State in your reactive system

Reactive Systems

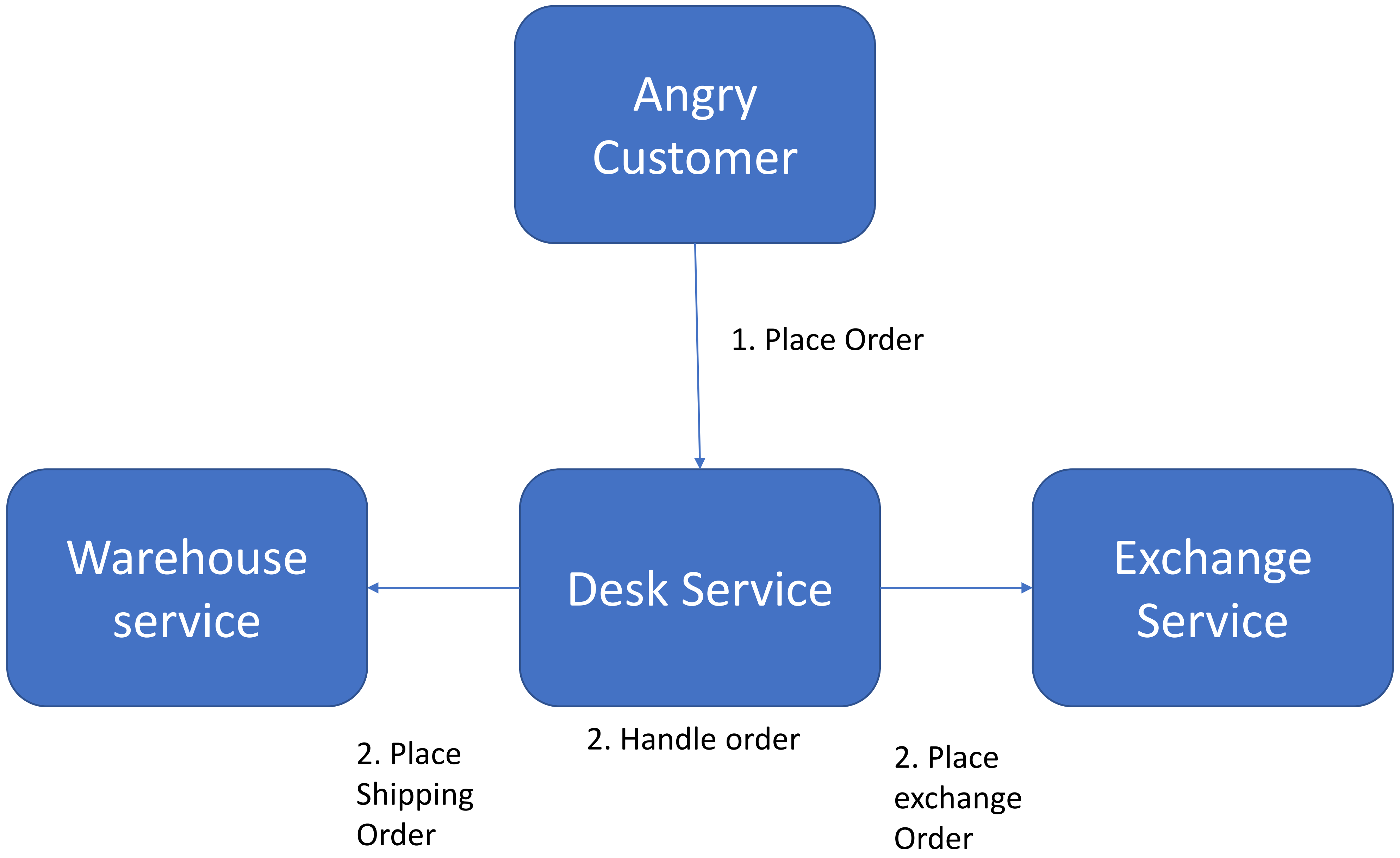
- Responsive
- Elastic
- Resilient
- Message Driven

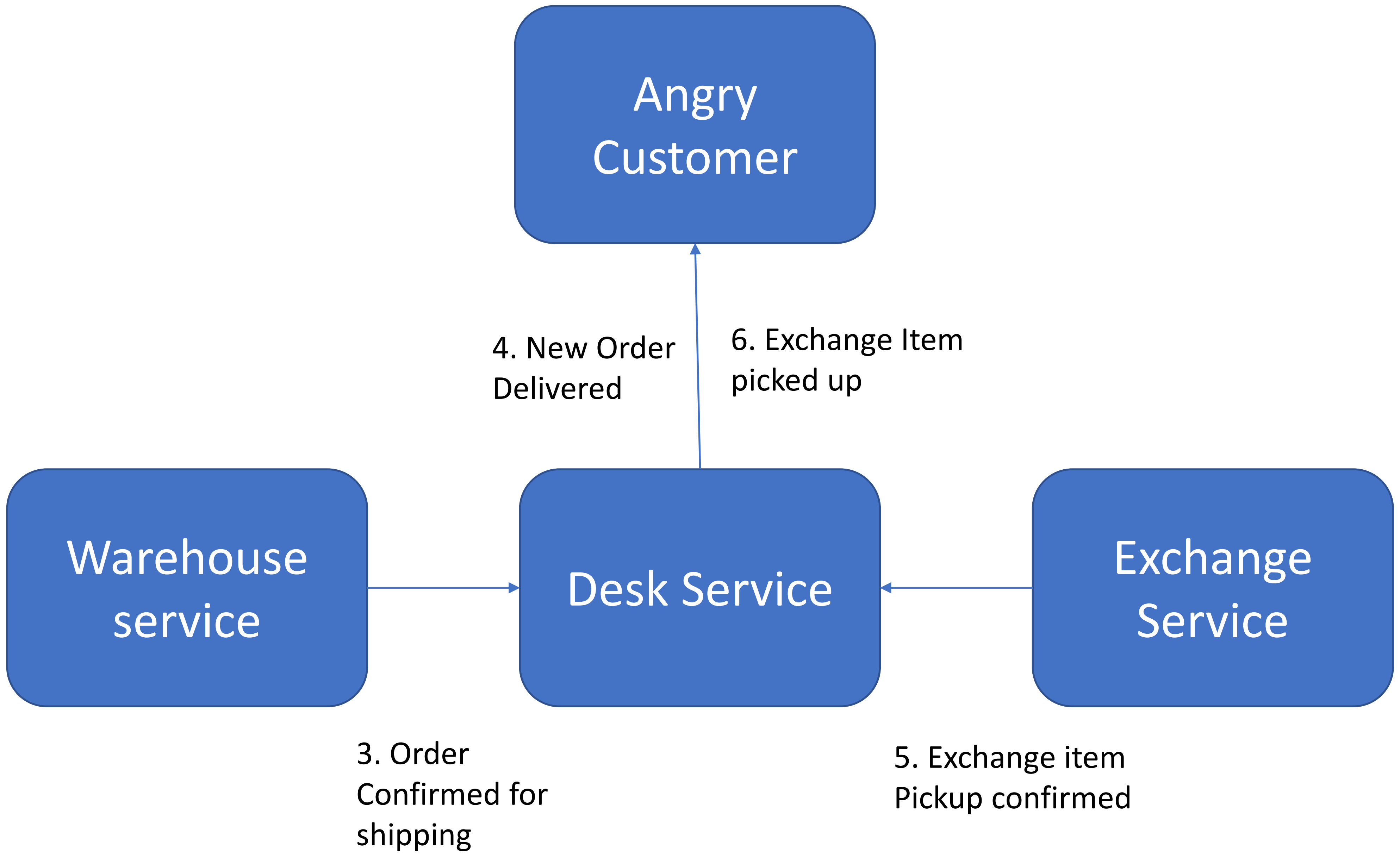
Micro Services

- Structure of Software and Teams

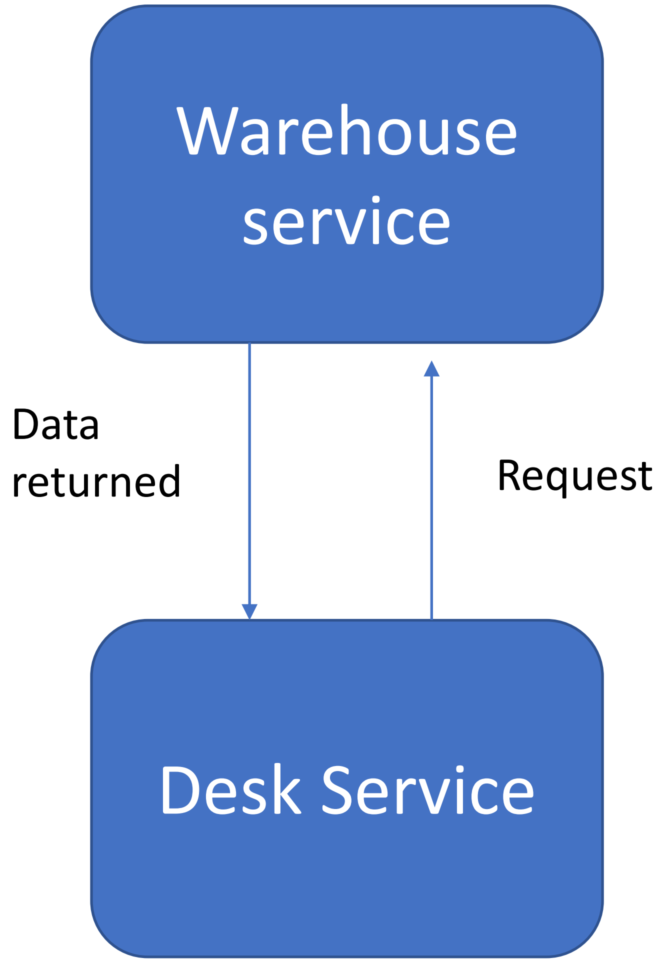
Various Definition

- Independently deployable
- Loose coupling
- Fault tolerant
- flexible


Angry Customer
1. Place Order
Warehouse service
Desk Service
Exchange Service
2. Place Shipping Order
2. Handle order
2. Place exchange Order

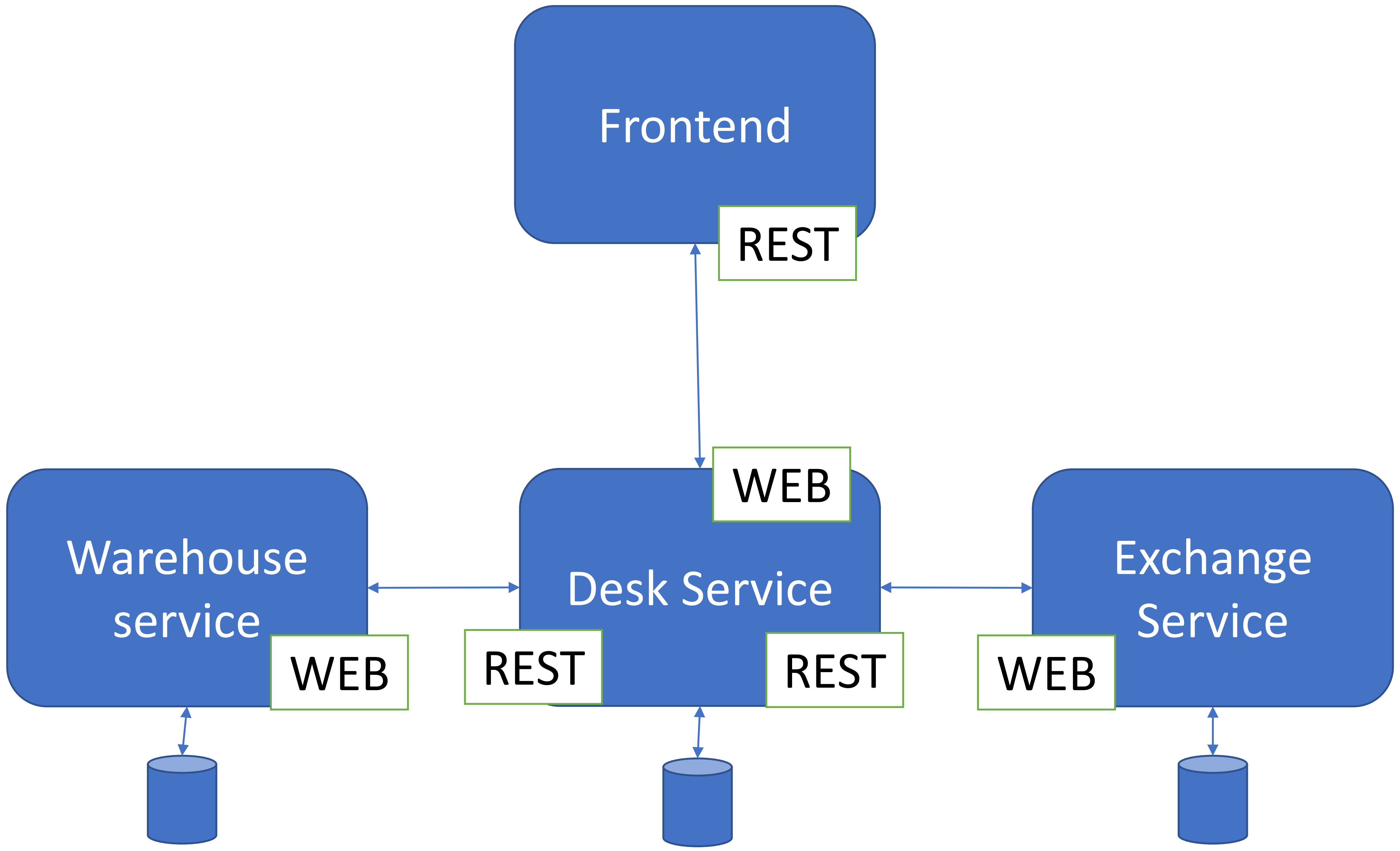

Angry Customer
4. New Order Delivered
6. Exchange Item picked up
Warehouse service
Desk Service
Exchange Service
3. Order Confirmed for shipping
5. Exchange item Pickup confirmed

Synchronous communication



```
//warehouse controller with endpoint
@PostMapping("/warehouse")
fun prepare(@RequestBody orders:Orders):Orders {
  return warehouseService.prepareOrders(orders)
}

//desk service calling warehouse endpoint
fun handleOrders(orders:Orders): Orders {
val savedOrders = deskRepository.save(orders)
Var warehouseResponse:Any? = null
Try{
warehouseResponse = restTemplate.postForObject(url, orders, Orders::class)
} catch(exception:Exception) {
 //do something
 }
// REST OF THE CODE
}
```


Frontend
REST
WEB
Warehouse service
WEB
Desk Service
REST
REST
Exchange Service
WEB

Blocking

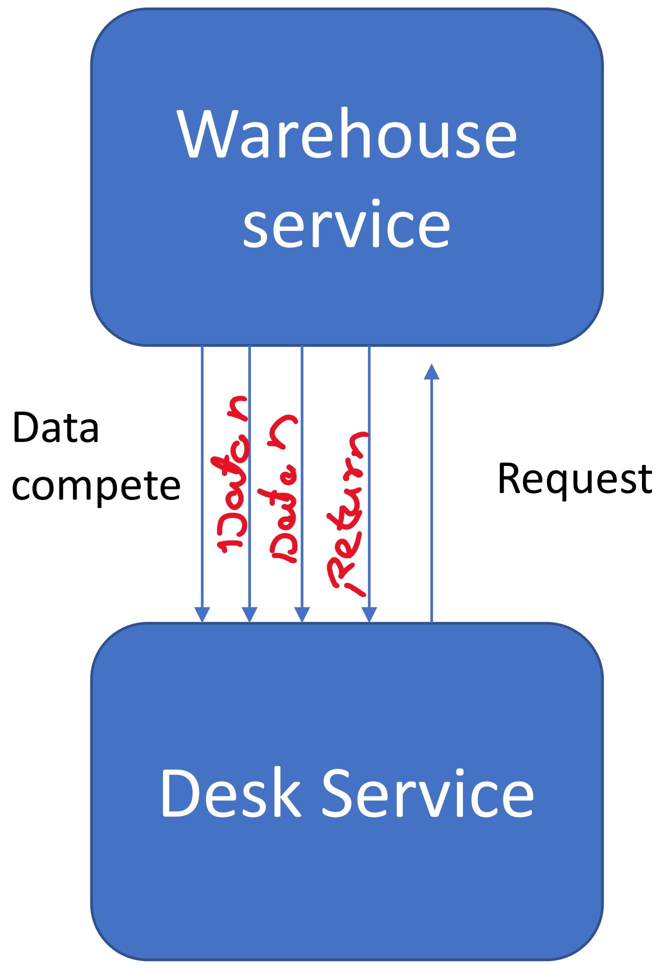
Responsive

Elastic

Resilient

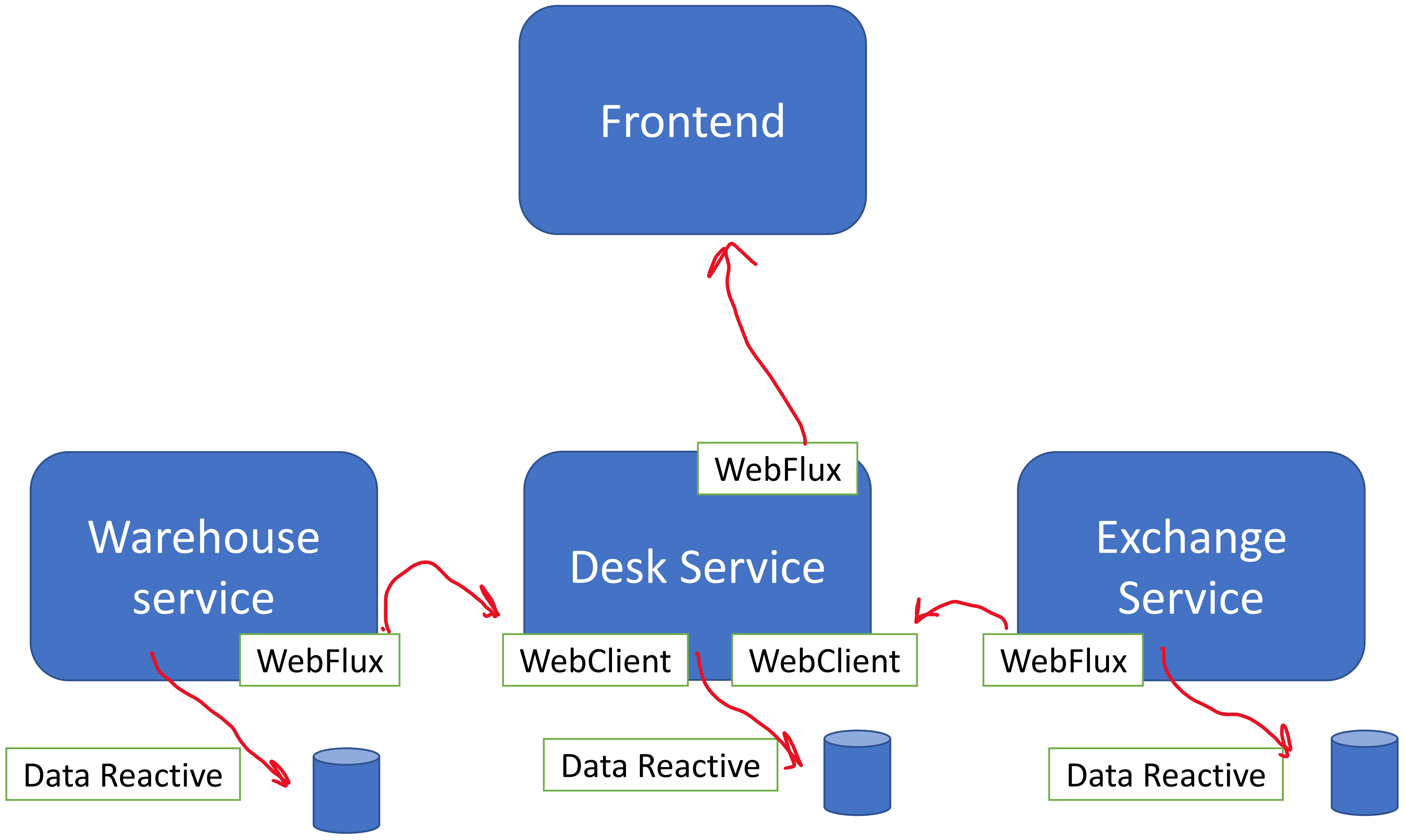
Message Driven

Asynchronous communication



```
//warehouse controller with endpoint
@PostMapping("/warehouse")
fun prepare(@RequestBody orders:Orders):Mono<Orders> {
  return warehouseService.prepareOrders(orders)
}

//desk service calling warehouse endpoint
fun handleOrder(order:Order){
Return webClient.method(POST)
.uri(warehouseServiceUrl)
.body(BodyInserter.fromValue(order.name)).retrive()
.bodyToMono(Order::class) }
```


Frontend
WebFlux
Warehouse service
Desk Service
Exchange Service
WebFlux
WebClient
WebClient
WebFlux
Data Reactive
Data Reactive
Data Reactive

Interface spaghetti

Responsive

Elastic

Resilient

Message Driven

Saga Pattern

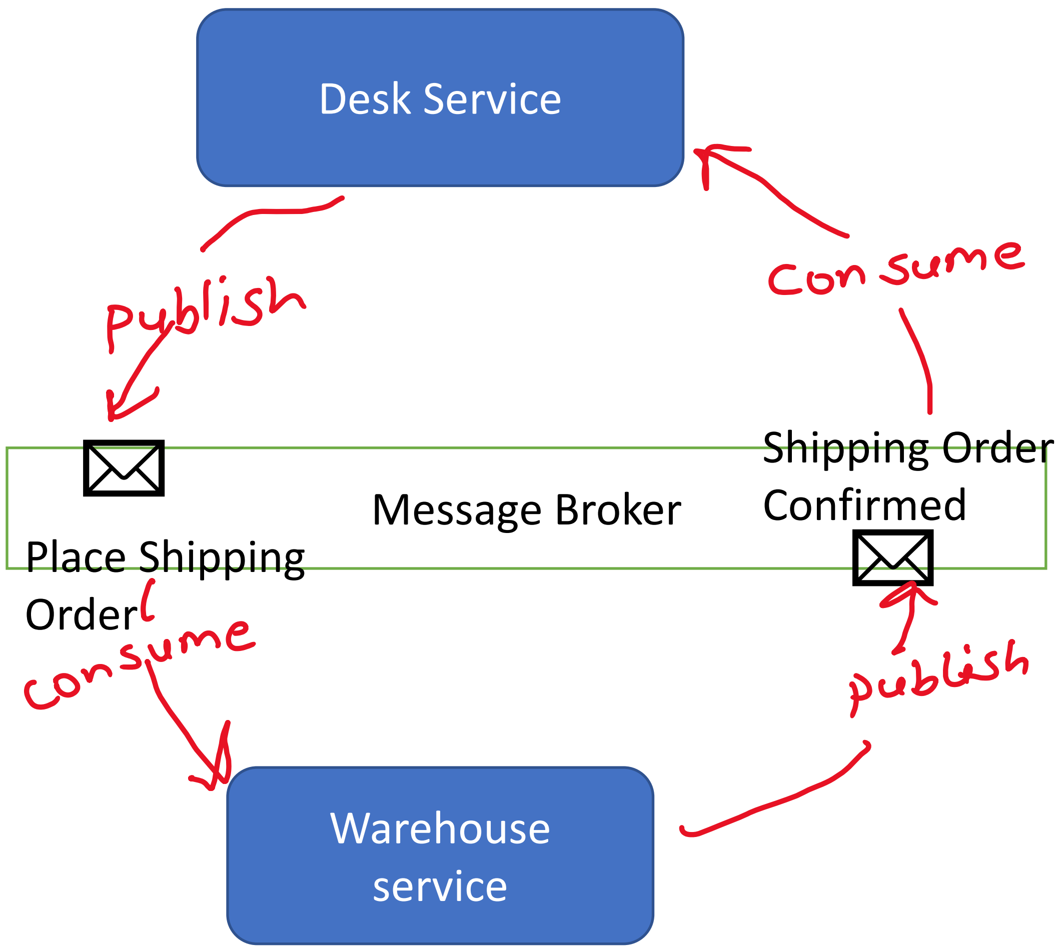
It prevent us from doing distributed transaction

It uses local transaction and also make sure that saga is completed

If something goes wrong, it make sure to roll back.

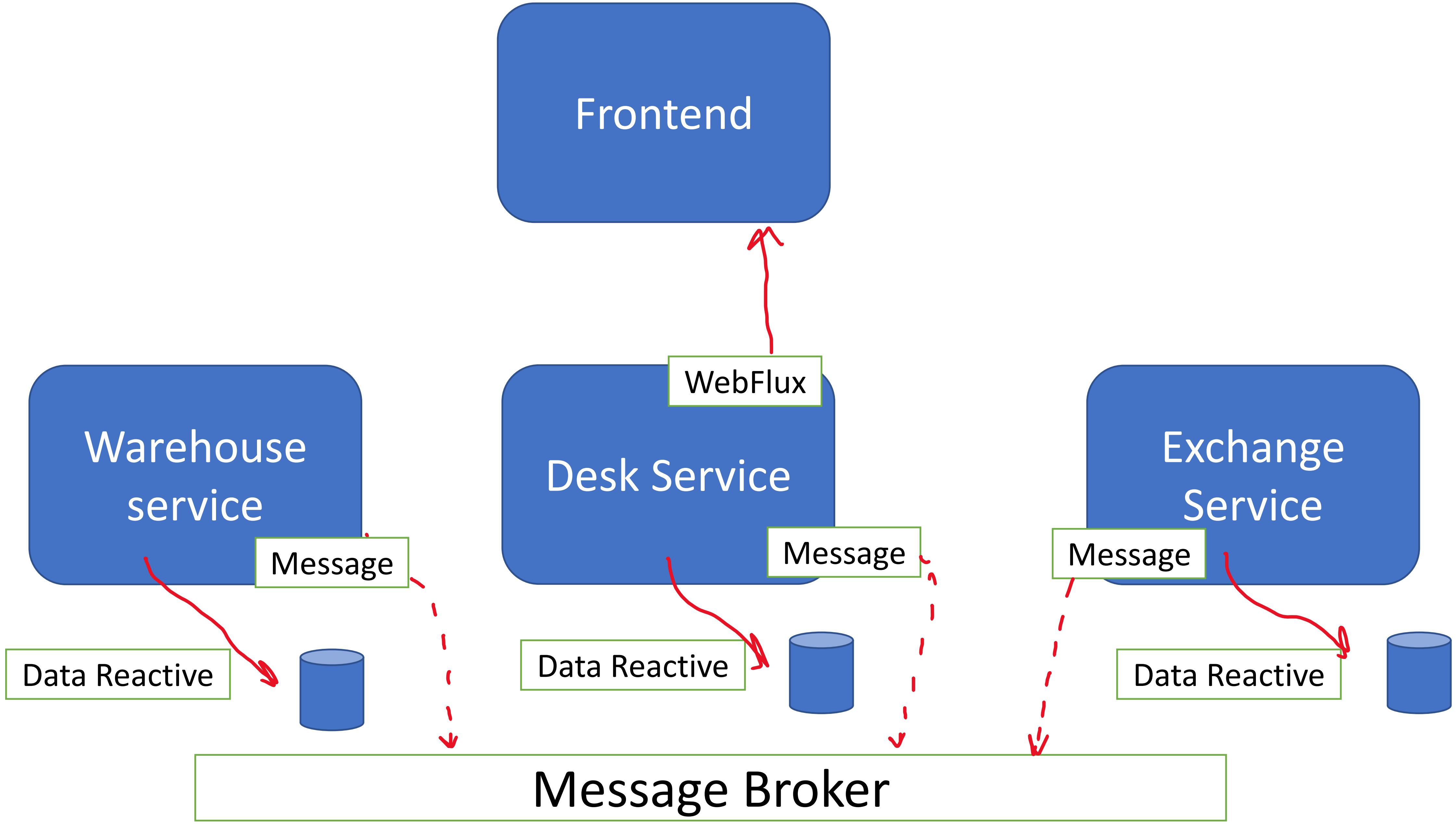
- Implementing saga pattern with reactive coding is just not possible
- We need to think about messaging in a distributed system
- Common way to do it by introducing a message broker
- Loose coupling with messages

Message Driven



```
fun sendShippingOrder(orderId:String){
kafkaTemplate.send("shippingOrdered", orderId)
}

@kafkaListener(topics="shippingOrdered")
fun prepareShippings(orderId:String) {
Println("Your shippings is ready")
shippingConfirmProducer.sendConfirmation(orderId)
}
```


Frontend
WebFlux
Warehouse service
Desk Service
Exchange Service
Message
Message
Message
Data Reactive
Data Reactive
Data Reactive
Message Broker

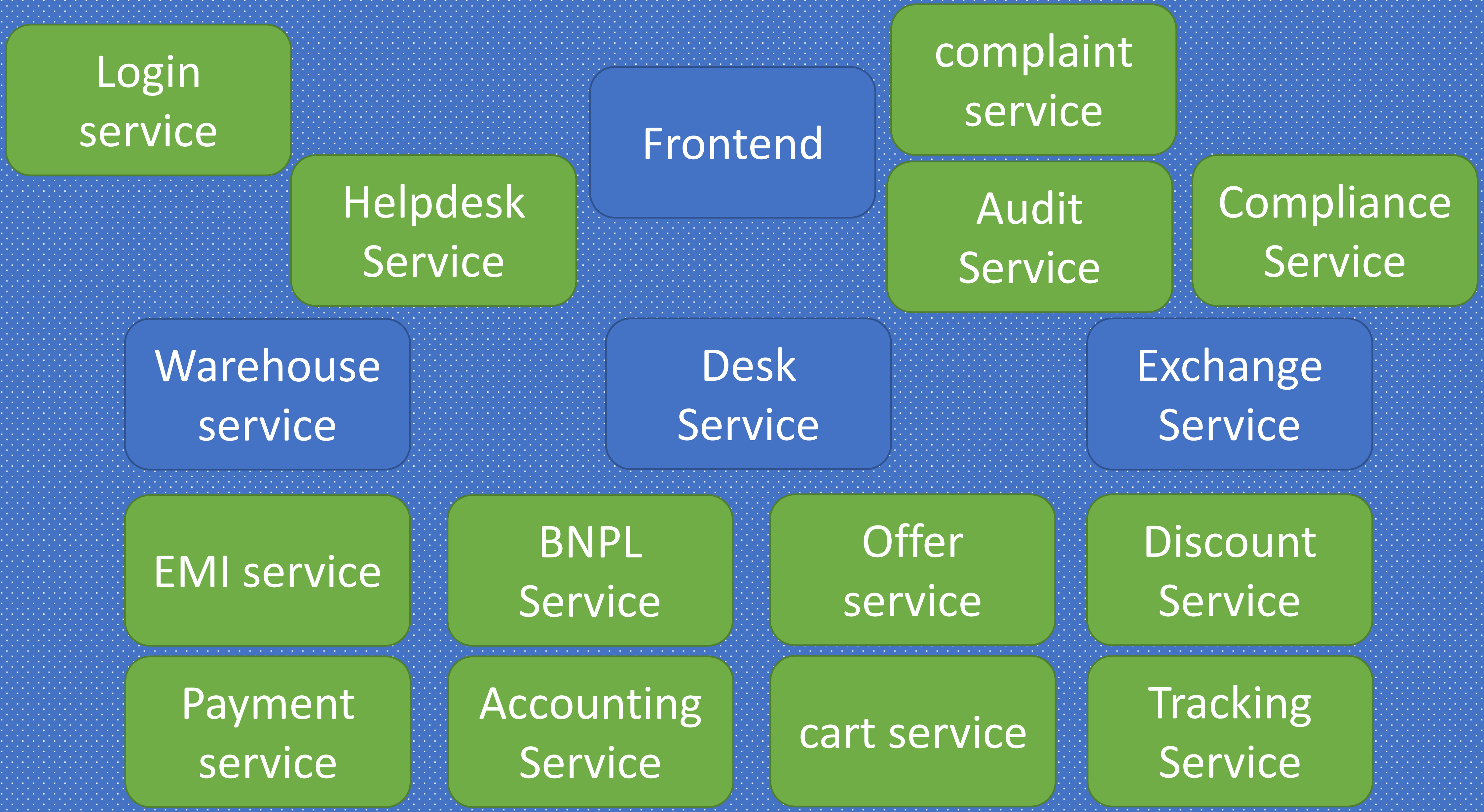
Awesome we achieved all four!!

Responsive

Elastic

Resilient

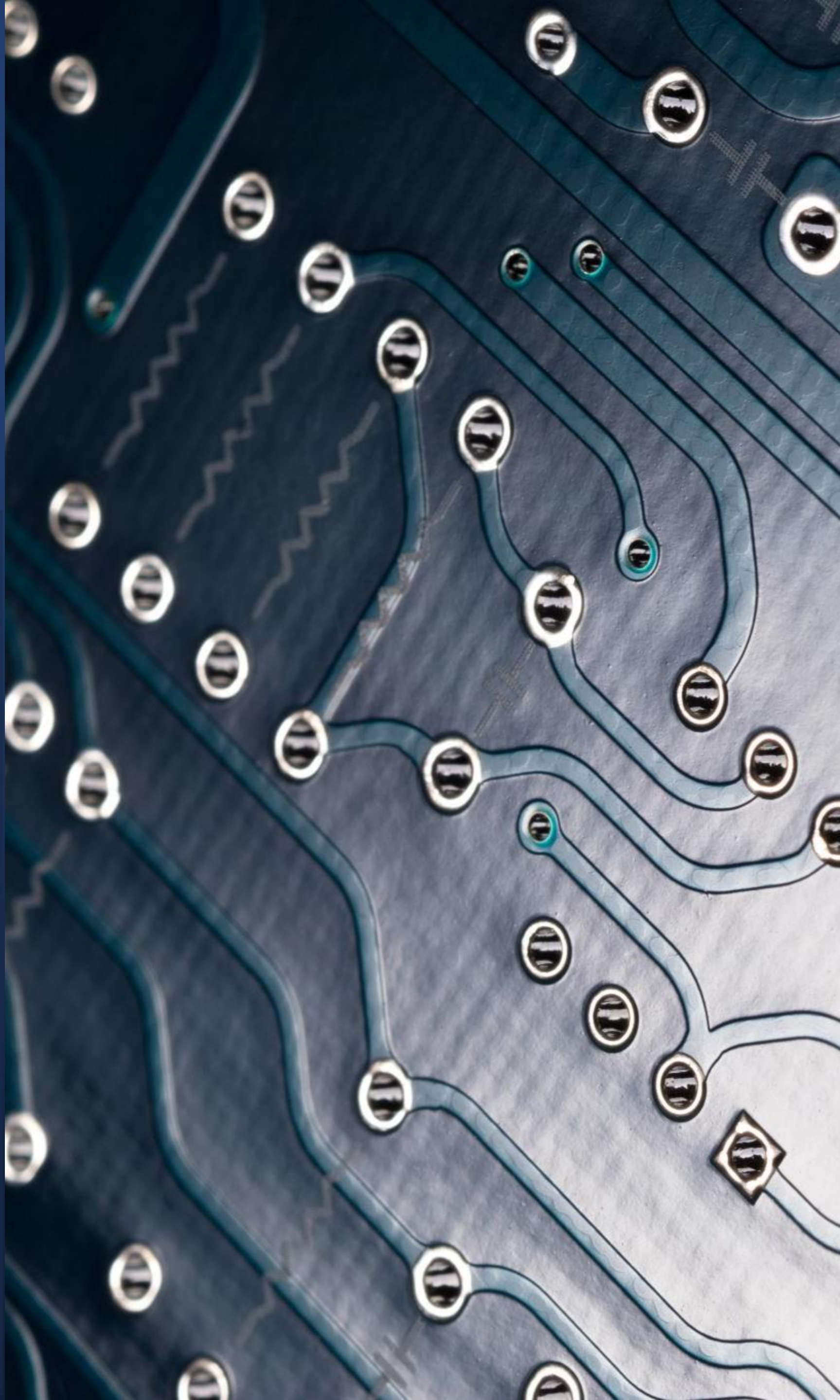
Message Driven


Login service
Helpdesk Service
Frontend
complaint service
Audit Service
Compliance Service
Warehouse service
Desk Service
Exchange Service
EMI service
BNPL Service
Offer service
Discount Service
Payment service
Accounting Service
cart service
Tracking Service

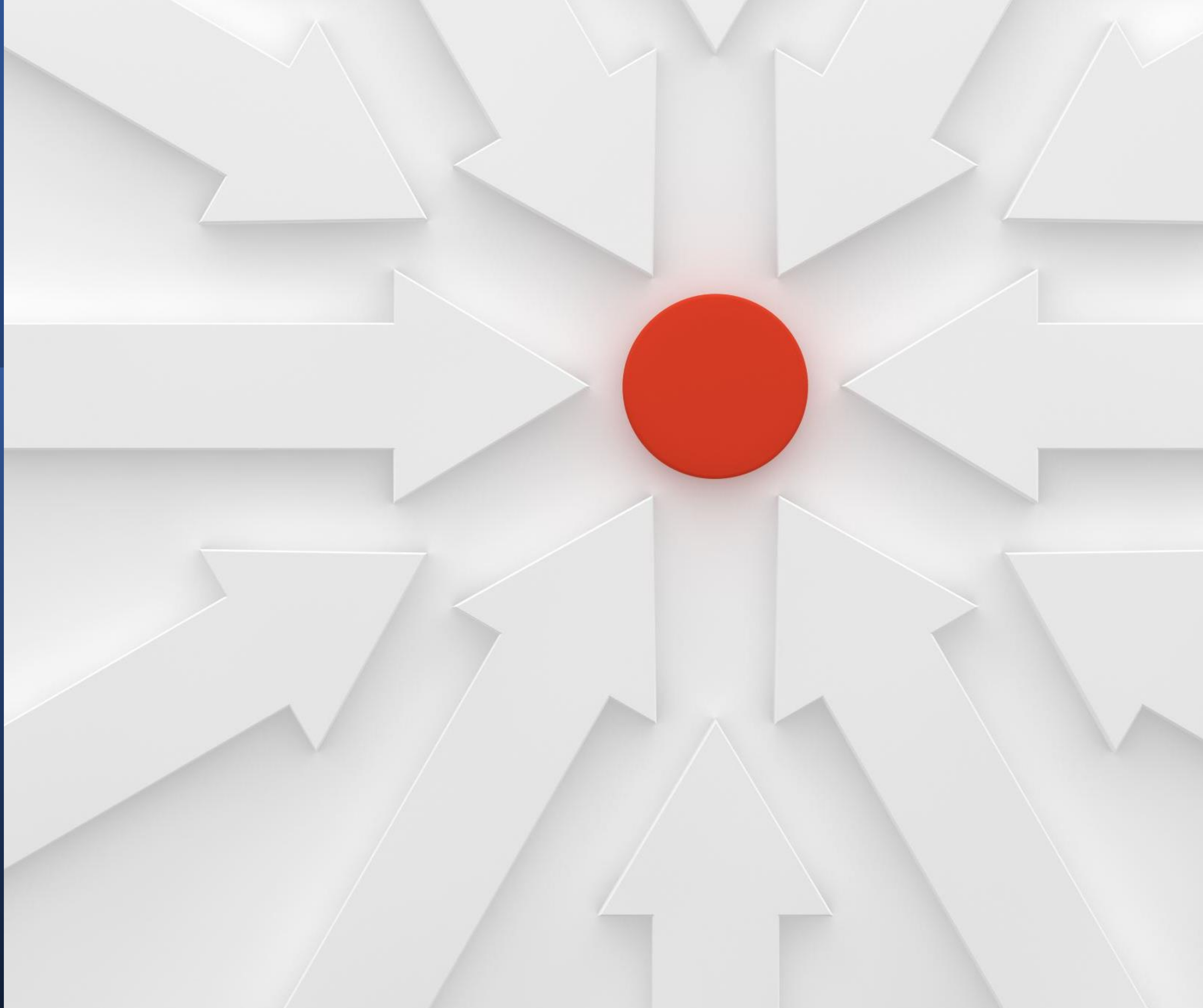
Cleaning up the party?

- Detect the problem quickly
- Observability
- Understanding what is going on

State in a reactive system



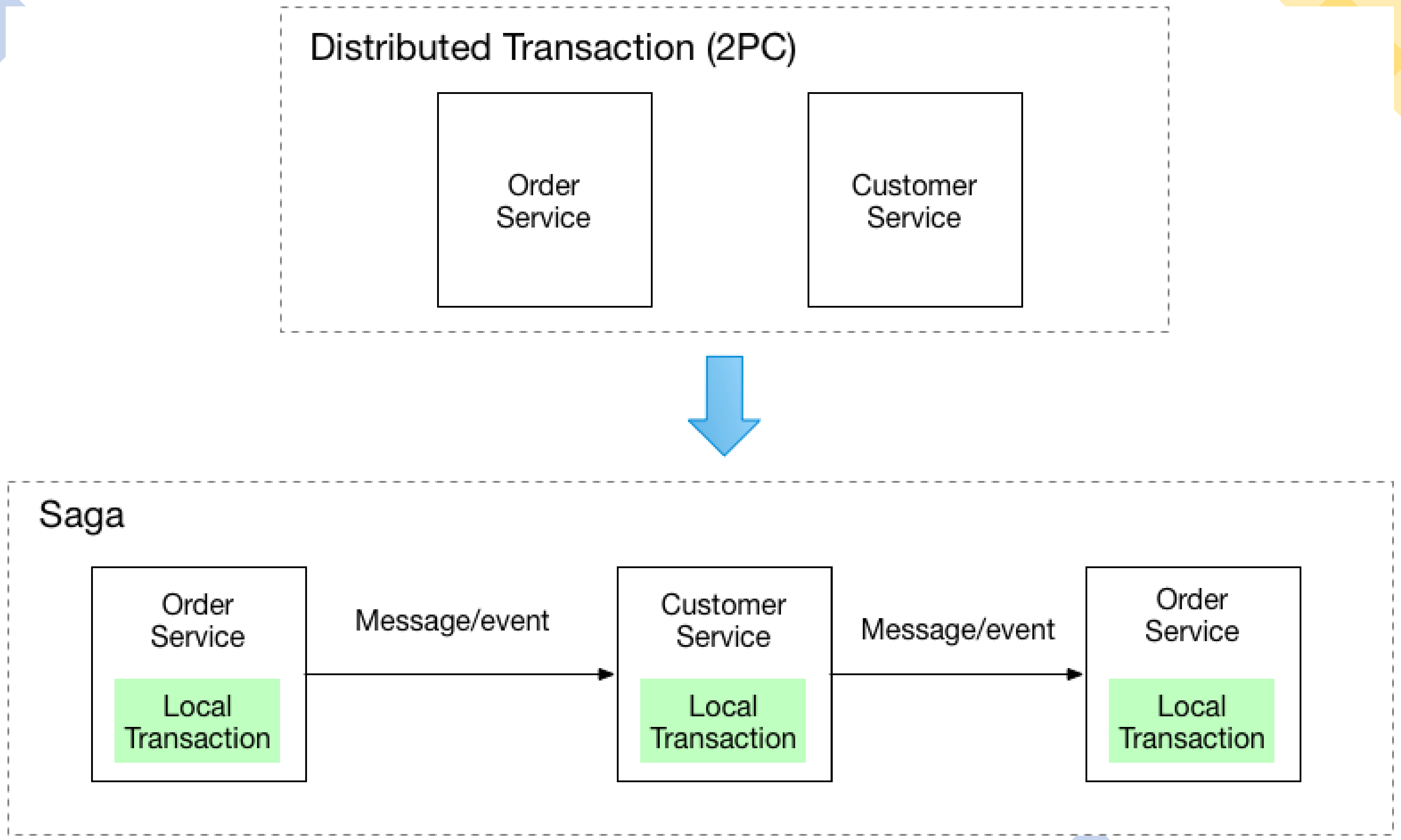
Solution: Define saga(process) in a orchestrated way

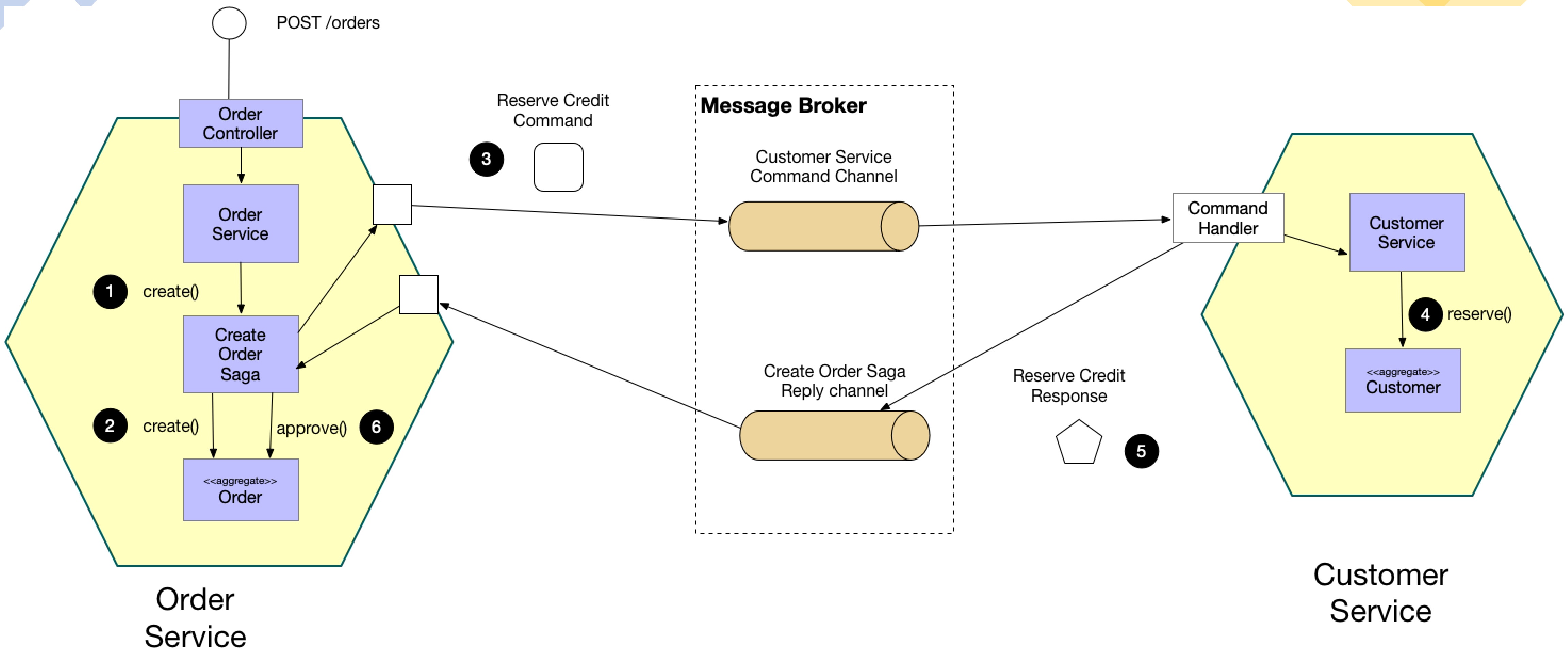


What is Saga Pattern?

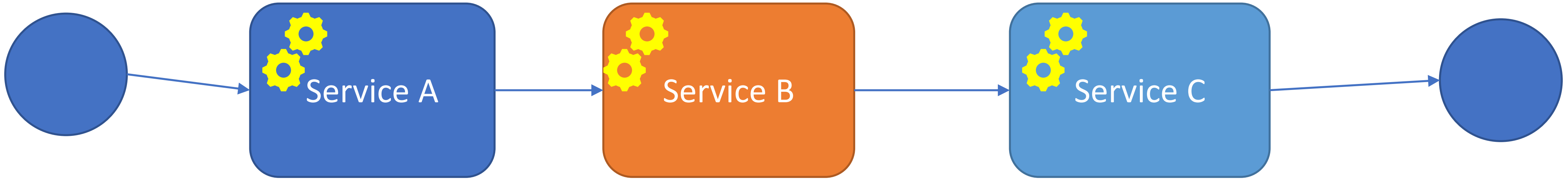
Implement each business transaction that spans multiple services is a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga.

If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

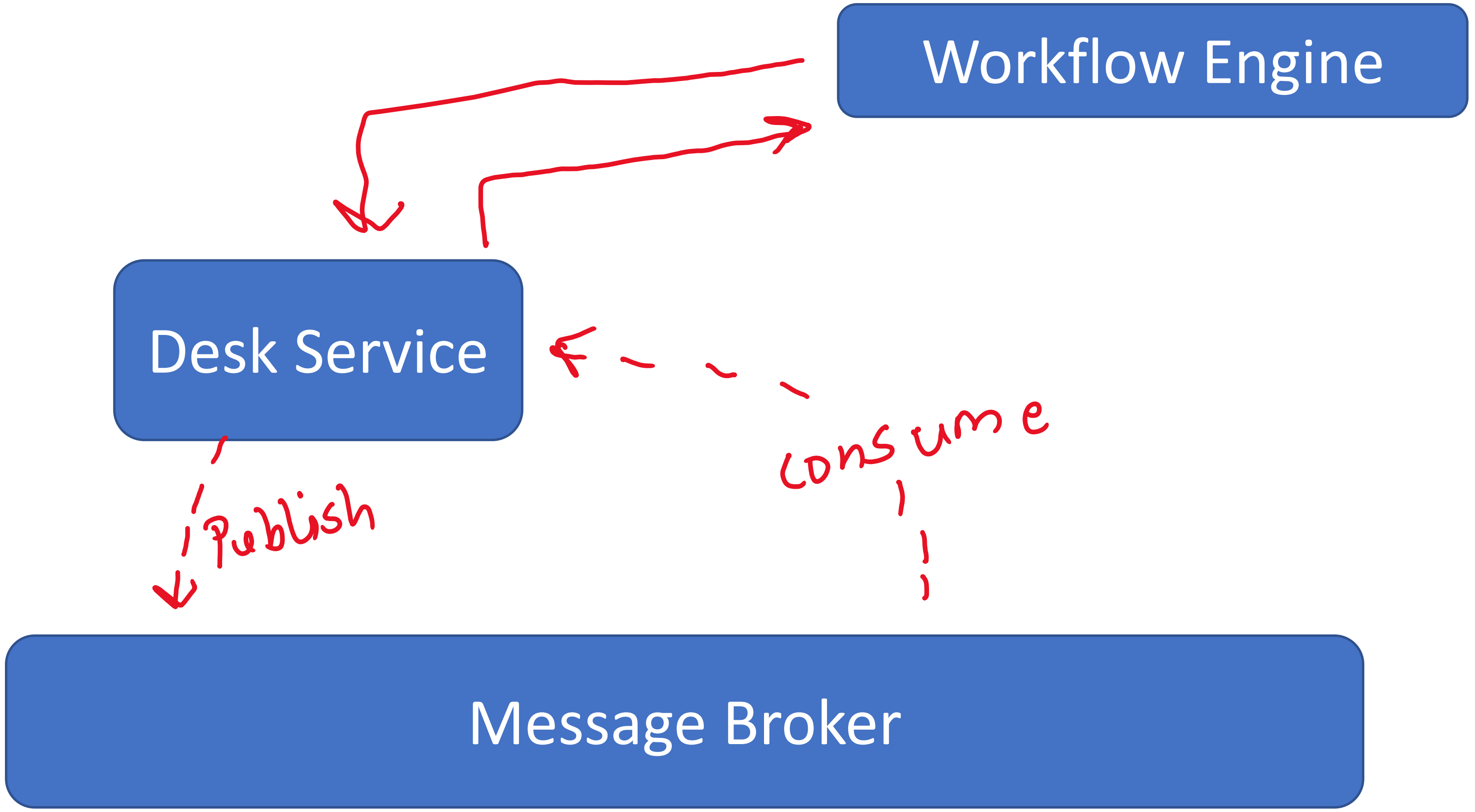

Distributed Transaction (2PC)
Order Service
Customer Service
Saga
Order Service
Local Transaction
Message/event
Customer Service
Local Transaction
Message/event
Order Service
Local Transaction

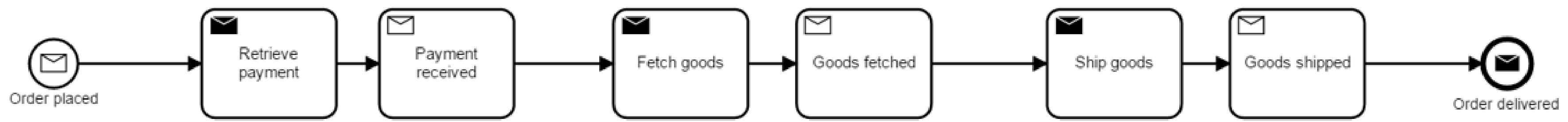

POST /orders
Order Controller
Order Service
1 create()
Create Order Saga
2 create()
approve() 6
<<aggregate>> Order
Order Service
3 Reserve Credit Command
Message Broker
Customer Service Command Channel
Create Order Saga Reply channel
Reserve Credit Response
5
Command Handler
Customer Service
4 reserve()
<<aggregate>> Customer
Customer Service

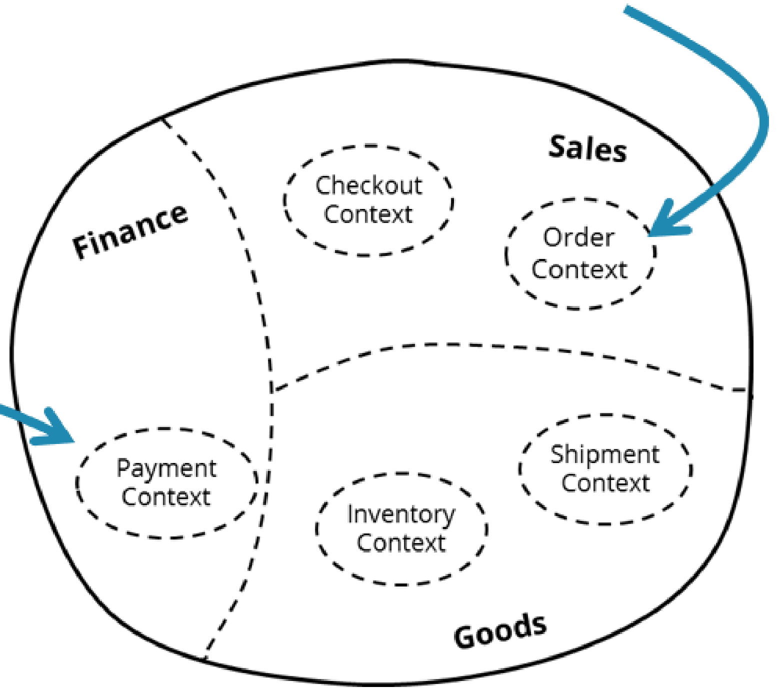
Workflow Engines

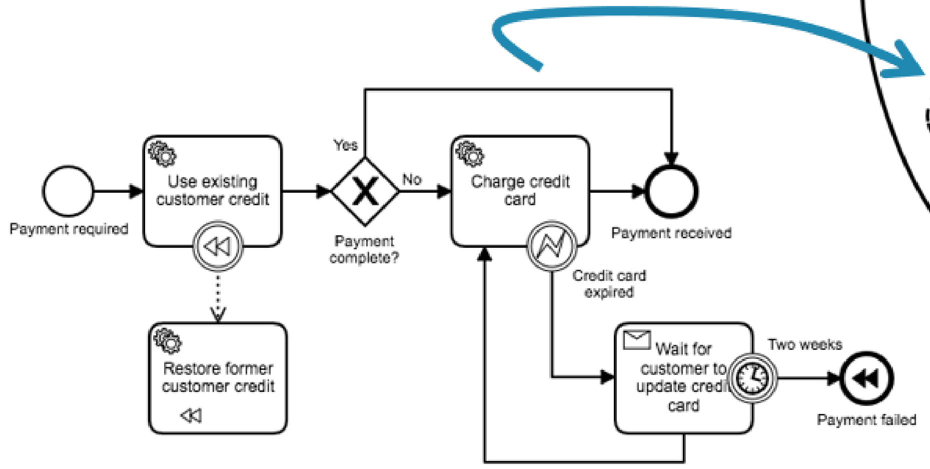


State


Workflow Engine
Desk Service
publish
consume
Message Broker


Order placed
Retrieve payment
Payment received
Fetch goods
Goods fetched
Ship goods
Goods shipped
Order delivered


Sales
Finance
Checkout Context
Order Context
Payment Context
Inventory Context
Shipment Context
Goods

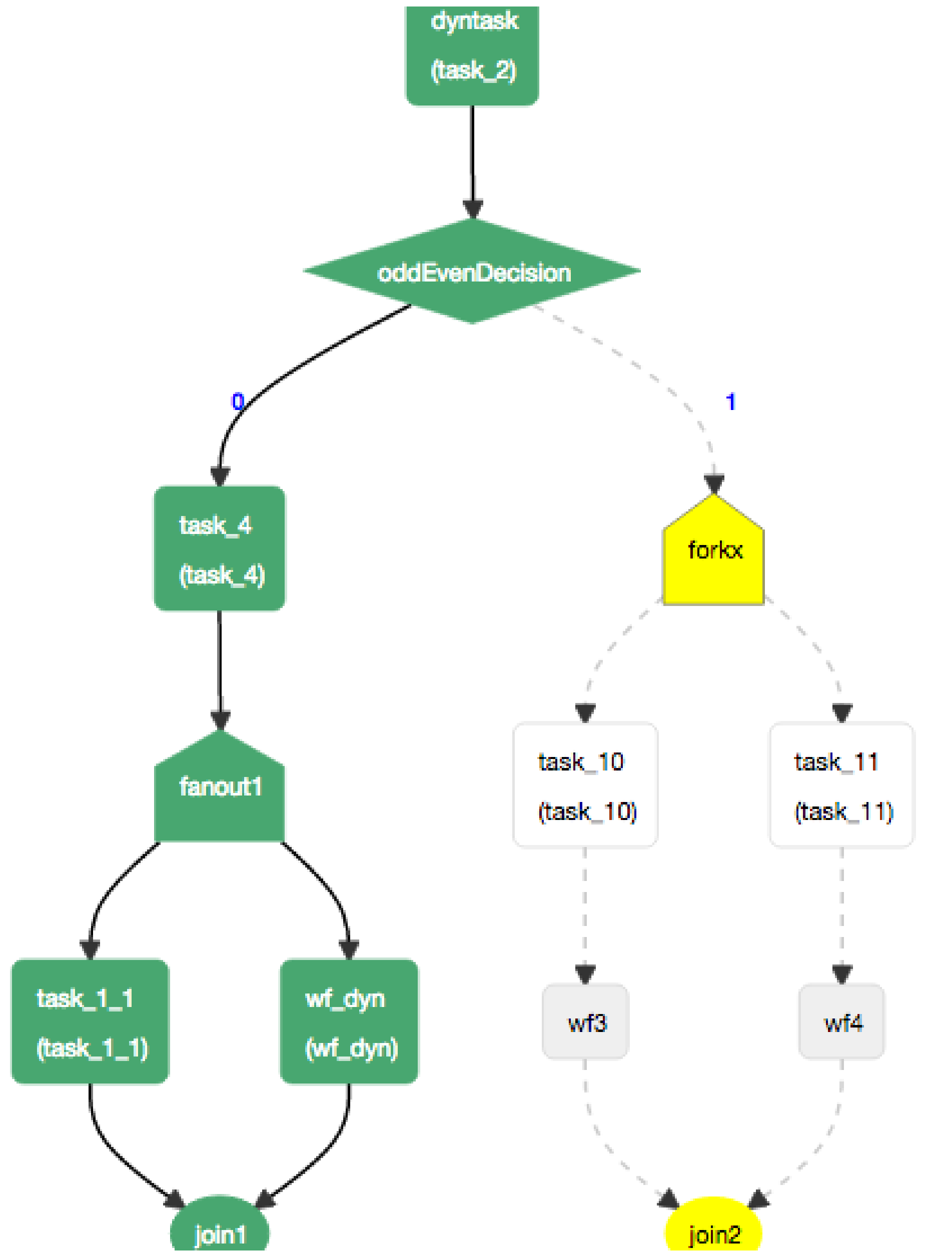

Payment required
Use existing customer credit
Yes
No
Payment complete?
Charge credit card
Payment received
Credit card expired
Restore former customer credit
Wait for customer to update credi card
Two weeks
Payment failed

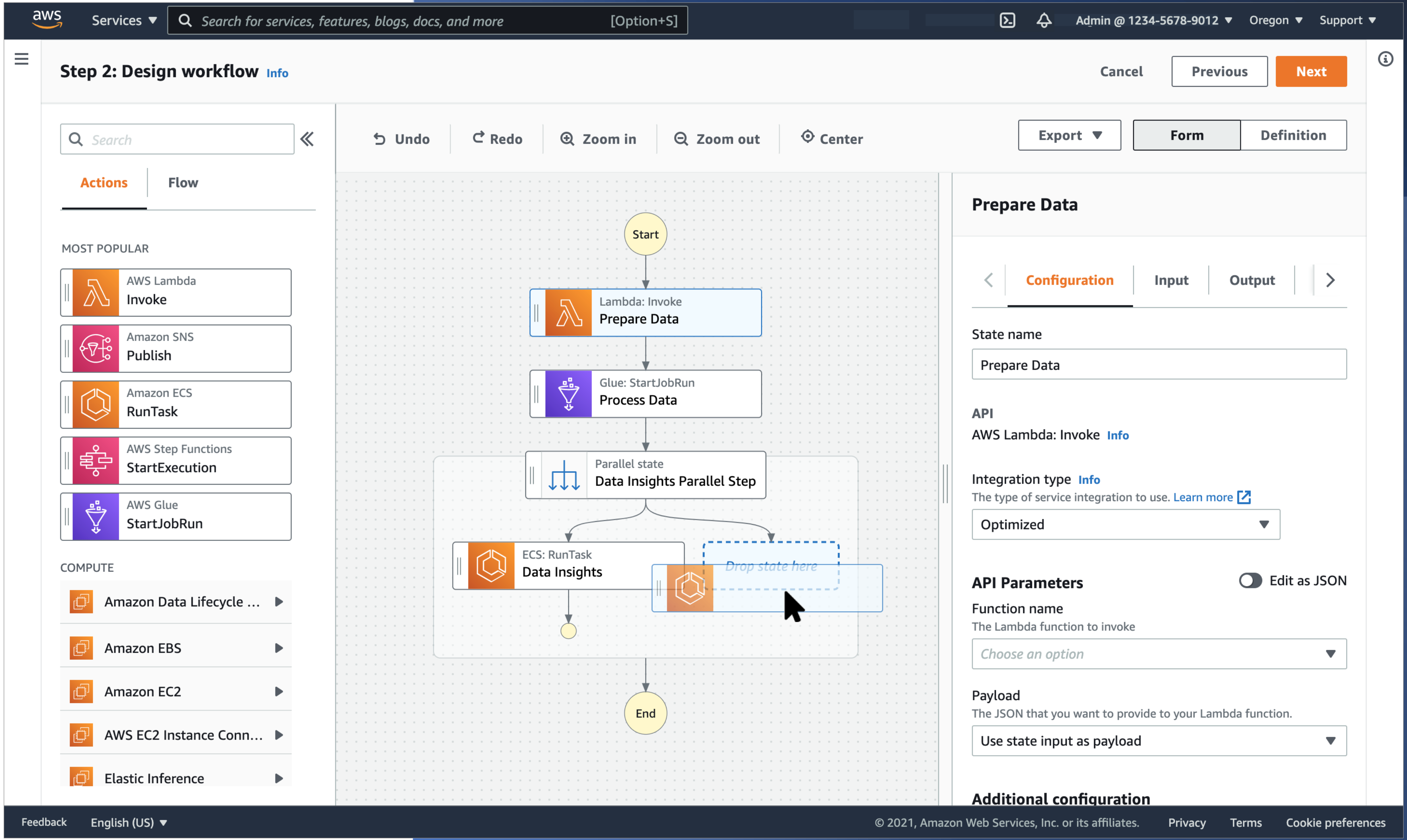
State

- In Axis, we are using orchestrator coordinated saga pattern which is nothing but orchestrator. In reality , It creates a saga(process) when ever new request is been sent to it from frontend.
- Orchestrator is responsible for containing the business logic and also state of the each request. Other services doesn’t know whether any state is been managed.
- If you re-login again, saga checks the state and based on the decided what to do next.
- If any thing goes wrong, we can check the state logs and realize where it is stuck and based on that debug the issue.
- One drawback, I will mention is that it is not visual. If something goes wrong. We have to login to db to check the problem. Which in production not everyone has that access and it takes more time.

Any tool/ framework that is available, which does this things?

Netflix Conductor Workflow example





Step 2: Design workflow Info

MOST POPULAR

- AWS Lambda Invoke
- Amazon SNS Publish
- Amazon ECS RunTask
- AWS Step Functions StartExecution
- AWS Glue StartJobRun

COMPUTE

- Amazon Data Lifecycle ...
- Amazon EBS
- Amazon EC2
- AWS EC2 Instance Conn...
- Elastic Inference

Prepare Data

State name: Prepare Data

API: AWS Lambda: Invoke Info

Integration type Info
The type of service integration to use. Learn more
Optimized

API Parameters — Edit as JSON

Function name
The Lambda function to invoke
Choose an option

Payload
The JSON that you want to provide to your Lambda function.
Use state input as payload

Additional configuration

Uber Cadence Workflow

Reference

- Spring I/O 2022: Be proactive about state in a reactive system by Nele Lea Uhlemann (https://www.youtube.com/watch?v=KUsPQGi3dFg&ab_channel=SpringI%2FO)
- Saga Pattern: https://microservices.io/patterns/data/saga.html
- Next,
  - Read Book: **Practical Process Automation: Orchestration and Integration in Microservices and Cloud Native Architectures** by Bernd Ruecker (Author)