Microservices Data Consistency Approaches: A tailored approach towards current limitations

Asim Riaz

1732102

PhD CS

Dr. Husnain Mansoor Ali

SZABIST Karachi Campus

Contents

INTRODUCTION

Microservices make it possible to divide large projects into a number of smaller, more manageable subprojects that may function independently of one another. In order to fulfil a single user request, an application that was created utilizing the architecture of microservices may need to coordinate with several internal microservices.

1 Adaptation Model

The main objective of the adaptation model lies in upholding transactions across the entire microservice environment. In such an environment, the developer is tasked with determining the order of microservice execution. In a microservices environment, it is customary for the initial service to initiate a transaction and subsequently send the corresponding transaction Id in all subsequent messages. This practice ensures that all microservices involved in the transaction record the transactionId in their respective database operations. The adaptation model is capable of monitoring and recording all transactions within the system. The transaction manager is responsible for maintaining comprehensive information regarding all ongoing transactions within a given environment. Additionally, it is tasked with tracking and executing rollbacks for any incomplete transactions.

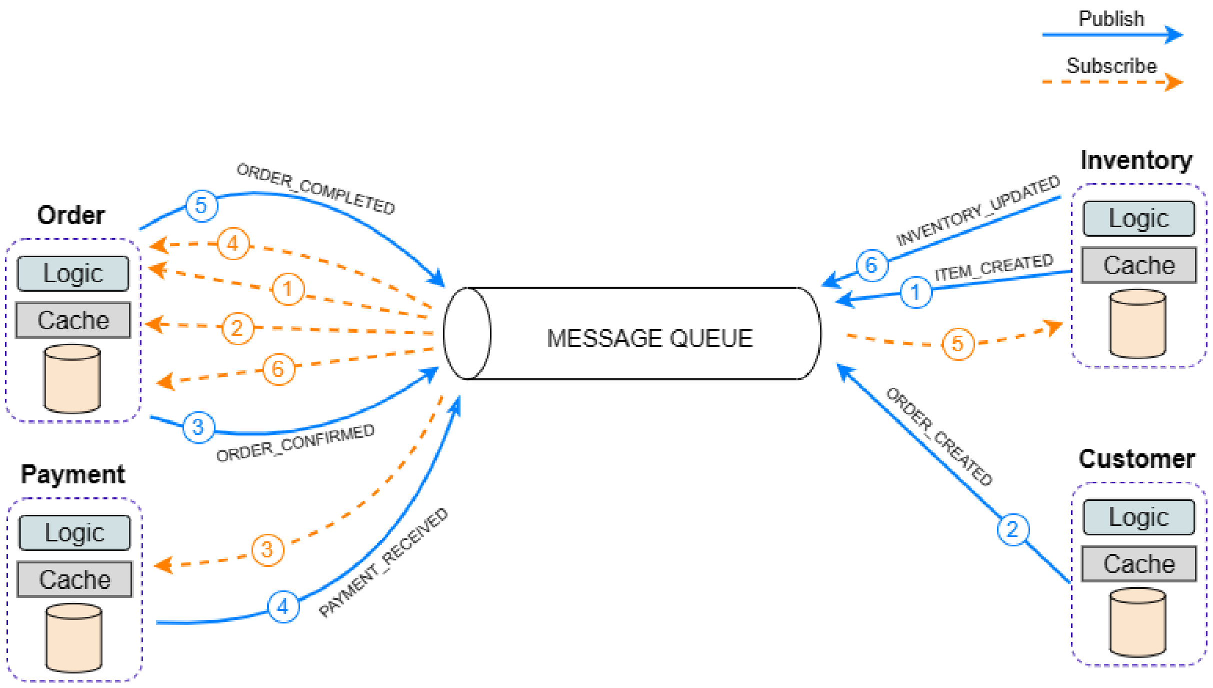


Figure 1: Basic Microservice

2 Basic Microservices Model

To evaluate the Adaptation model, it is necessary to define a basic environment in which any transaction in a microservice environment may works, for this purpose basic sales information system was considered as an example to study all its requirement. In this example total 4 microservices we involved to complete a basic process. The process involved 4 microservices in which order, inventory, customer and payment services are involved.

The fundamental operational context is established through the incorporation of inventory within inventory services, which is subsequently cached within the order service to facilitate monitoring of any modifications transpiring between the two aforementioned services. The inventory service is responsible for storing comprehensive information pertaining to inventory, whereas the order service solely retains pertinent fields necessary for querying or listing data associated with each order. The order services retain the information pertaining to the items in the inventory surface within a temporary table. It is beneficial to have a system in place that enables the tracking of all orders. Furthermore, it serves as a preventative measure against the need for the order service to make multiple API calls to the inventory service. The aforementioned practice has the potential to contribute to the occurrence of data inconsistency. However, in the context of saga choreography, the utilization of a message queue can effectively manage such instances of inconsistency. Hence, the exclusive and accurate data can solely be accessed through the inventory service.

2.1 Message Queue Events

1. The initiation of the transaction occurs upon the publication of an event denoted as ORDER_CREATED by the customer service, which is subsequently subscribed to by the order service. A service order is generated containing an Order Number, Item ID, and Quantity, with a status of ”Pending”.

2. The Order Service issues an event denoted as ORDER_CONFIRMED, which is subscribed to by the payment service. The received service order includes essential details such as the order ID, amount, and date, as well as a pending status. However, the information regarding the transactionId, bank transactionId, and bank name is incomplete.

Subsequent to this event, the order service's status was modified to ”Paid in Transit”.

3. Upon receipt of payment, the payment service publishes an event PAYMENT_RECEVIED, indicating the receipt of payment, which is subsequently subscribed to by the order service. During this particular event, the payment service obtains comprehensive payment information, such as the transactionId, online transactionId, and bank name. The order service subscribes to this event and updates the status to ”paid” upon receipt of the payment information.

4. The Order Service triggers the publication of a new event, ORDER_COMPLETED, upon receipt of payment for an order. The Inventory Service subscribes to this event and updates its data accordingly, resulting in the updated inventory information in the Inventory Service.

5. The most recent inventory update event, INVENTORY_UPDATED, is disseminated by the inventory service and received by the order service, which in turn updates the placeholder table of items. This enables the orderly processing of subsequent orders.

2.1.1 ITEM CREATED

Message Event Number # 1	
Event Name :	ITEM_CREATED
Description :	Customer select an item from list of products
Publisher :	Inventory Service
Subscriber :	Order Service
Event Details :	Item details are stored in a Order Services, to keep track of item for a particular order

An event of ITEM_CREATED is published by inventory service which is subscribed by Order service. Order service stores all the information of item(s) in cache storage. Cache storage is utilized to verify the order and status listing of items, thereby preventing the need for regeneration of an API call in case an order item is needed.

Order Service

Order

OrderId	OrderNo	ItemId	Quantity	Status

Items

ItemId	ItemName	Price
66	Pencil	50
45	Pen	30
75	Marker	25
37	Scale	10

Inventory Service

Inventory

ItemId	ItemName	Quantity	Price	Status
66	Pencil	1	50	
45	Pen	4	30	
75	Marker	7	25	
37	Scale	2	10	

2.1.2 ORDER CREATED

Message Event Number # 2	
Event Name :	ORDER_CREATED
Description :	Order is created in Order Service
Publisher :	Customer Service
Subscriber :	Order Service
Event Details :	Order is created in order table of Order service.

An event of ORDER_CREATED is published by Customer service which is subscribed by Order service. Order information is stored in order table in which *OrderId*, *OrderNumber* and *ItemId* is stored whereas Status field is Pending for further processing.

Customer Service

Customer

CustomerNo	**Name**	**Address**	**Email**
ALFIKA	Andrew Richards	25 Street Down Town	andrew.richard@gbl.com

Order Service

Order

OrderId	**OrderNo**	**ItemId**	**Quantity**	**Status**
34	44567	66	1	Pending

Items

ItemId	**ItemName**	**Price**
66	Pencil	50
45	Pen	30
75	Marker	25
37	Scale	10

2.1.3 ORDER CONFIRMED

Message Event Number # 3	
Event Name :	ORDER_CONFIRMED
Description :	Customer is proceeded to Payment Service after order is confirmed
Publisher :	Order Service
Subscriber :	Payment Service
Event Details :	When Order is confirmed by customer. Payment service initialized for a particular order.

An event of ORDER_CONFIRMED is published by Order service which is subscribed by Payment service. Order service has already had Order details when order is confirmed Status field is marked as Paid in Transit for further processing of payment.

Order Service

Order

OrderId	OrderNo	ItemId	Quantity	Status
34	44567	66	1	Paid In Transit

Items

ItemId	ItemName	Price
66	Pencil	50
45	Pen	30
75	Marker	25
37	Scale	10

Payment Service

Payment

OrderId	Amount	Date	Status	TransactionId	BankName
34	50	10 April 2023	Pending		

2.1.4 PAYMENT RECEIVED

Message Event Number # 4	
Event Name :	PAYMENT_RECEIVED
Description :	Payment is received by Payment Service
Publisher :	Payment Service
Subscriber :	Order Service
Event Details :	When Payment is completed.

An event of PAYMENT_RECEVIED is published by Payment service which is subscribed by Order service. Pending status of Payment Service has been updated to completed with *TransactionId* and *BankName*. The Order service updates is status to Paid and Payment service updates its status from Pending to Complete

Order Service

Order

OrderId	**OrderNo**	**ItemId**	**Quantity**	**Status**
34	44567	66	1	Paid

Items

ItemId	**ItemName**	**Price**
66	Pencil	50
45	Pen	30
75	Marker	25
37	Scale	10

Payment Service

Payment

OrderId	**Amount**	**Date**	**Status**	**TransactionId**	**BankName**
34	50	10 April 2023	Complete	66534287765	HBL

2.1.5 ORDER COMPLETED

Message Event Number # 5	
Event Name :	ORDER_COMPLETED
Description :	When Order completed it updates Inventory
Publisher :	Order Service
Subscriber :	Inventory Service
Event Details :	Order Service status updated to paid and Inventory service updates its database.

An event of ORDER_COMPLETED is published by Order service which is subscribed by Inventory service. In this event Order service send *ItemId* to Inventory service which is the confirmation that order has been completed and inventory need to be updated.

Order Service

Order

OrderId	OrderNo	ItemId	Quantity	Status
34	44567	66	1	Paid

Items

ItemId	ItemName	Price
66	Pencil	50
45	Pen	30
75	Marker	25
37	Scale	10

Inventory Service

Inventory

ItemId	ItemName	Quantity	Price	Status
66	Pencil	0	50	
45	Pen	4	30	
75	Marker	7	25	
37	Scale	2	10	

2.1.6 INVENTORY UPDATED

Message Event Number # 6	
Event Name :	INVENTORY_UPDATED
Description :	Inventory updates in Order service
Publisher :	Inventory Service
Subscriber :	Order Service
Event Details :	Inventory service generates this event to acknowledge all subscriber services.

An event of INVENTORY_UPDADTED is published by inventory service which is subscribed by Order service. In this event Inventory service generate an acknowledgement that inventory has been updated. So that all subscriber services should update accordingly.

Order Service

Order

OrderId	OrderNo	ItemId	Quantity	Status
34	44567	66	1	Paid

Items

ItemId	ItemName	Price
66	Pencil	50
45	Pen	30
75	Marker	25
37	Scale	10

Inventory Service

Inventory

ItemId	ItemName	Quantity	Price	Status
66	Pencil	0	50	
45	Pen	4	30	
75	Marker	7	25	
37	Scale	2	10	

3 Evaluation of Adaptation Model

3.1 Data Validation Across Microservices

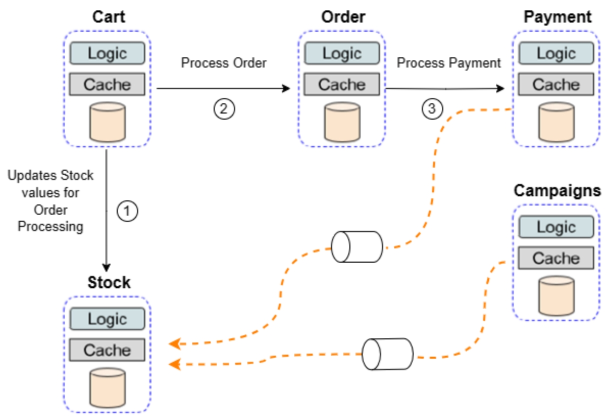


Figure 2: Data Validation Across Microservices

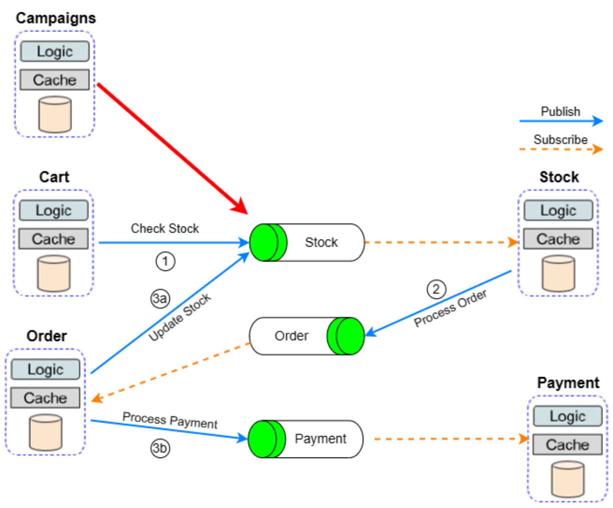
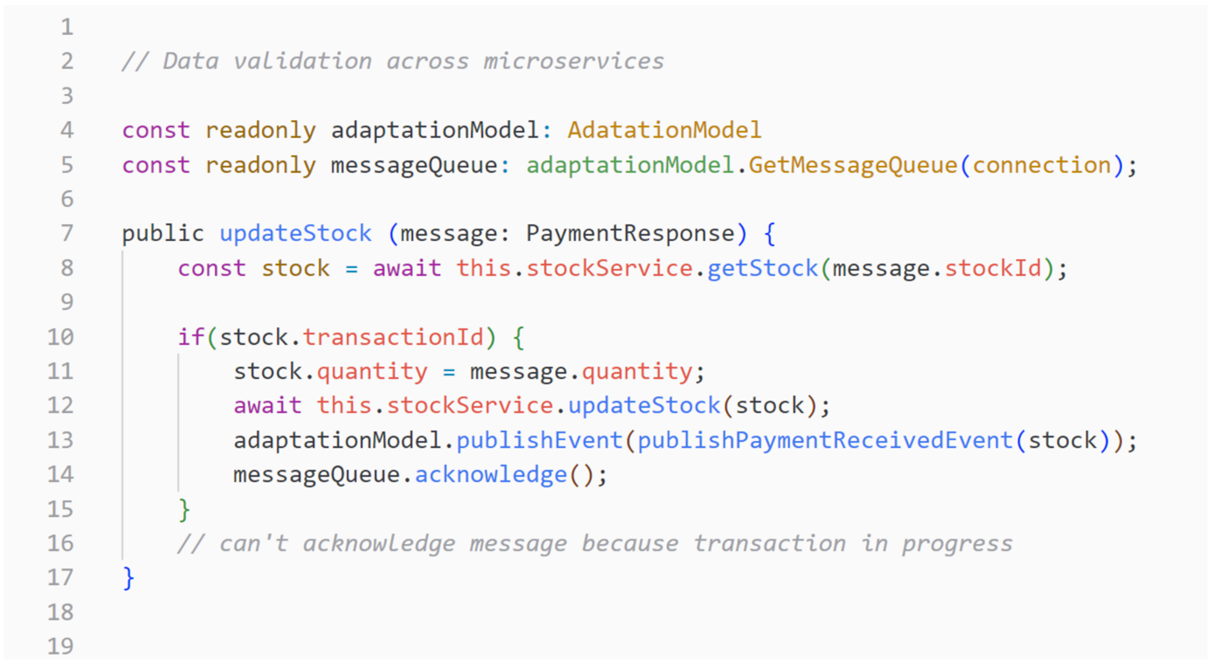


Figure 3: Data Validation with Adaptation Model

As shown in Figure 2, the issue of data validation in microservices typically arises within the context of orchestration, wherein each microservice is capable of accessing Paramore procedure calls or asynchronous API calls to retrieve data from other microservices. The aforementioned scenario can be effectively managed through the implementation of saga choreography. This approach involves the utilization of a message broker to facilitate communication between microservices and their respective databases. By leveraging this mechanism, it becomes possible to analyze the current state of data and preemptively prevent any potential issues from arising.

The utilization of a transaction manager within an adaptation model can effectively manage scenarios wherein specific data is concurrently accessed by multiple transactions. In Figure 3, the system monitors transactions through the use of a distinct transaction identifier. Each message queue event includes the transactionId associated with the corresponding transaction. The presence of a transactionId within a given data or record prompts the interpretation model to either delay or refrain from interfering with the ongoing transaction. Upon the completion of the aforementioned transaction, the transactionId was subsequently released, thereby rendering the pertinent information available for utilization in other transactions and within the microservice environment.



```

// Data validation across microservices

const readonly adaptationModel: AdatationModel
const readonly messageQueue: adaptationModel.GetMessageQueue(connection);

public updateStock (message: PaymentResponse) {
    const stock = await this.stockService.getStock(message.stockId);

    if(stock.transactionId) {
        stock.quantity = message.quantity;
        await this.stockService.updateStock(stock);
        adaptationModel.publishEvent(publishPaymentReceivedEvent(stock));
        messageQueue.acknowledge();
    }
    // can't acknowledge message because transaction in progress
}


```

Figure 4: Code Snippet for Data Validation With Adaptation Model

In the above mention code, in Figure 4, a Stock service return a specific stock retrieved by its stockId available in message context. It can only update its quantity if and if only message contain a valid transactionId. If a stock contains transactionId it can not be modified by another service or transaction.

3.2 Feral Ordering

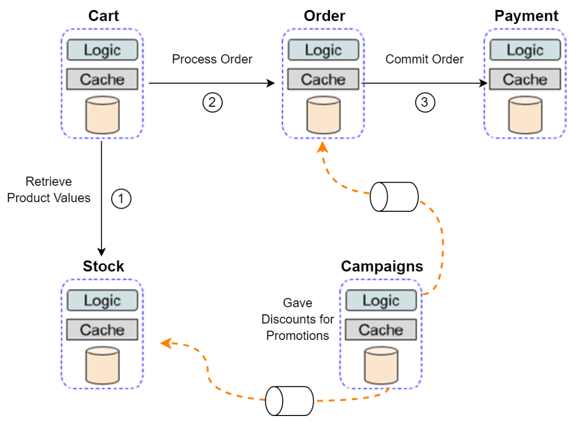


Figure 5: Feral Ordering

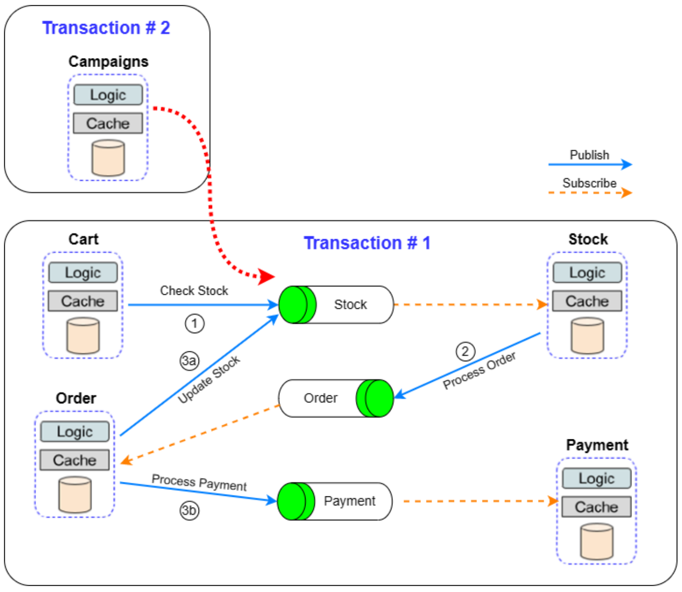
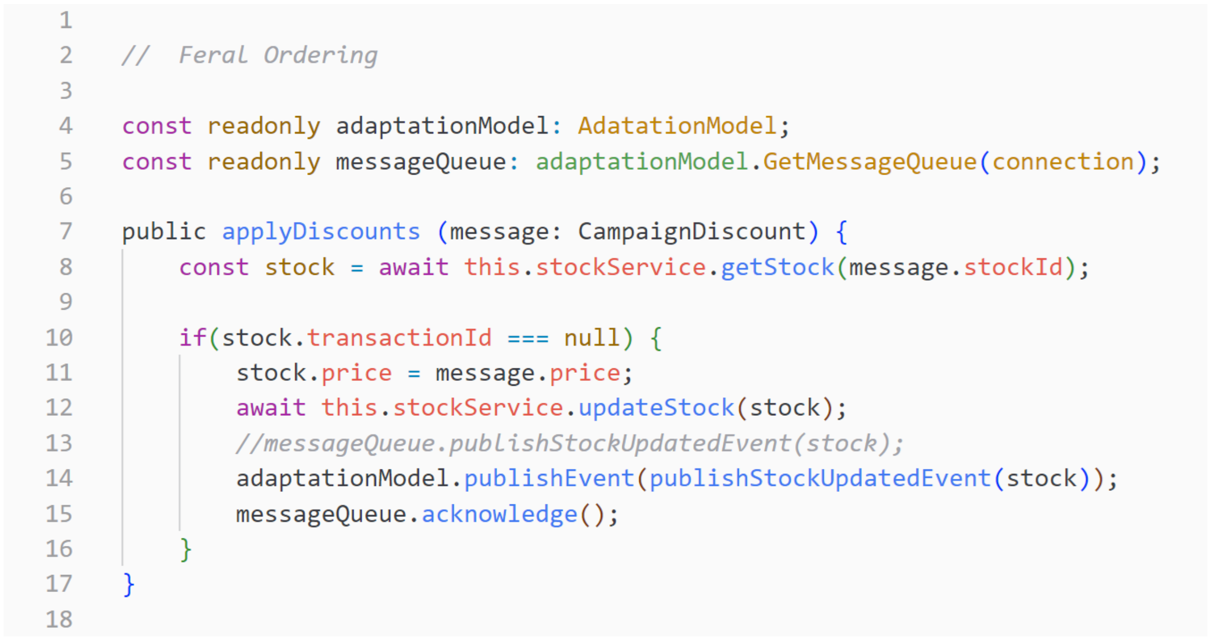


Figure 6: Feral Ordering with Adaptation Model

Federal ordering refers to a scenario where an order is being processed, but another service is responsible for updating the database, which may not be accurately reflected in the service, as shown in Figure 5. The occurrence of dissimilar data between two transactions that access comparable information is known as data inconsistency. During the processing of a transaction, any updates made to the stocks cannot be reflected in the transaction until it has been completed. Microservices orchestration typically involves federal ordering, wherein each service interacts with other servers through an asynchronous API call.

The issue at hand can be effectively addressed in, Figure 6, microservice choreography through the utilization of message broker acknowledgment. The microservices choreography approach involves the publication of an event by each service, which is subsequently subscribed to by multiple servers. This process facilitates the fulfillment of the business model and transaction. This situation can be effectively addressed through choreography. Specifically, if a particular stock has not been subscribed to by any transaction, it can be updated through the campaign service. Upon the alteration of the stock service message broker, it becomes capable of transmitting an acknowledgment signal to all subscribed services of the stock service. If the stock undergoes an update during this procedure and transaction #1 is triggered subsequent to this alteration, an automatically updated version of the stock can be utilized. In the second scenario, when transaction #1 triggers an event that has already been subscribed to by the stock service, it will utilize the previous value of the updated stock. Subsequently, solely new transactions will be able to access the updated stock information.



```

//  Feral Ordering

const readonly adaptationModel: AdatationModel;
const readonly messageQueue: adaptationModel.GetMessageQueue(connection);

public applyDiscounts (message: CampaignDiscount) {
    const stock = await this.stockService.getStock(message.stockId);

    if(stock.transactionId === null) {
        stock.price = message.price;
        await this.stockService.updateStock(stock);
        //messageQueue.publishStockUpdatedEvent(stock);
        adaptationModel.publishEvent(publishStockUpdatedEvent(stock));
        messageQueue.acknowledge();
    }
}

```

Figure 7: Code Snippet for Feral Ordering With Adaptation Model

The aforementioned code , in Figure 7, includes a method known as ”apply discount” which receives a message from the campaign discount. This method verifies that the stock in question has not been accessed by any transaction, in order to update its value. If the stock has been utilized by another transaction, the campaign service is unable to apply a discount to that particular stock. In a comparable vein, in the event that a stock undergoes an update via a campaign service, it will subsequently update the stock and produce an acknowledgment event to all subscribed services. This allows said services to update the value of the aforementioned stock accordingly.

3.3 Foreign Key Constraint

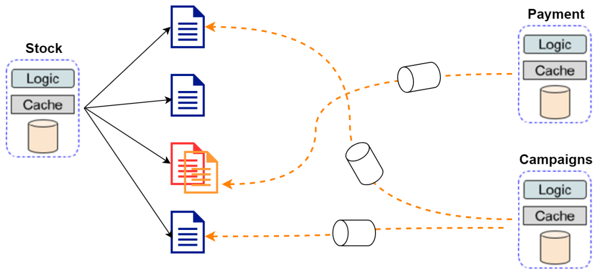


Figure 8: Implicit Associations across Microservices

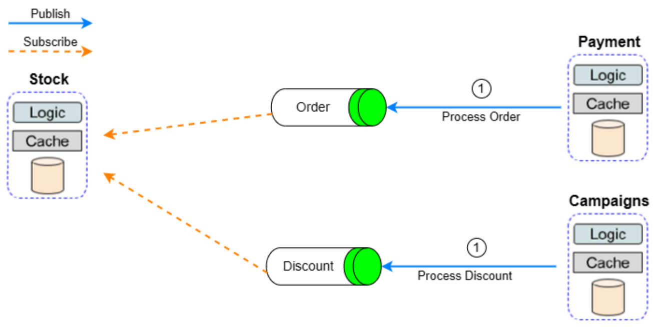


Figure 9: Foreign Key Constraint with Adaptation Model

The issue of foreign key constraints is presented in the context of implicit cross microservice association. When two microservices that can establish a one-to-many relationship? Managing this anomaly poses a challenge due to the autonomous nature of microservices in a microservice environment, where their database relationships are independent of any other microservice's data repository. In Figure 8, the presence of such a relationship between microservices poses challenges in preserving the integrity of cascade updates or deletions across both the master and detail microservices.

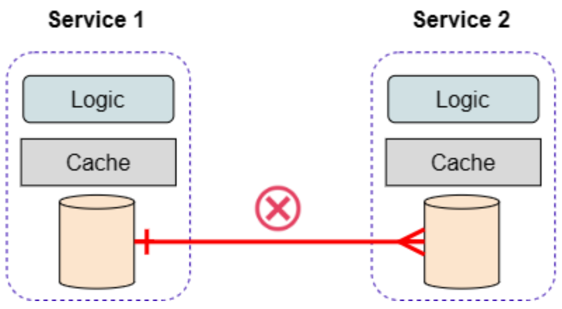


Figure 10: Across Microservice Relationship

The choreography of microservices, in Figure 9, is capable of autonomously managing inter-microservice decisions, owing to the fact that the functional decomposition of microservices is inherently atomic in terms of its functionality. The aforementioned constraint limits the ability of a given microservice to access data that is functionally dependent on a distinct service within the same business model as depicted in Figure 10.