoliths, which have all the drawbacks of a monolith and all the difficulties of microservices.

5 Migration Issues

5.1 Functional Decomposition

As the fundamental building blocks for development, deployment, and scalability, services are physically segregated in microservice systems. Serious quality issues might result from im-

proper service decomposition, like performance and scalability. However, physical isolation renders it hard for outsiders to access a service's core logics. Adopting microservices is largely motivated by the application's functional breakdown. A software development process called Domain Driven Design (DDD) places emphasis on translating ideas from the issue domain into artefacts from the solution domain. Despite not being tailored specifically for microservices, DDD is frequently utilized in microservices systems to enable a more efficient service decomposition.

5.2 Capture Service Dependencies

The quality of service decomposition is more challenging to assess than the modularity of monolithic systems. Monolithic systems' modularity is often assessed using evolutionary coupling of files and static dependencies. For microservice systems, mapping the service IP addresses or service names in the code to service repositories is necessary to perform static analysis to capture service dependencies. This mapping is frequently difficult and prone to mistakes. On the other side, it is impossible to detect evolutionary coupling by looking at revision histories because services are independently built-in distinct repositories. It is challenging to measure service coupling and further assess the quality-of-service decomposition when service dependencies are lacking.

5.3 Migration Complexity of Legacy Systems

It is sometimes challenging to transition a monolithic legacy system to a microservice architecture while simultaneously ensuring the ongoing delivery of business services because of the complex relationship between files. Since gradual migration to microservice systems typically affects microservice isolation, this issue mostly affects Independent Development and Deployment.

To gradually transition a legacy system to a microservice architecture, **Anti-Corruption Layer** and **Strangler Pattern** approaches are frequently combined.

5.3.1 Strangler Pattern

This pattern recommends gradually replacing particular capabilities with freshly created services. The old system and the new services are both utilized in tandem to serve the business during the migration process, which can last for a very long period. As all the legacy system are replaced by new services, it is eventually strangled and replaced by a microservice system and thus can be decommissioned. [2, 18]

5.3.2 Anti-corruption Layer

This pattern design recommends putting an anti-corruption layer in between the new services and the old system to isolate them from one another. This layer interprets communications so that the old system can maintain its original functionality while the new services can preserve its original design choices. [2, 15]

5.4 Database Decomposition

When the data components (such as fields or tables) in different services' databases are used in the same data query or transaction, this is known as data coupling. Cascading queries and conventional transaction management cannot be employed in microservice systems because of the physical isolation. This restriction on database sharing among services refers to centralized databases or database sharing among some services. This issue impacts all capabilities of Microservices architecture which includes Independent Development and Deployment, High Scalability and Availability and Service Ecosystem.

5.5 Deployment

Usually, sophisticated service configurations are a part of microservice systems. For instance, inappropriate or inconsistent service settings (such as inconsistent JVM and Docker memory constraints) may result in runtime errors. It indirect impacts microservices scalability, and availability and minimize the chances of extension of new requirements

PROS AND CONS: MICROSERVICES DATA CONSISTENCY APPROACHES / FRAMEWORKS

Maintaining data consistency in a microservice architecture is a difficult task as compared to traditional monolith applications because monolith architecture usually handles data consistency through the shared relational database management system. The microservice architecture uses a database per service pattern, in which each service has its data store. This independence of data store allows services to main autonomous behavior which is an essential part of the design principle for microservices. This autonomy enables to implementation of different technologies to handle data repositories like NoSQL, Redis, etc. Furthermore, this situation permits loosely coupling in microservices architecture which promotes agility, high availability, and scalable solutions.

Along with these perks of microservice architecture, there are critical points concerning data management related to transaction management and data integrity or consistency. Each microservice has its runtime and data storage in the distributed architecture, data is highly accessible and scalable.

6 The Problem: Data Consistency in Distributed Systems

A shared relational database handles and ensures data consistency for monolithic applications via ACID transactions.

Atomicity: A transaction's steps are all passed or failed at the same time. It makes sure that every step should execute completely or nothing.

Consistency: It monitors that at the end of every transaction, all data in the database is consistent.

Isolation: Always a single transaction can have exclusive access to data at the instance of time; all other transactions must wait for the working transaction to complete.

Durability: Data persistence in the database is ensured at the end of the transaction.

But in a microservice architecture, there is no single unit of work and no central database due to which every microservice has its runtime and a datastore with a diverse technology. The business logic is distributed across various native transactions. This means that in a microser-

vices architecture, a single transaction unit of work cannot be used across databases, which violates the rule of an ACID transaction, but it is necessary for data consistency.

In a sample scenario, Stock management, payment, and order management are all possible services in an order management system. Assume that these services are built using microservice architecture and that a database per service pattern is used. Each service will call its subsequent service to complete the order. Order service initializes the process by calling stock service for reservati0n and stock management through which products are reserved. To complete the process payment service will be called by the order service. All services are working autonomously because each service, updates to the separate database, are committed within its scope. In the last step, if the order is not generated, the payment was deducted from a customer's credit card. This situation raised data inconsistency in the process.

6.1 Possible Solutions

To begin with, there is no single solution that works well in every situation. Depending primarily on the use case, various solutions can be implemented. In a microservice architecture there are two main approaches to maintaining data consistency:

- Distributed Transaction
- Eventual Consistency

6.1.1 Distributed Transaction

[2]In a distributed transaction, transitions are carried out on multiple services, databases, and message queues. A distributed transaction manager ensures data integrity through all different databases. The complexity of distributed transactions is high because multiple resources are affected in a distributed transaction.

6.1.2 Eventual Consistency

The eventual consistency model emphasizes high availability in distributed systems. In this model, discrepancies are tolerated for a brief period until the distributed data problem is solved. This model uses the BASE database model instead of following distributed ACID transactions between microservices. As compared ACID model which delivers consistency, the BASE

database model offers high availability. The eventual consistency model is used by the common pattern known as SAGA.[22, 24]

[3] **BASE** model abbreviated as:

Basically **A**vailable: It replicates the data among the nodes of the database cluster to guarantee its availability

Soft State: Data may fluctuate over time due to a lack of significant consistency. The developers are in charge of maintaining consistency.

Eventual Consistency: Although immediate consistency may not be attainable with the BASE, it will be achieved gradually ensured relatively after a short time.

Maintaining data consistency between distant data stores can be quite difficult. To design new apps, a different perspective is required. In microservices architecture responsibility for data consistency shifts from the database to the application level. Microservices architecture has many advantages, including high availability, scalability, automation, autonomous teams, and so on. To get the most out of the microservice architectural style, a few technological adaptations to typical methodologies are required.[33]

7 Microservices Data Consistency Approaches / Frameworks

7.1 Two Phase Commit

[6, 21]This protocol is a technique to handle database transactions in a distributed environment like relational database management systems. It confirms the serialization of the transaction to maintain a centralized atomic commit. It is well defined that this protocol performance degrades in high throughput systems like cloud computing or microservices. To maintain data constancy in microservice concurrency can be maintained in a decentralized manner like in every microservice individually. In 2PC, a coordinator transaction observes the whole business process's integrity. If all sub-transactions are complete successfully, it commits the transaction successfully. On the other hand, if any sub-transaction fails to complete, the coordinator transaction calls rollback all successful transactions to a consistent state. The reason why 2PC is not efficient in cloud computing and the microservices environment is that it was difficult to maintain a state for coordinator transactions to track sub-transactions. 2PC follows a rule in which

all sub-transactions should be available at the time of commitment. If any sub-transaction fails, the whole transaction cannot be completed. To ensure the correctness of the transaction 2PC sub-transaction maintain a locking mechanism which may lead to a critical situation and can affect the throughput of the transaction. Another lacking in this technique is that support for modern technologies is missing e.g., message broker and NoSQL databases.

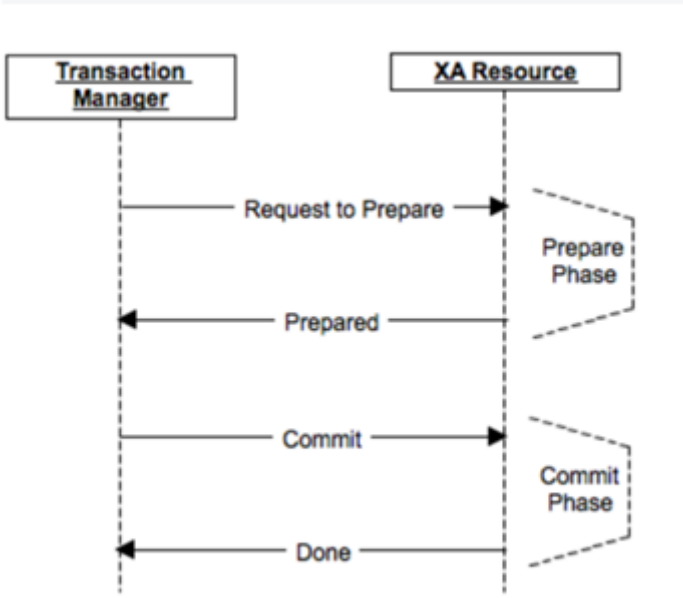

Figure 4: Two Phase Commit.[7]

The normal implementations of such a coordinator are typically synchronous, which could eventually result in a reduction in throughput.

7.2 Saga Pattern

The Saga pattern handle transaction in a distributed manner. Transaction control is dependent upon each microservices as compared to a whole transaction handling all microservices. This helps to reduce the dependency of transactions and supports a microservices architecture isolation environment. It does not ensure a transnational guarantee like 2PC but takes a long time to complete transaction requirements. In Saga, local transactions work sequentially, where each transaction is controlled by a single microservice. Initialization starts from an external event-based request. All transactions will be executed sequentially one after another. If any subsequent microservice failed to complete its transaction, a compensation action will be triggered to roll back the transaction to maintain consistency.[24]

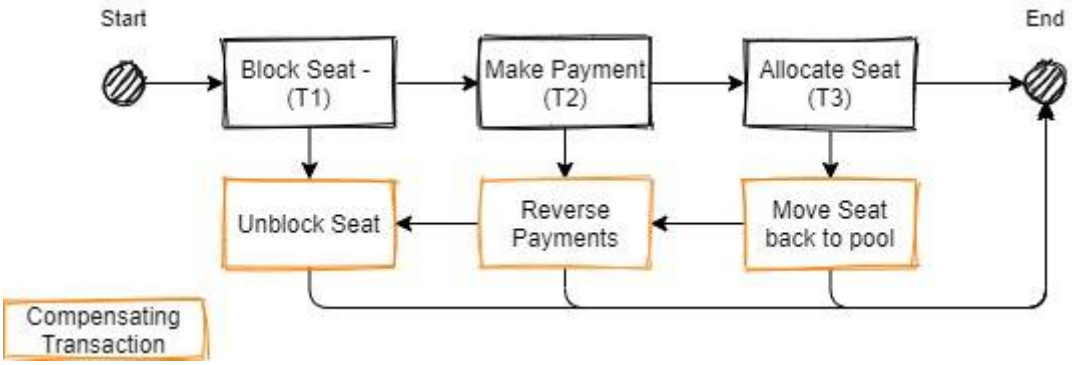

Figure 5: SAGA Pattern.[6]

The SAGA pattern makes use of the eventual consistency model. It employs an asynchronous paradigm based on several services. It handles distributed transactions locally in which each service controls its data for the transaction. In this manner, the SAGA pattern governs how the series of services are delivered. Two strategies **orchestration** and **choreography** can be used to get microservices to cooperate to achieve distributed transactions in the SAGA Pattern.

7.2.1 Orchestration

In orchestration, transactions are sequenced in accordance with business logic by a coordinator service known as Saga Execution Orchestrator (SEG). The orchestrator chooses which operation needs to be carried out. The orchestrator should reverse the prior actions if an operation fails, this process is called compensation operation. To maintain the system operating consistently, compensation operations are the measures to take when a breakdown occurs.[22]

Camunda and Apache Camel are two frameworks that are used to implement the Saga orchestration pattern.

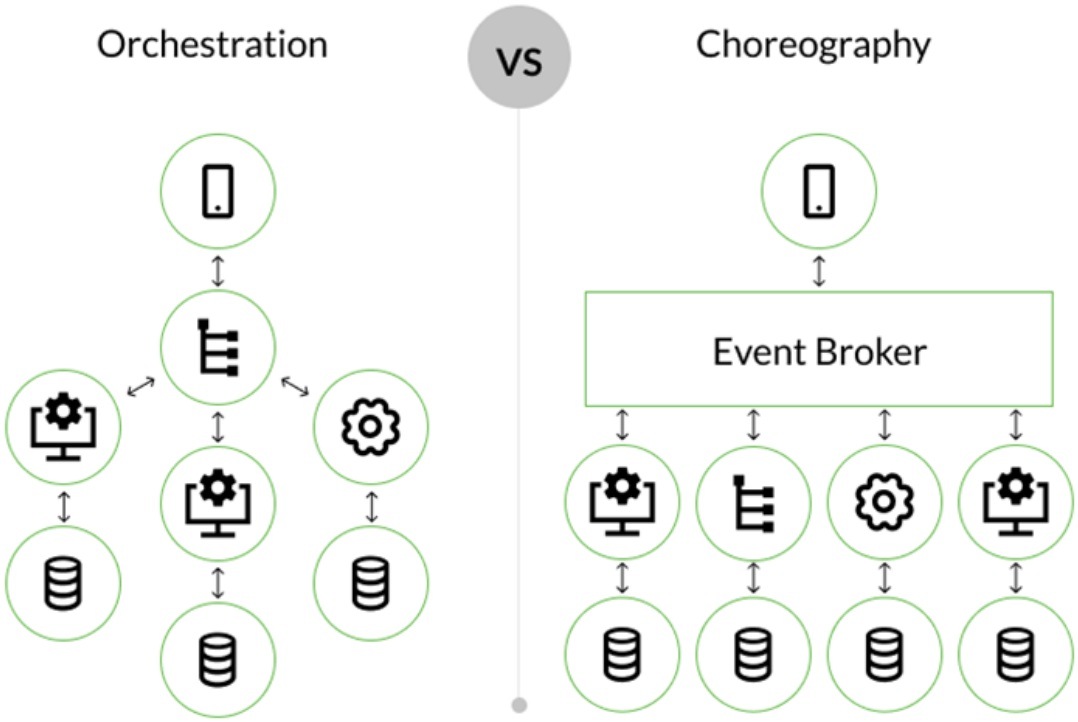

Figure 6: Orchestration VS Choreography.[12]

7.2.2 Choreography

In choreography, an event broker has a primary role if you want to coordinate between microservices and have a mechanism of sending messages between them whenever anything occurs. Following the completion of a task, each service creates an event, and each service monitors events to take appropriate action. An established event-driven infrastructure is needed for this pattern.[22]

7.3 SEATA

SEATA is an open-source framework for handling distributed transactions based on java. SEATA works on different modes in which AT, XA, TCC, and Saga are included. It provides solutions based on the domain model. It also implements a hybrid solution based on the requirement of a certain condition. In SEATA XA mode is implemented as a 2PC model and Saga works as it is. XA is the major feature of the framework which is unaware of the business code. It

focuses on business transactions (SQL) so that it can generate rollback data after analyzing. In the first stage, all sub-transactions or microservice involve in the transaction for local commit to generate roll back log. In the second phase, a global transaction works asynchronously to decide on commit or rollback depending upon the situation of all sub-transactions. TCC works on a Try, Confirm and Cancel basis. In the trial phase, all sub-transaction participated in the distributed transaction to acquire business resources. In confirm phase, if all sub-transaction were successful, it executes commit. Cancel phase initiates the cancellation of the business resource used in the trial phase [19]

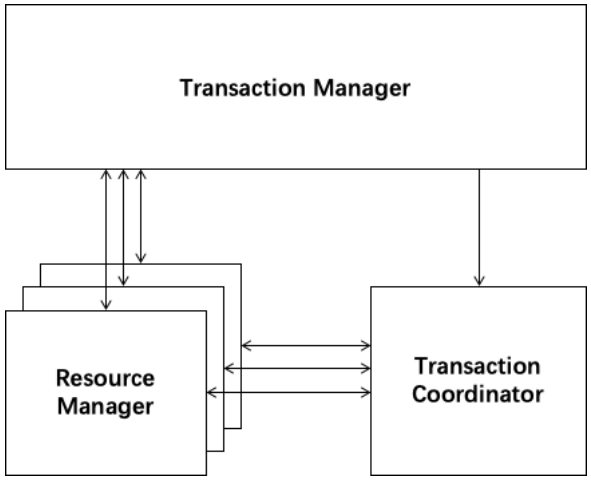

Figure 7: SEATA.[19]

RESEARCH WORK

8 Limitations

8.1 Two Phase Commit

- There is no mechanism to roll back the other transaction if one microservice goes down during the commit step.
- While the slowest service completes its confirmation, other services must wait. Until the entire transaction is finished, the resources needed by the services remain locked.
- Due to their reliance on the transaction coordinator, 2PC commits are slow by design. This can lead to scalability problems, especially in microservices-based applications and in scenarios where there are a lot of services involved in a roll-back.

8.2 SAGA Pattern

- SAGA does not provide a precise technique to estimate the length of time it will take for a system to reach a consistent state, hence eventual data consistency is not guaranteed.
- Doesn't have read isolation, requires more work, for example, the user might view the operation being done, but it is cancelled owing to a compensation transaction in a few seconds.
- When a service instance count gets higher, it was challenging to debug
- An increase in development costs is a result of the real service developments and the compensation that goes along with them.
- Compensation transactions make designs more complicated.

8.2.1 Orchestration

- Due to the possibility of multiple calls inside the retry mechanism, compensation operation must be idempotent.

- When data has been altered by another transaction, it might not already be possible to undo the changes.
- Compensation operations have a high complexity rate in a multi-instance microservice environment.

8.2.2 Choreography

- The state of the service is maintained by the event store usually worked as a message broker. States can be rebuilt by repeating an event from the event store for services.
- The scope of the choreograph microservice may impact the performance of the transaction. The higher number of steps in a transaction may lead to increase complexity, It is challenging to keep track of which services attend various events.

9 Design Considerations

1. If at all feasible, keep away of utilizing distributed transactions between microservices. Complex challenges arise while working with dispersed transactions.
2. Create your system with as little reliance on distributed consistency as you can. To do this, note the following transaction boundaries.
    - Decide which operations must take place inside the same work unit. For this kind of task, need strong consistency.
    - Determine whether operations are consistent enough to withstand any potential delays. For this kind of procedure, use eventual consistency.
3. If you want to use asynchronous, non-blocking service calls, think about employing event-driven architecture.
4. To maintain consistency, create fault-tolerant systems using compensation and reconciliation procedures.
5. A shift in perspective is necessary for creation and development of eventual consistent patterns.

10 Research Roadmap

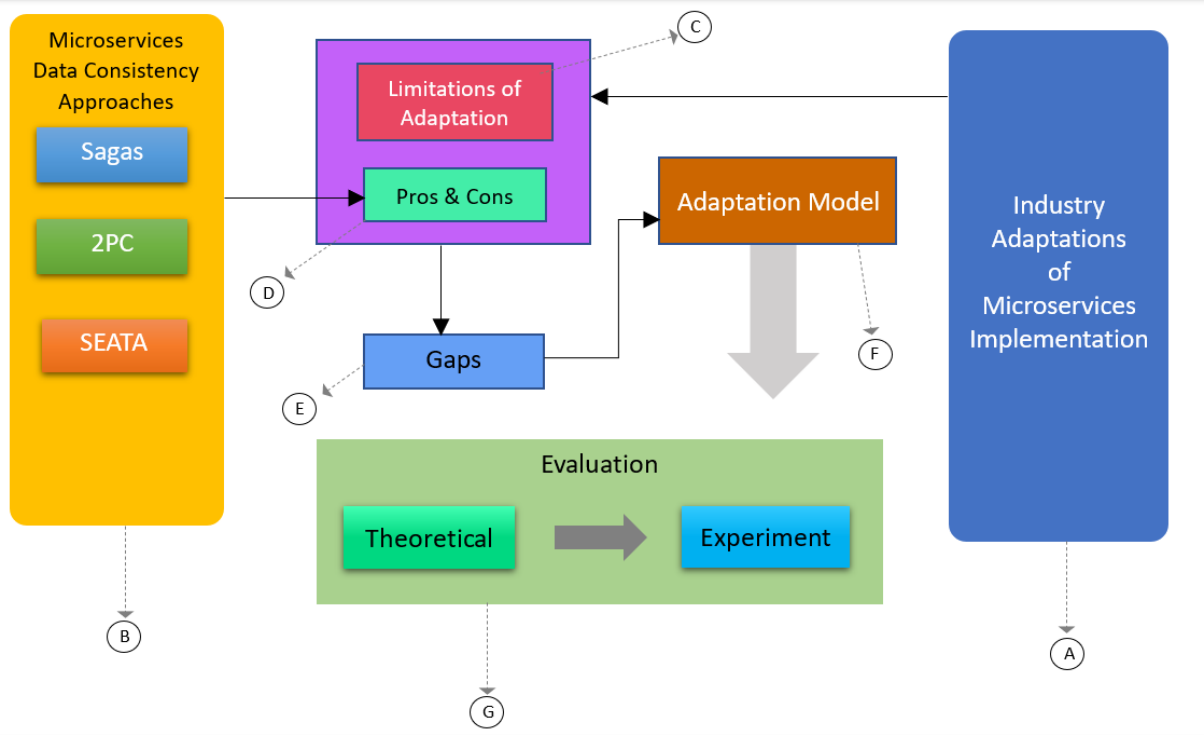

Figure 8: Research Roadmap.

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] Laigner, R., Zhou, Y., Salles, M.A.V., Liu, Y. and Kalinowski, M., 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. arXiv preprint arXiv:2103.00170.

[2] Xiang, Q., Peng, X., He, C., Wang, H., Xie, T., Liu, D., Zhang, G. and Cai, Y., 2021. No Free Lunch: Microservice Practices Reconsidered in Industry. arXiv preprint arXiv:2106.07321.

[3] Waseem, M., Liang, P., Shahin, M., Ahmad, A. and Nassab, A.R., 2021. On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study. In Evaluation and Assessment in Software Engineering (pp. 201-210).

[4] BaÅ¡karada, S., Nguyen, V. and Koronios, A., 2018. Architecting microservices: Practical opportunities and challenges. Journal of Computer Information Systems.

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

[6] Kuryazov, D., Jabborov, D. and Khujamuratov, B., 2020, October. Towards decomposing monolithic applications into microservices. In 2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT) (pp. 1-4). IEEE.

[7] Two Phase Commit. (2022). Available at: https://tbindi.github.io/tale/images/two.png.

[8] Microservices in the enterprise, 2021: Real benefits, worth the challenges. (2021). IBM Market Development Insights.

[9] Daja, D., 2020. Data Transformation: An Overview.

[10] Laigner, R., Zhou, Y. and Salles, M.A.V., 2021, June. A distributed database system for event-based microservices. In Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems (pp. 25-30).

[11] Bailis, P. and Ghodsi, A., 2013. Eventual consistency today: limitations, extensions, and beyond. Communications of the ACM, 56(5), pp.55-63.

[12] Microservice Orchestration and Choreography by Dev.Pro. (n.d.). Available at: https://dev.pro/wp-content/uploads/2022/03/Microservice-Orchestration-and-Choreography-by-Dev.Pro.*png*[*Accessed*24*Jun*.2022].

[13] Taibi, D., Lenarduzzi, V. and Pahl, C., 2017. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. IEEE Cloud Computing, 4(5), pp.22-32.

[14] Vale, G., Correia, F.F., Guerra, E.M., de Oliveira Rosa, T., Fritzsch, J. and Bogner, J., 2022, March. Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs. In 2022 IEEE 19th International Conference on Software Architecture (ICSA) (pp. 69-79). IEEE.

[15] HaselbÃ¶ck, S., Weinreich, R. and Buchgeher, G., 2018, November. An expert interview study on areas of microservice design. In 2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA) (pp. 137-144). IEEE.

[16] Bogner, J., Fritzsch, J., Wagner, S. and Zimmermann, A., 2019, March. Microservices in industry: insights into technologies, characteristics, and software quality. In 2019 IEEE international conference on software architecture companion (ICSA-C) (pp. 187-195). IEEE.

[17] SÃ¶ylemez, M., Tekinerdogan, B. and KolukÄ±*saTarhan*,*A*.,2022.*Feature* − *DrivenCharacterizationofMicroserviceArchitectures*:*ASurveyoftheStateofthePractice*.*AppliedSciences*,1

[18] Blinowski, G., Ojdowska, A. and PrzybyÅek, A., 2022. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. IEEE Access, 10, pp.20357-20374.

[19] seata.io. (n.d.). Seata. [online] Available at: https://seata.io/en-us/ [Accessed 2 Feb. 2022].

[20] Wu, M., Zhang, Y., Liu, J., Wang, S., Zhang, Z., Xia, X. and Mao, X., On the Way to Microservices: Exploring Problems and Solutions from Online QA Community.

[21] Faustino, D., GonÃ§alves, N., Portela, M. and Silva, A.R., 2022. Stepwise Migration of a Monolith to a Microservices Architecture: Performance and Migration Effort Evaluation. arXiv preprint arXiv:2201.07226.

[22] DÃ¼rr, K., LichtenthÃ¤ler, R. and Wirtz, G., 2021. An Evaluation of Saga Pattern Implementation Technologies. In ZEUS (pp. 74-82).

[23] Malyuga, K., Perl, O., Slapoguzov, A. and Perl, I., 2020, April. Fault tolerant central saga orchestrator in RESTful architecture. In 2020 26th Conference of Open Innovations Association (FRUCT) (pp. 278-283). IEEE.

[24] Kuryazov, D., Jabborov, D. and Khujamuratov, B., 2020, October. Towards decomposing monolithic applications into microservices. In 2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT) (pp. 1-4). IEEE.

[25] Docker (2018). Enterprise Application Container Platform — Docker. [online] Docker. Available at: https://www.docker.com/.

[26] Kubernetes (2019). Production-Grade Container Orchestration. [online] Kubernetes.io. Available at: https://kubernetes.io/.

[27] Apache Mesos. (n.d.). Apache Mesos. [online] Available at: https://mesos.apache.org/.

[28] DevOps.com. (n.d.). Home. [online] Available at: https://devops.com/ [Accessed 2 Feb. 2022].

[29] Di Francesco, P., Malavolta, I. and Lago, P., 2017, April. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In 2017 IEEE International Conference on Software Architecture (ICSA) (pp. 21-30). IEEE.

[30] Esposito, C., Castiglione, A. and Choo, K.K.R., 2016. Challenges in delivering software in the cloud as microservices. IEEE Cloud Computing, 3(5), pp.10-14.

[31] Yarygina, T. and Bagge, A.H., 2018, March. Overcoming security challenges in microservice architectures. In 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE) (pp. 11-20). IEEE.

[32] Taibi, D. and Lenarduzzi, V., 2018. On the definition of microservice bad smells. IEEE software, 35(3), pp.56-62.

[33] Richards, M., 2016. Microservices antipatterns and pitfalls. O'Reilly Media, Incorporated.

[34] Soldani, J., Tamburri, D.A. and Van Den Heuvel, W.J., 2018. The pains and gains of microservices: A systematic grey literature review. Journal of Systems and Software, 146, pp.215-232.

[35] Rademacher, F., Sorgalla, J. and Sachweh, S., 2018. Challenges of domain-driven microservice design: a model-driven perspective. IEEE Software, 35(3), pp.36-43.

[36] Hassan, S. and Bahsoon, R., 2016, June. Microservices and their design trade-offs: A self-adaptive roadmap. In 2016 IEEE International Conference on Services Computing (SCC) (pp. 813-818). IEEE.

[37] Bogner, J., Fritzsch, J., Wagner, S. and Zimmermann, A., 2019, March. Microservices in industry: insights into technologies, characteristics, and software quality. In 2019 IEEE international conference on software architecture companion (ICSA-C) (pp. 187-195). IEEE.

[38] Ford, N., 2018. The State of Microservices MaturityâSurvey Results.

[39] Kleppmann, M., Beresford, A.R. and Svingen, B., 2019. Online event processing: Achieving consistency where distributed transactions have failed. Queue, 17(1), pp.116-136.

[40] Helland, P., 2020. Data on the Outside vs. Data on the Inside: Data kept outside SQL has different characteristics from data kept inside. Queue, 18(3), pp.43-60.

11 Appendix - A

11.1 2PC - Log

```
Listing files to watch...
 - syedmustafacs/auth-service-jira
 - asmryz276299/client
 - mazahir10/order-service-jira
 - mazahir10/order-service-jira
Generating tags...
 - syedmustafacs/auth-service-jira -> syedmustafacs/auth-service-jira:d05eac9
 - asmryz276299/client -> asmryz276299/client:d05eac9
 - mazahir10/order-service-jira -> mazahir10/order-service-jira:d05eac9
 - mazahir10/order-service-jira -> mazahir10/order-service-jira:d05eac9
 - mazahir10/order-service-jira -> mazahir10/order-service-jira:d05eac9
Checking cache...
 - syedmustafacs/auth-service-jira: Found Locally
 - asmryz276299/client: Found Locally
 - mazahir10/order-service-jira: Found Locally
 - mazahir10/order-service-jira: Found Locally
Tags used in deployment:
 - syedmustafacs/auth-service-jira ->
        syedmustafacs/auth-service-jira:39e77d319dde92b5c5f2d7a048466d49b24e13b7f9815f20f78344a6ad0a8f2e
 - asmryz276299/client ->
        asmryz276299/client:c85b726a7fcf277304ac2c8ed8d4a0ee918c8ce569f4d16d0ddd5df6b40920b5
 - mazahir10/order-service-jira ->
        mazahir10/order-service-jira:22c49d9fcb4bbf5c042589154a7606b79a8e39b34ace27dbce7186575b6ef15e
 - mazahir10/order-service-jira ->
        mazahir10/order-service-jira:22c49d9fcb4bbf5c042589154a7606b79a8e39b34ace27dbce7186575b6ef15e
 - mazahir10/order-service-jira ->
```

```
        mazahir10/order-service-jira:22c49d9fcb4bbf5c042589154a7606b79a8e39b34ace27dbce7186575b6ef15e
Starting deploy...
 - deployment.apps/auth-depl created
 - service/auth-srv created
 - deployment.apps/client-depl created
 - service/client-srv created
 - deployment.apps/order-depl created
 - service/order-srv created
 - ingress.networking.k8s.io/ingress-service created
Waiting for deployments to stabilize...
 - deployment/client-depl is ready. [2/3 deployment(s) still pending]
 - deployment/auth-depl is ready. [1/3 deployment(s) still pending]
 - deployment/order-depl is ready.
Deployments stabilized in 2.633 seconds
Press Ctrl+C to exit
Watching for changes...
[order]
[order] > order-service@1.0.0 start
[order] > node index.js
[order]
[client]
[client] > comm-gen-react-app@0.1.0 start
[client] > react-scripts start
[client]
[auth]
[auth] > auth@1.0.0 start
[auth] > ts-node-dev src/index.ts
[auth]
[auth] [INFO] 11:09:51 ts-node-dev ver. 1.1.8 (using ts-node ver. 9.1.1, typescript ver. 4.7.4)
```

```
[order] Order Service Stared
[client] Initialize Prepare Mode
[order] Initialize Prepare Mode
[client] UPDATE customer SET fund = ? WHERE customer_id = ?
[client] Done
INSERT INTO order  ( id, order_id, p_id )  VALUES  ( ?, ?, ? )
[order] Done
[client] Commit Phase ... Done
[order] Commit Phase  ... Done
```

11.2 Saga Choreography

Request

```
curl --location --request POST 'http://localhost:8081/order/create' \
--header 'Content-Type: application/json' \
--data-raw '{
    "userId": 103,
    "productId": 33,
    "amount": 4000
}'
```

Kafka Payload

```
{"eventId":"b0e47448-eeb8-4cf4-bd29-b3a4315fc592",
    "date":"2022-06-03T17:26:46.777+00:00","orderRequestDto":{"userId":103,
    "productId":33,"amount":4000,"orderId":1},"orderStatus":"ORDER_CREATED"}


{"eventId":"c48c5593-9f81-4ab4-9de8-b9fca2d2bef2",
    "date":"2022-06-03T17:26:51.989+00:00","paymentRequestDto":{"orderId":1,
    "userId":103,"amount":4000},"paymentStatus":"PAYMENT_COMPLETED"}
```

Request

```
curl --location --request POST 'http://localhost:8081/orders' \
--header 'Content-Type: application/json' \
--data-raw '{
    "userId": 103,
    "productId": 12,
    "amount": 800
}'
```

Insufficient Amount


```
{"eventId":"fecacc77-017d-49cd-bdfa-58e47170da49",
    "date":"2022-06-03T17:28:23.126+00:00","orderRequestDto":{"userId":103,
    "productId":12,"amount":800,"orderId":2},"orderStatus":"ORDER_CANCELLED"}

{"eventId":"46378bbc-5d15-4436-bed1-c6f3ddb1dc31",
    "date":"2022-06-03T17:28:15.940+00:00","paymentRequestDto":{"orderId":2,
    "userId":103,"amount":800},"paymentStatus":"PAYMENT_FAILED"}
```


Request

```
curl --location --request GET 'http://localhost:8081/orders' \
--header 'Content-Type: application/json' \
--data-raw ''
```

Response


```
[
    {
        "id": 1,
        "userId": 103,
        "productId": 33,
        "price": 4000,
        "orderStatus": "ORDER_COMPLETED",
        "paymentStatus": "PAYMENT_COMPLETED"
    },
```

```
    {
        "id": 2,
        "userId": 103,
        "productId": 12,
        "price": 800,
        "orderStatus": "ORDER_CANCELLED",
        "paymentStatus": "PAYMENT_FAILED"
    }
]
```


User Balance

```
+----+------------+------+
| id | user_id | balance  |
+----+---------+----------+
|  1 |     101 |     5000 |
|  2 |     102 |     3000 |
|  3 |     103 |      200 |
|  4 |     104 |    20000 |
|  5 |     105 |      999 |
+----+-----------+-------+
2 rows in set (0.00 sec)
```

Order Repo

```
+----+--------------------+------+-----------+---------------------+----------
| id | order_status         | payment_status      | price   | product_id | user_id  |
+----+--------------------+------+-----------+---------------------+----------
|  1 | ORDER_COMPLETED      | PAYMENT_COMPLETED   | 40000   | 33         |      103 |
|  2 | CRDER_CANCELLED      | PAYMENT_FAILED      | 800     | 12         |      103 |
+----+--------------------+------+-----------+---------------------+----------+
2 rows in set (0.00 sec)
```

11.3 SEATA - TCC

```
2022-06-21 15:57:35.909  INFO 9104 --- [nio-8070-exec-1] i.seata.tm.api.DefaultGlobalTransaction
2022-06-21 15:57:39.033  INFO 9104 --- [nio-8070-exec-1] ShardingSphere-SQL
2022-06-21 15:57:39.033  INFO 9104 --- [nio-8070-exec-1] ShardingSphere-SQL
2022-06-21 15:57:39.063  INFO 9104 --- [nio-8070-exec-1] c.d.m.o.a.impl.UserOrderTccActionImpl
2022-06-21 15:57:39.149  INFO 9104 --- [h_RMROLE_1_1_12] i.s.c.r.p.c.RmBranchRollbackProcessor
2022-06-21 15:57:39.157  INFO 9104 --- [h_RMROLE_1_1_12] io.seata.rm.AbstractRMHandler
2022-06-21 15:57:39.174  INFO 9104 --- [h_RMROLE_1_1_12] c.d.m.o.a.impl.UserOrderTccActionImpl
2022-06-21 15:57:39.191  INFO 9104 --- [h_RMROLE_1_1_12] ShardingSphere-SQL
2022-06-21 15:57:39.191  INFO 9104 --- [h_RMROLE_1_1_12] ShardingSphere-SQL
2022-06-21 15:57:39.197  INFO 9104 --- [h_RMROLE_1_1_12] io.seata.rm.AbstractResourceManager
2022-06-21 15:57:39.199  INFO 9104 --- [h_RMROLE_1_1_12] io.seata.rm.AbstractRMHandler
2022-06-21 15:57:39.314  INFO 9104 --- [nio-8070-exec-1] i.seata.tm.api.DefaultGlobalTransaction
2022-06-21 15:57:39.338 ERROR 9104 --- [nio-8070-exec-1] o.a.c.c.C.[.[.[/].[dispatcherServlet]

: Begin new global transaction [192.168.1.188:8091:27149561796388640]
: Rule Type: master-slave
: SQL: INSERT INTO user_order  ( id, order_id, p_id ) VALUES ( ?, ?, ? ) ::: DataSources: master
: geneOrder--------------------1387677438737305602
: rm handle branch rollback process:xid=192.168.1.188:8091:27149561796388640,branchId=27149561796388658,
            branchType=TCC,resourceId=gene-order,applicationData={"actionContext":{"action-start-time":16196
            "sys::prepare":"geneOrder","sys::rollback":"cancel","sys::commit":"commit","id":13876774387373
            "host-name":"169.254.174.68","actionName":"gene-order"}}

: Branch Rollbacking: 192.168.1.188:8091:27149561796388640 27149561796388658 gene-order
: cancel--------------------1387677438737305602
: Rule Type: master-slave
: SQL: DELETE FROM user_order WHERE id=? ::: DataSources: master
: TCC resource rollback result : true, xid: 192.168.1.188:8091:27149561796388640,
```

```
            branchId: 27149561796388658, resourceId: gene-order
: Branch Rollbacked result: PhaseTwo_Rollbacked
: [192.168.1.188:8091:27149561796388640] rollback status: Rollbacked
: Servlet.service() for servlet [dispatcherServlet] in context with path []
            threw exception [Request processing failed;
            nested exception is java.lang.ArithmeticException: / by zero] with root cause

java.lang.ArithmeticException: / by zero



2022-06-21 15:57:39.221  INFO 4656 --- [h_RMROLE_1_1_12] i.s.c.r.p.c.RmBranchRollbackProcessor
2022-06-21 15:57:39.229  INFO 4656 --- [h_RMROLE_1_1_12] io.seata.rm.AbstractRMHandler
2022-06-21 15:57:39.251  INFO 4656 --- [h_RMROLE_1_1_12] c.d.m.p.a.impl.CompanyProductActionImpl
2022-06-21 15:57:39.252  INFO 4656 --- [h_RMROLE_1_1_12] ShardingSphere-SQL
2022-06-21 15:57:39.252  INFO 4656 --- [h_RMROLE_1_1_12] ShardingSphere-SQL
2022-06-21 15:57:39.257  INFO 4656 --- [h_RMROLE_1_1_12] ShardingSphere-SQL
2022-06-21 15:57:39.257  INFO 4656 --- [h_RMROLE_1_1_12] ShardingSphere-SQL
2022-06-21 15:57:39.264  INFO 4656 --- [h_RMROLE_1_1_12] io.seata.rm.AbstractResourceManager
2022-06-21 15:57:39.266  INFO 4656 --- [h_RMROLE_1_1_12] io.seata.rm.AbstractRMHandler

: rm handle branch rollback process:xid=192.168.1.188:8091:27149561796388640,branchId=27149561796388642,
            branchType=TCC,resourceId=order-decuct,applicationData={"actionContext":{"action-start-time":161
            "sys::prepare":"deduct","sys::rollback":"cancel","sys::commit":"commit","id":1,
            "host-name":"169.254.174.68","actionName":"order-decuct"}}

: Branch Rollbacking: 192.168.1.188:8091:27149561796388640 27149561796388642 order-decuct
: cancel--------------------1
: Rule Type: master-slave
```

```
: SQL: SELECT  id,product_name,account  FROM company_product WHERE (id = ?) ::: DataSources: slave02
: Rule Type: master-slave
: SQL: UPDATE company_product  SET product_name=?,account=?  WHERE id=? ::: DataSources: master
: TCC resource rollback result : true, xid: 192.168.1.188:8091:27149561796388640,
          branchId: 27149561796388642, resourceId: order-decuct
: Branch Rollbacked result: PhaseTwo_Rollbacked
```