

State in your reactive system

Reactive Systems Vs Micro Services

Reactive Systems (2014)

- Responsive
- Elastic
- Resilient
- Message Driven

Micro Services (2011)

- structures an application as a loosely coupled set of services
- Each service is owned by a team, which has sole responsibility for making changes.

Other Definitions,

- Independently deployable
- Loose coupling
- Fault tolerant
- flexible



Impatient
Customer

1. Place Order

Warehouse
service

Desk Service

Exchange
Service

2. Place
Shipping
Order

2. Handle order

2. Place
exchange
Order



Impatient
Customer

4. New Order
Delivered

6. Exchange Item
picked up

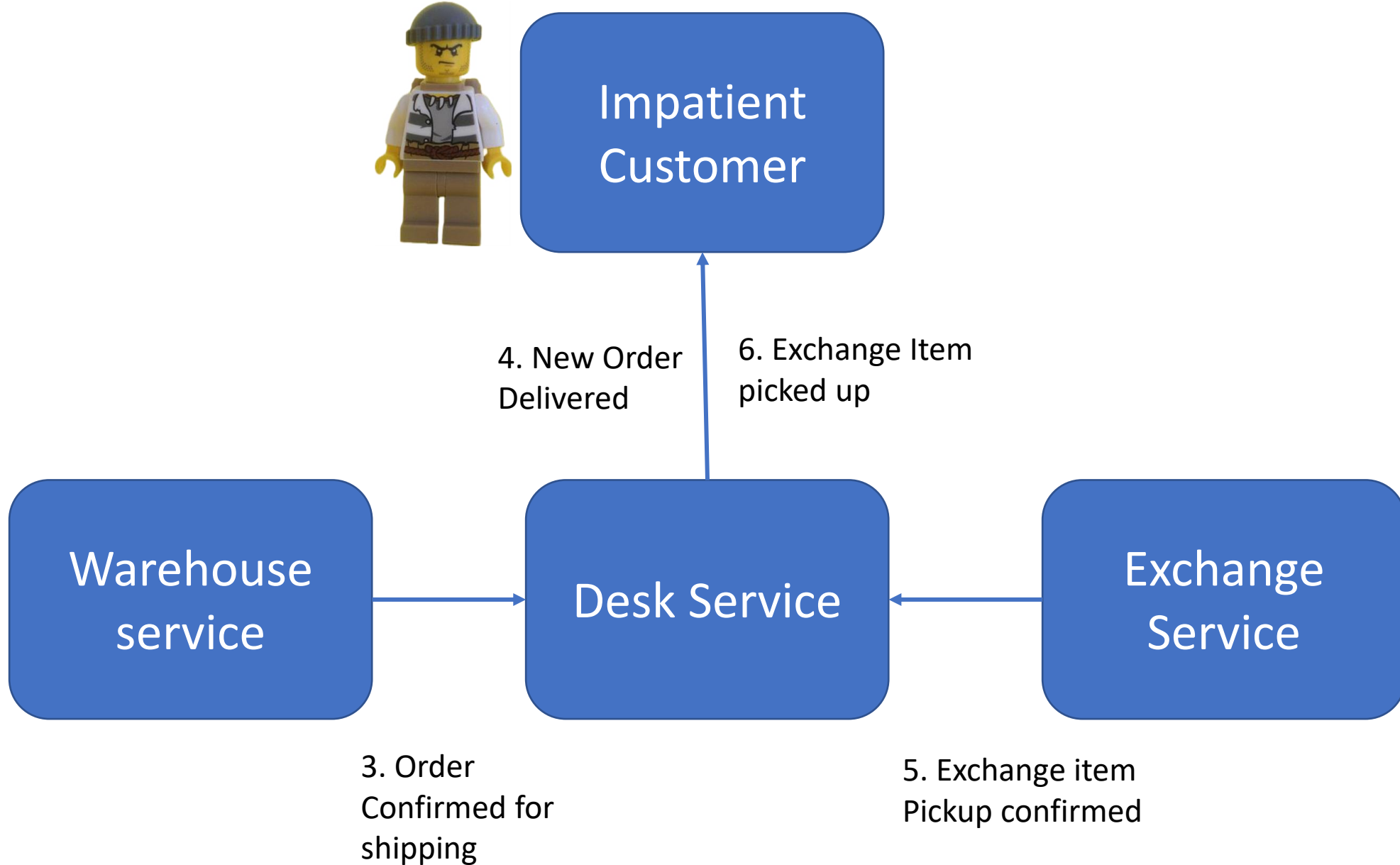
Warehouse
service

Desk Service

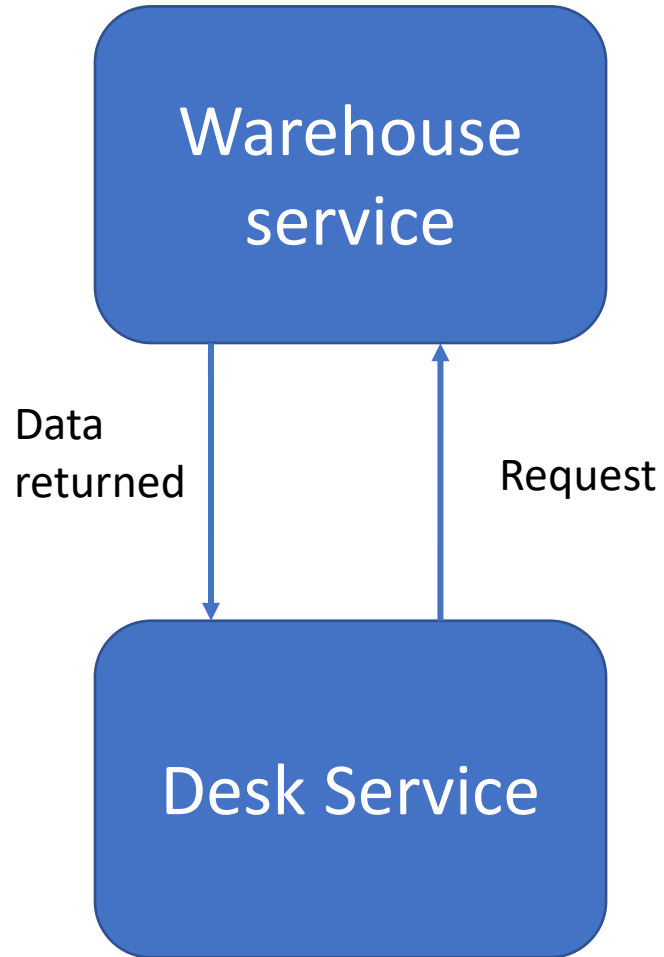
Exchange
Service

3. Order
Confirmed for
shipping

5. Exchange item
Pickup confirmed



Synchronous communication



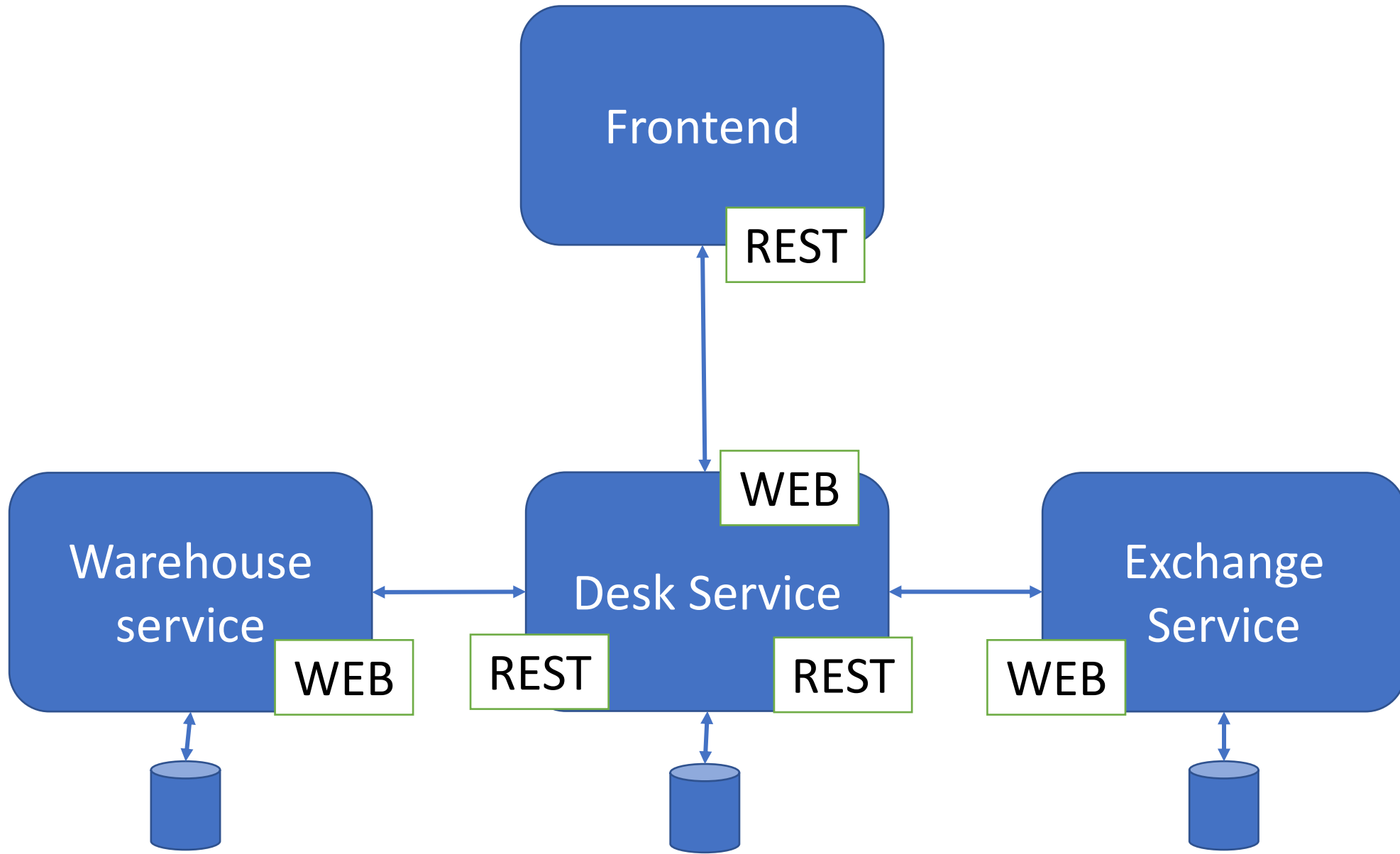
```
//warehouse controller with endpoint
```

```
@PostMapping("/warehouse")
```

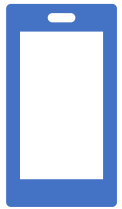
```
fun prepare(@RequestBody orders:Orders):Orders {  
    return warehouseService.prepareOrders(orders)  
}
```

```
//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  
}
```



Blocking



Responsive



Elastic

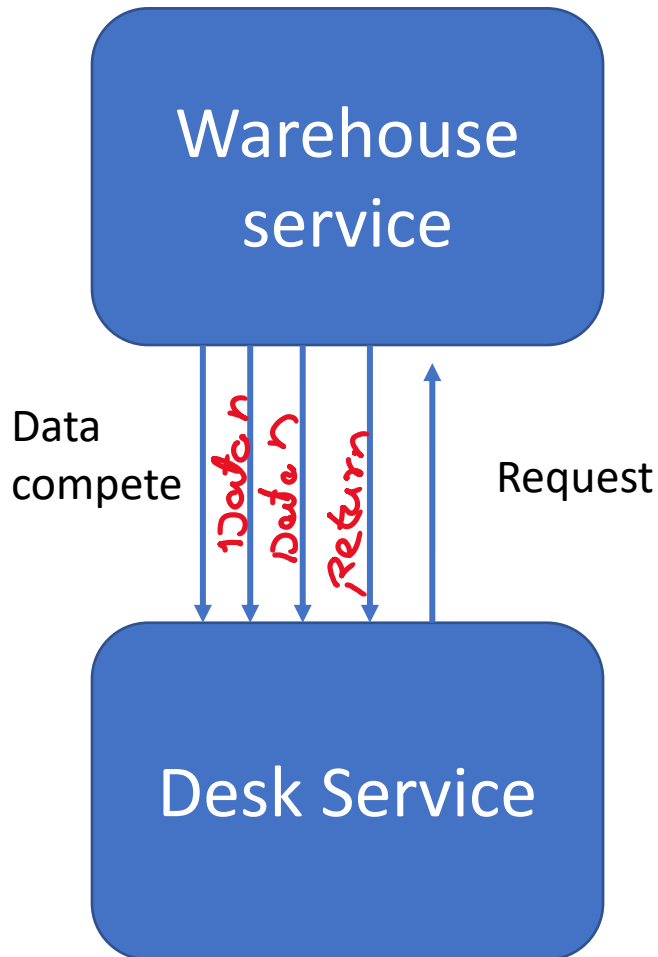


Resilient



Message Driven

Asynchronous communication



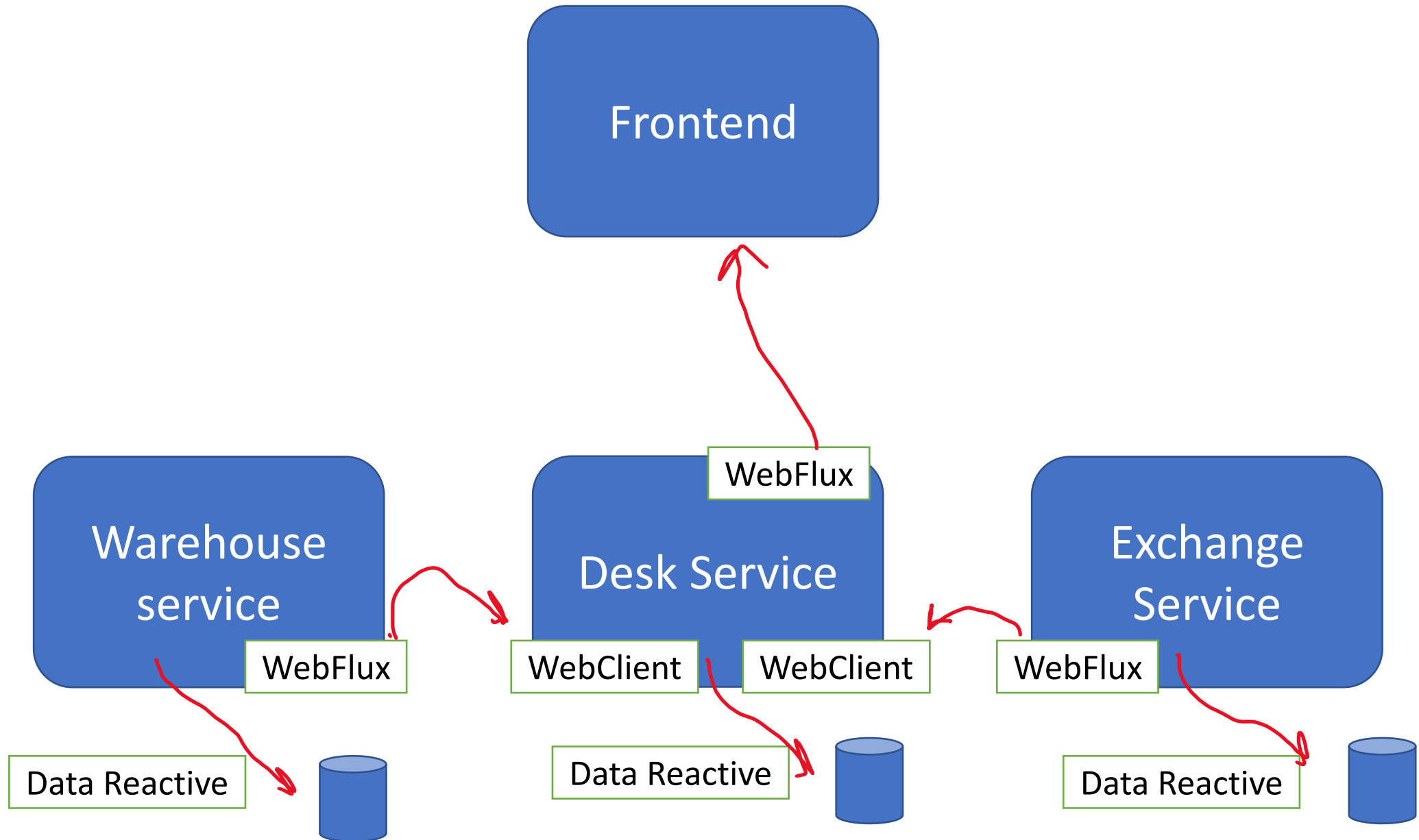
//warehouse controller with endpoint

`@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) }
```

Interface spaghetti



Responsive



Elastic



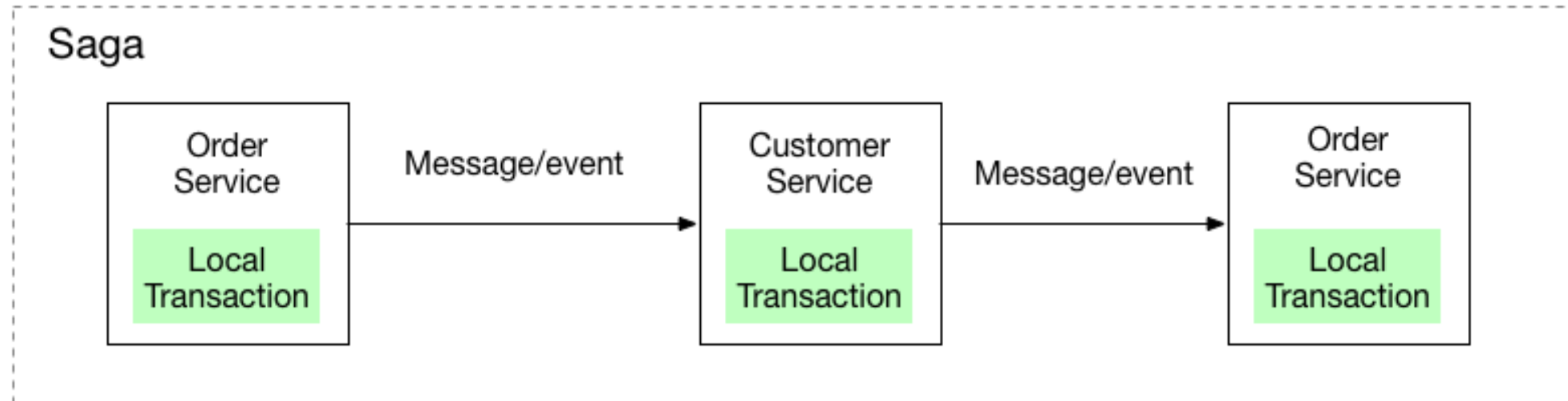
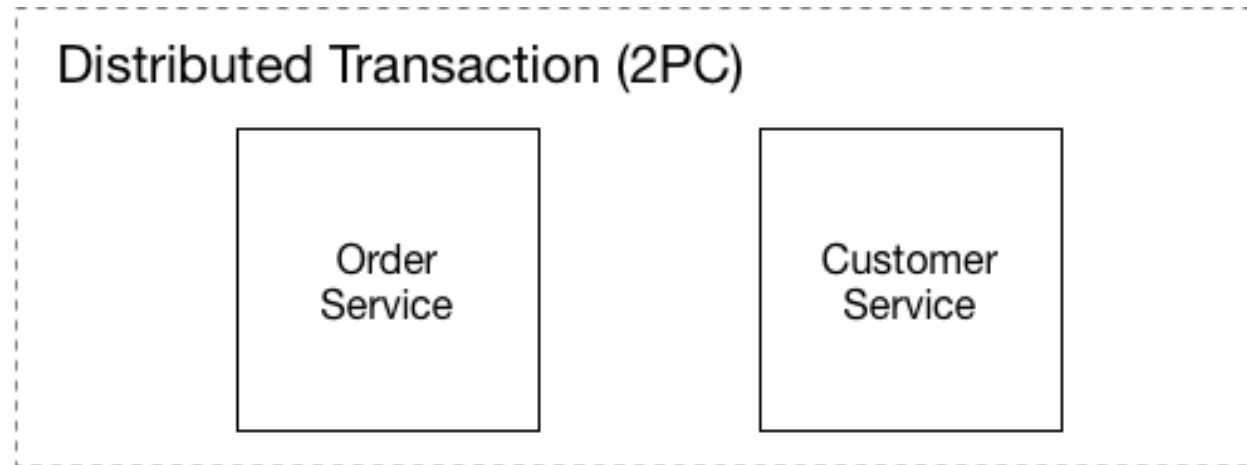
Resilient



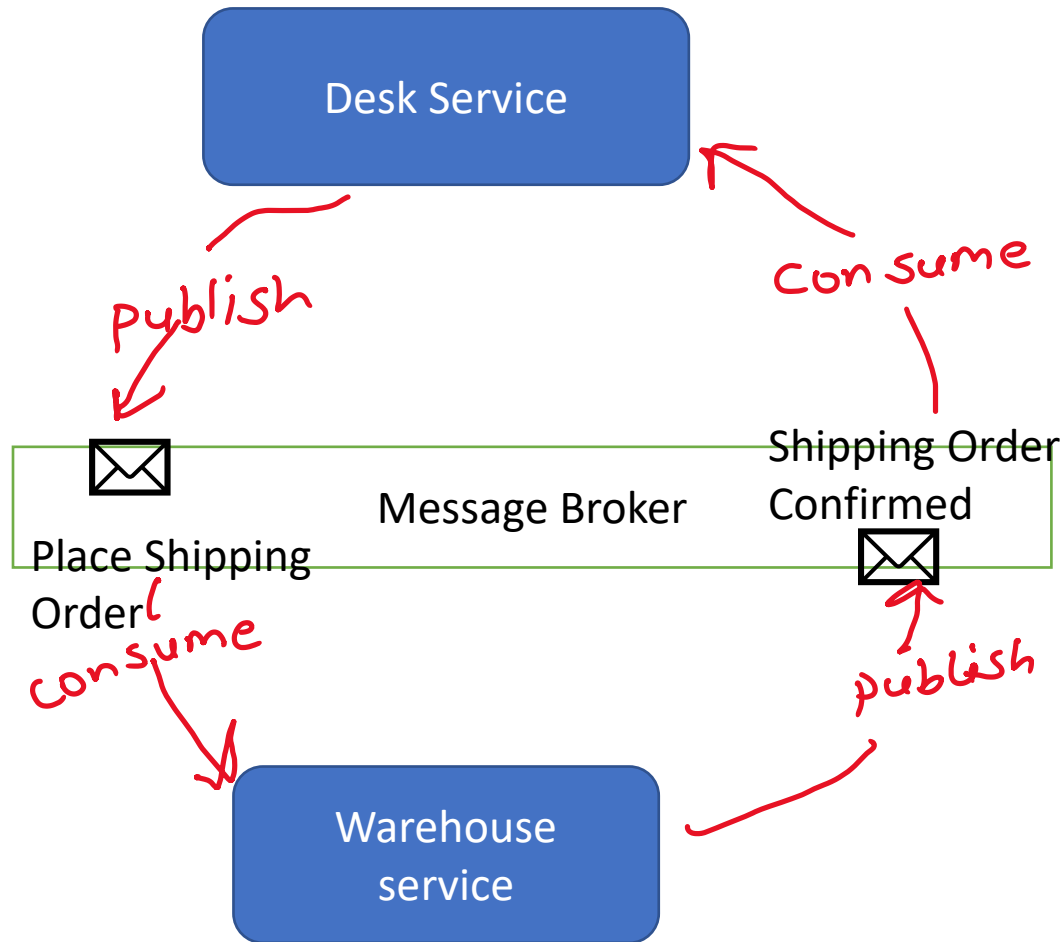
Message Driven

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**.
- Implementing saga pattern with reactive coding is just not possible

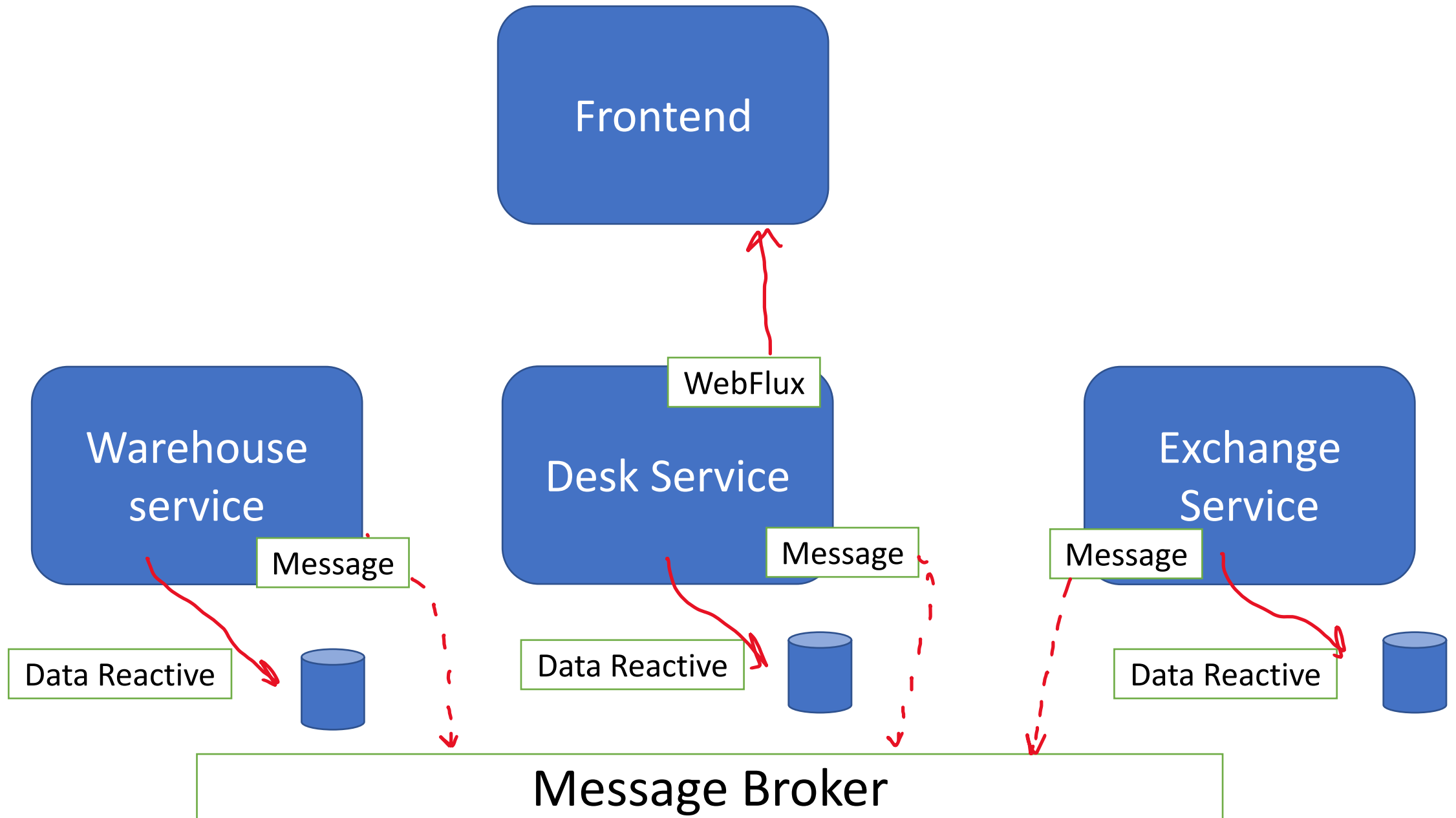


Message Driven



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

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



Awesome we achieved all four!!



Responsive



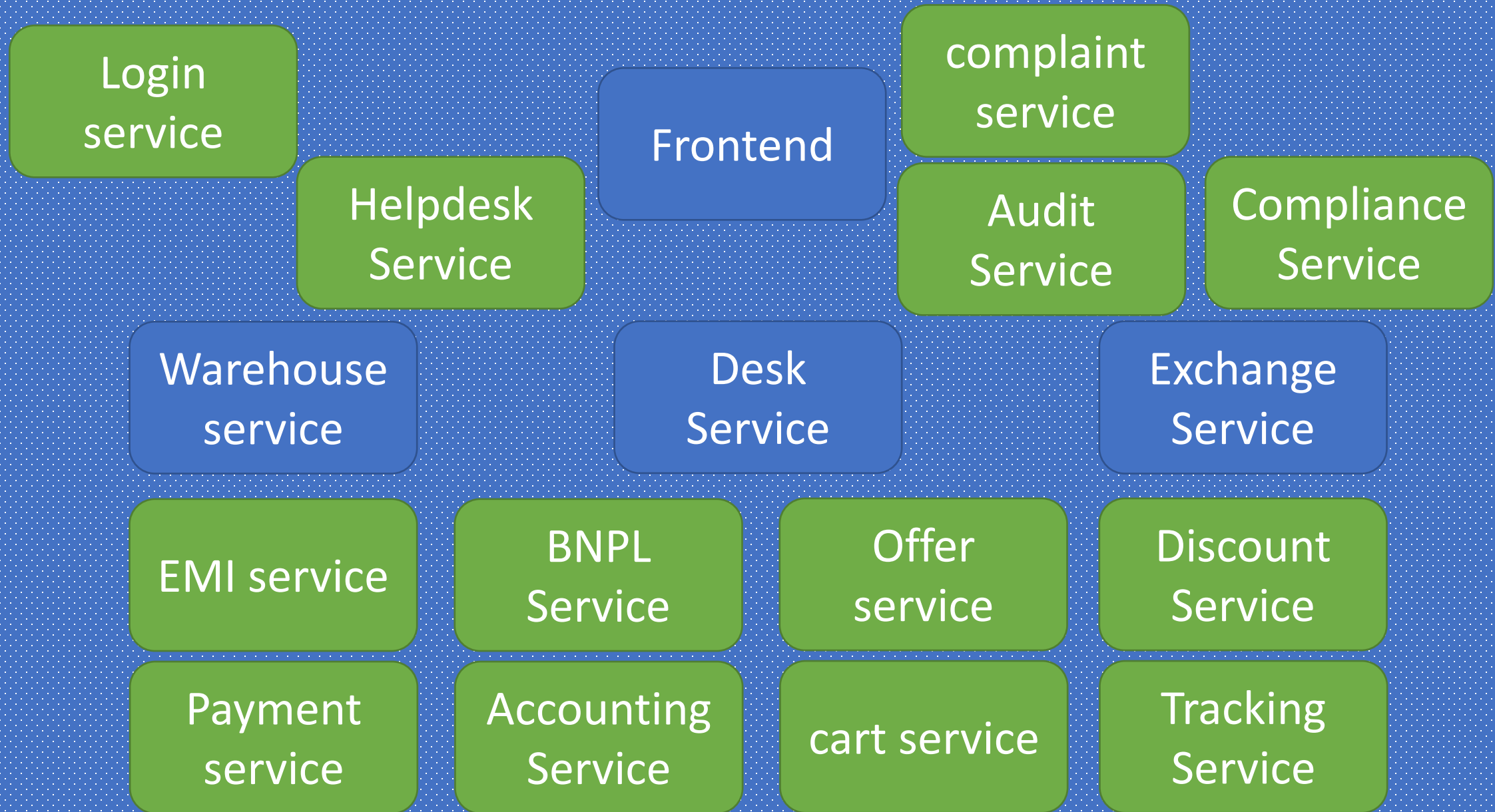
Elastic



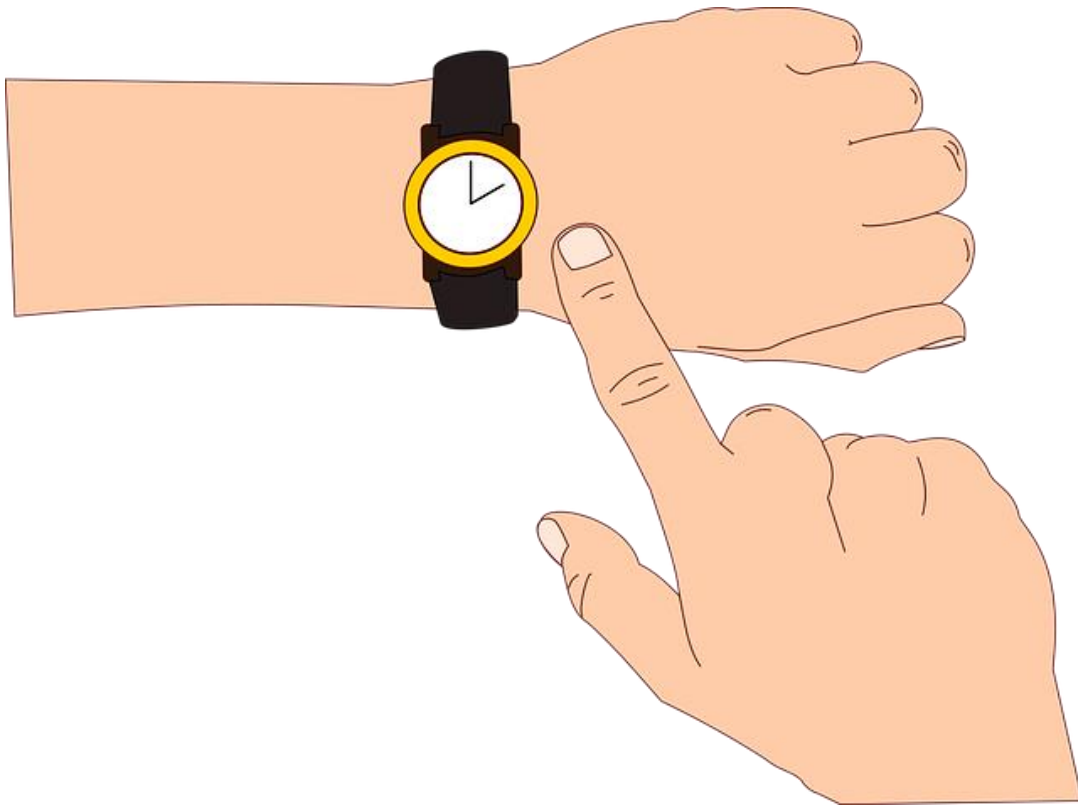
Resilient



Message Driven



I haven't received my order yet, Where is it?

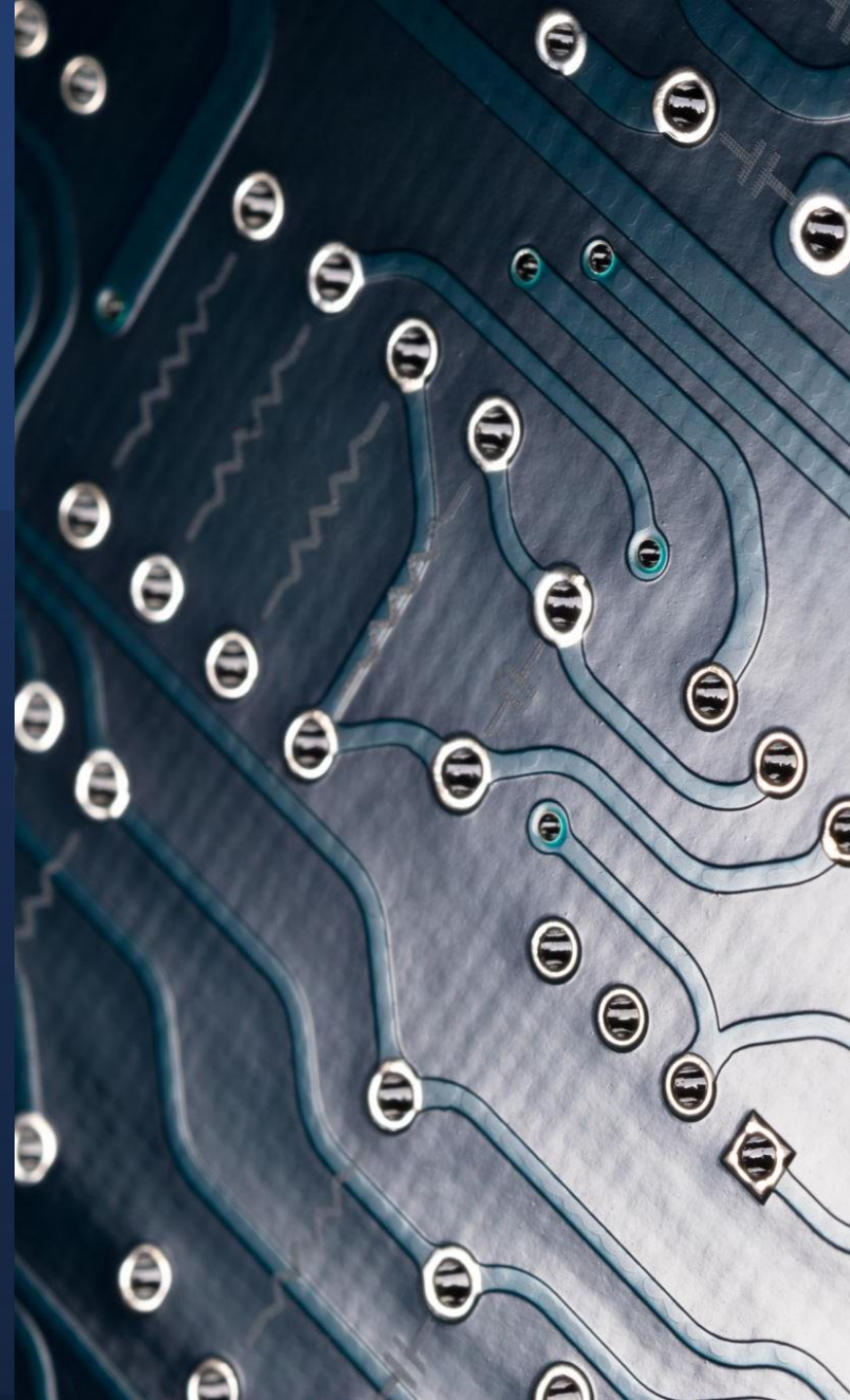




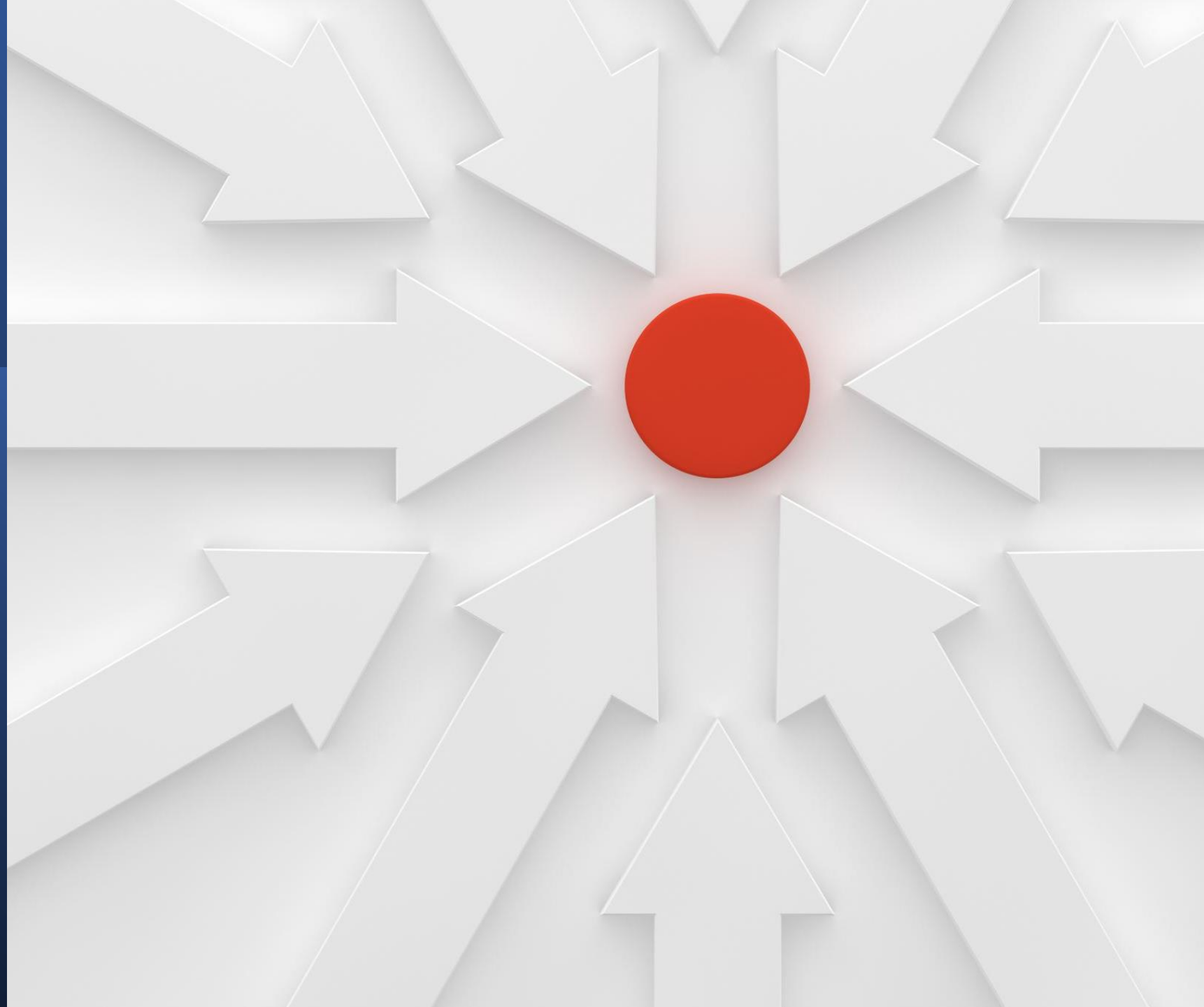
How to bring order in chaos?

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

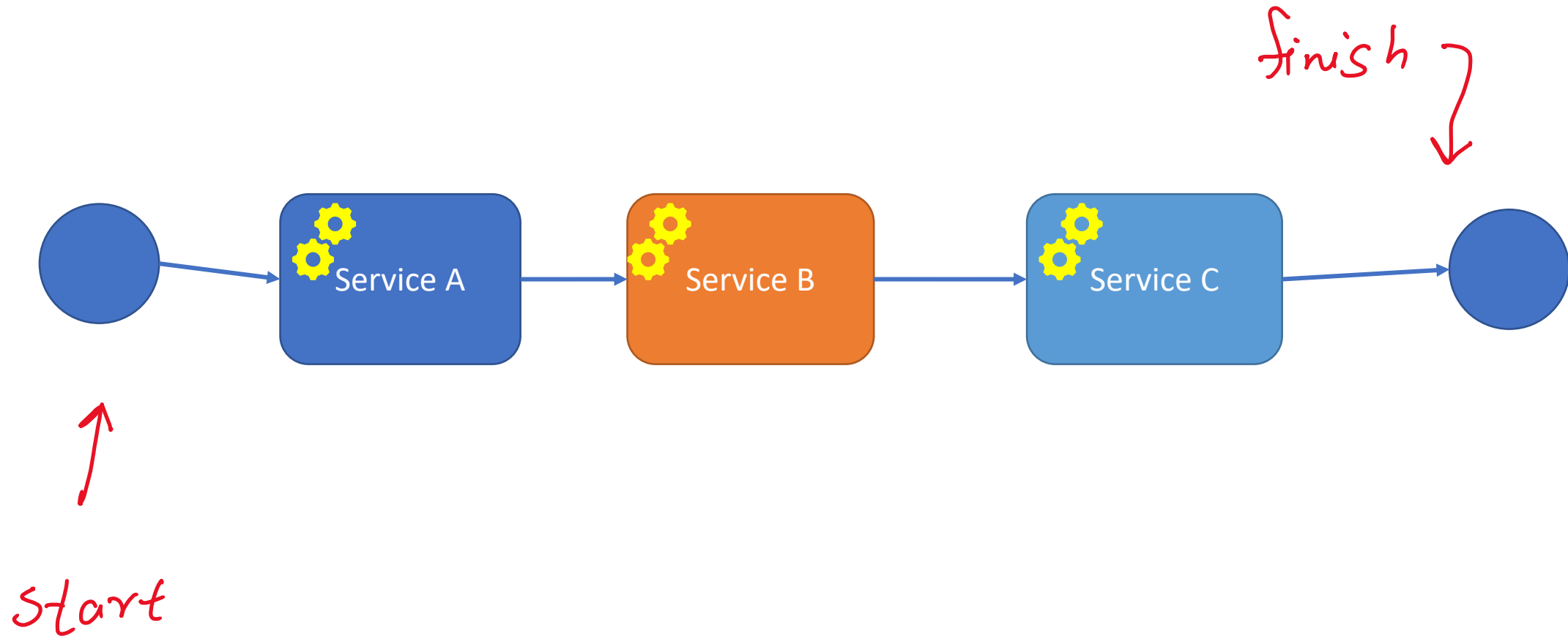
State in a reactive system



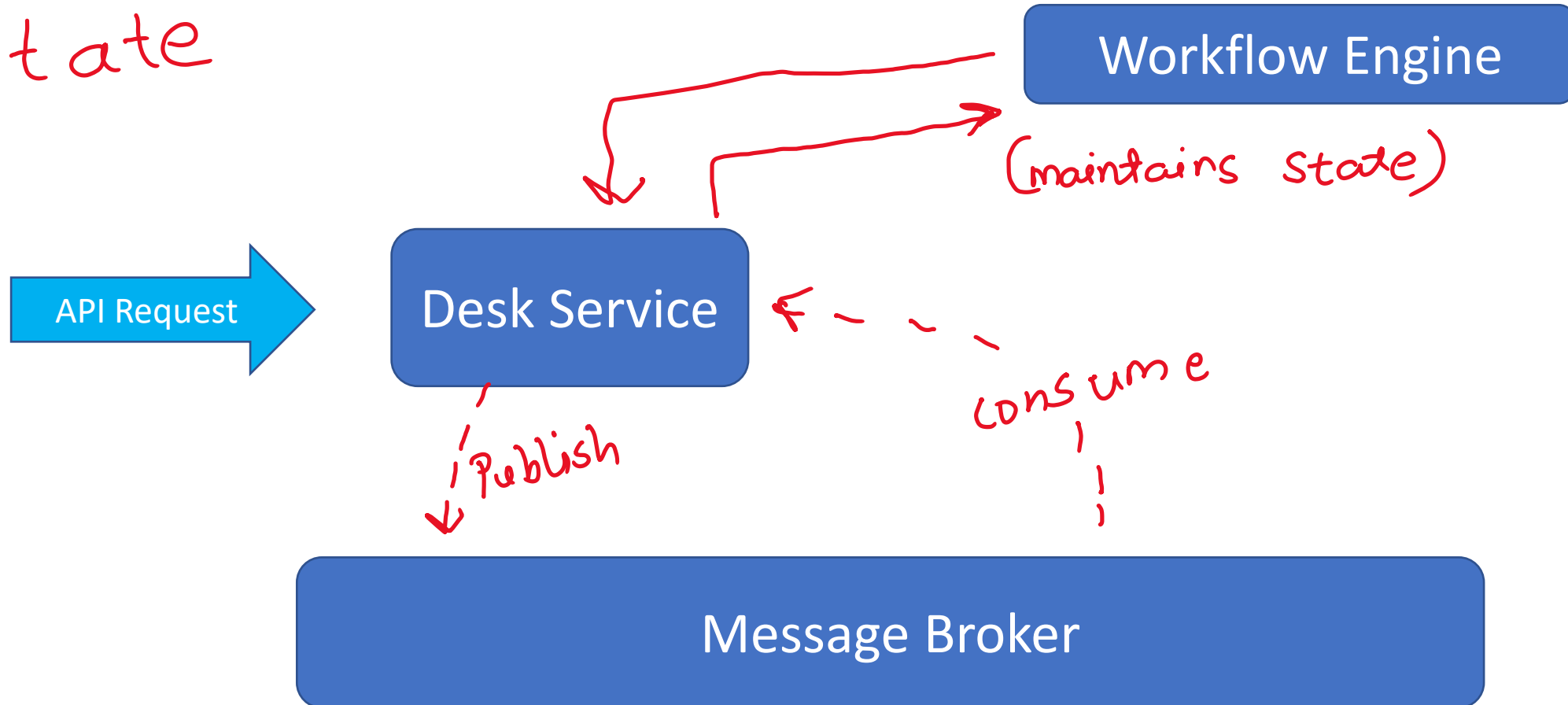
Solution:
Define
saga(process)
in an
orchestrated
way



Workflow Engines



state



Desk Service

enterShop

.next(orderProduct)

.next(placeOrder)

.next(makePayment, dependsOn= placeOrder)

.next(checkPaymentStatus)

.next(createInvoice, dependsOn = checkPaymentStatus)

.next(sendCommunication)

val placeOrder = decisionBranch(WITH_EXCHANGE= flow1,
WITHOUT_EXCHANGE= flow2)

val flow1 = parallel(placeShippingOrder, placeExchangeOrder)

val flow2 = placeShippingOrder

Front-end
API
calls

Place Orders

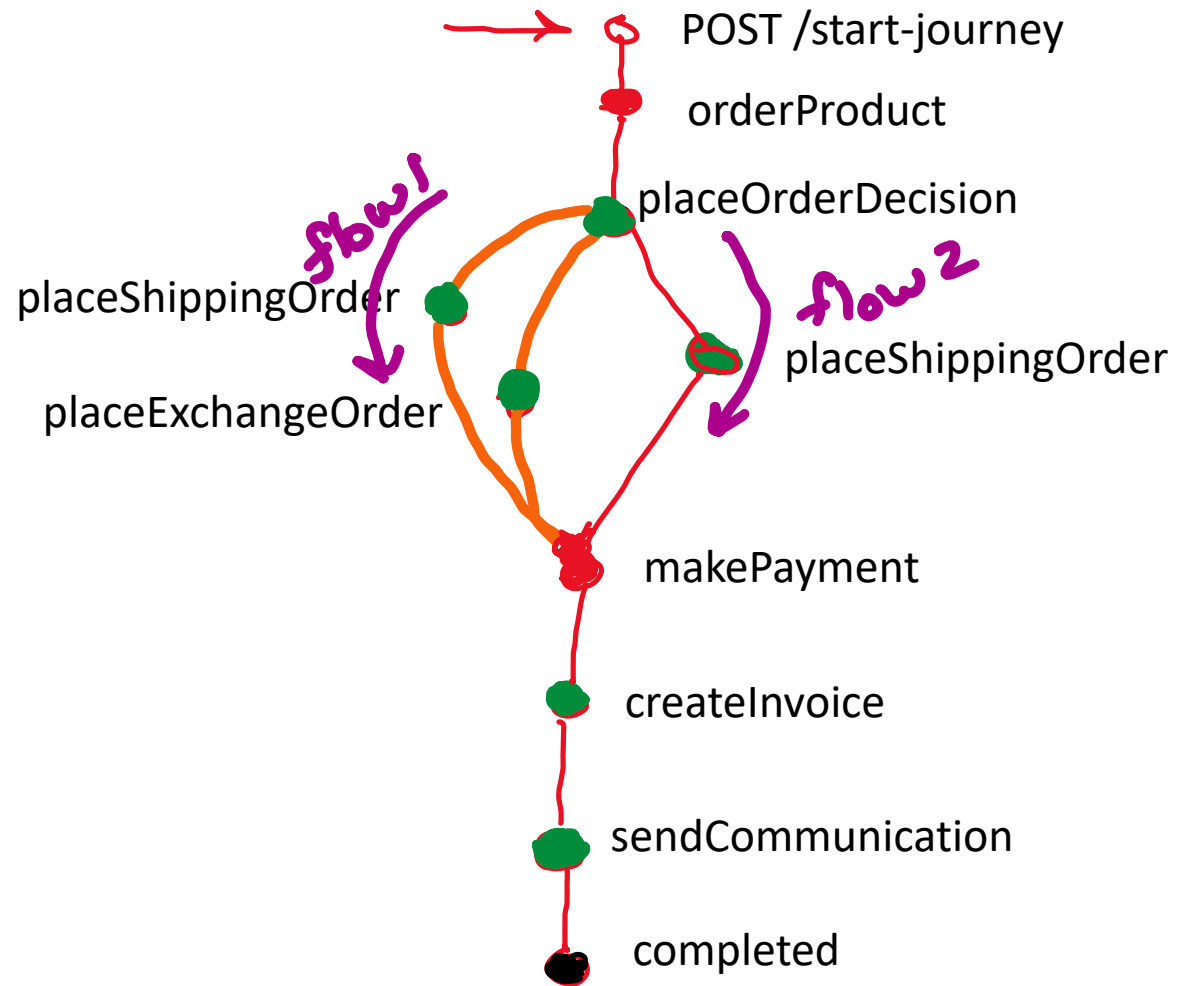
OrderPlaced,
nextStep: waitForConfirmation

Make Payment

PaymentCompleted,
nextStep: Completed

Email,
SMS

Notify Customer



Desk Service

//desk service

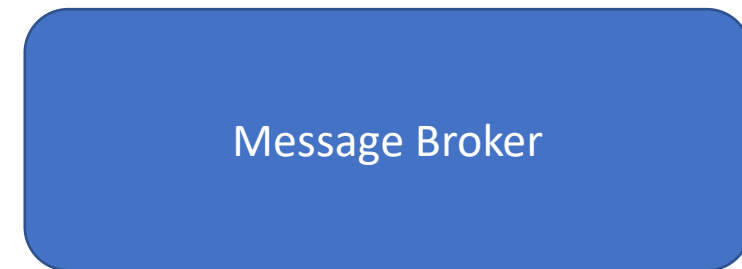
```
class PlaceOrder(  
    withExchange: Step,  
    withoutExchange: Step  
) : DecisionStep() {  
  
    override fun execute(orders: Orders) : Step {  
        return if(orders.withExchange){ withExchange  
        } else { withoutExchange }  
    }  
}
```

//desk service

```
class PlaceShippingOrder(  
    val producer: ShippingOrderProducer  
) : SystemStep() {  
  
    override fun execute(orders: Orders) {  
        doSomething(orders)  
        producer.send(orders.identifier)  
    }  
}
```

//warehouse service

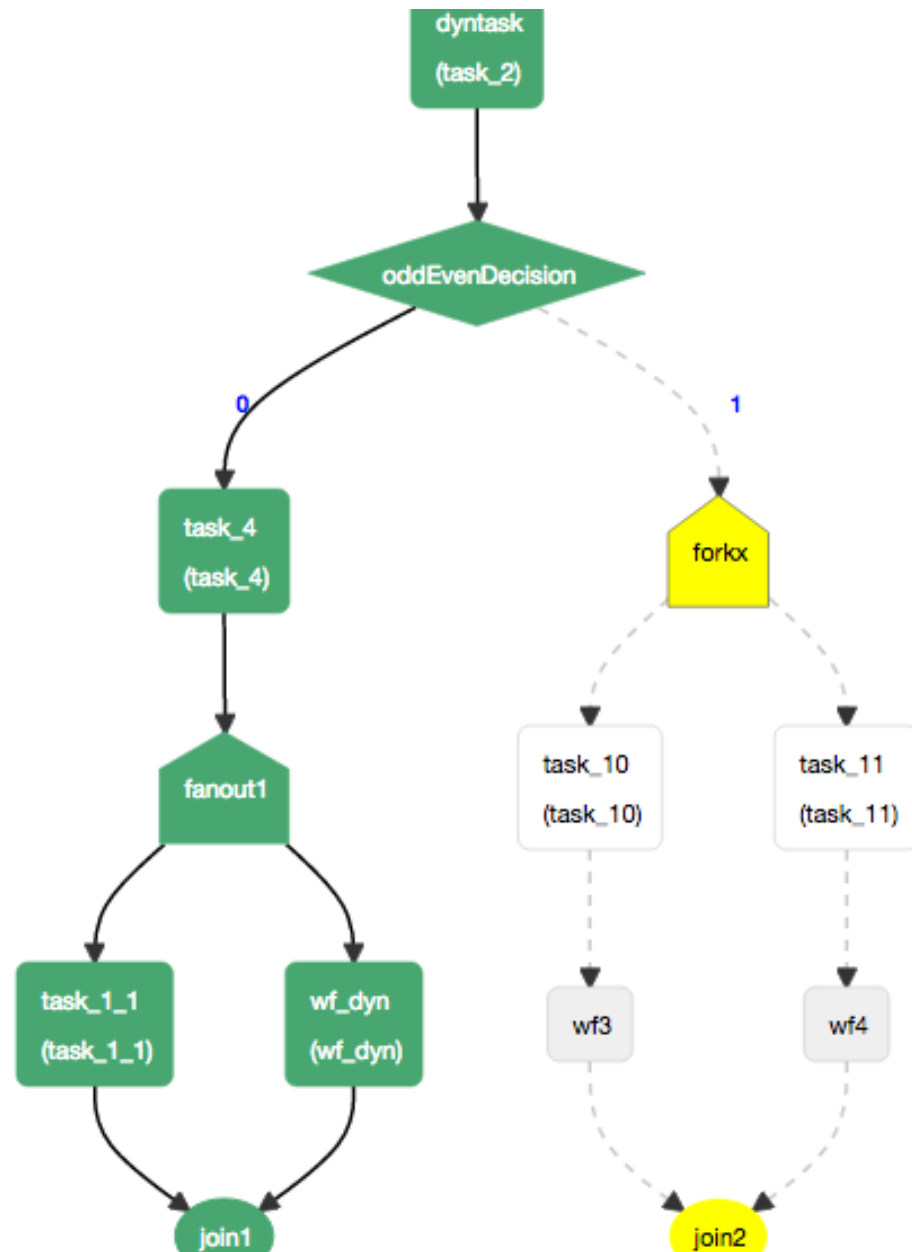
```
@kafkaListener(topics="shippingOrdered")  
fun prepareShippings(orderId:String) {  
    Println("Your shippings is ready")  
    shippingConfirmProducer.sendConfirmation(orderId)  
}
```



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



Netflix Conductor Workflow example





Services ▾

[Option+S]



Admin @ 1234-5678-9012 ▾

Oregon ▾

Support ▾

Step 2: Design workflow [Info](#)

Cancel

Previous






Next








Actions

Flow

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 ▶

Undo

Redo

Zoom in

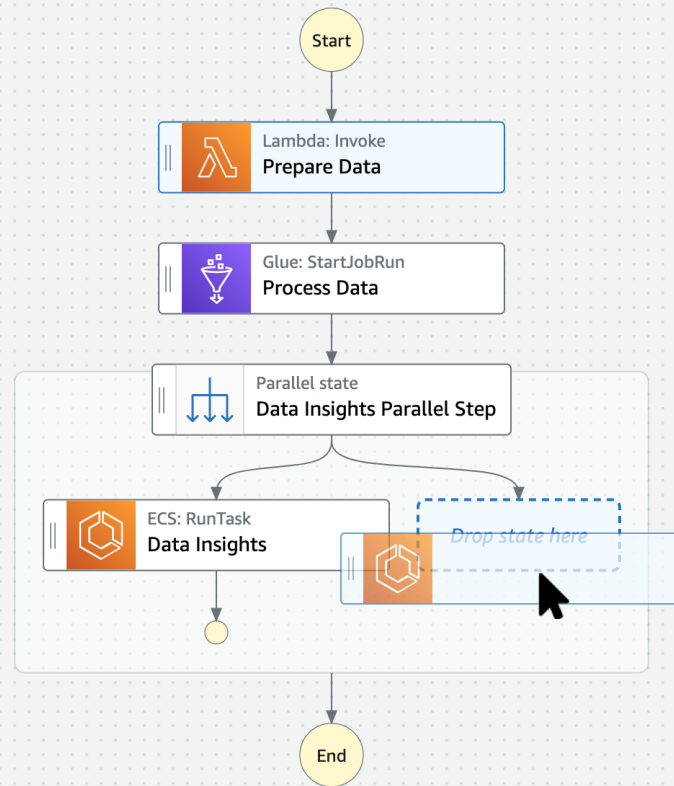
Zoom out

Center

Export ▾

Form

Definition



Prepare Data



Configuration

Input

Output



State name

API

AWS Lambda: Invoke [Info](#)Integration type [Info](#)The type of service integration to use. [Learn more](#)

API Parameters

☐ Edit as JSON

Function name

The Lambda function to invoke

Payload

The JSON that you want to provide to your Lambda function.

Additional configuration

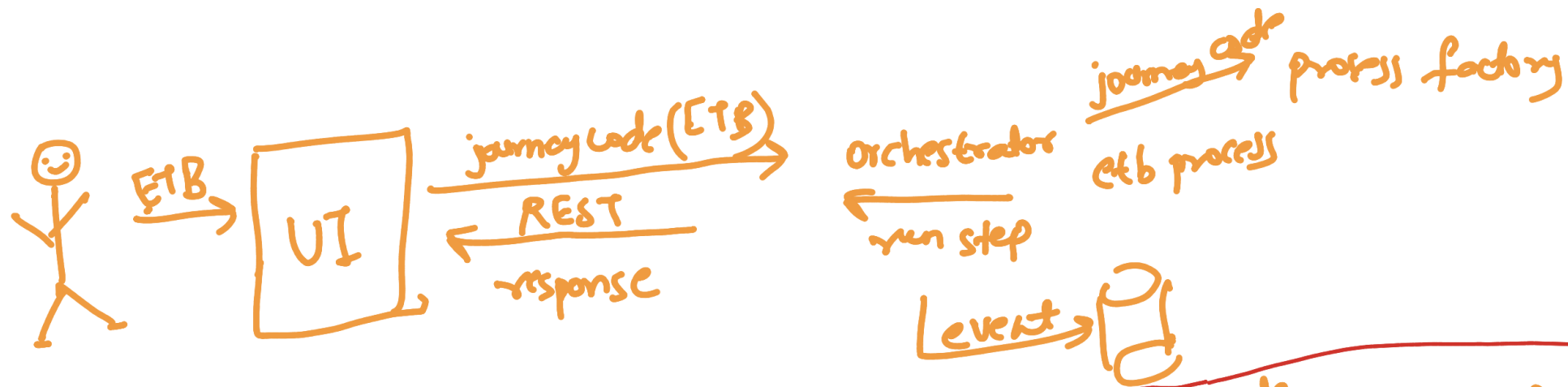
[Feedback](#)

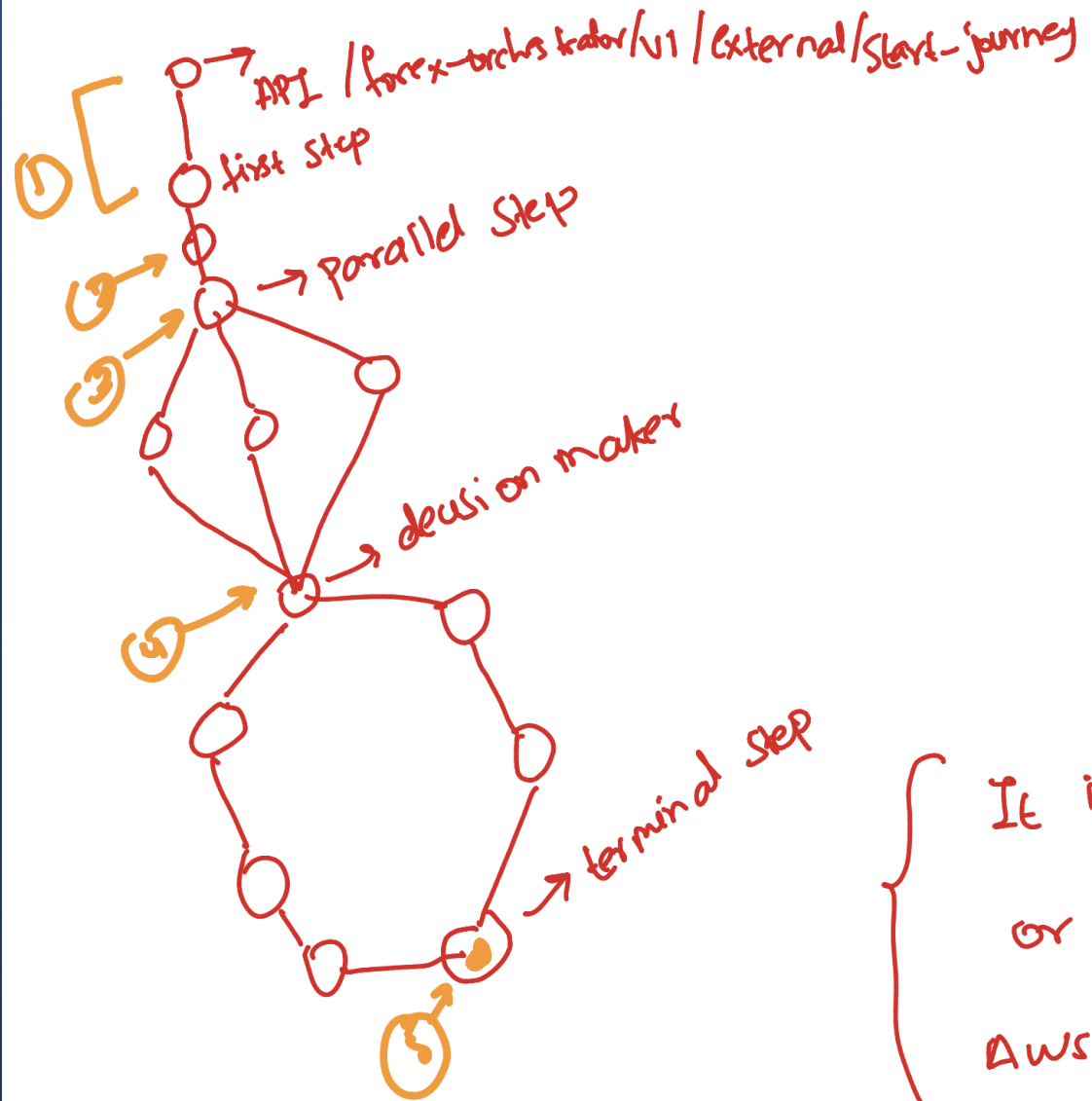
English (US) ▾

© 2021, Amazon Web Services, Inc. or its affiliates.

[Privacy](#)[Terms](#)[Cookie preferences](#)

Let's see another example. (bit complex)





- ① Here most of the logic is in core library
- ② Customer / system / Async Step
- ③ parallel processing
- ④ if...else... ladder like logic
- ⑤ Completion of the journey

It is almost similar to work flow engine
or what we see other tools like
AWS Step function / Redfury conductor / etc. are

claiming

State



In Forex, we are using **orchestrator coordinated saga pattern** (orchestrator core and orchestrator). In reality , **It creates a saga(process)** whenever new request is been sent to it from frontend.



Orchestrator is responsible for containing the **business logic** and also **state of 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.

Thank you

