Cleipnir
Resilient Functions

Resilient Functions - **Prerequisites**

- **dotnet** sdk
- **IDE** - Visual Studio, Rider or Visual Studio Code
- **Docker**
- Clone saga challenge repository @ **cleipnir.net**

What is it?

A .NET framework

assisting with the **implementation** of code

which needs to be executed in its

entirety

Who am I?

- Software Developer with 10 years experience
  Currently Software Consultant @ Brølstærk 🤓
- MSc Computer Science from Aarhus University
- 2 years unfinished PhD in Distributed Systems at Stavanger University
- Cleipnir Resilient Functions are the scattered remains of my failed PhD rebuild into something practical
- Live in Randers with my wife and 3 children
- Like to do Cross Fitness in my spare time
  Let me know if you can help with changing my barbell hang clean into a proper barbell clean!

Use Cases - What can you **use it for**?

- **‘Atomic Business Processes’**
  Any business process which must be executed in its entirety
  in order to avoid inconsistencies

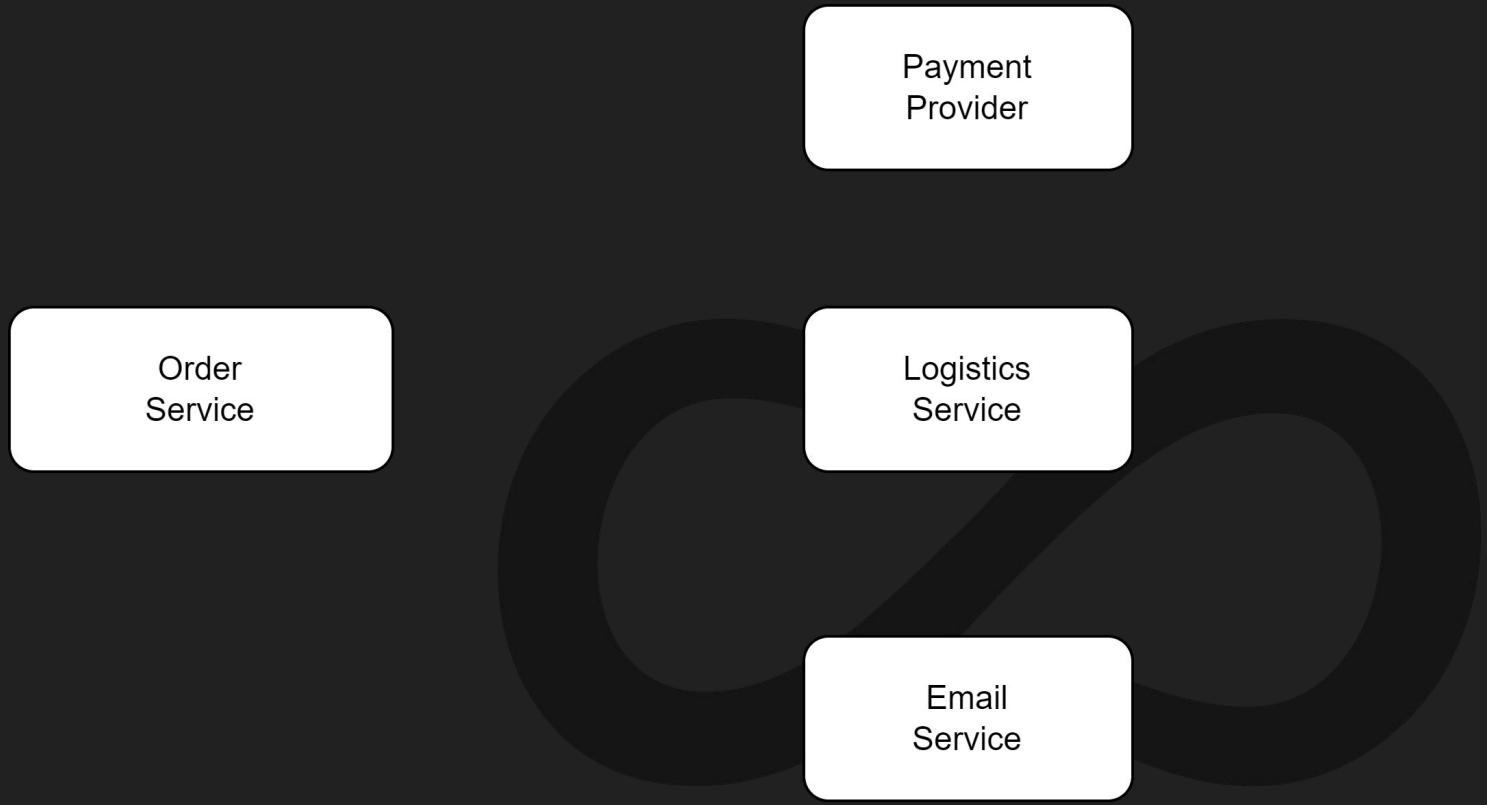
- The situation is more common than we might expect in our microservice system’s landscape.

- I.e. do you have any methods/classes - in your code base - which communicates with multiple external systems?
  What happens if the flow is only partly completed?

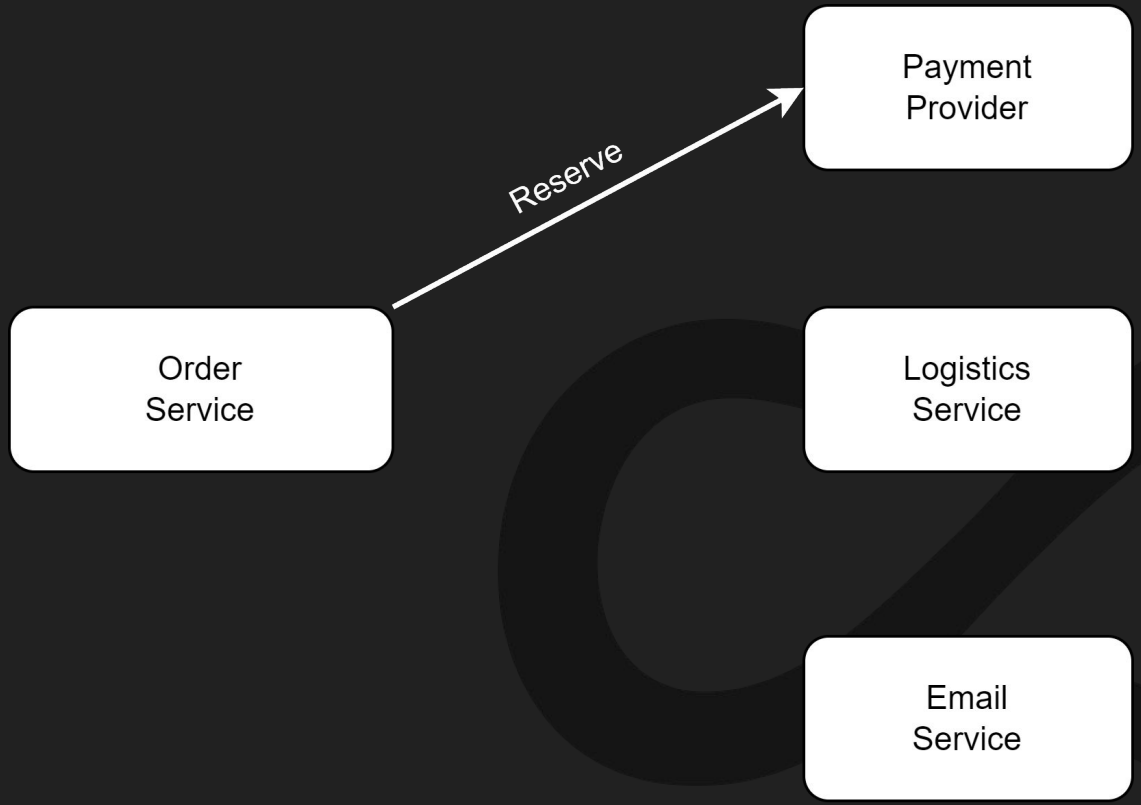
Order Processing Example

```
public async Task ProcessOrder(Order order)

    var transactionId = await _paymentProviderClient.Reserve(order.CustomerId, order.TotalPrice);

    await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, order.ProductIds);
```

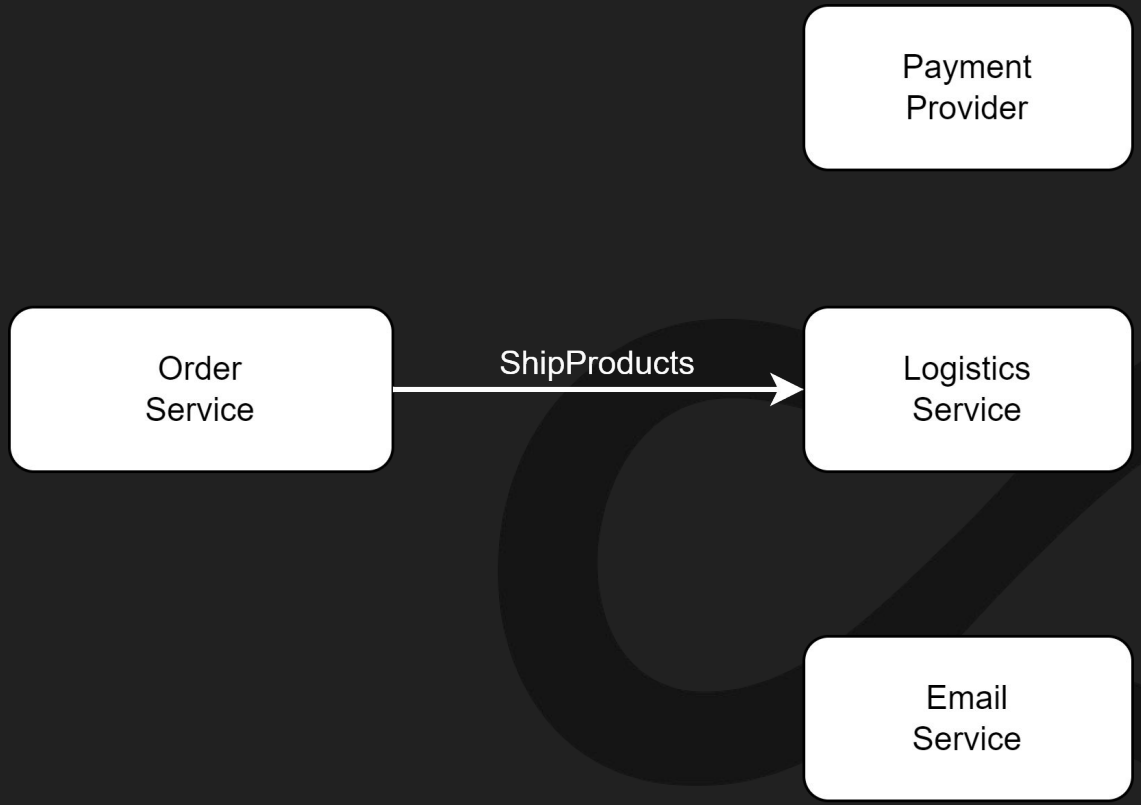
Order Processing Example



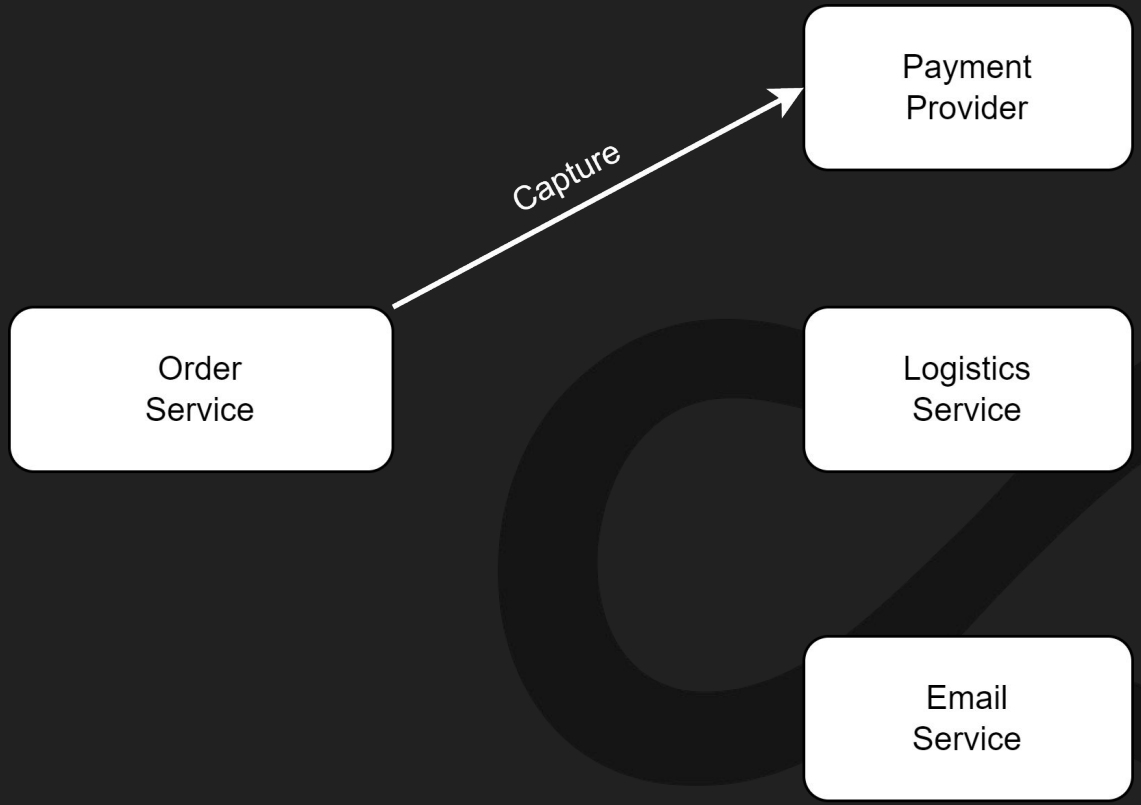
Order Processing Example



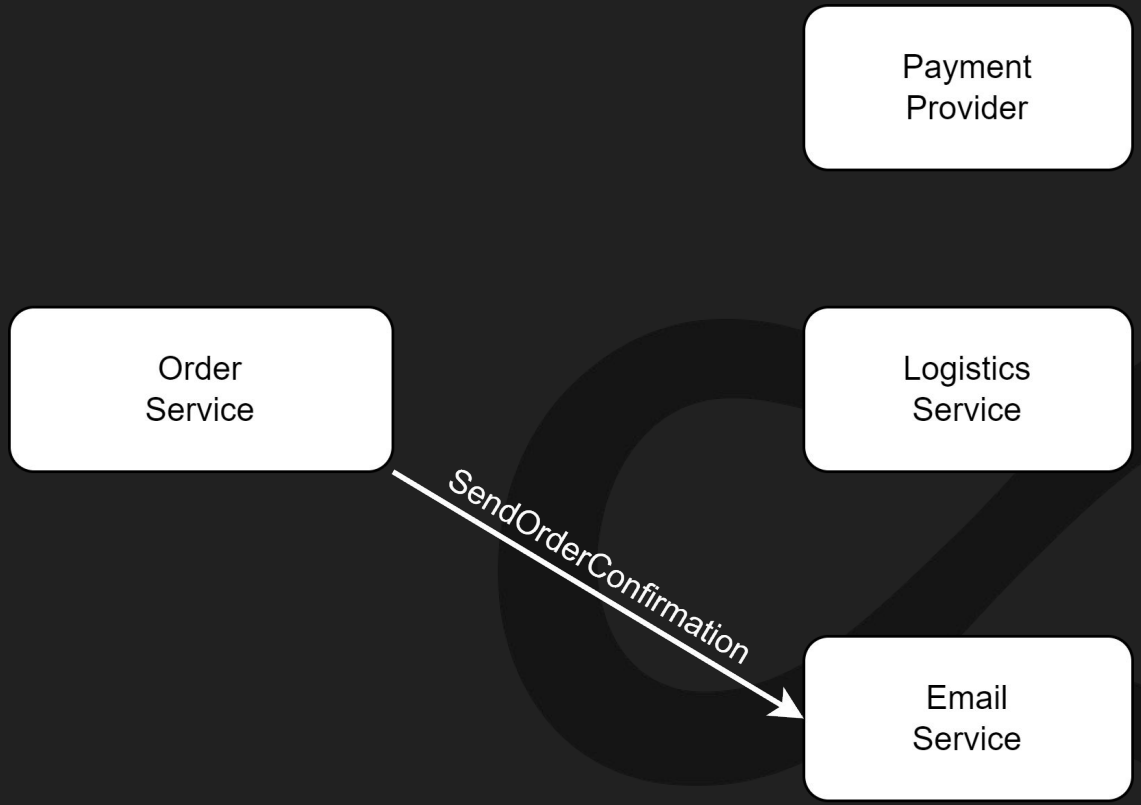
Order Processing Example



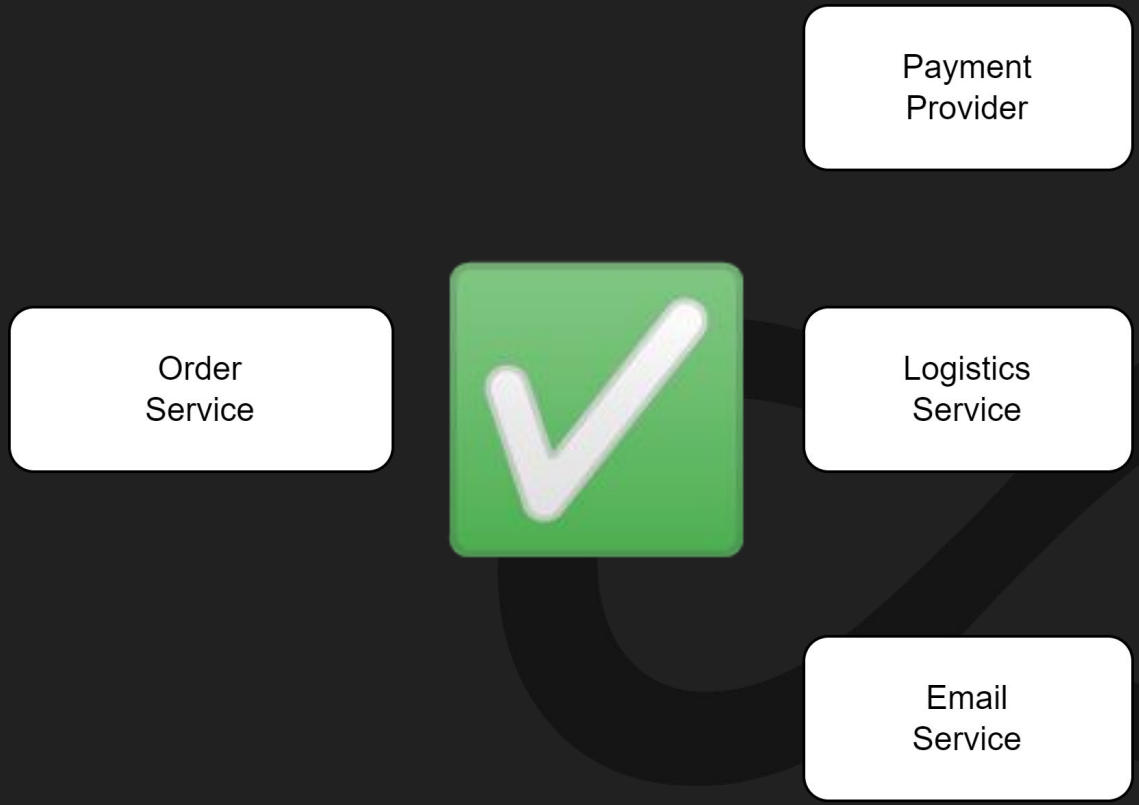
Order Processing Example



Order Processing Example



Order Processing Example



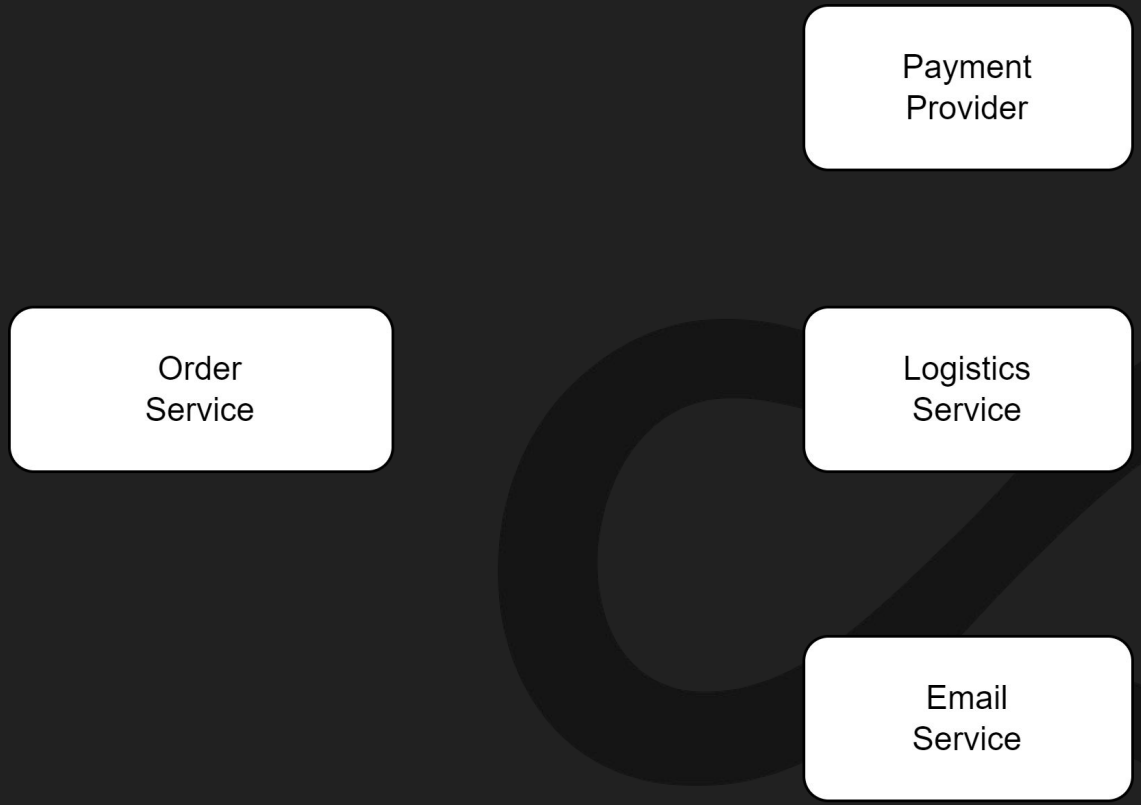
Order Processing Example

```
public async Task ProcessOrder(Order order)

    var transactionId = await _paymentProviderClient.Reserve(order.CustomerId, order.TotalPrice);

    await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, order.ProductIds);
```

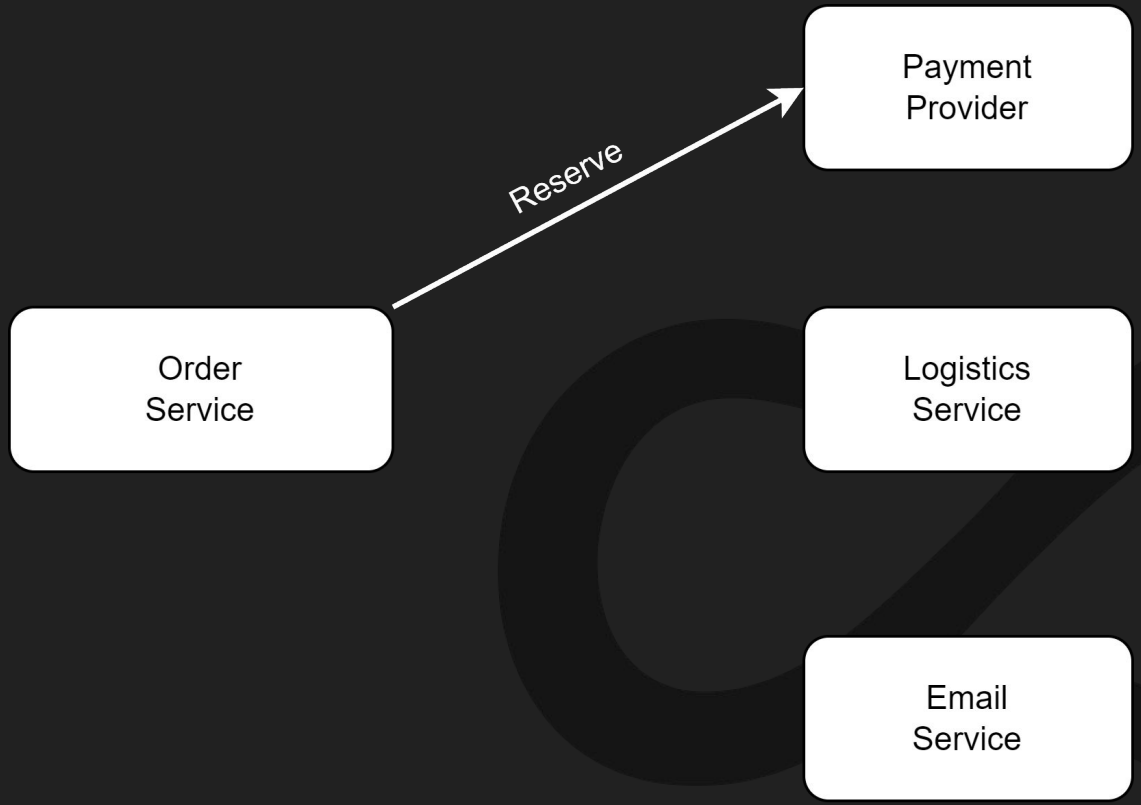
Question:

What can go wrong if we can **crash** at any point during the execution?

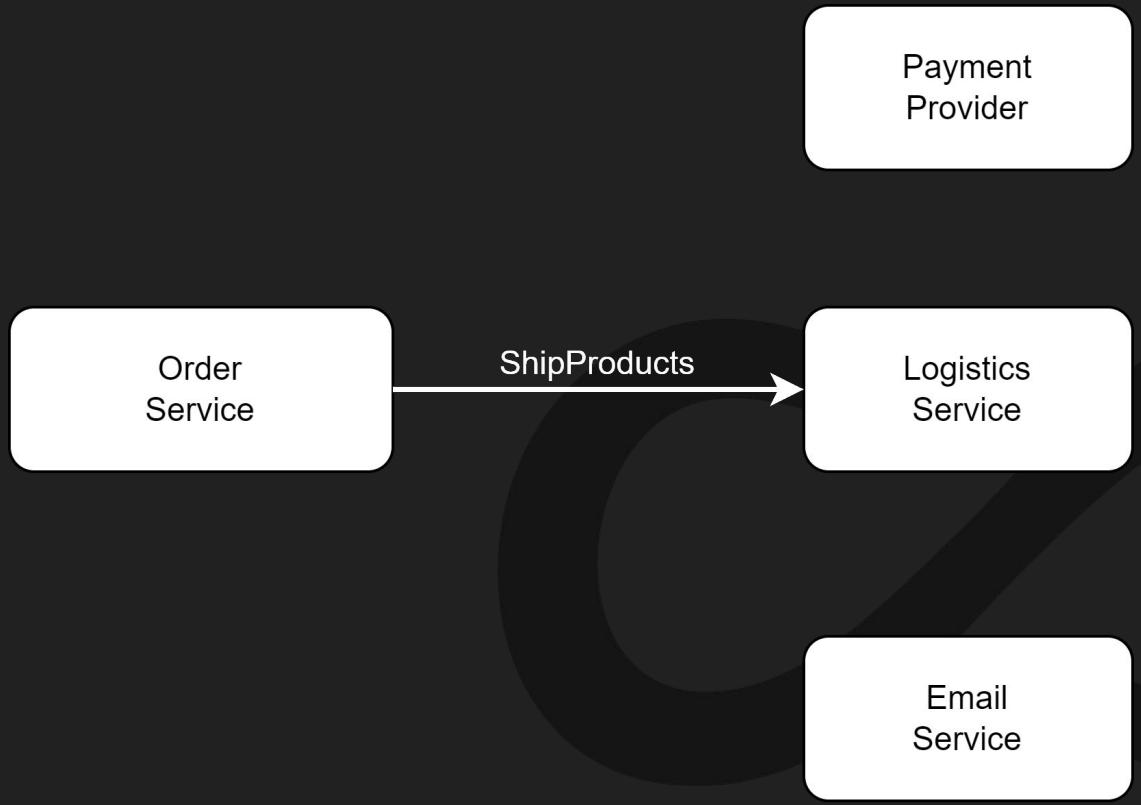
Order Processing Example - Crash



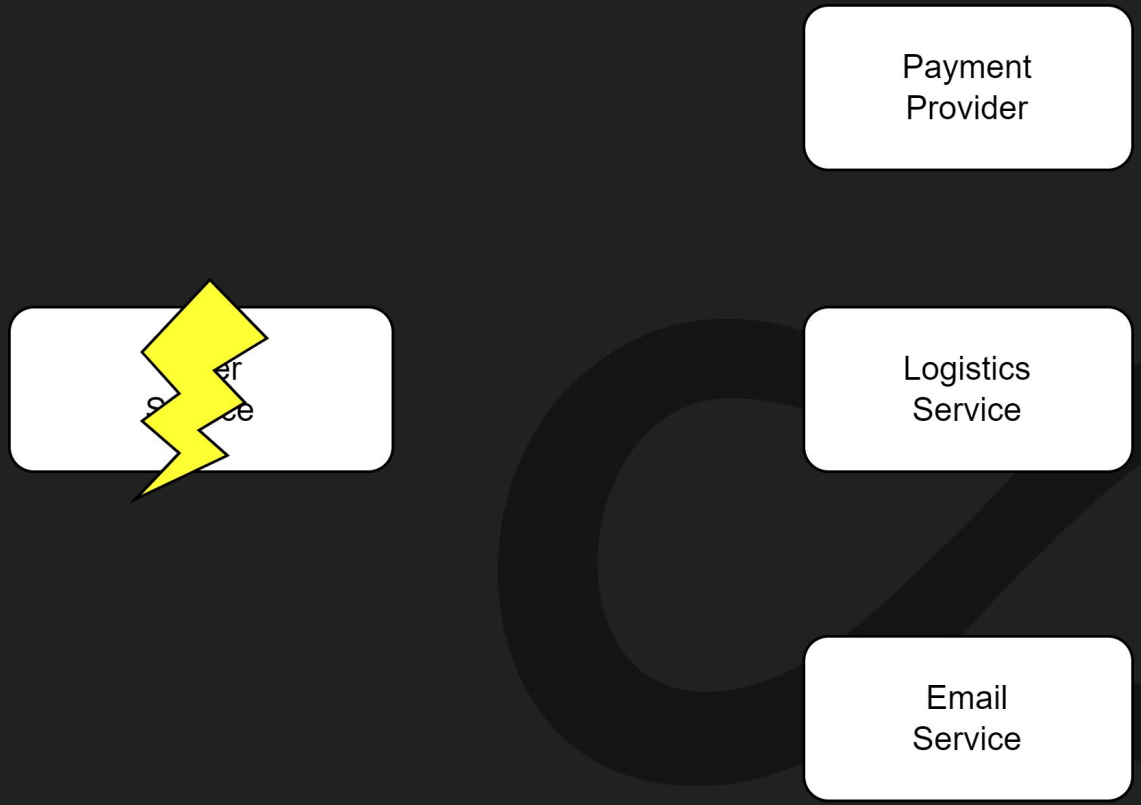
Order Processing Example - Crash



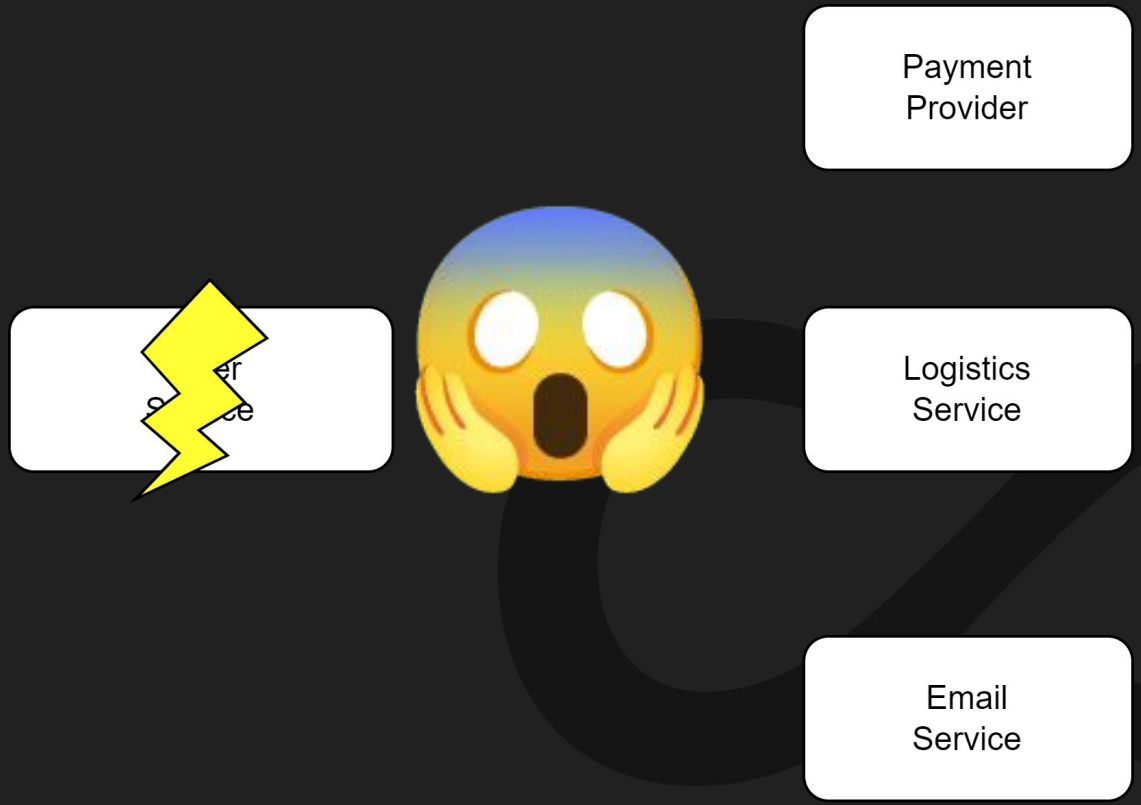
Order Processing Example - Crash



Order Processing Example - Crash



Order Processing Example - Crash



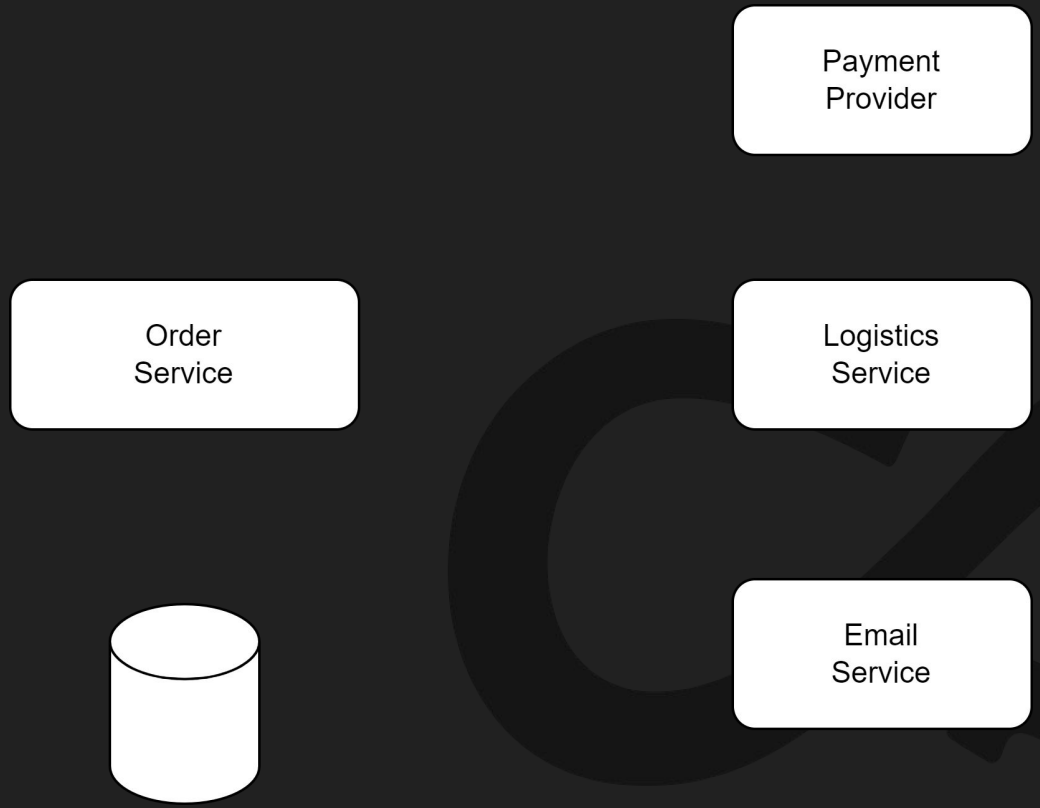
Order Processing Example

```
public async Task ProcessOrder(Order order)

    var transactionId = await _paymentProviderClient.Reserve(order.CustomerId, order.TotalPrice);

    await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, order.ProductIds);
```

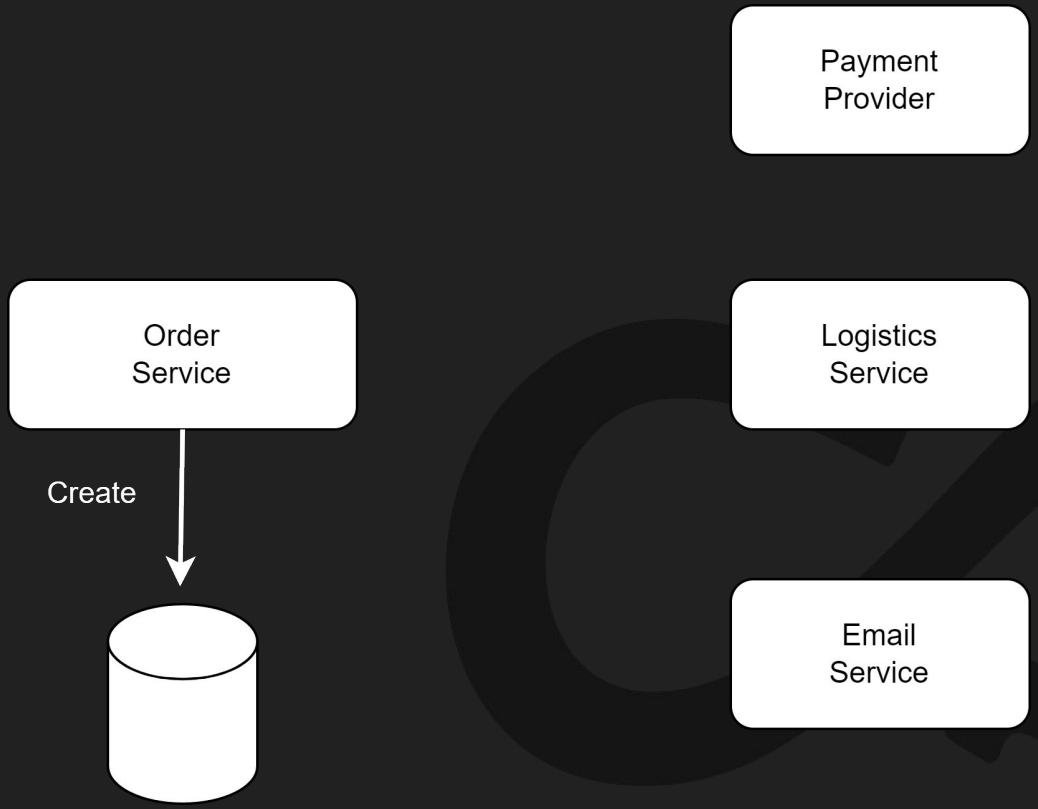
Question:

Can we just **restart** the flow after a crash?

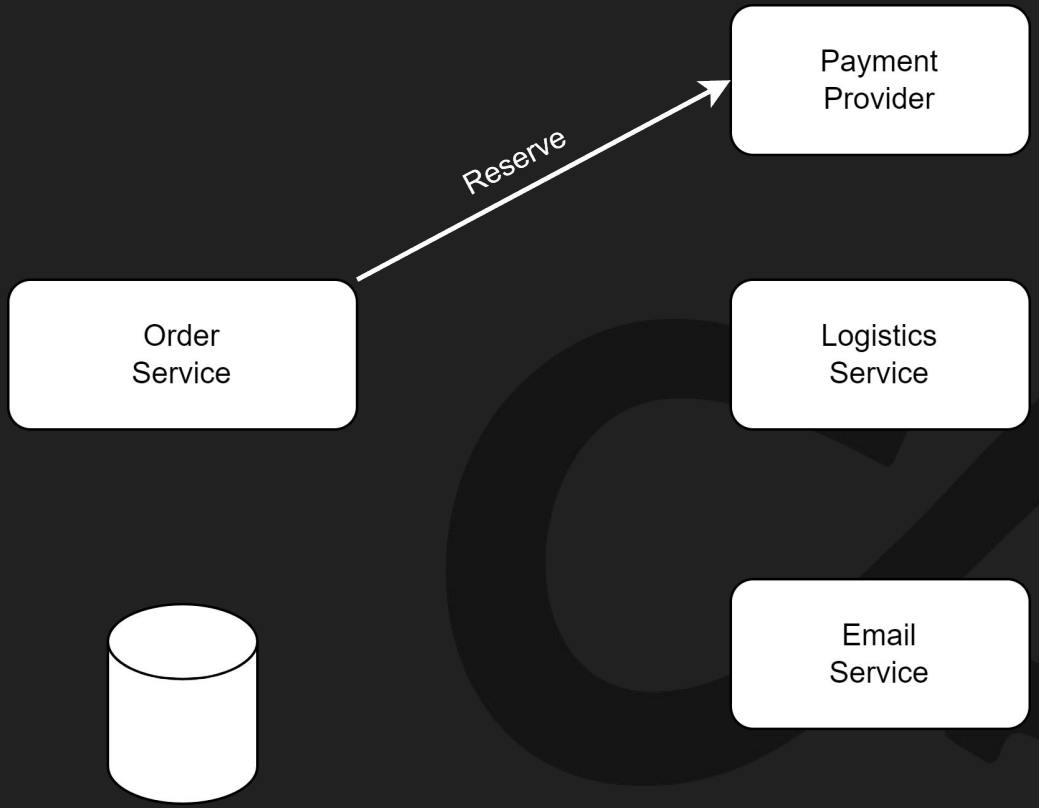
Order Processing Example - Restartable



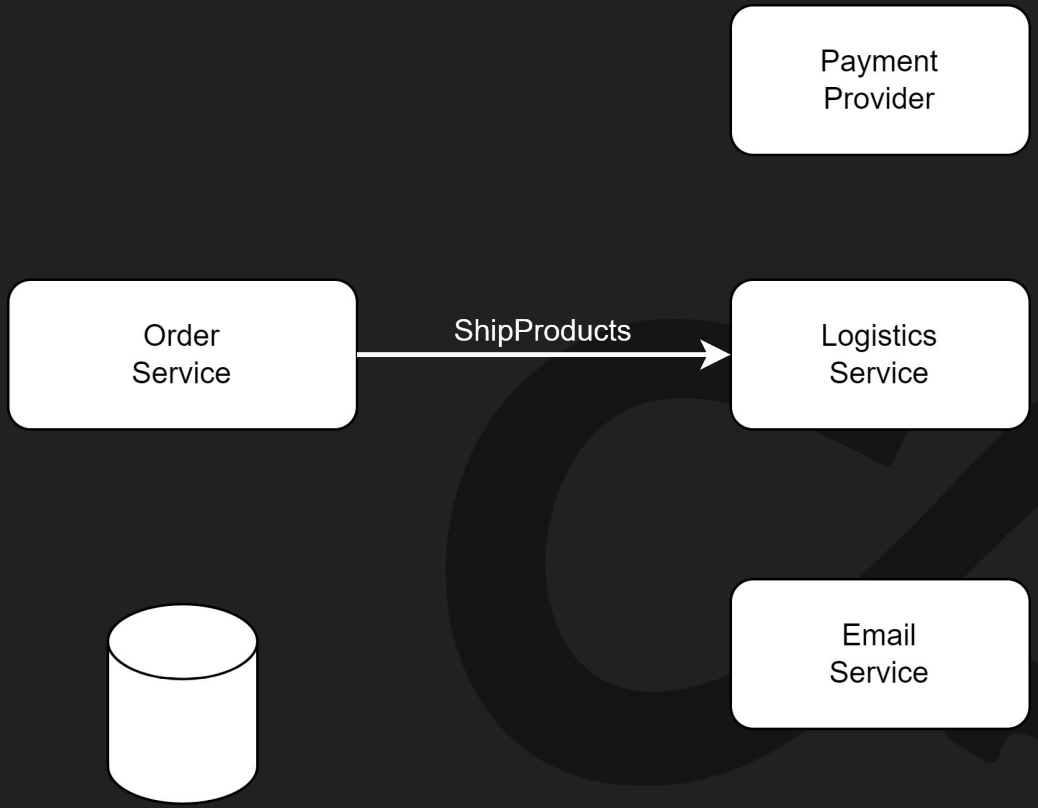
Order Processing Example - Restartable



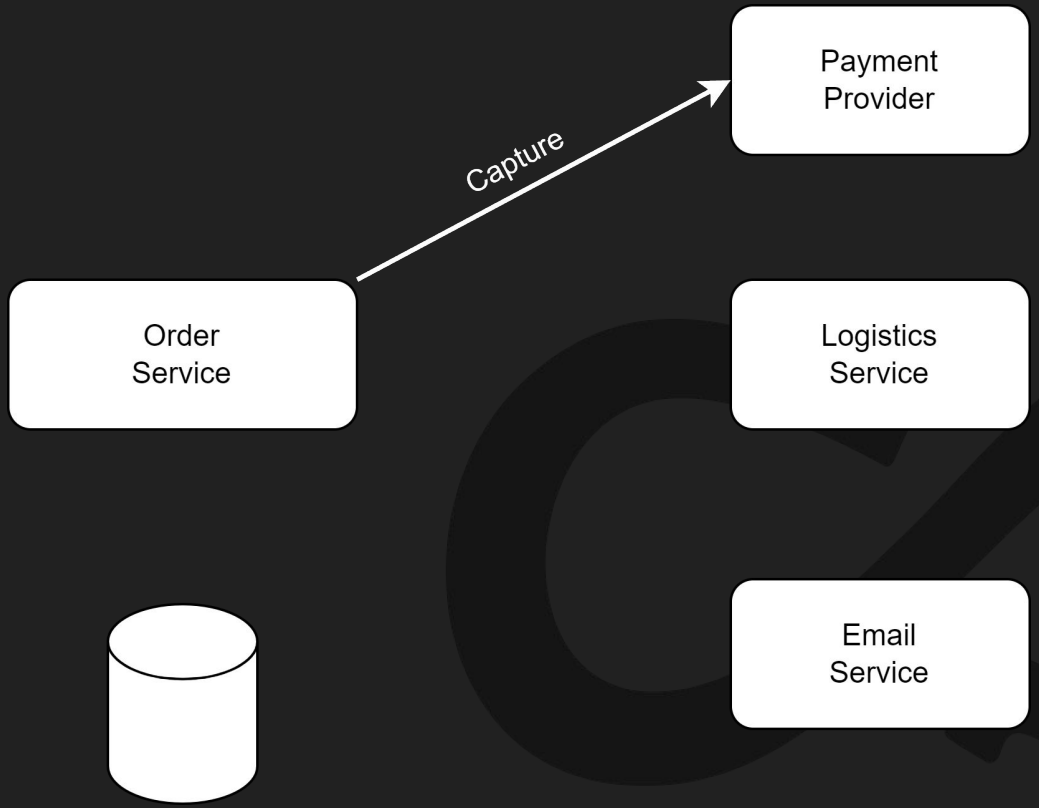
Order Processing Example - Restartable



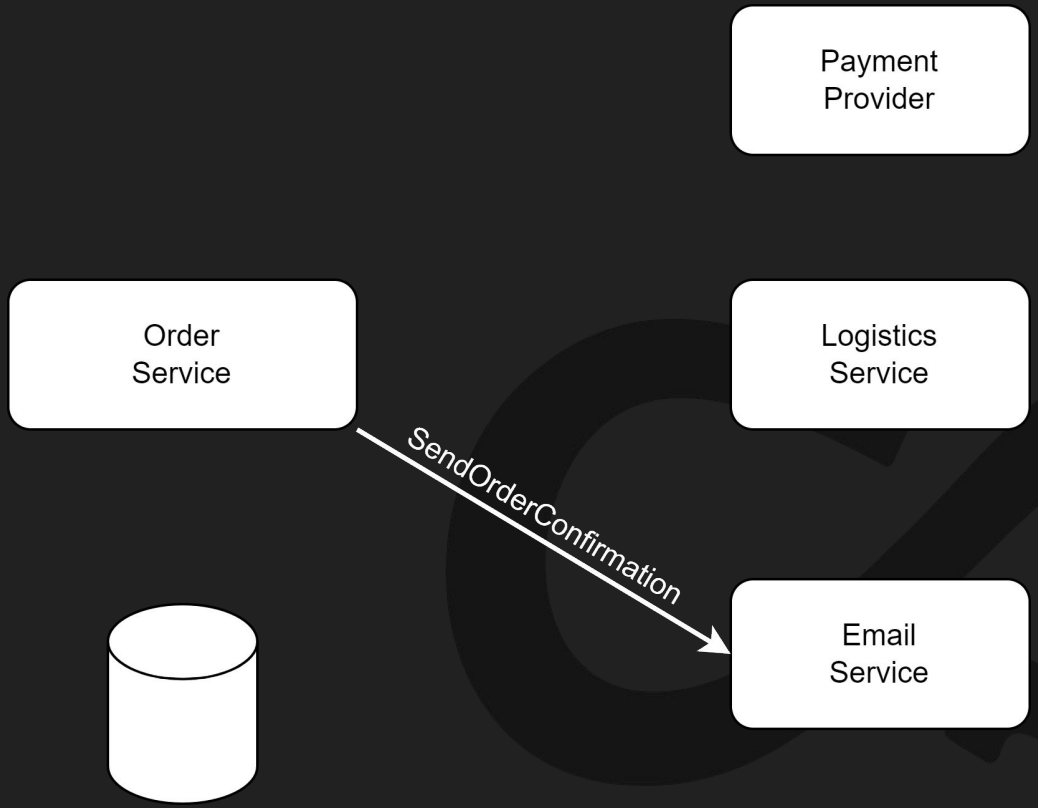
Order Processing Example - Restartable



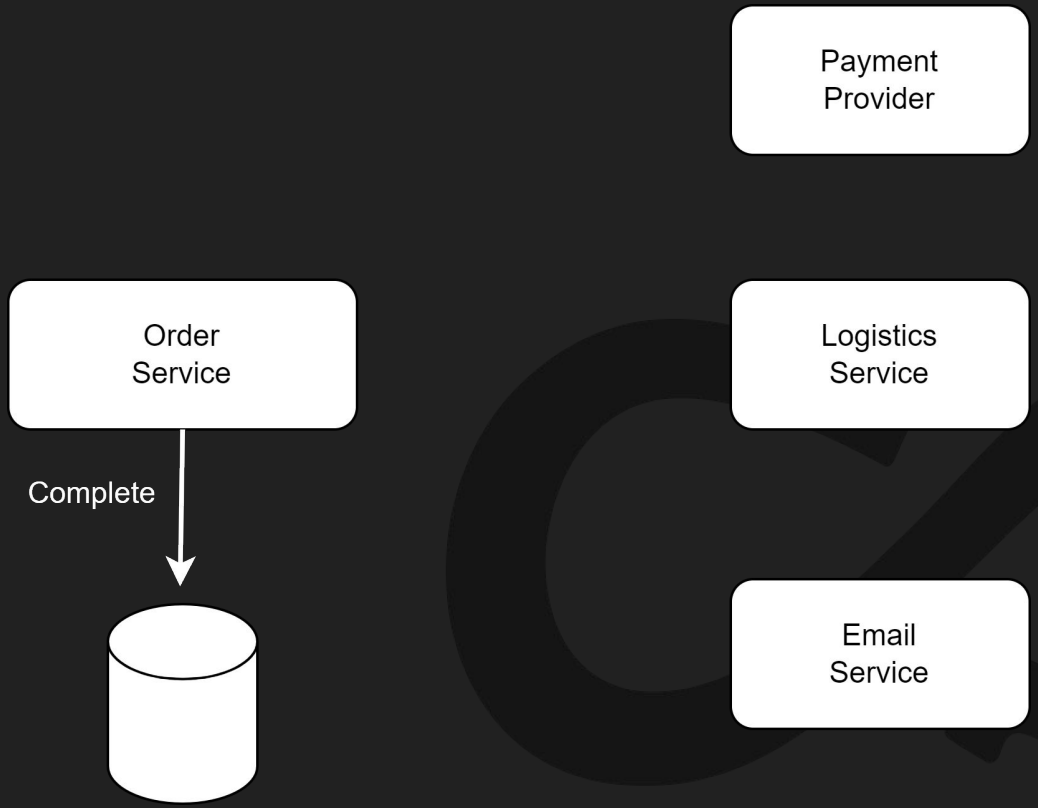
Order Processing Example - Restartable



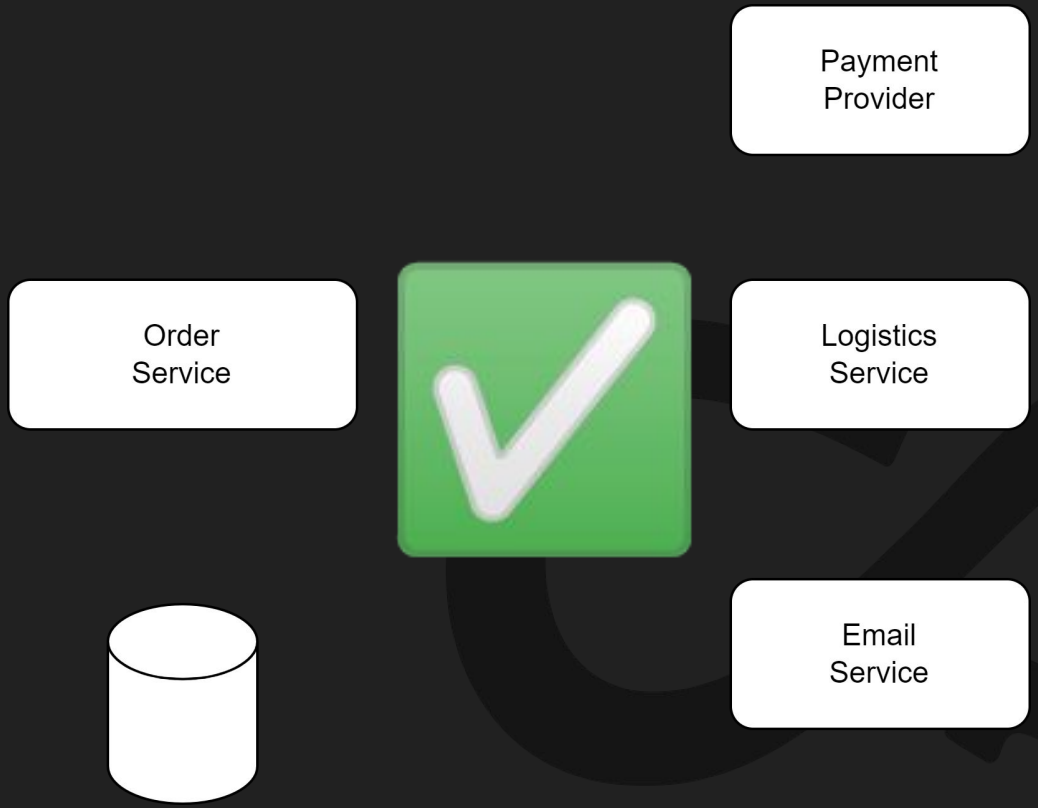
Order Processing Example - Restartable



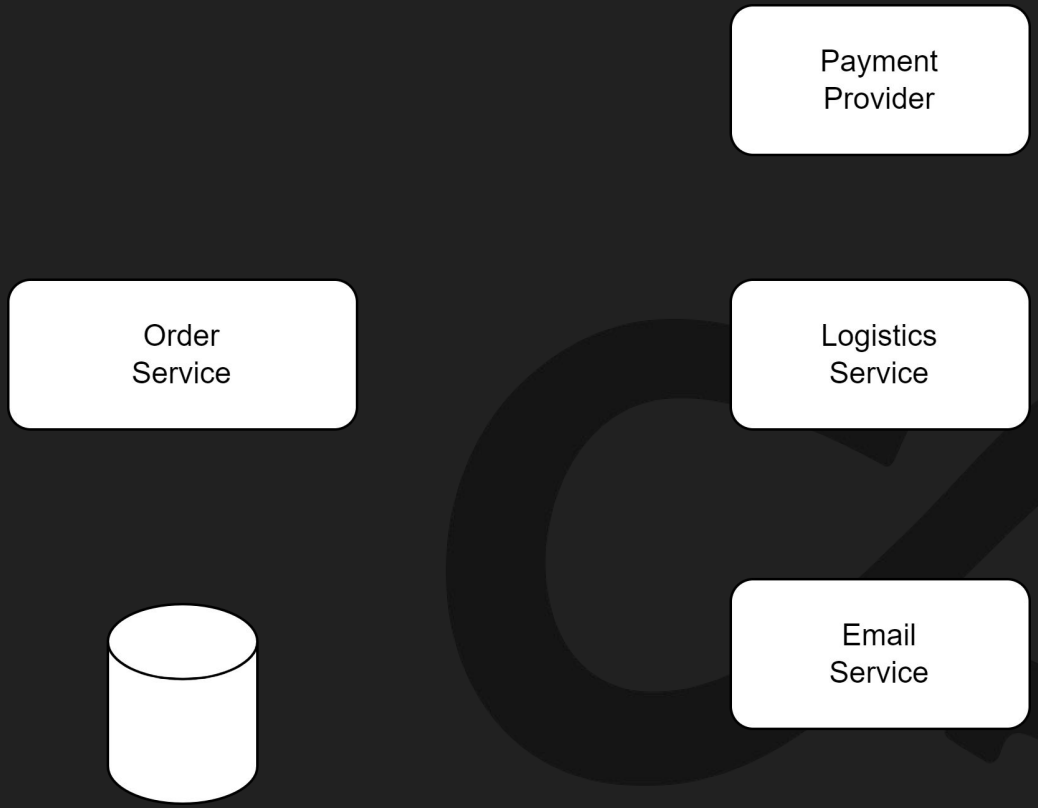
Order Processing Example - Restartable



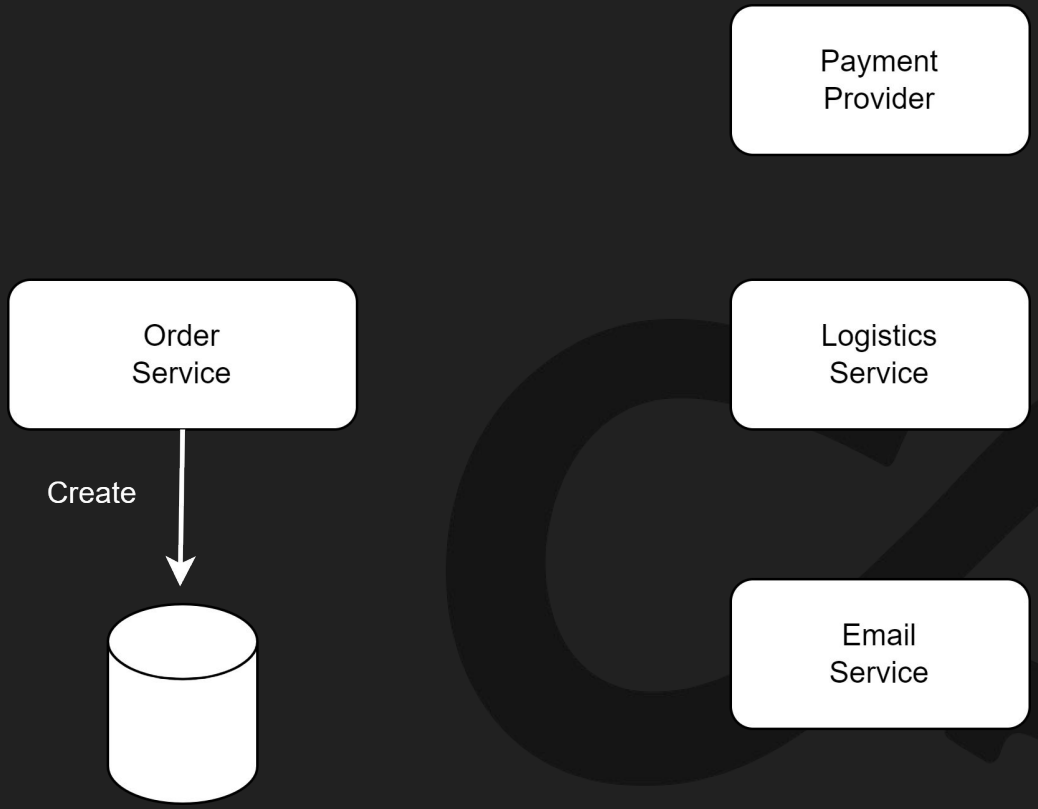
Order Processing Example - Restartable



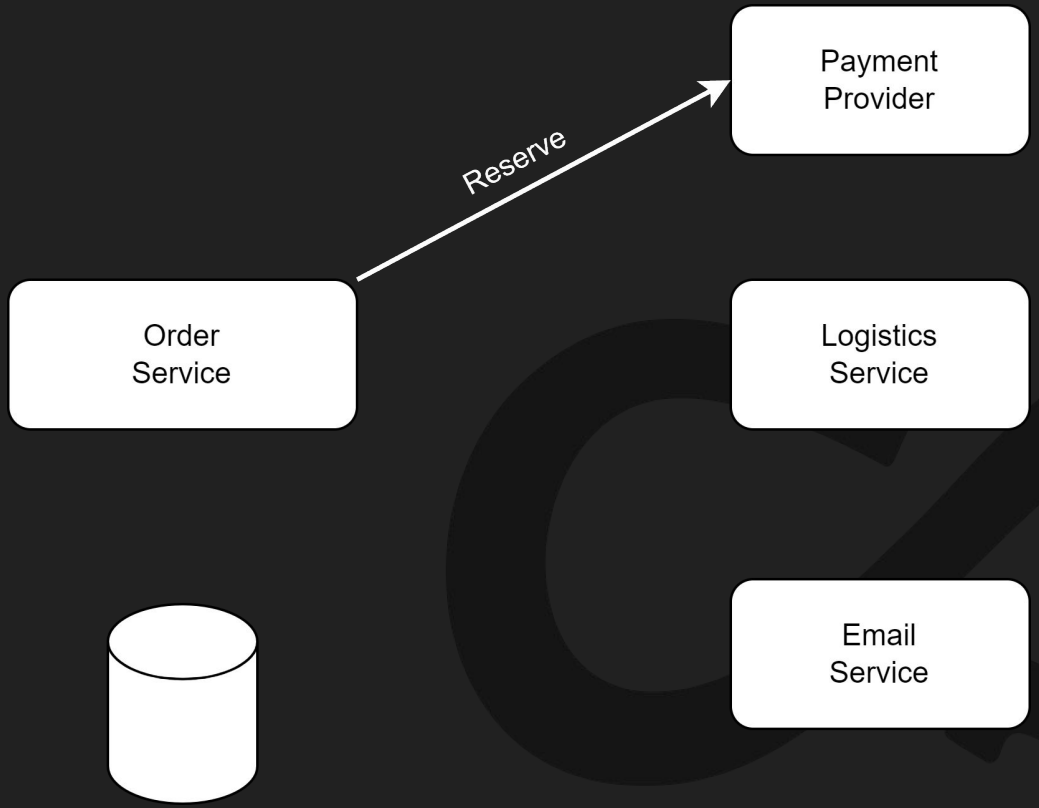
Order Processing Example - Restartable with 💀



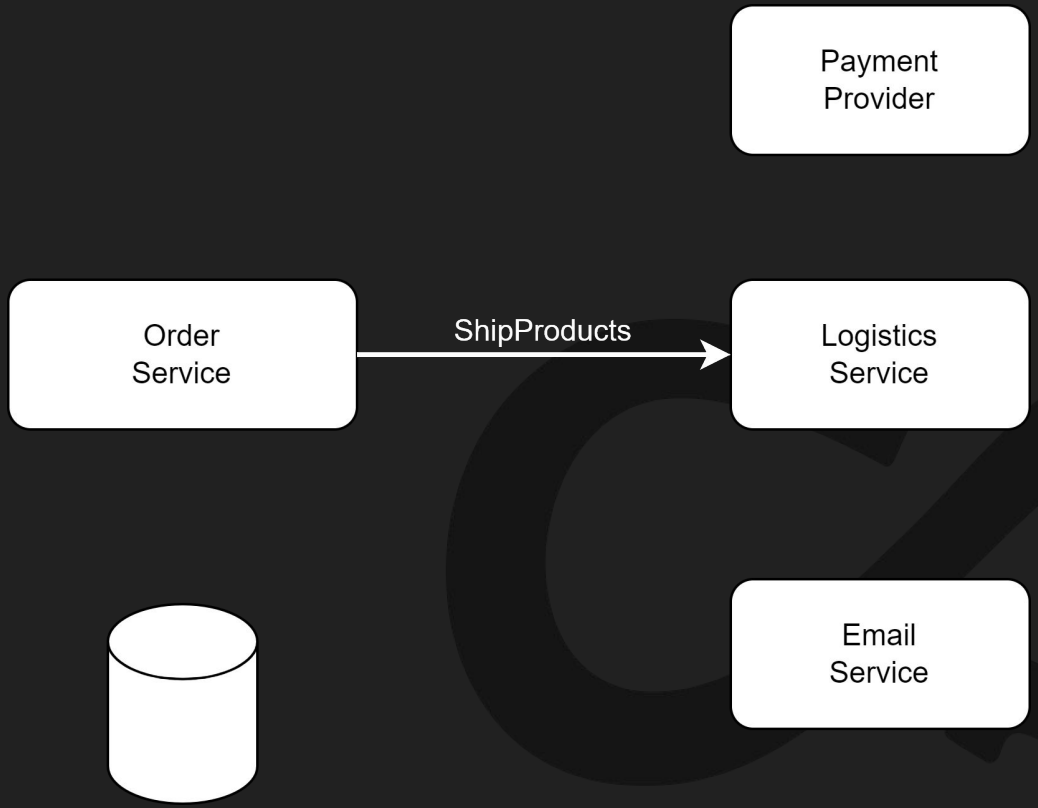
Order Processing Example - Restartable with 💀



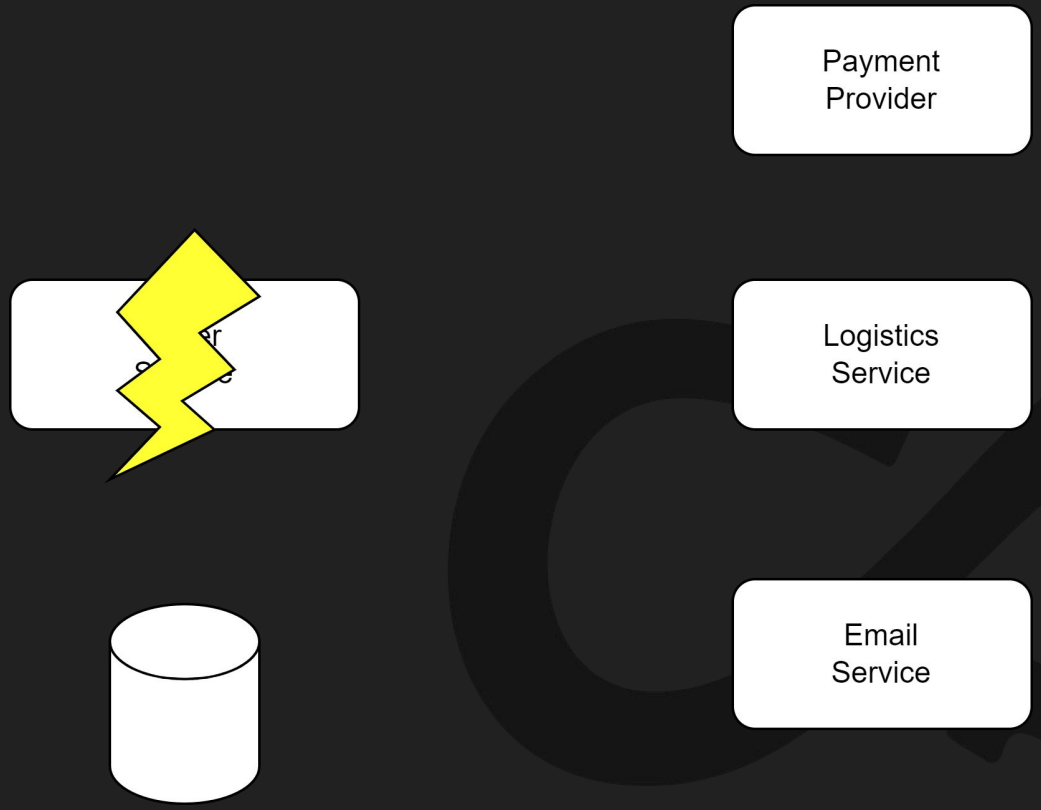
Order Processing Example - Restartable with 💀



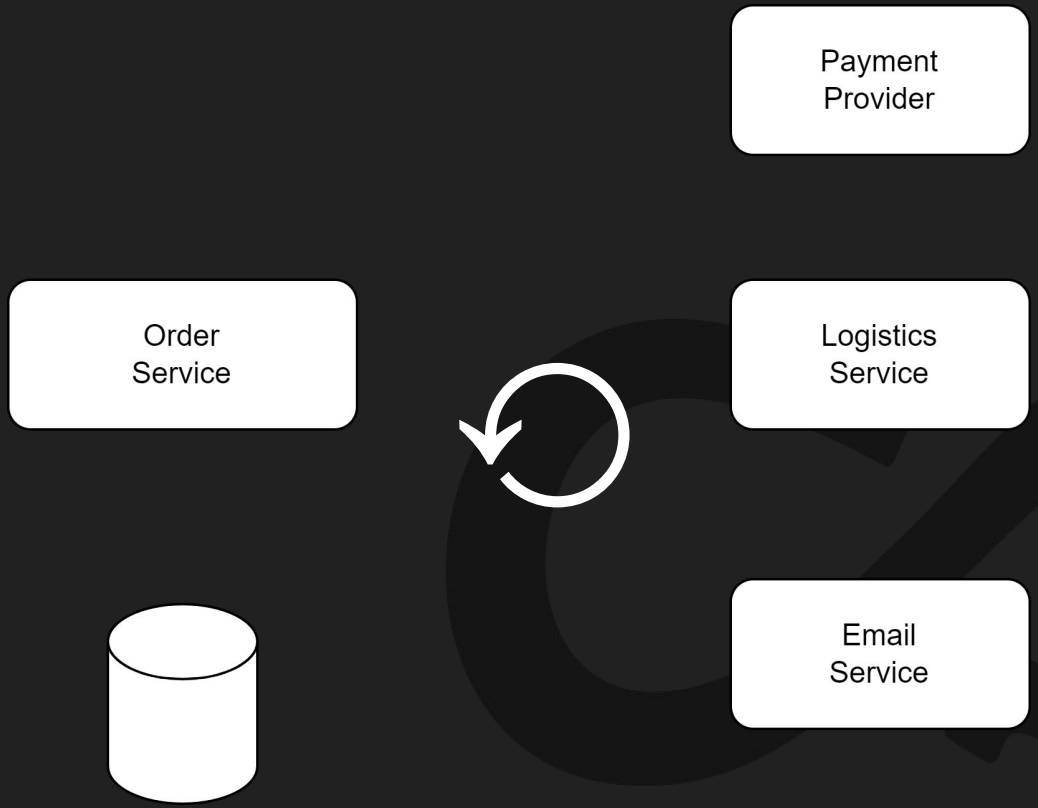
Order Processing Example - Restartable with 💀



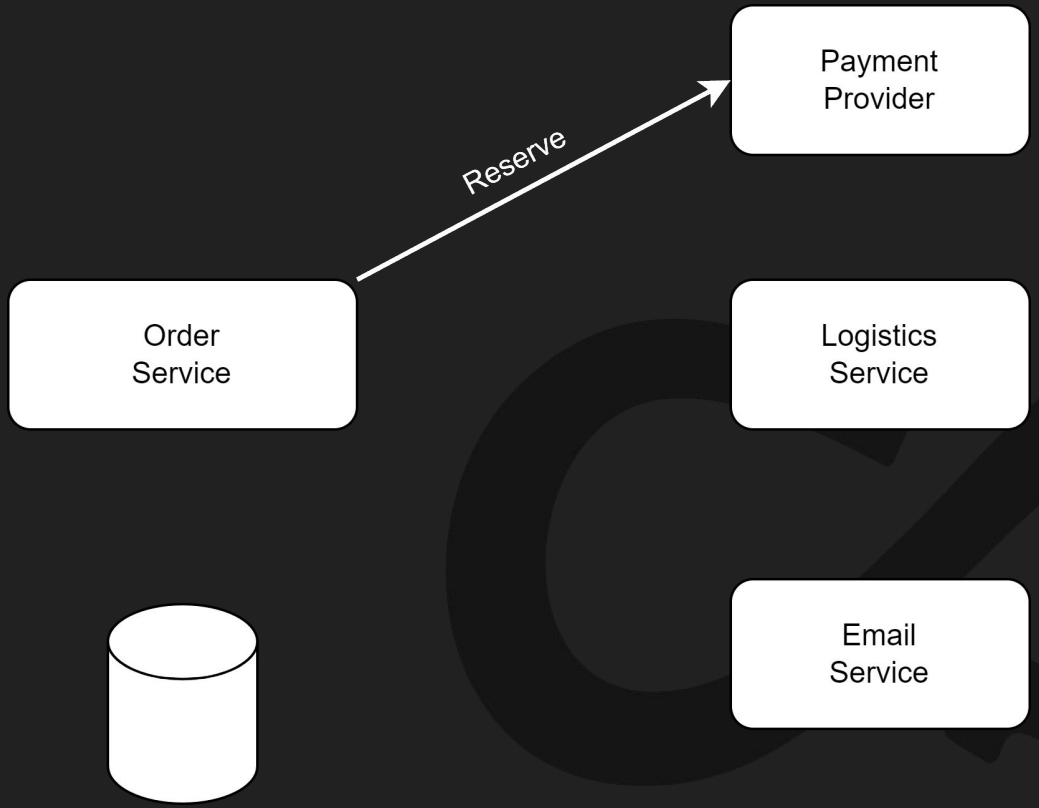
Order Processing Example - Restartable with 💀



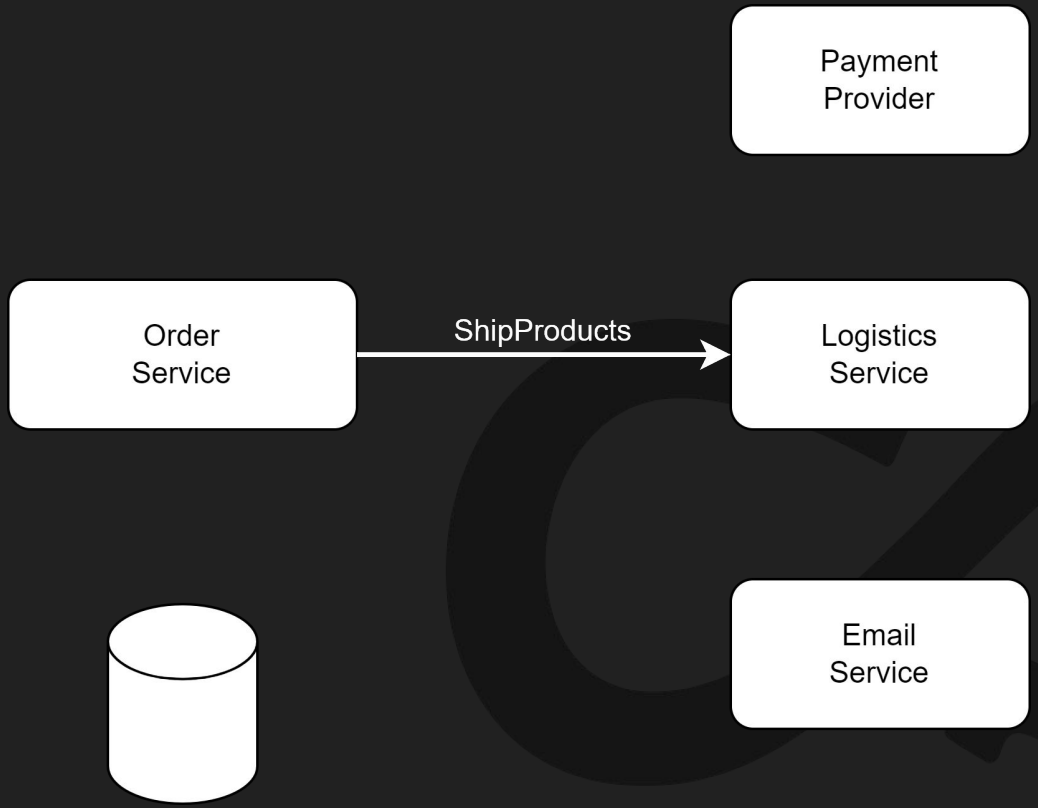
Order Processing Example - Restartable with 💀



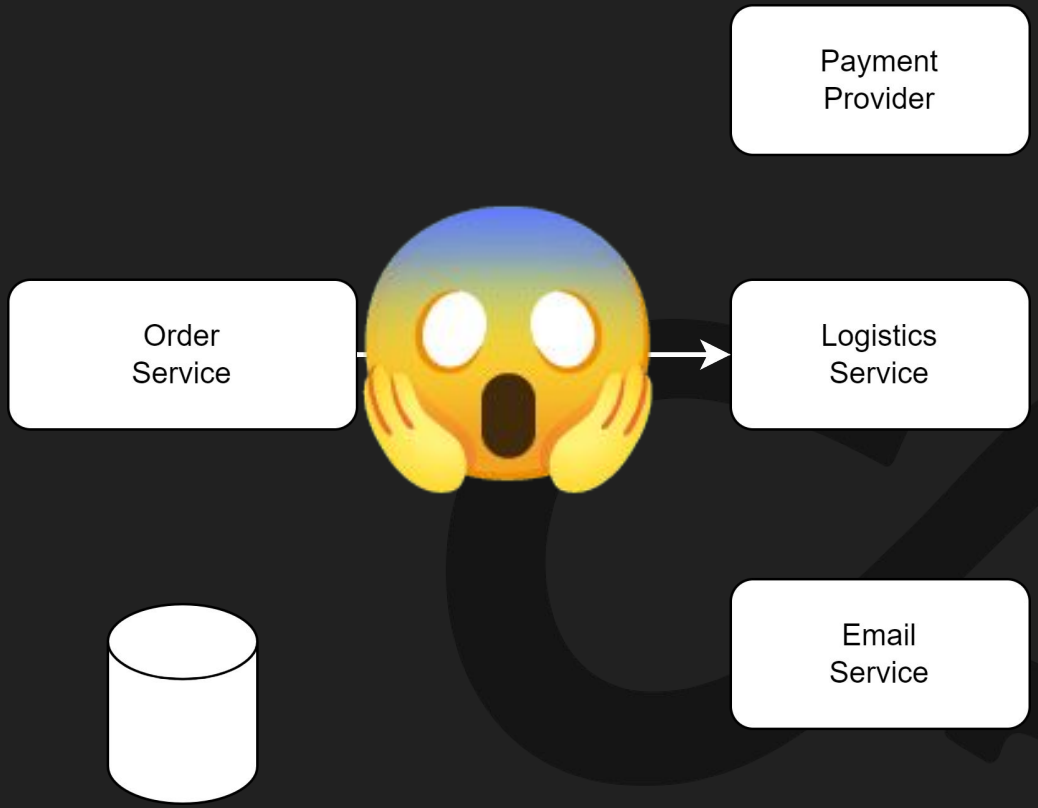
Order Processing Example - Restartable with 💀



Order Processing Example - Restartable with 💀



Order Processing Example - Restartable with 💀



Resilient Functions - **Introduction**

Any function or method can be registered with the framework.

```
var store = new PostgreSqlFunctionStore (connectionString);

var rFunctions = new RFunctions(store);

rFunctions

    .RegisterMethod<OrderProcessor>()

    .RegisterAction<Order>(

        functionTypeId : nameof(OrderProcessor),

        inner : orderProcessor => orderProcessor.ProcessOrder

    );
```

<u>This:</u>

- ensures the invocation will complete despite crashes and restarts
- simplify handling of re-invocations
- allows evolving the implementation (versioning) while remaining backwards compatible

Resilient Functions - **Concepts**

A Resilient Function is identified by function type and function instance id:

```
var rAction = rFunctions
    .RegisterMethod<OrderProcessor>()
    .RegisterAction<Order>(
        functionTypeId: nameof(OrderProcessor),
        inner: op => op.ProcessOrder
    );
await rAction.Invoke(functionInstanceId: order.OrderId, order);
```

All functions with same function type point to the same method/function.

Resilient Functions - **Concepts**

A Resilient Function invocation may also return a value:

```
var rFunc = rFunctions.RegisterFunc(

   functionTypeId: "StickyRandomNumber",

   int (Interval interval) => Random.Shared.Next(interval.From, interval.To)

).Invoke;


var dieOutcome = await rFunc("DiceThrow#1", new Interval(1, 6));
```

Resilient Functions - **Concepts**

It is safe to invoke the same Resilient Function across replicas multiple times:

```
await rAction.Invoke(functionInstanceId: order.OrderId, order);

await rAction.Invoke(functionInstanceId: order.OrderId, order);
```

The second invocation will:

1. Wait for the first invocation to complete
2. Return the same result without invoking the underlying action/func

Resilient Functions - **Concepts**

Signature of a resilient function: *rf*(param, scrapbook, context) → result

- **Parameter**
  Information required for the business process to perform its task at hand.
  I.e. Order information (order number, products ordered…)

- **Scrapbook** (optional)
  Invocation progress information useful after a restart
  I.e. has an endpoint been invoked before or a reply received

- **Context** (optional)
  Invocation metadata and function's private event source

- **Result** (may also be void or null)
  The returned value of the function

Resilient Functions - **Signature Examples**

RAction:

```
public Task ProcessOrder(Order order)

public Task ProcessOrder(Order order, Scrapbook scrapbook)

public Task ProcessOrder(Order order, Scrapbook scrapbook, Context context)
```

RFunc:

```
public Task<Guid> ProcessOrder(Order order, Scrapbook scrapbook, Context context)

public Task<Guid> ProcessOrder(Order order, Scrapbook scrapbook)

public Task<Guid> ProcessOrder(Order order)
```

Resilient Functions - **Scrapbook**

- Allows track-keeping of the **invocation's progress** and holding other useful information for when/if the function is **retried**

- A scrapbook is a **user-defined type** which must be subtype of RScrapbook and have empty constructor

- Contains a Save-method allowing its current state to be persisted.
  I.e.
  ```
  await scrapbook.Save()
  ```

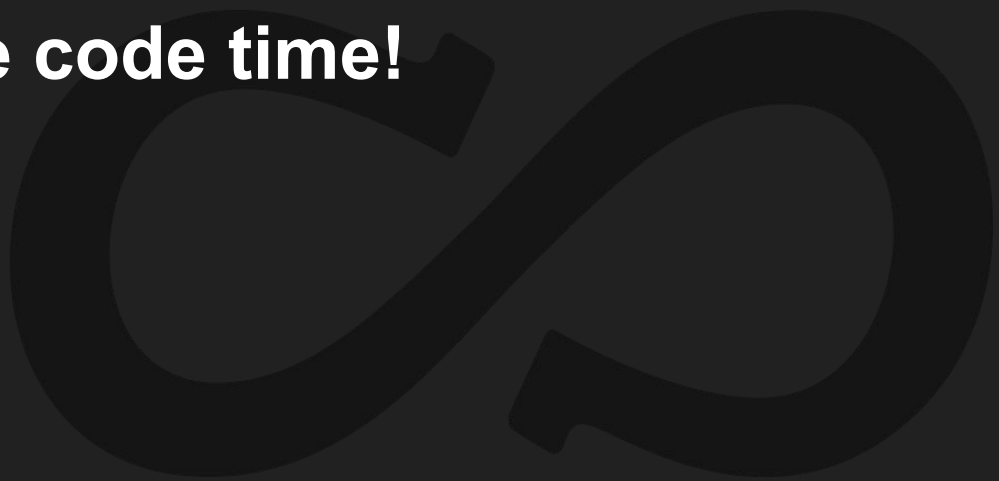
Resilient Functions - **Scrapbook Example**

```
public class Scrapbook : RScrapbook

  public Guid? BankTransactionId { get; set; }

  public bool EmailSent { get; set; }
```

Resilient Functions - **Order Processing**

Web-API Sample

Source code time!



Resilient Function Correctness - Rule of thumb

When reasoning about the **code's correctness** of a resilient function it helps:

1. To repeatedly "run" the code of the method mentally from top to bottom
2. While doing so imagine that the execution may stop at any point and restart from the top of the method again.

If the code is correct in all such different "runs" then the code is robust.

Resilient Function Correctness - Rule of thumb

```
public async Task ProcessOrder(Order order)

    var transactionId = await _paymentProviderClient.Reserve(order.CustomerId, order.TotalPrice);

    await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, order.ProductIds);
```

Resilient Functions - **Order Processing Challenge #1**

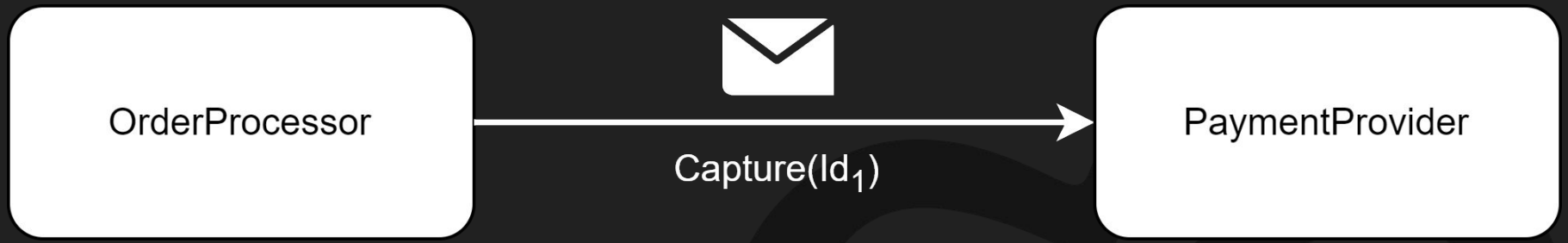
Payment Provider API has been changed
Now:

- Accepts client provided **‘TransactionID’**

- Thus, if same transaction id is used in multiple requests the underlying operation is **only performed once**

Order Processing - Client Provided Transaction Id

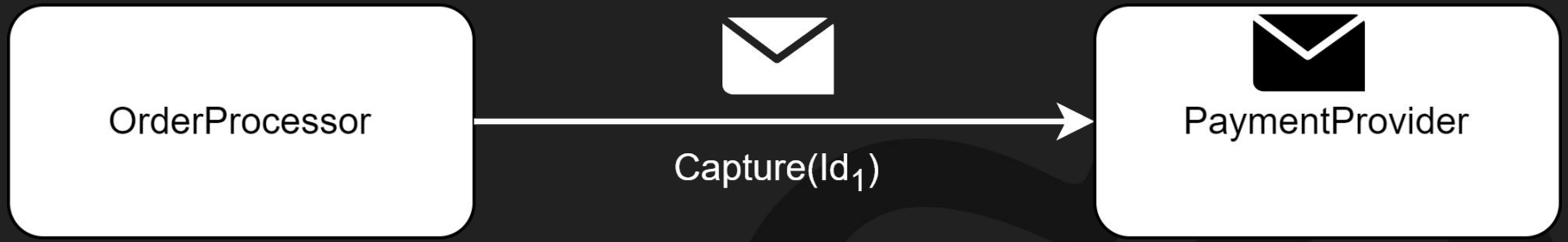
Order Processing - Client Provided Transaction Id



Order Processing - Client Provided Transaction Id

Order Processing - Client Provided Transaction Id

Order Processing - Client Provided Transaction Id



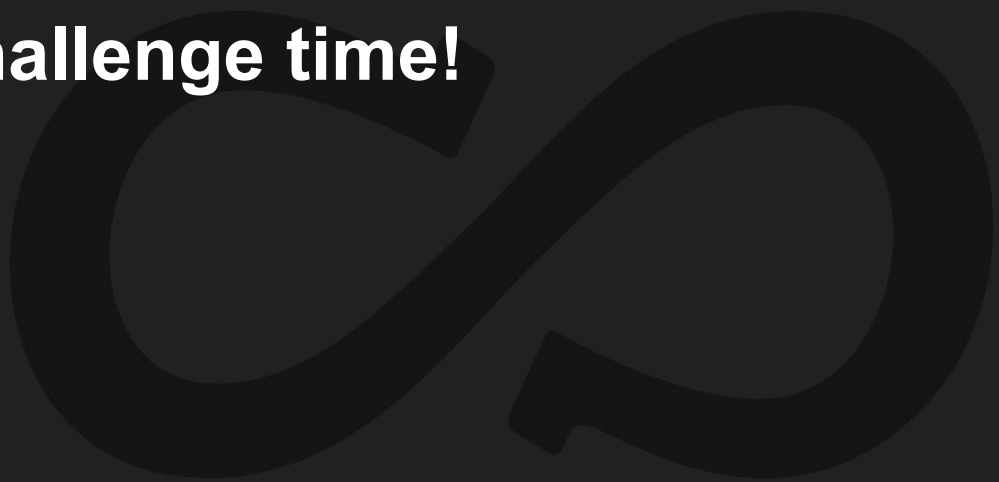
Order Processing - Client Provided Transaction Id

Order Processing - Client Provided Transaction Id

Resilient Functions - **Order Processing**

Payment Provider API

Code Challenge time!

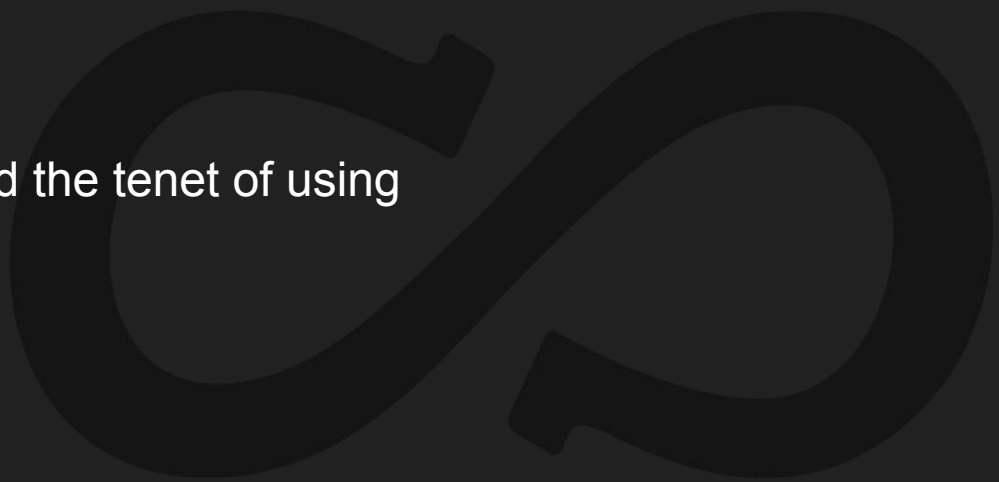


Handling **Failures**

Failure Handling - Human Intervention

Writing code addressing **all failure scenarios**
in a distributed setting
is often **infeasible**

As such, the framework is built around the tenet of using
human intervention.



Failure Handling - Failing an invocation

A function invocation **fails** when:

- the invocation throws an **unhandled exception**

A failed function invocation is *not* retried automatically by the framework.

Failure Handling - When to **fail** an invocation?

It may make sense to fail a function, when:

- an external API is consistently **unreachable**
- a **runtime exception** occurs during the invocation
  I.e. when deserializing a response from external service
- an **at-most-once API** call has been started but no reply has yet been received when the invocation was restarted
- other suggestions?

Failure Handling - When to **fail** an invocation?

It may make sense to fail a function, when:

- an external API is consistently **unreachable**
- a **runtime exception** occurs during the invocation
  I.e. when deserializing a response from external service
- other suggestions?

Failure Handling - Re-invoking a failed function

Re-invocation:

In order to invoke a failed function **again,** it must be explicitly re-invoked.
This can be accomplished by using the function's associated **ControlPanel**.
I.e.

```
var rFunc = rFunctions.RegisterFunc(functionTypeId, …);

var controlPanel = await rFunc.ControlPanel.For(functionInstanceId);

await controlPanel.ReInvoke();
```

Failure Handling - Changing **state** of a failed function

Before re-invoking a method it is sometimes beneficial to change the resilient function's state.

This can also be accomplished using the ControlPanel:

```
var controlPanel = await rAction.ControlPanel.For("MK-4321");


controlPanel.Scrapbook.LogisticsServiceCallStatus  = WorkStatus.NotStarted;

await controlPanel.ReInvoke();
```

Failure Handling - Postponing an invocation

A function can be **postponed**, if it is unable to complete its execution.

This can be accomplished by throwing an: `PostponeInvocationException`

Question:

Why can this be beneficial over simply using: `await Task.Delay(delay)`

Resilient Functions - **Order Processing Challenge #2**

Unlike the PaymentProvider API the **Logistics Service API**:

- Performs its ‘side-effect’ **everytime** it is called
- It does *not* accept an ID

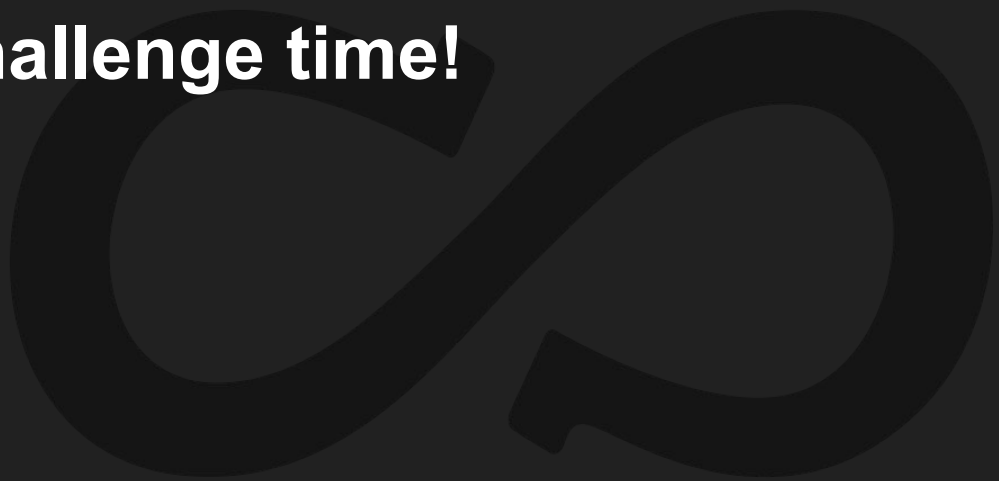
Thus:

- We must call it at-most-once
- **Stop** the invocation and investigate if we are about to call it **multiple times!**

Resilient Functions - **Order Processing**

Logistics Service API

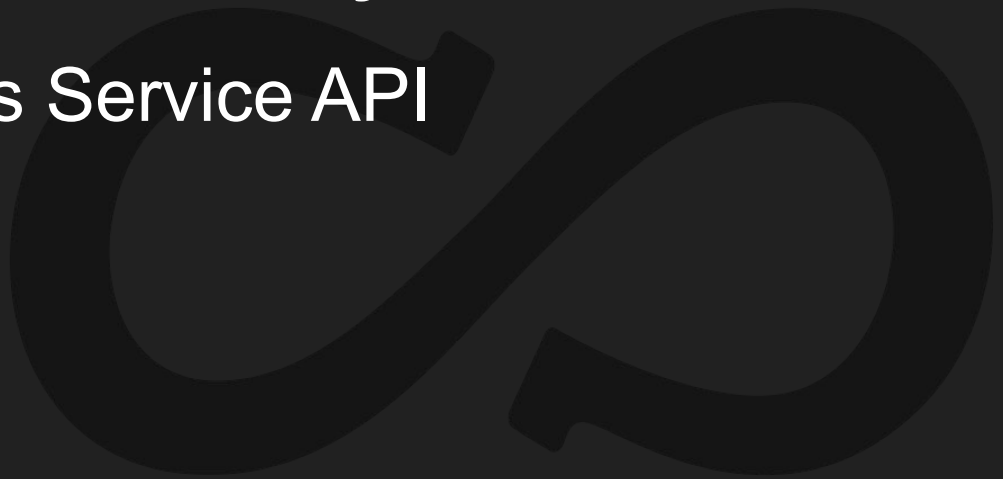
Code Challenge time!



Resilient Functions

Limits of Distributed Systems

Logistics Service API

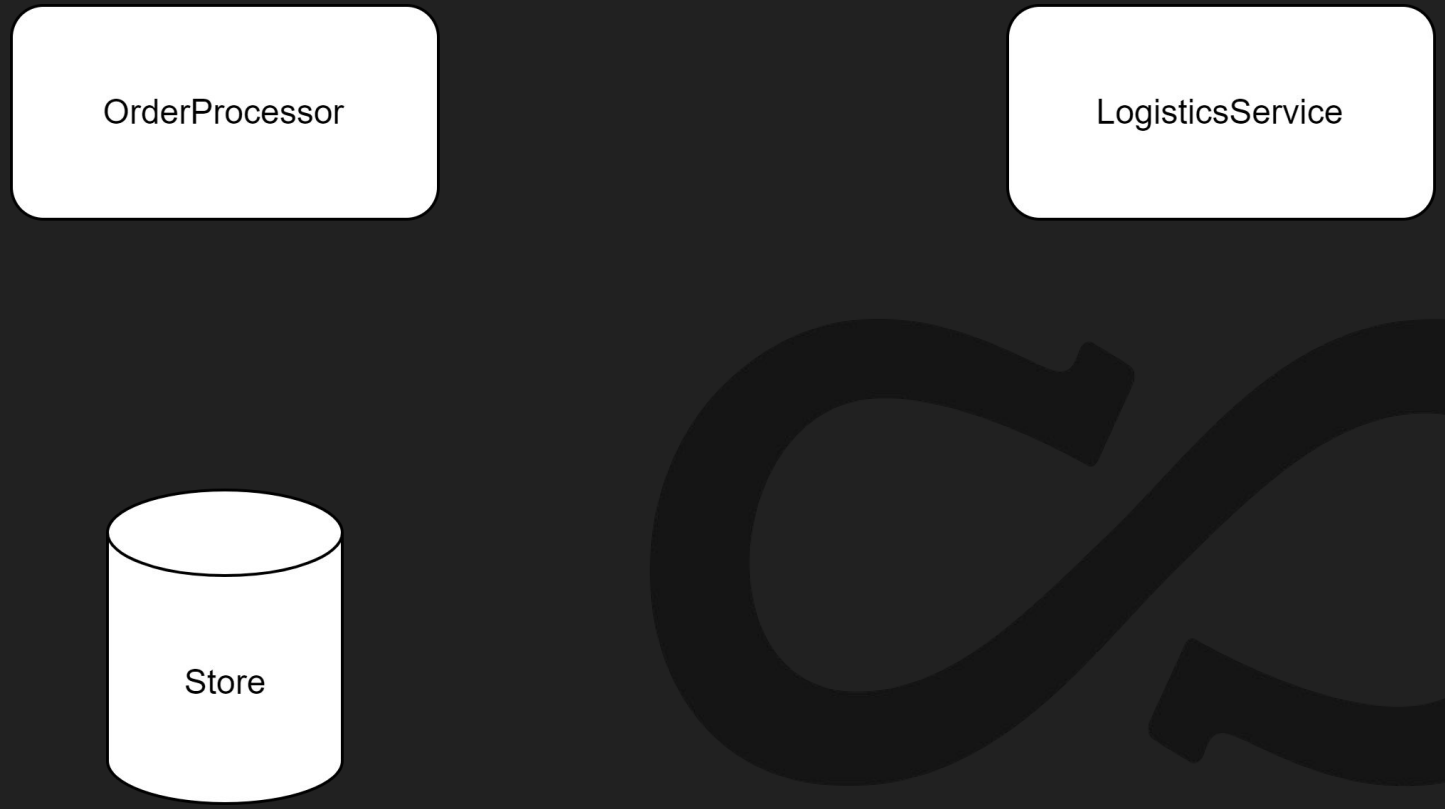


Resilient Functions

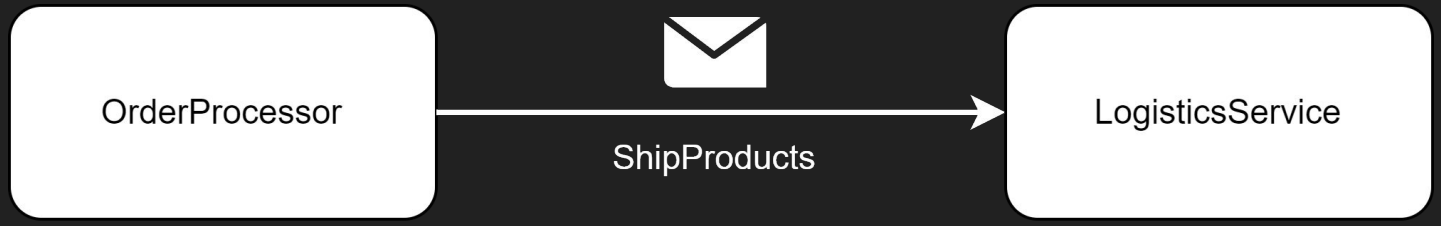
Limits of Distributed Systems

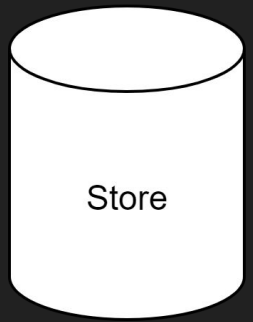
First Approach

Order Processing - **Call then Save** (1)

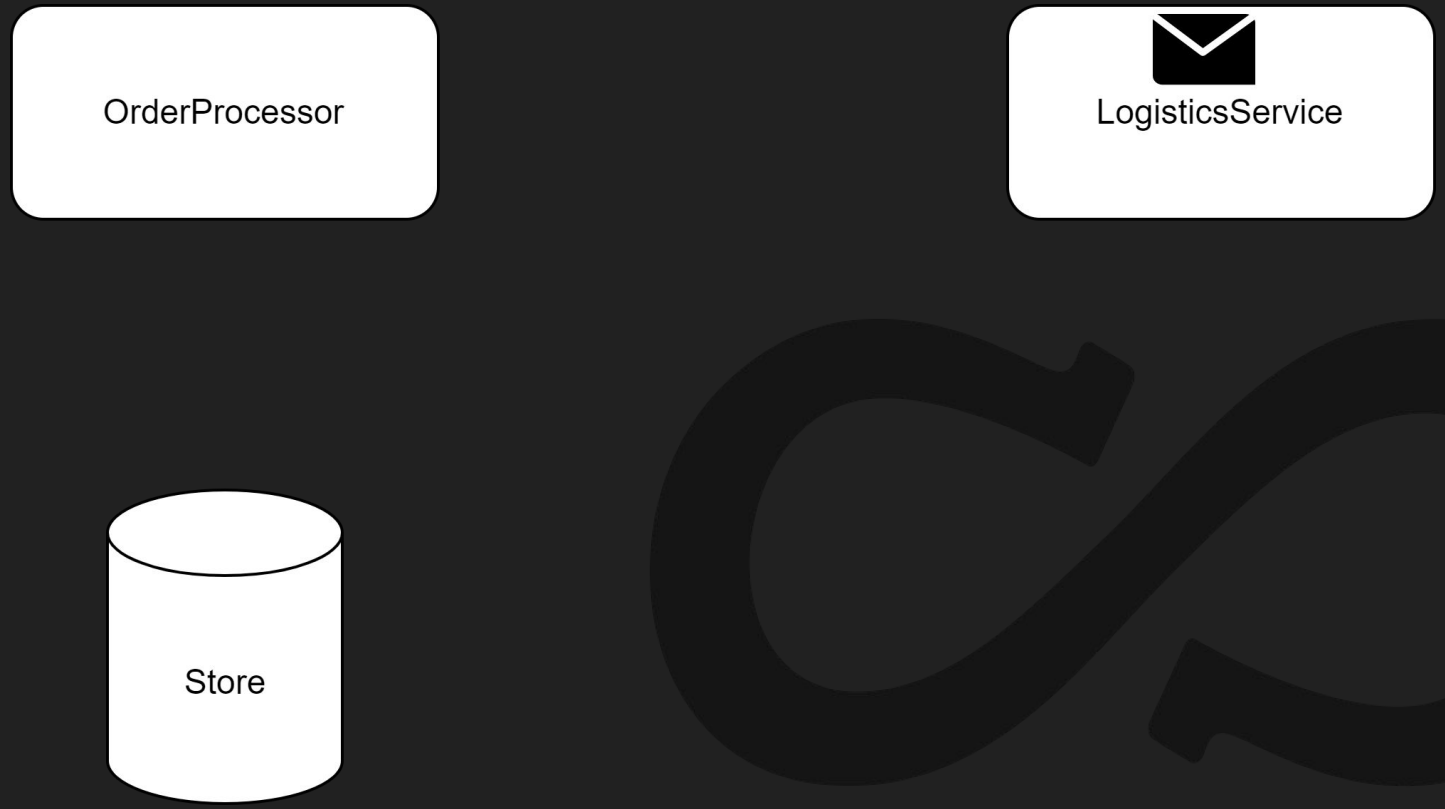


Order Processing - **Call then Save** (1)

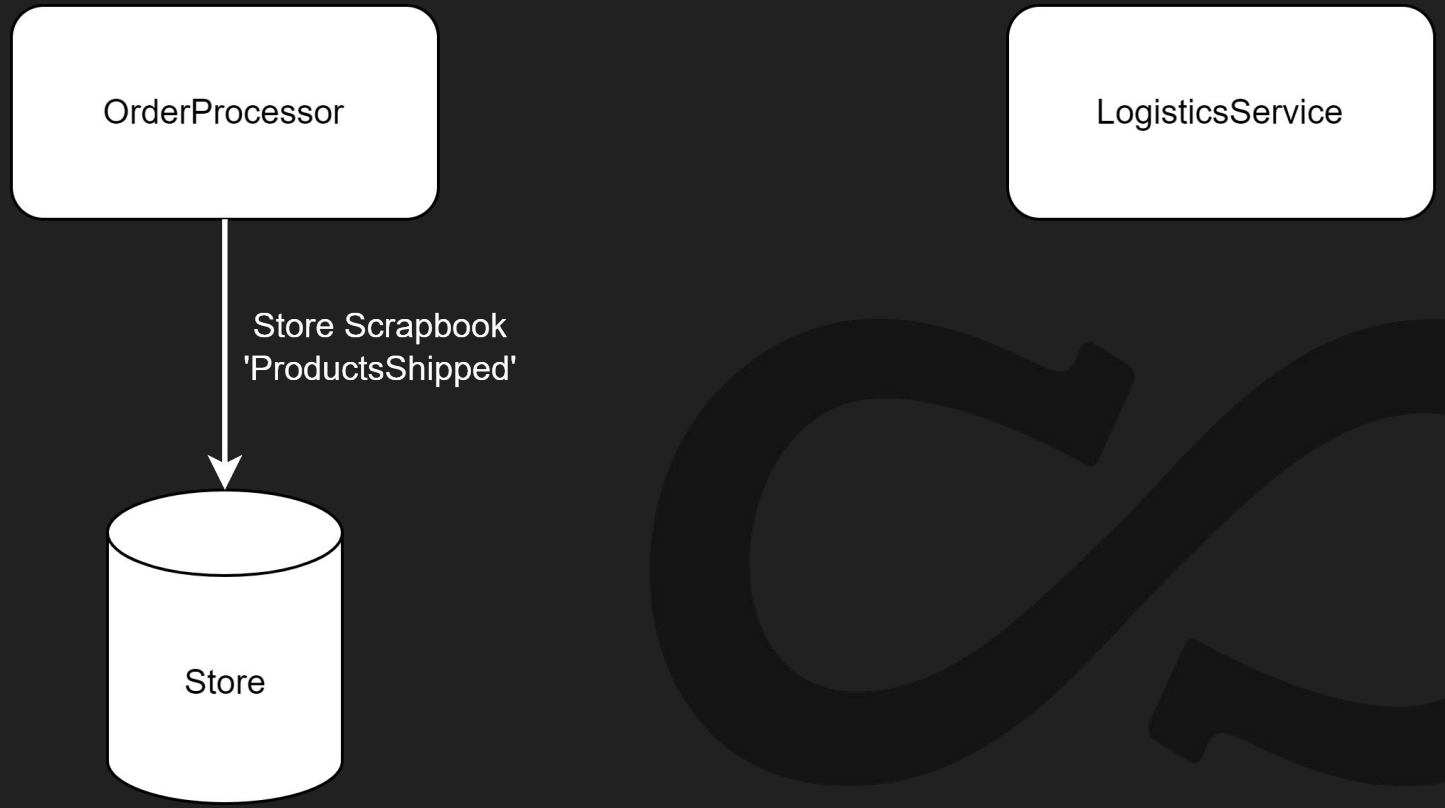




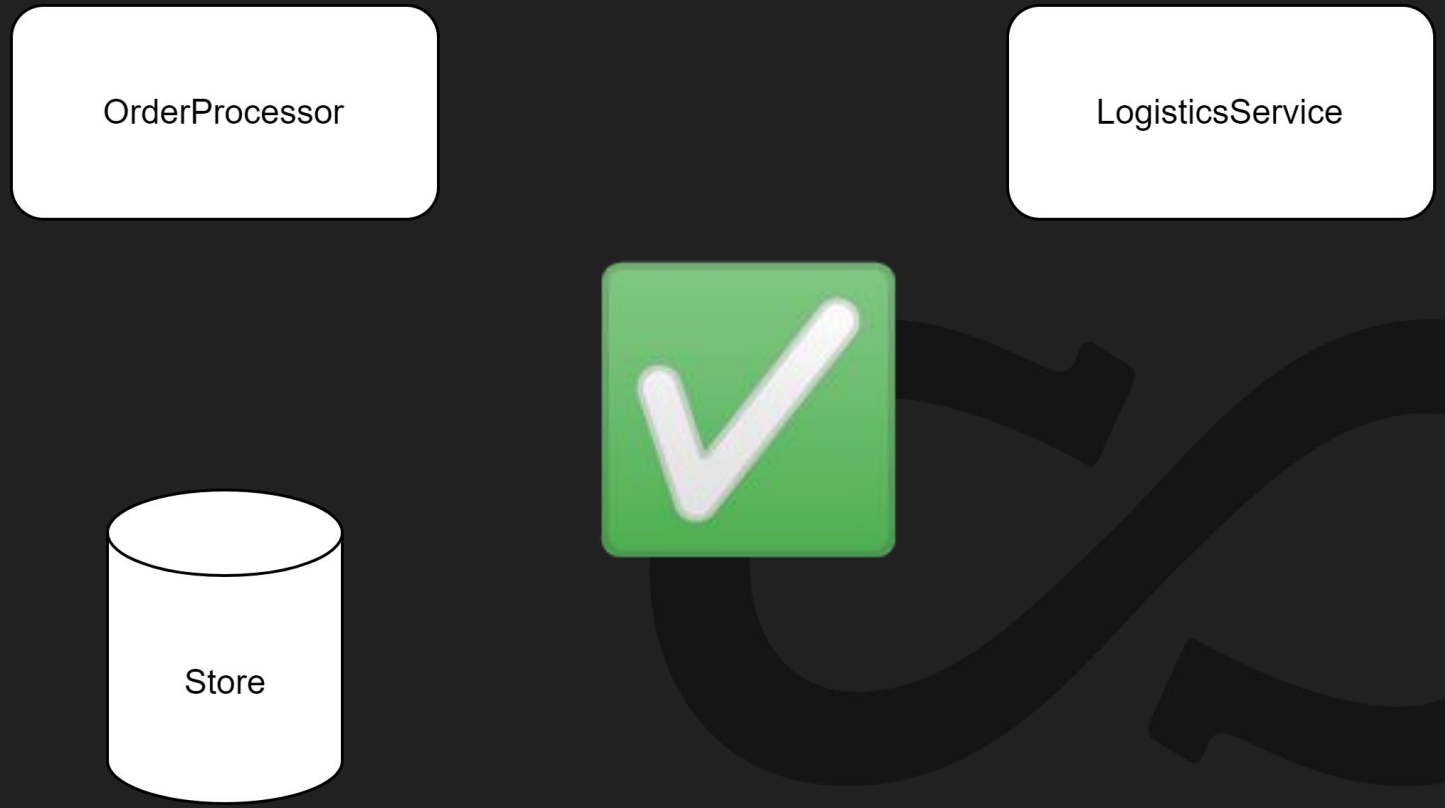
Order Processing - **Call then Save** (1)



Order Processing - **Call then Save** (1)

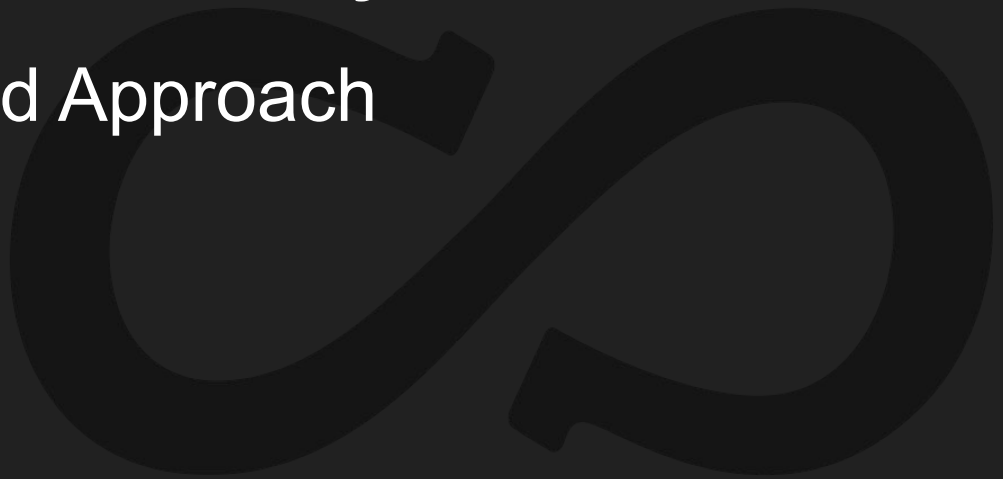


Order Processing - **Call then Save** (1)

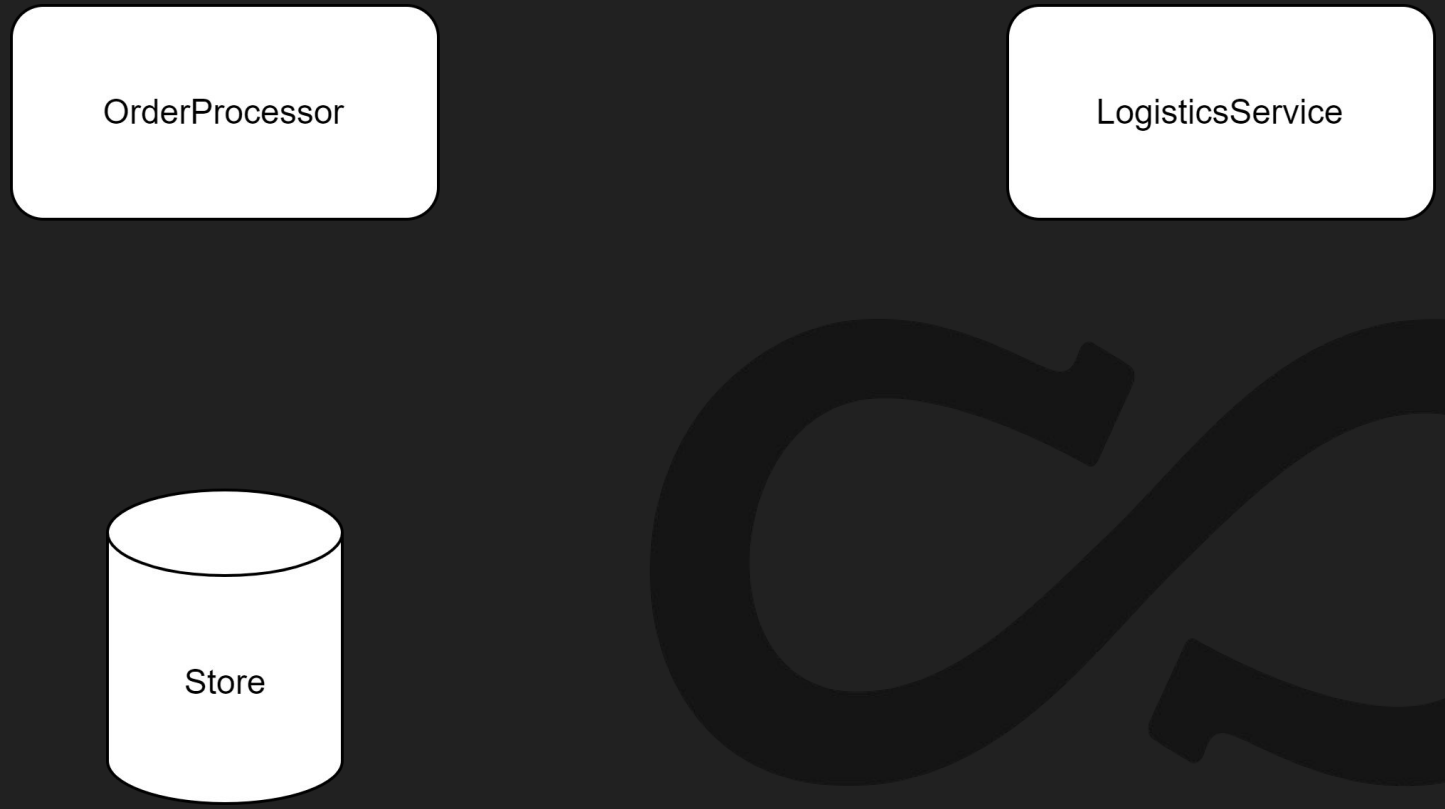


Limits of Distributed Systems

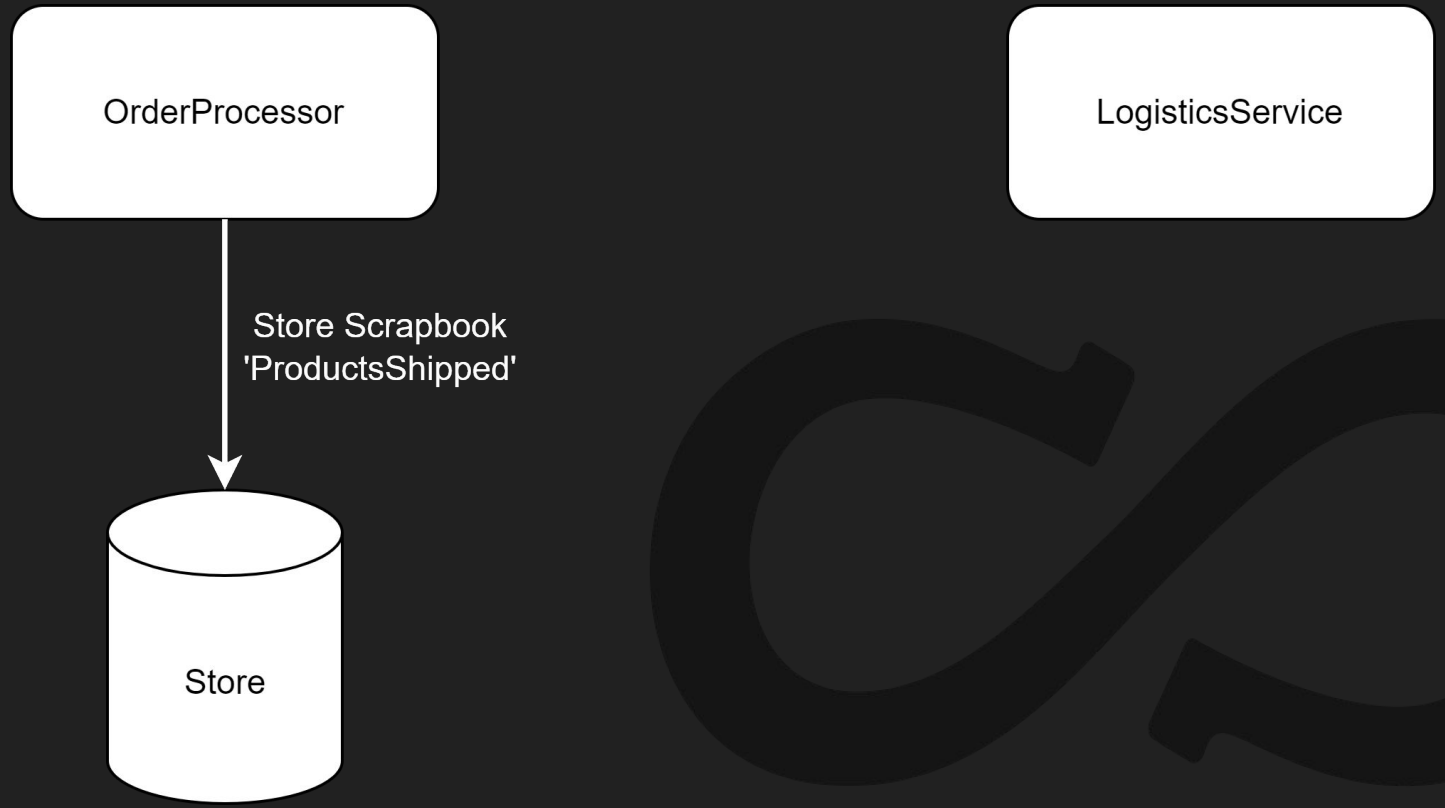
Second Approach



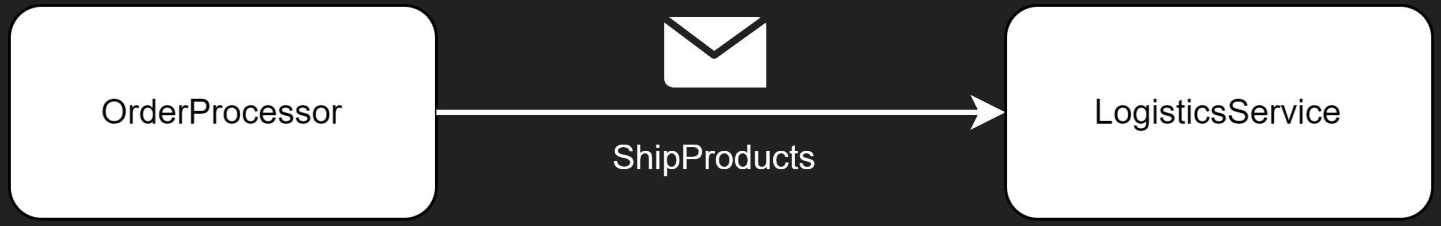
Order Processing - **Save then Call** (2)

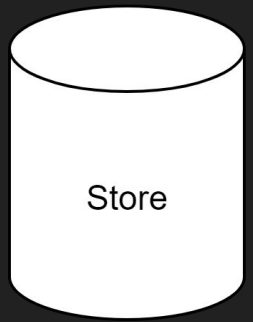


Order Processing - **Save then Call** (2)

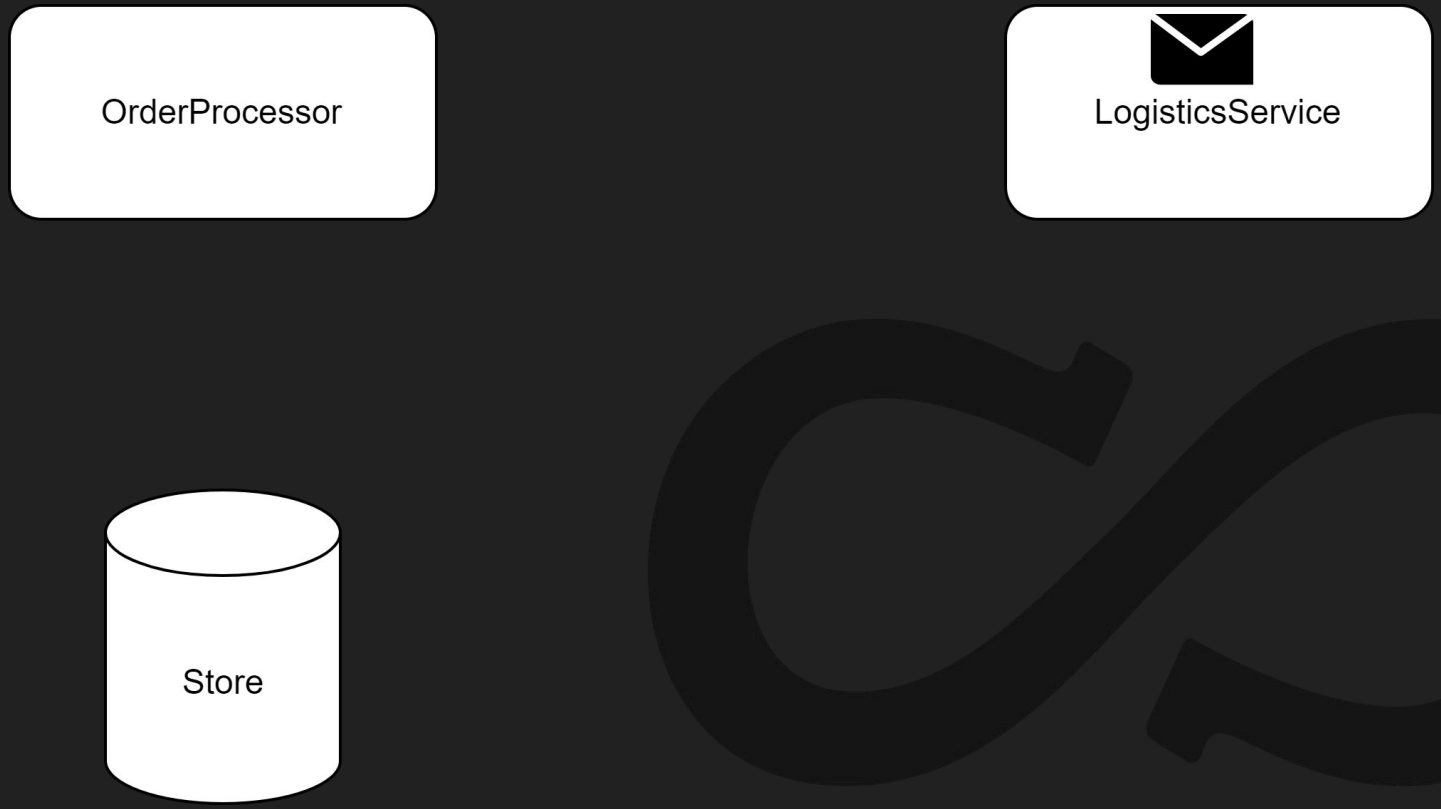


Order Processing - **Save then Call** (2)





Order Processing - **Save then Call** (2)



Order Processing - **Save then Call** (2)

OrderProcessor

LogisticsService

Store

Resilient Functions - **Limits of Distributed Systems**

Which approach is best?

(1)

```
if (ShipProductsWorkStatus == NotStarted)
  await _logisticsClient.ShipProducts(
    order.CustomerId,
    order.ProductIds
  )

  scrapbook.ShipProductsWorkStatus = Started

  await scrapbook.Save()
```

(2)

```
if (ShipProductsWorkStatus == NotStarted)
  scrapbook.ShipProductsWorkStatus = Started

  await scrapbook.Save()


  await _logisticsClient.ShipProducts(
    order.CustomerId,
    order.ProductIds
  )
```

Unit & Integration **Testing**

Resilient Function - Unit Testing

It is simple to test a resilient method without spinning the whole framework up:

1. An inner function is just an ordinary method:

```
public async Task ProcessOrder(Order order, Scrapbook scrapbook)
```

2. A Scrapbook is simple to instantiate or mock.
3. This allows one - *as in the challenge just completed* - to easily cover different scenarios.

Resilient Function - Integration Testing

Performing **integration tests** is also simplified by using an **InMemoryStore**:

```
var store = new InMemoryFunctionStore();

var rFunctions = new RFunctions(store);

...
```

Functional Programming Approach

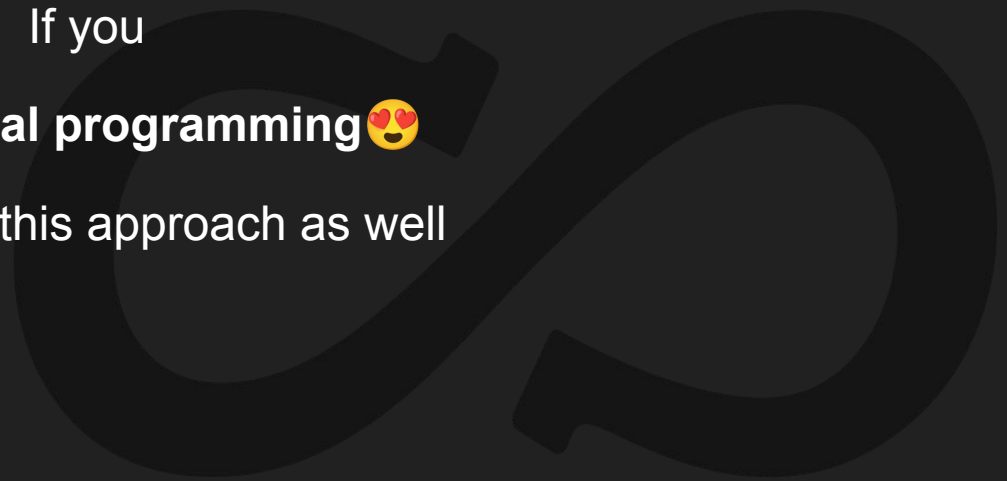
Functional Programming Approach

Under the hood the framework uses a functional programming approach.

If you

😍**functional programming**😍

you can use this approach as well



Functional Programming Approach 😍 **Success**

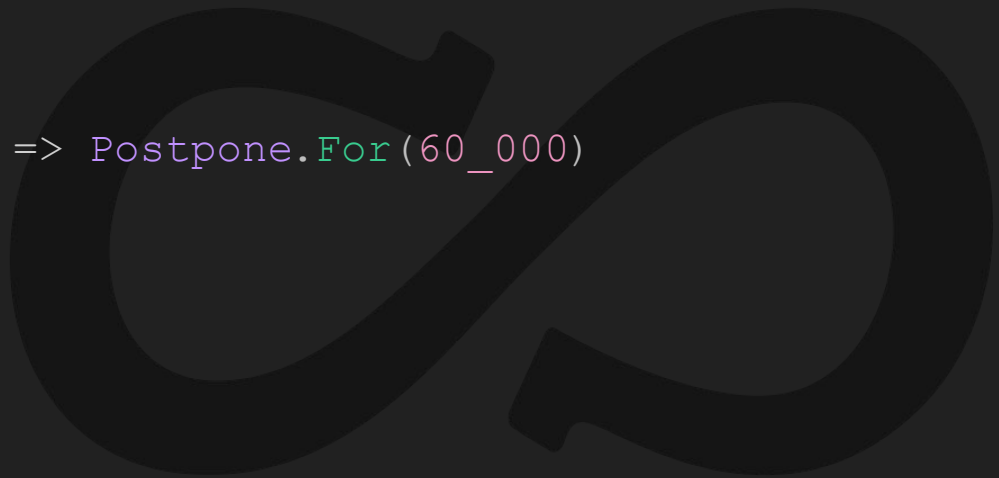
A resilient function can also return a Result-wrapper.
Thus, the following is allowed:

```
rFunctions.RegisterFunc(
    "FunctionalFlow",
    Result<string>(string param) => param
)
```

Functional Programming Approach 😍 **Postpone**

A resilient function can also return a Result-wrapper.
Thus, the following is allowed:

```
rFunctions.RegisterFunc(

    "FunctionalFlow",

    Result<string>(string param) => Postpone.For(60_000)

)
```



Functional Programming Approach 😍 **Failure**

A resilient function can also return a Result-wrapper.
Thus, the following is allowed:

```
rFunctions.RegisterFunc(
    "FunctionalFlow",
    Result<string>(string param)
      => Fail.WithException(new ArgumentException("...")))
```

Coffee Break Problem ☕ Recurring Job

Description:

From the constructs seen so far is it possible to create a recurring job?

For example, implement a resilient function **executes each night at midnight**?

Technical **Architecture**

What problem?!

The problem Resilient Functions solve is **seemingly** simple.
However there are **subtleties** lurking behind the innocent facade.

1. Ensuring two replicas are not executing the same function at the same time (Failure detection)

2. Versioning - handling evolving code without having down time or risking executing the same functionality multiple times

3. A single source of truth for the status of the specific business process

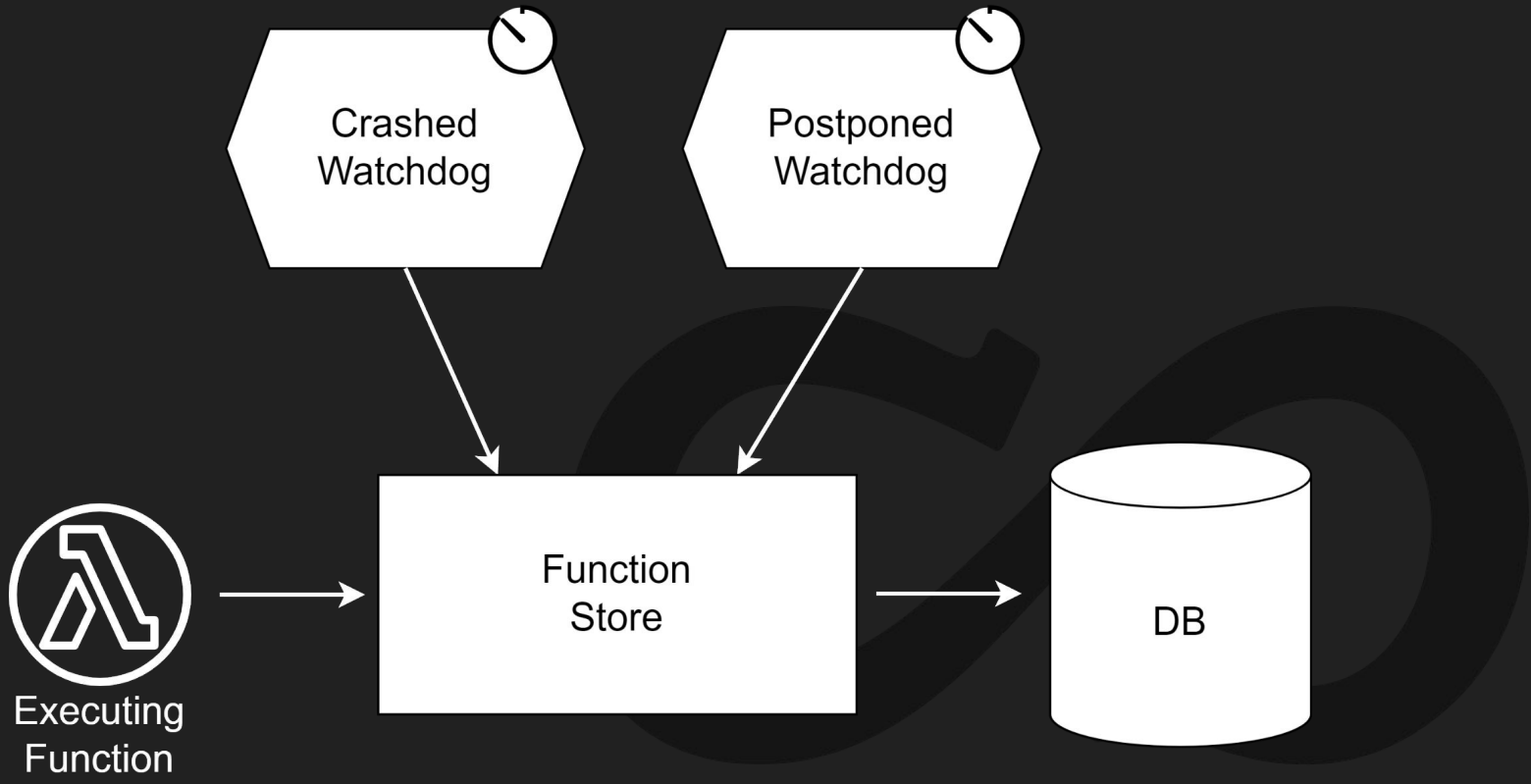
4. Flexibility - it is simple to change data if needed

Framework **Design Principles**

- Create a **simple** and **intuitive abstraction** facilitating with the implementation of **business processes** which must **complete** in their entirety

- **Optimize** the developer's **degrees of freedom**
  - aka you are free to write the code how you like it
  - opposed to having fully declarative configuration (*heavy* abstraction)

Framework **Technical Architecture** (single replica)



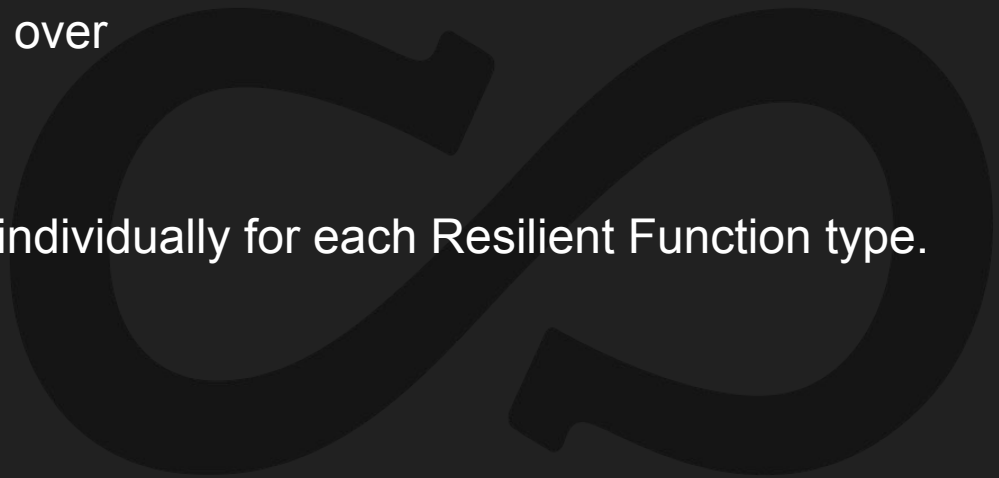
Function Invocation - **Failure Detection**

When a resilient function is invoking a **heartbeat** is repeatedly emitted.

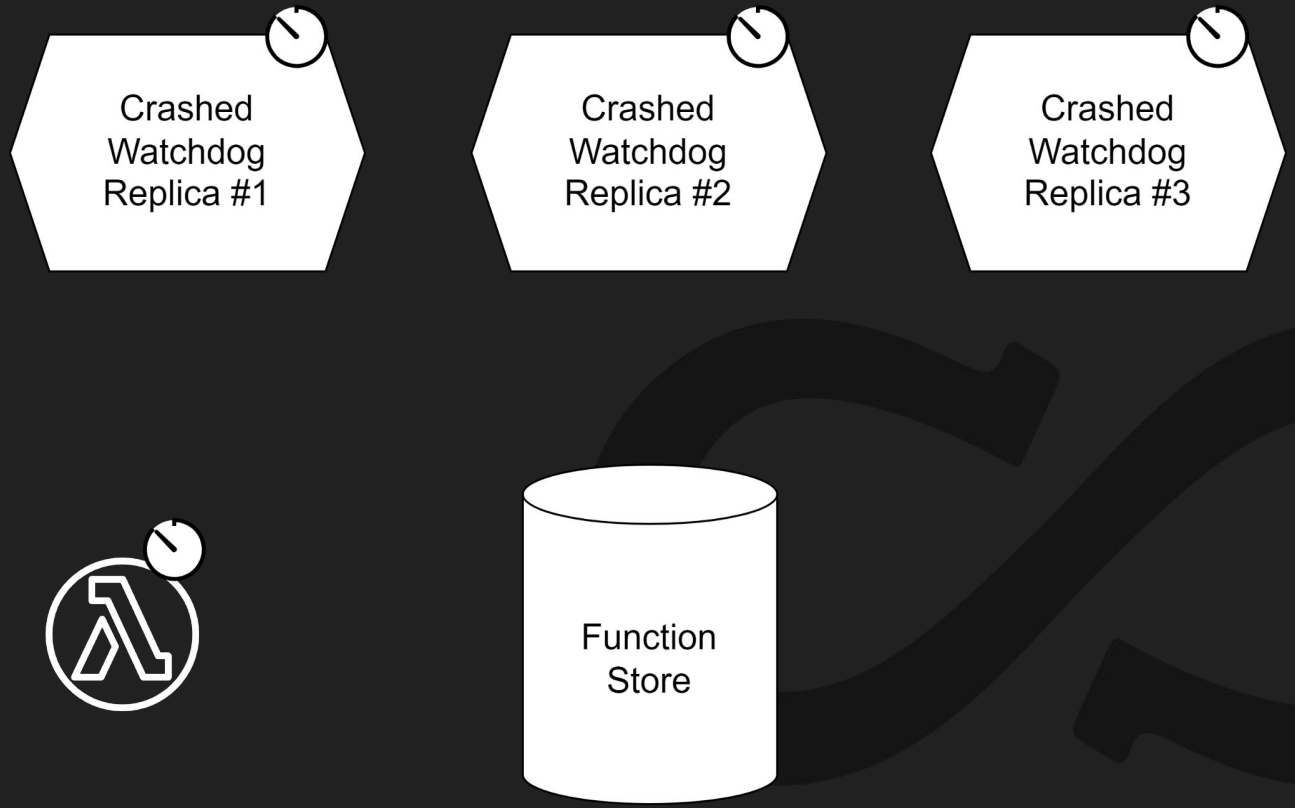
If the heartbeat **stops** then:

- one of the other replicas will take over
- and re-invoke the function.

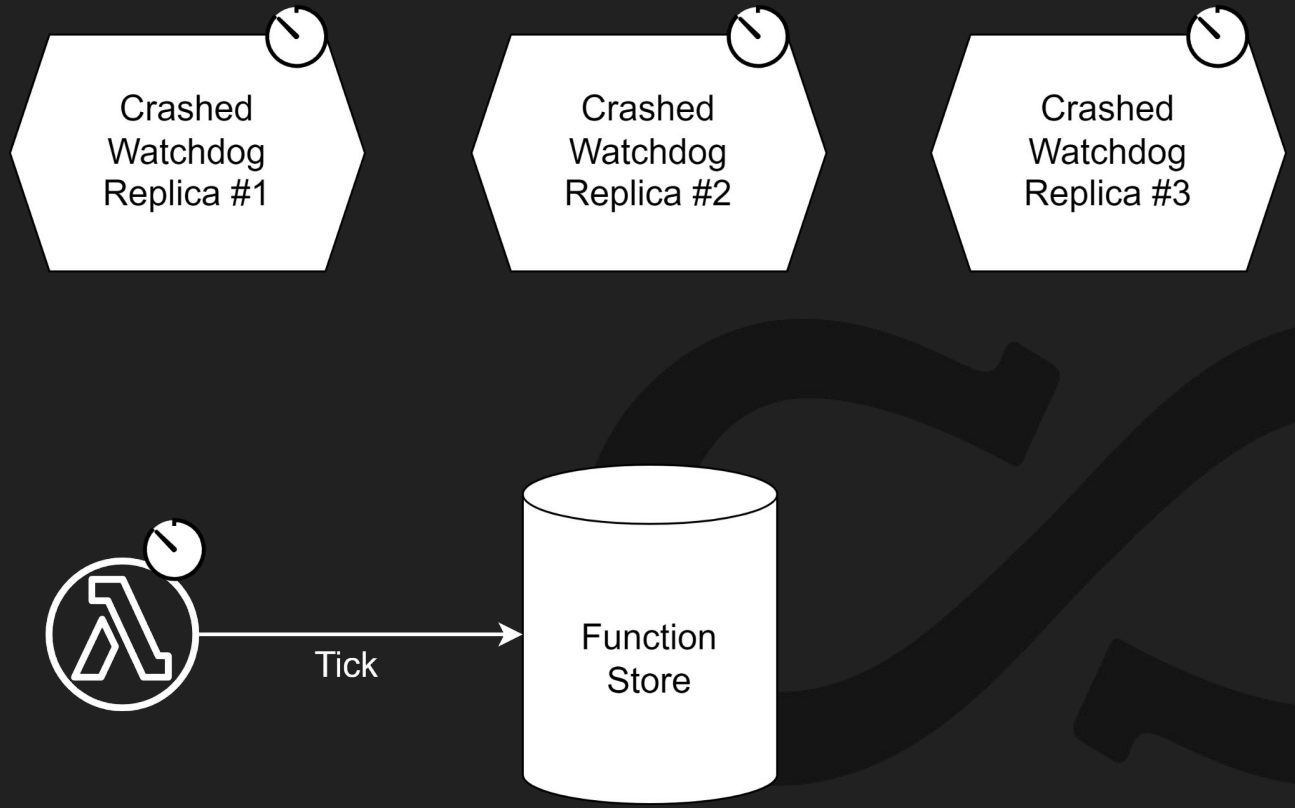
The heartbeat **frequency** can be set individually for each Resilient Function type.



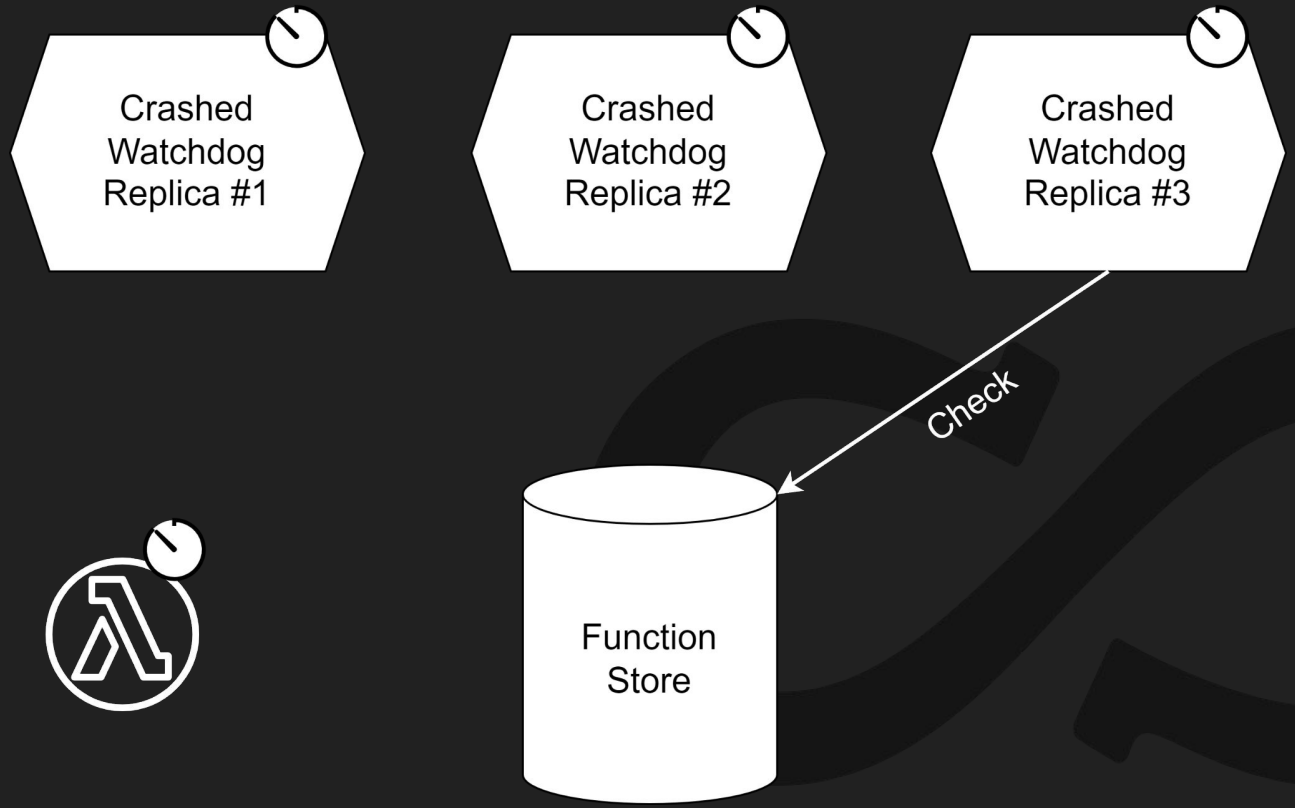
Function Invocation - **Failure Detection**



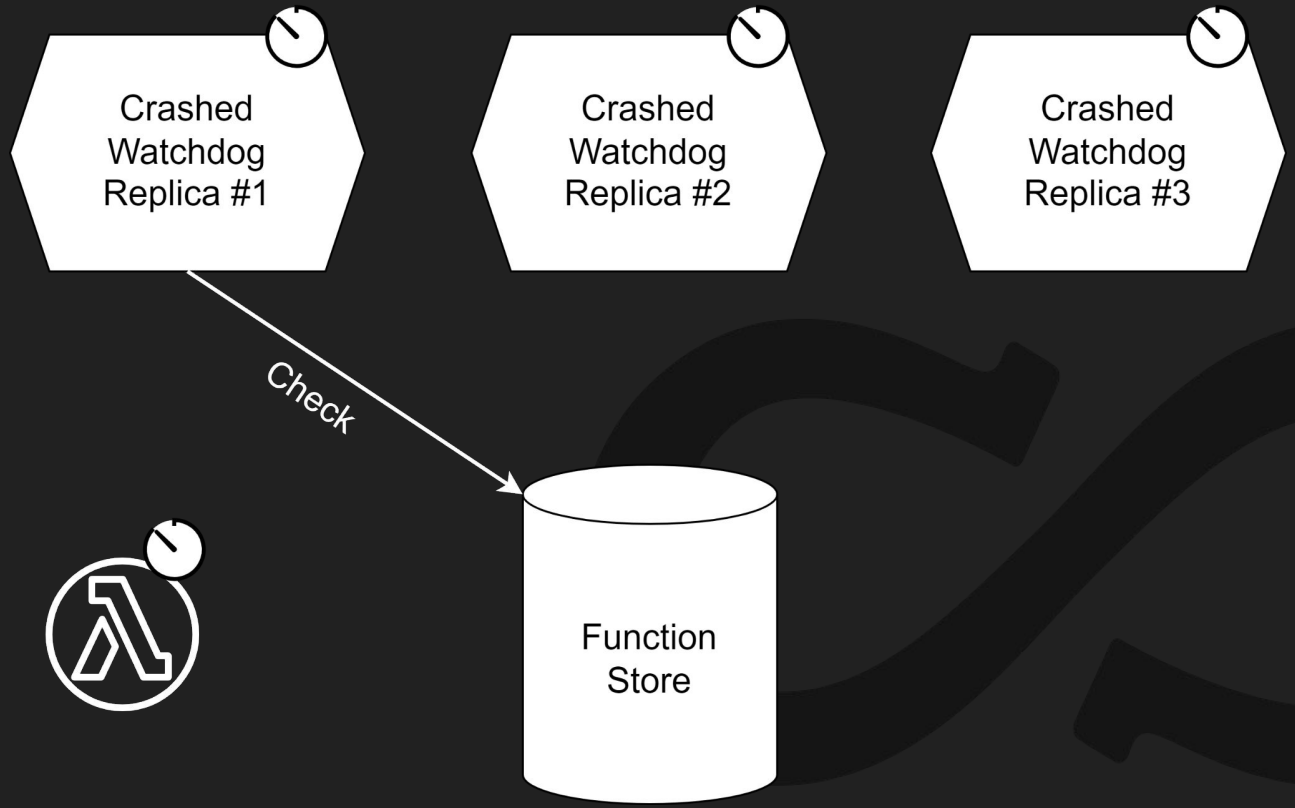
Function Invocation - **Failure Detection**



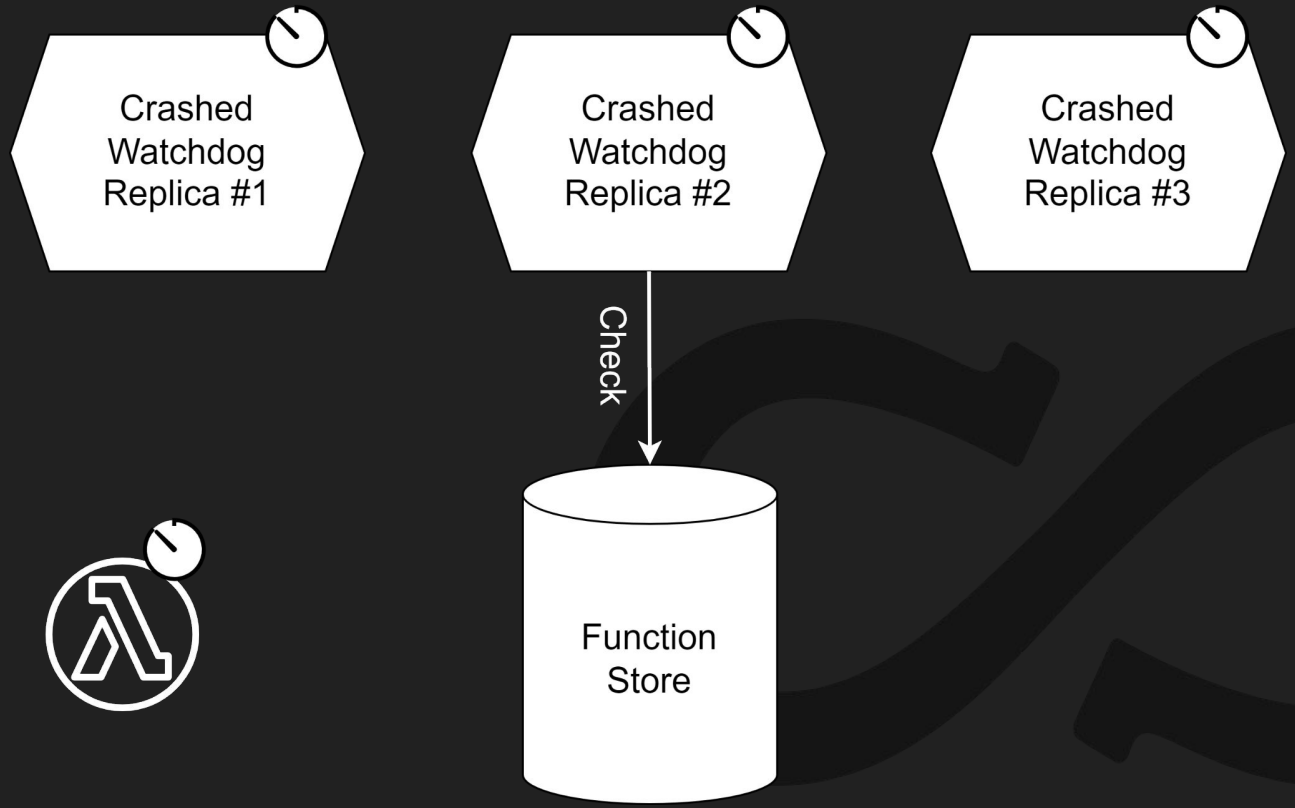
Function Invocation - **Failure Detection**



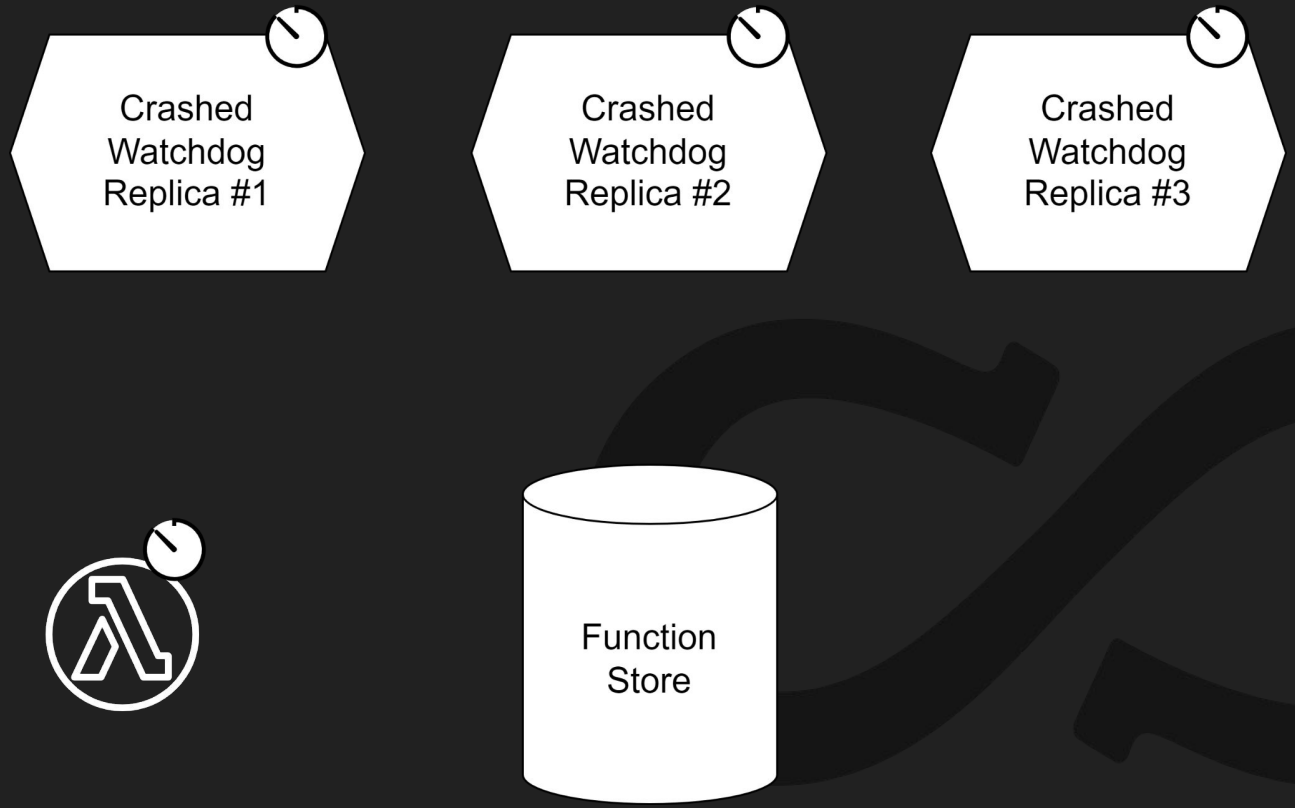
Function Invocation - **Failure Detection**



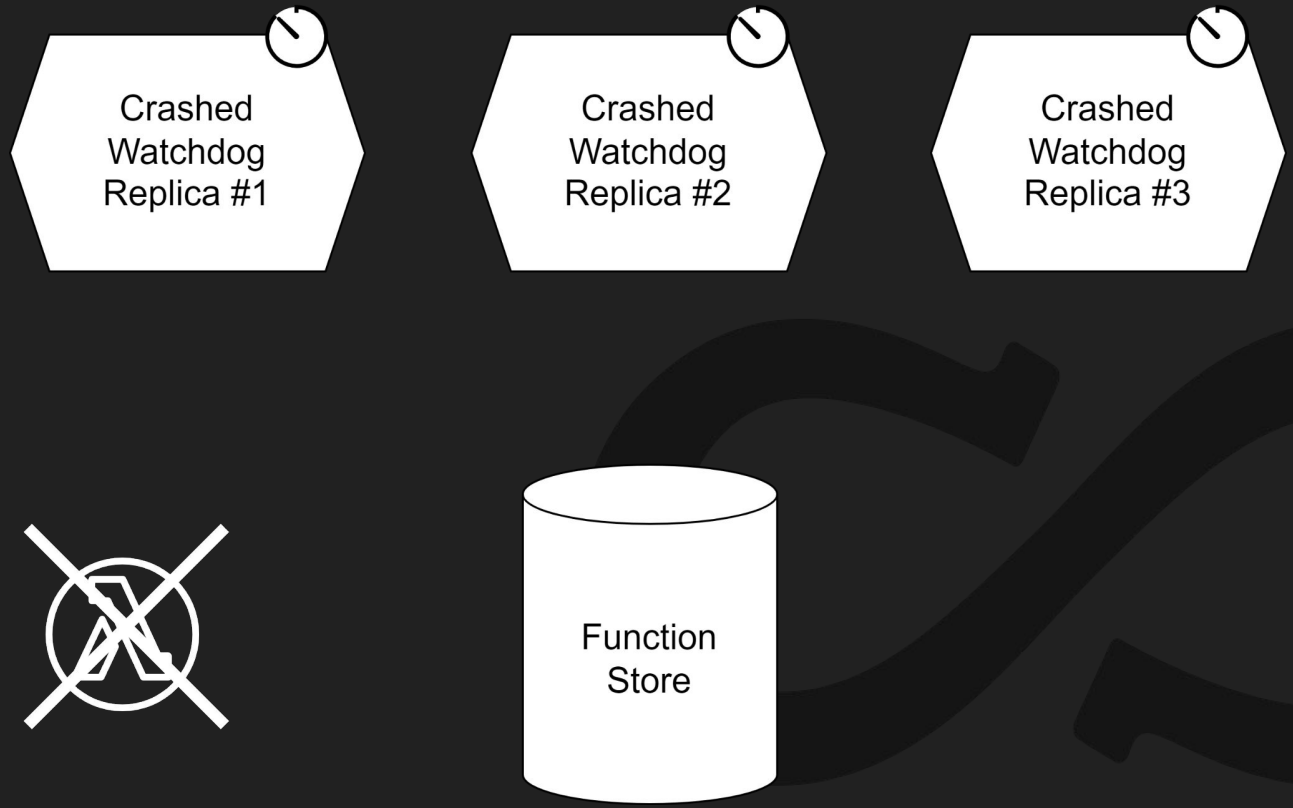
Function Invocation - **Failure Detection**



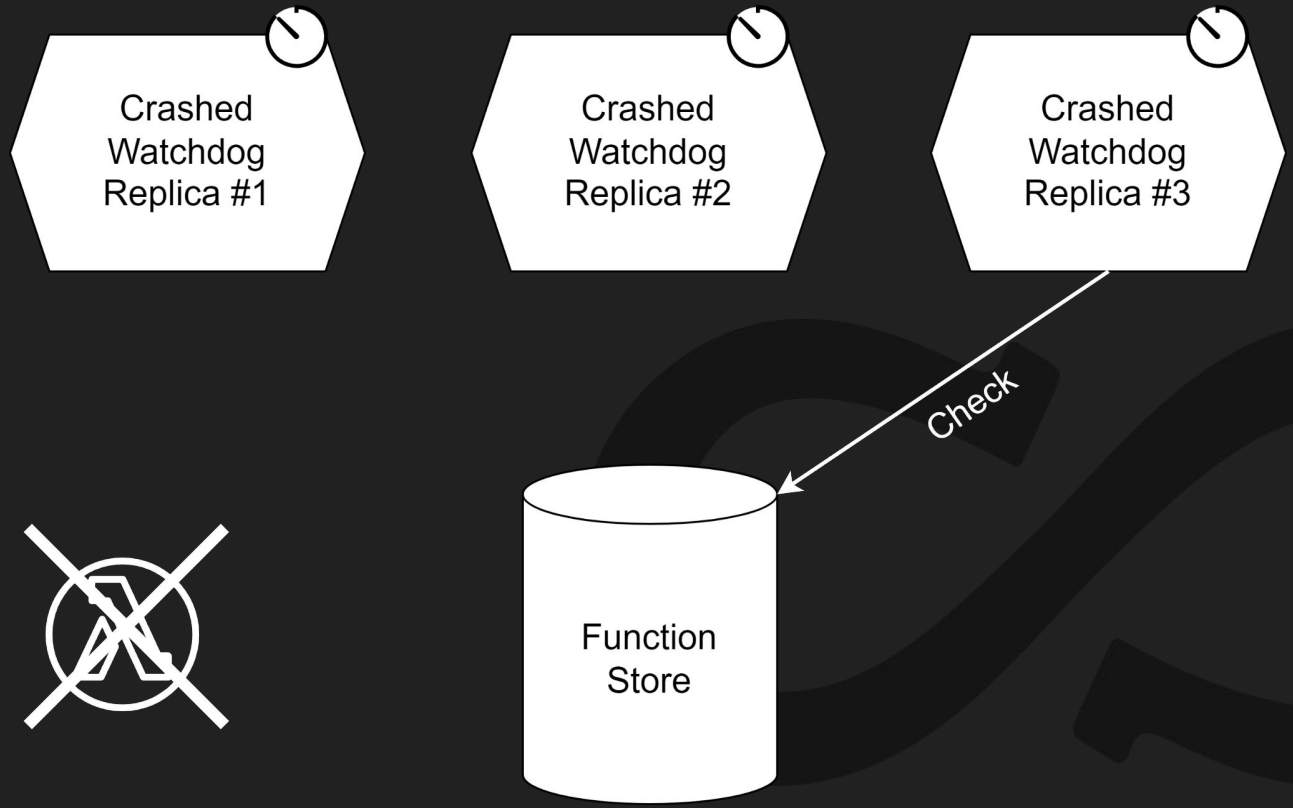
Function Invocation - **Failure Detection**



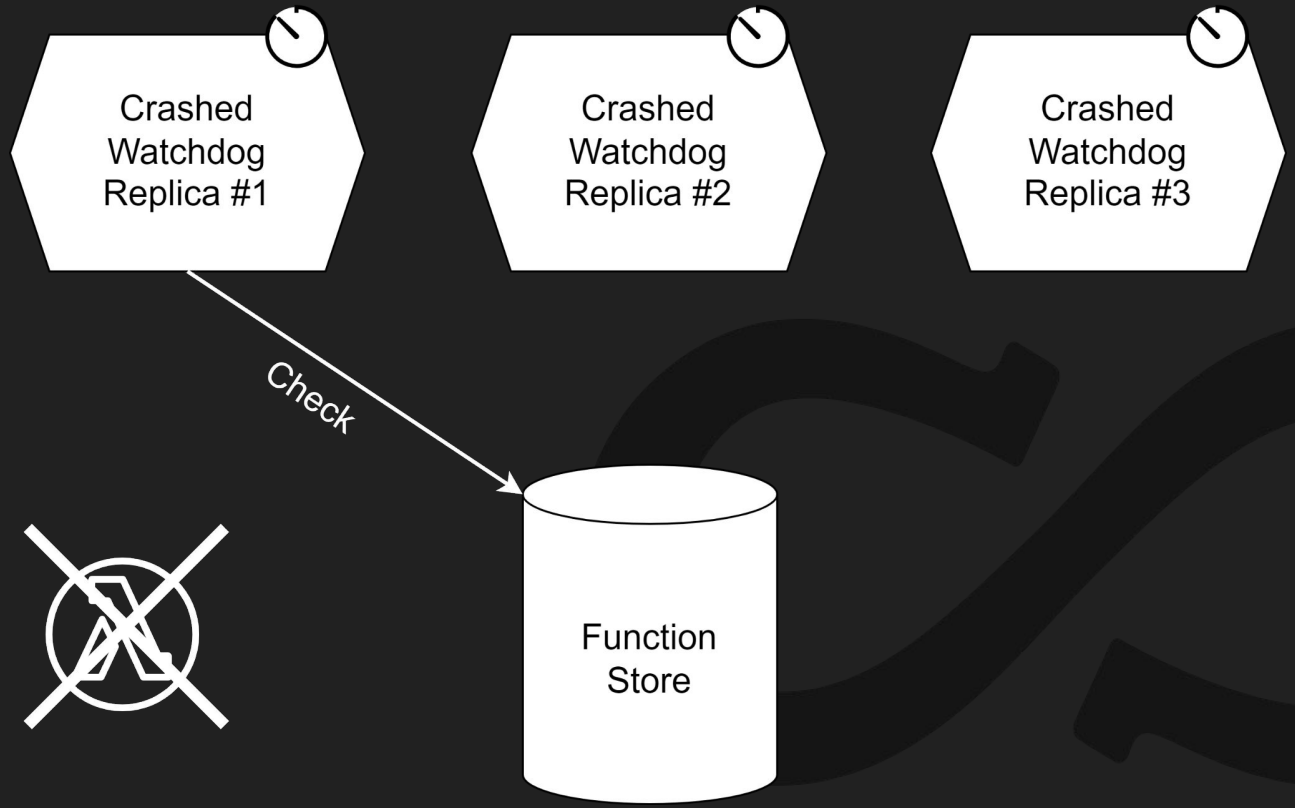
Function Invocation - **Failure Detection**



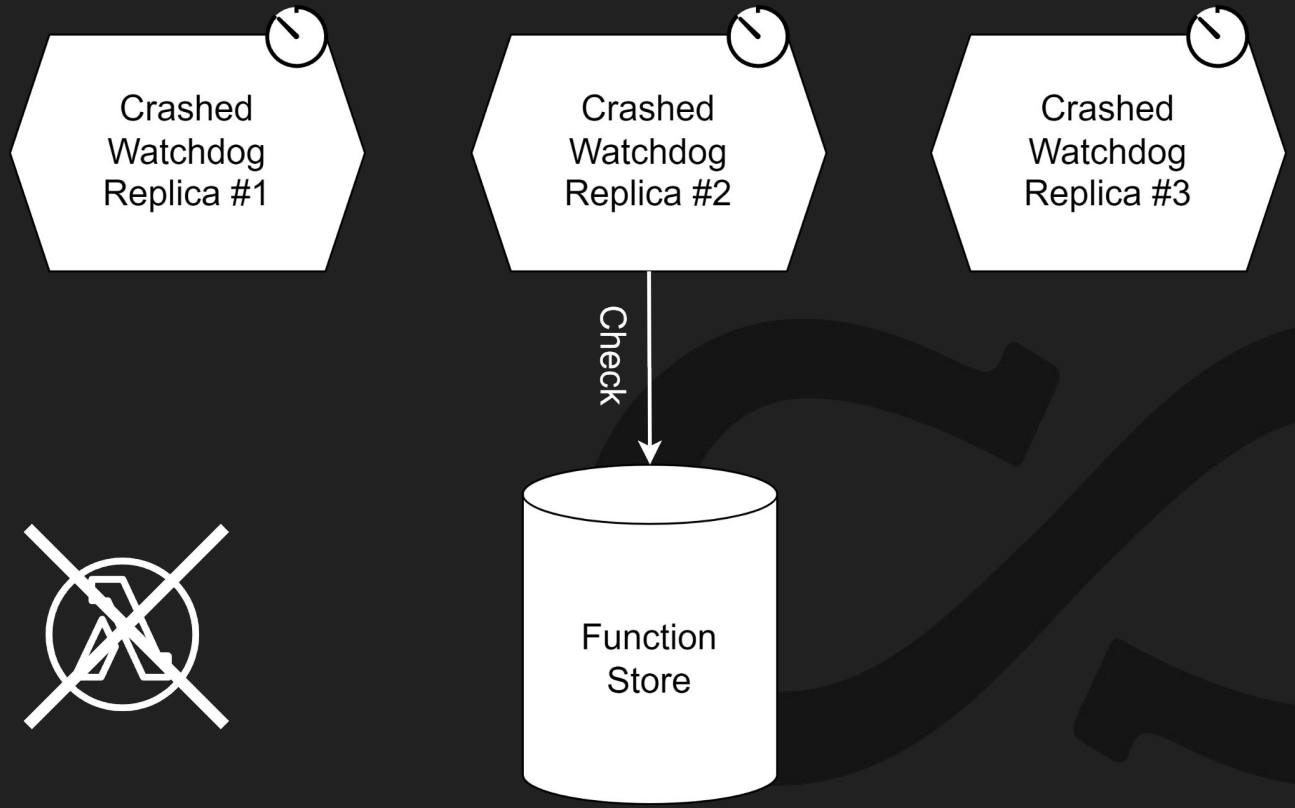
Function Invocation - **Failure Detection**



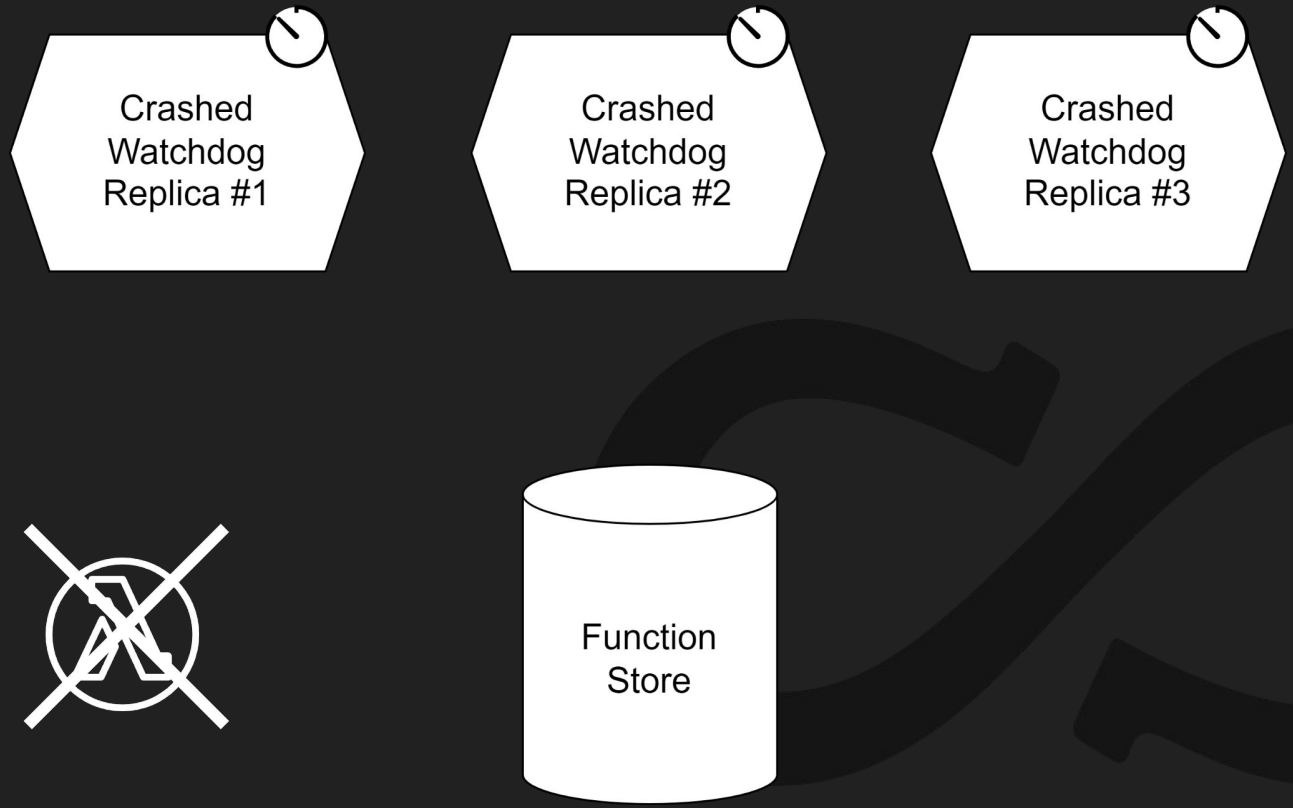
Function Invocation - **Failure Detection**



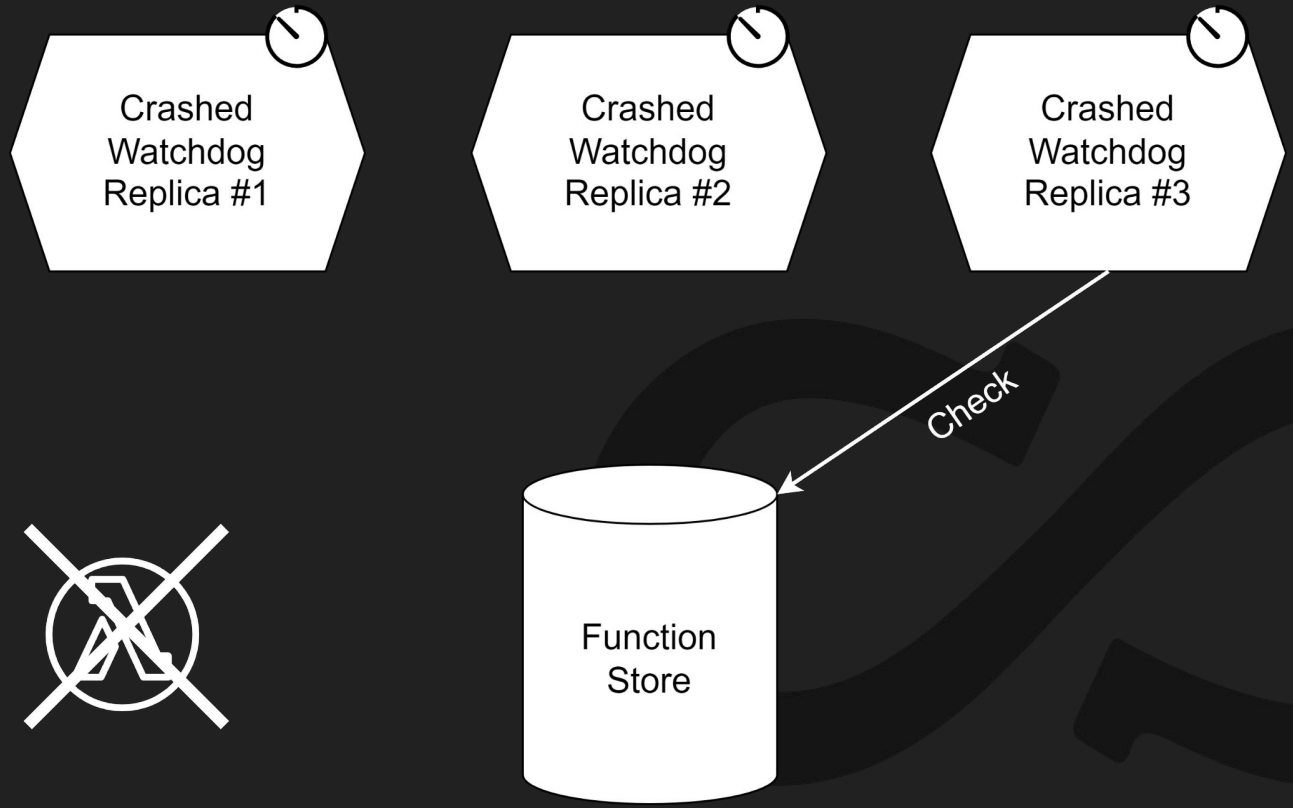
Function Invocation - **Failure Detection**



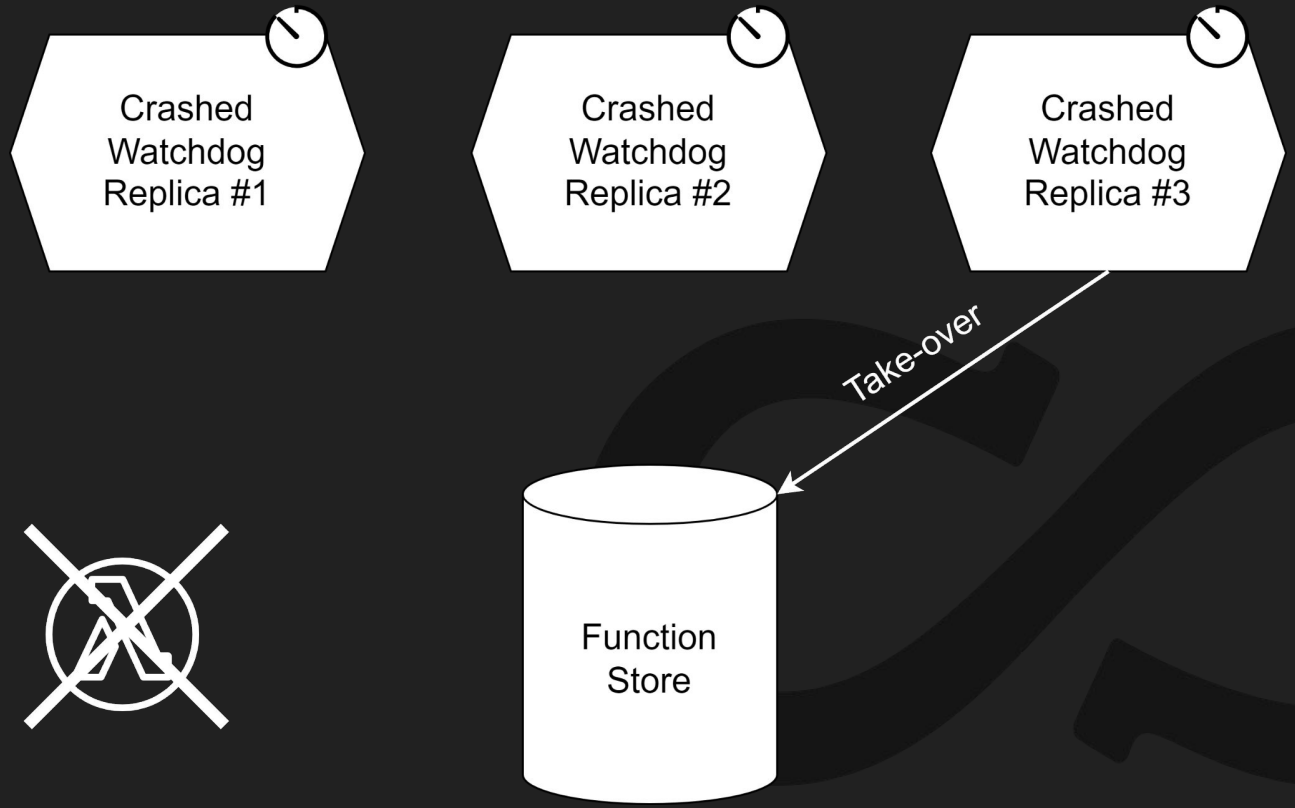
Function Invocation - **Failure Detection**



Function Invocation - **Failure Detection**



Function Invocation - **Failure Detection**



Function Invocation - **Failure Detection**

Questions:

1. What happens if the heartbeat frequency is decreased or increased?
2. How many **round trips** to the database does a successfully executed resilient function invocation at-least entail?

What Cluster size?

The framework has been designed

from the **ground up**

to work

regardless of **# of replicas**

and

with **automatic reconfiguration!**

Function Invocation - **Inlined** vs **Retry** Behavior

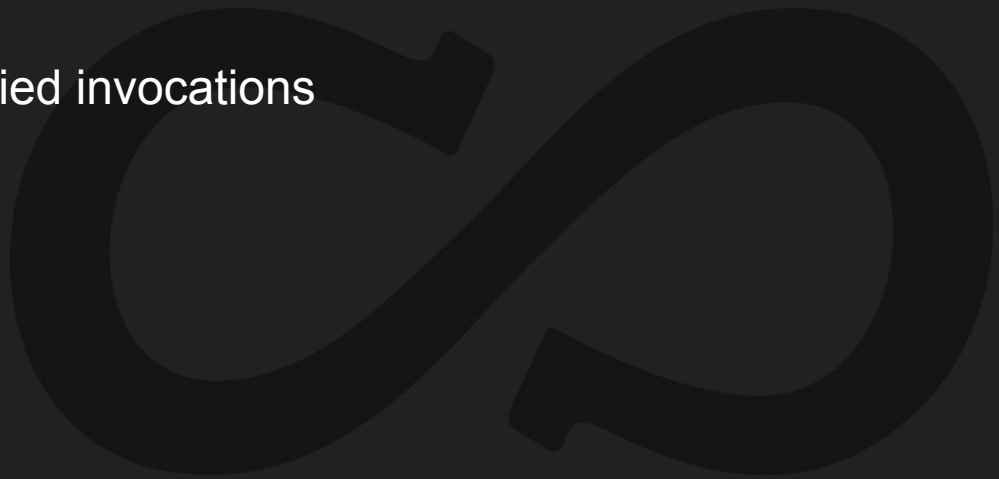
In order to **optimize performance** and **simplify debugging**

the first time a specific resilient function is invoked

the invocation occurs as a **direct call** (i.e. without scheduling)

You also get:

- **cloud independance** & **multiple database** support
- graceful shutdown
- automatic **load balancing** of retried invocations



Coffee Break Problem ☕ Scheduling an invocation

Description:

Assume we want to start processing an order; but not wait for the processing to complete before returning a reply to the caller.

Question:

Is the following sufficient?

```
Task.Run(() => rAction.Invoke(functionInstanceId, param, scrapbook));
return "order processing started";
```

Versioning

Versioning - Handling *always* evolving code

The only thing about code that **doesn't** change

is that

our code **must** change

When downtime is **not** allowed this poses a challenging problem…

Versioning - Handling *always* evolving code

The only thing about code that **doesn't** change

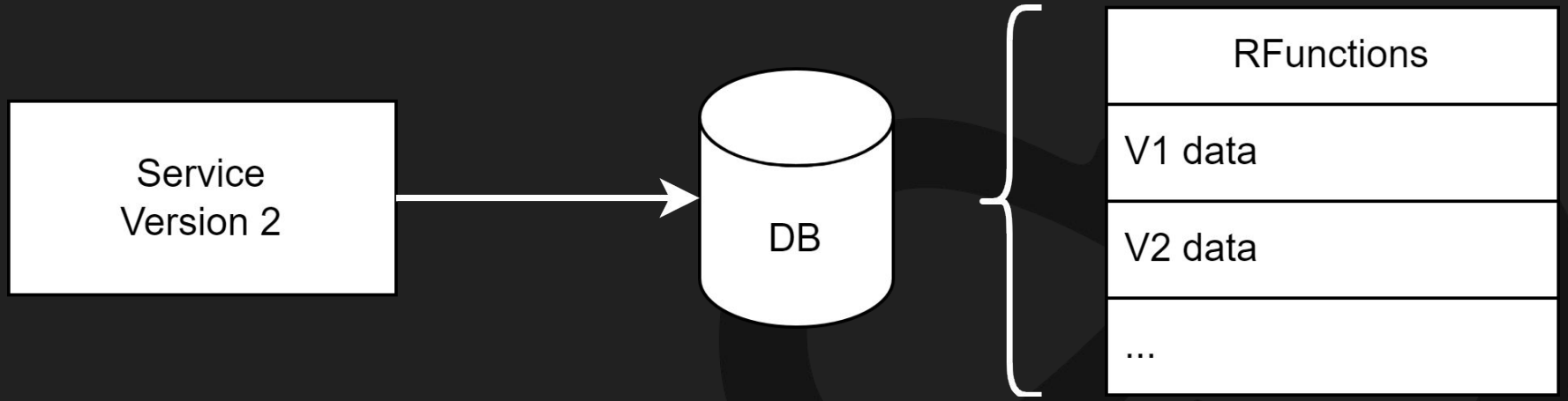
is that

our code **must** change

When downtime is **not** allowed this poses a challenging problem…

As the code must be able to handle both new and previous versions.

Versioning - Handling *always* evolving code 🤔



Versioning - **Backwards Compatibility**

Versioning APIs is commonplace for software developers today.

Versioning of Resilient Functions is actually quite similar but also different:

- Parameter and Scrapbook are (de)serialized from/to **json** out-of-the-box

  Thus, you can **version** by changing the code in a way that is **compatible** with this.

  🤫 if you do not like json - you can also provide your own (de)serializer!

Versioning - **Backwards Compatibility**

(De)serialization is based on the runtime type and not the generic type specified

when registering a resilient function.

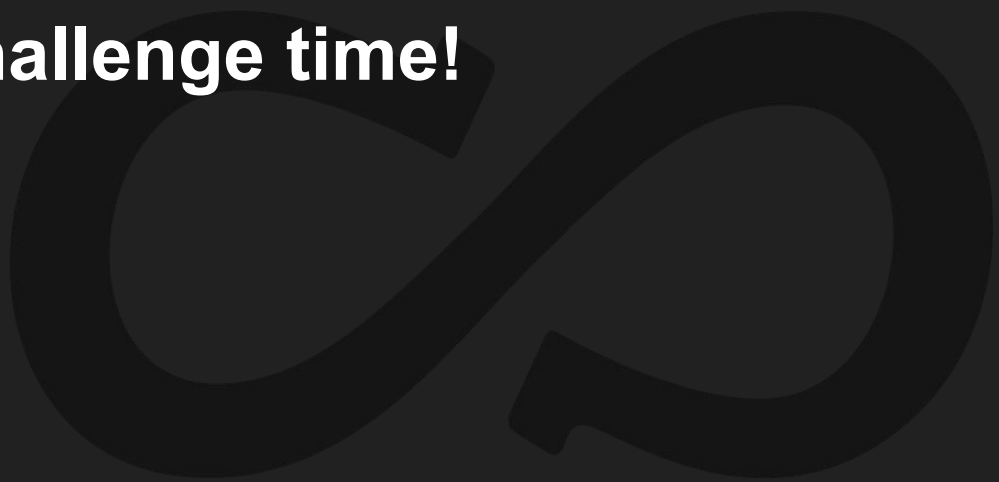
Thus, the following is possible:

```
public async Task ProcessOrder(object order, Scrapbook scrapbook)

   if (order is OrderV1 v1)...

   else if (order is OrderV2 v2)...
```

Resilient Functions - **Order Processing**

“Added Brand” Scenario

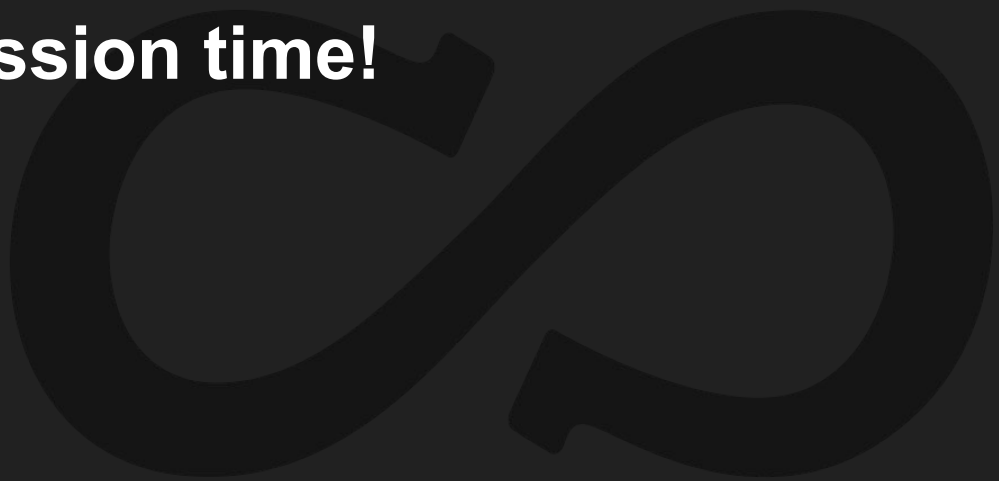
Code Challenge time!



Resilient Functions - **Order Processing**

“New Payment Provider” Scenario

Discussion time!

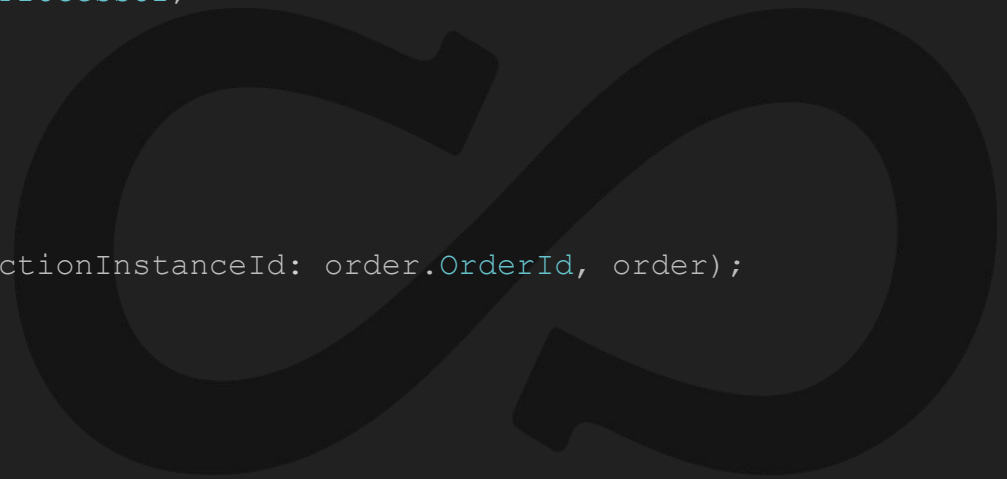


~~Coffee Break~~ Problem ☕ Versioning & Distinct Functions

Description:

What if we prefer to not change the existing code but use different Resilient Functions when versioning? I.e.

```
private readonly V1.OrderProcessor _orderProcessor;


[HttpPost]

public async Task Post(V1.Order order)

   await _orderProcessor.ProcessOrder(functionInstanceId: order.OrderId, order);
```



~~Coffee Break~~ Problem ☕ Versioning & Distinct Functions

Description:

What if we prefer to not change the existing code but use different Resilient Functions when versioning? I.e.

```
private readonly V1.OrderProcessor _orderProcessor;


[HttpPost]

public async Task Post(V1.Order order)

   await _orderProcessor.ProcessOrder(functionInstanceId: order.OrderId, order);
```

~~Coffee Break~~ Problem ☕ Versioning & Distinct Functions

Description:

What if we prefer to not change the existing code but use different Resilient Functions when versioning? I.e.

```
private readonly V2.OrderProcessor _orderProcessor;


[HttpPost]

public async Task Post(V2.Order order)

   await _orderProcessor.ProcessOrder(functionInstanceId: order.OrderId, order);
```

~~Coffee Break~~ Problem☕Versioning & Distinct Functions

Description:

What if we prefer to not change the existing code but use different Resilient Functions when versioning?

Question:

Is there any issue with this approach?

Resilient Functions - 2x **Order Processing**

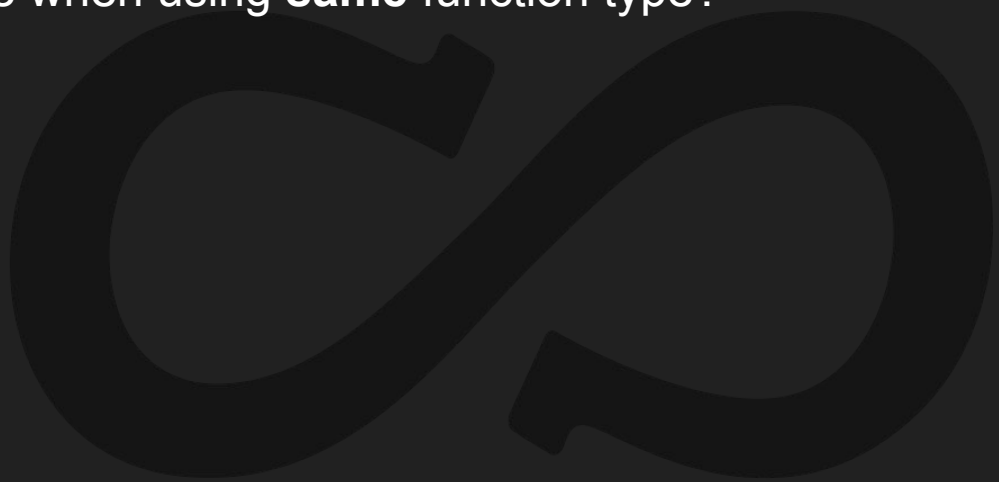
Works with **upcoming**

SharedState<T>

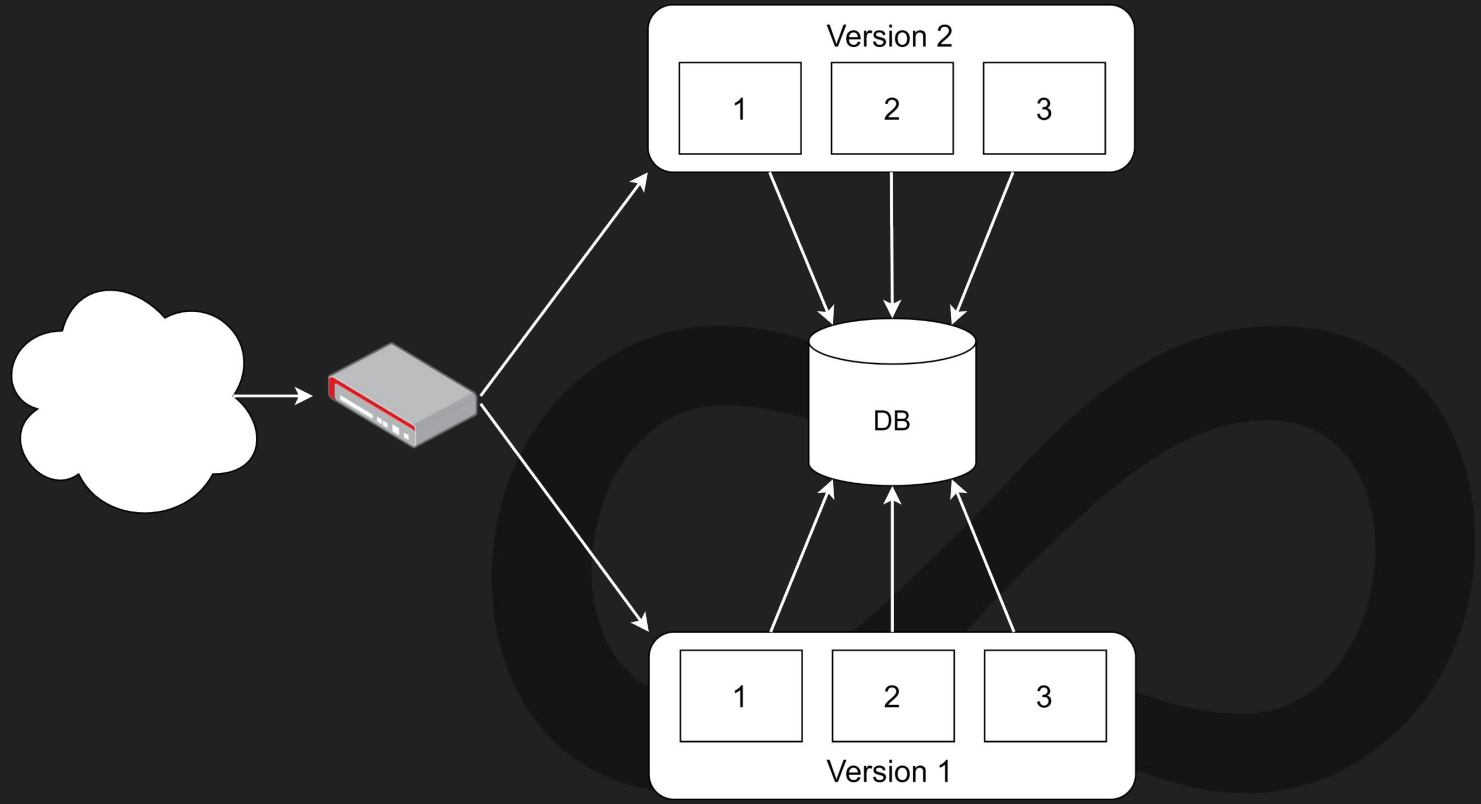
Type

Deployment & Versioning - Same FunctionType

Do we have a similar issue when using **same** function type?



Deployments & Versioning - 🐙Octopus Problem🐙



Deployment & Versioning - CPR Example

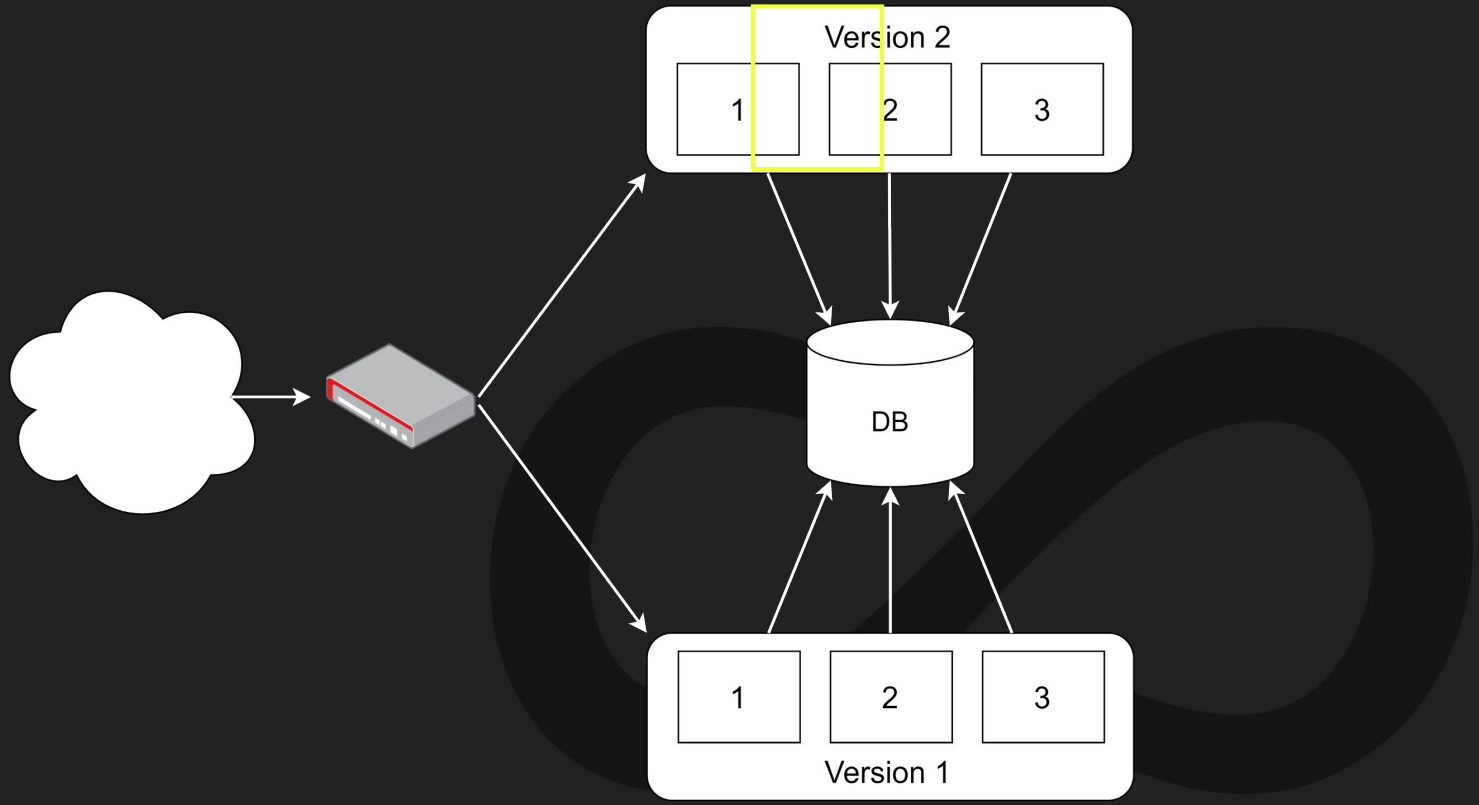
V1:

```
public record Person(string Birthdate, string SequenceNumber);
```

V2:

```
public record Person(string Birthdate, string SequenceNumber, string? Cpr);

public void Process(Person person)

    if (person.Cpr == null)

        person = person with { Cpr = $"{person.Birthdate}-{person.SequenceNumber}" };
```

Deployments & Versioning - 🐙Octopus Problem🐙



Deployment & Versioning - Same FunctionType

Solution #1:

- Delay use of feature until deployment has completed (i.e. with feature flag)

  (*be careful with rollbacks*)

Deployment & Versioning - Same FunctionType

Solution #2:

- Add version to parameter or scrapbook which you can verify on invocation

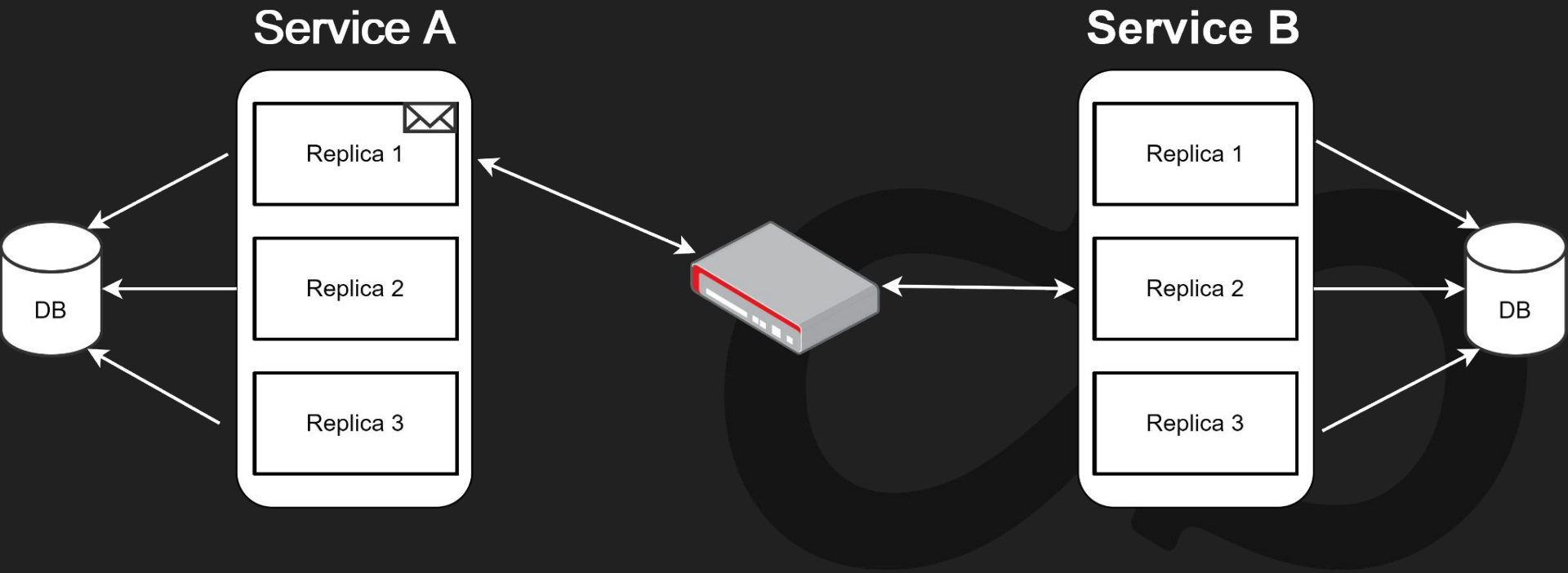
```
rFunctions.RegisterAction(
    "OrderProcessor",
    void (Order order) =>
        if (order.Version > 1)
            throw new ArgumentException("Unsupported version");
        //...
);
```

Shifting gears…

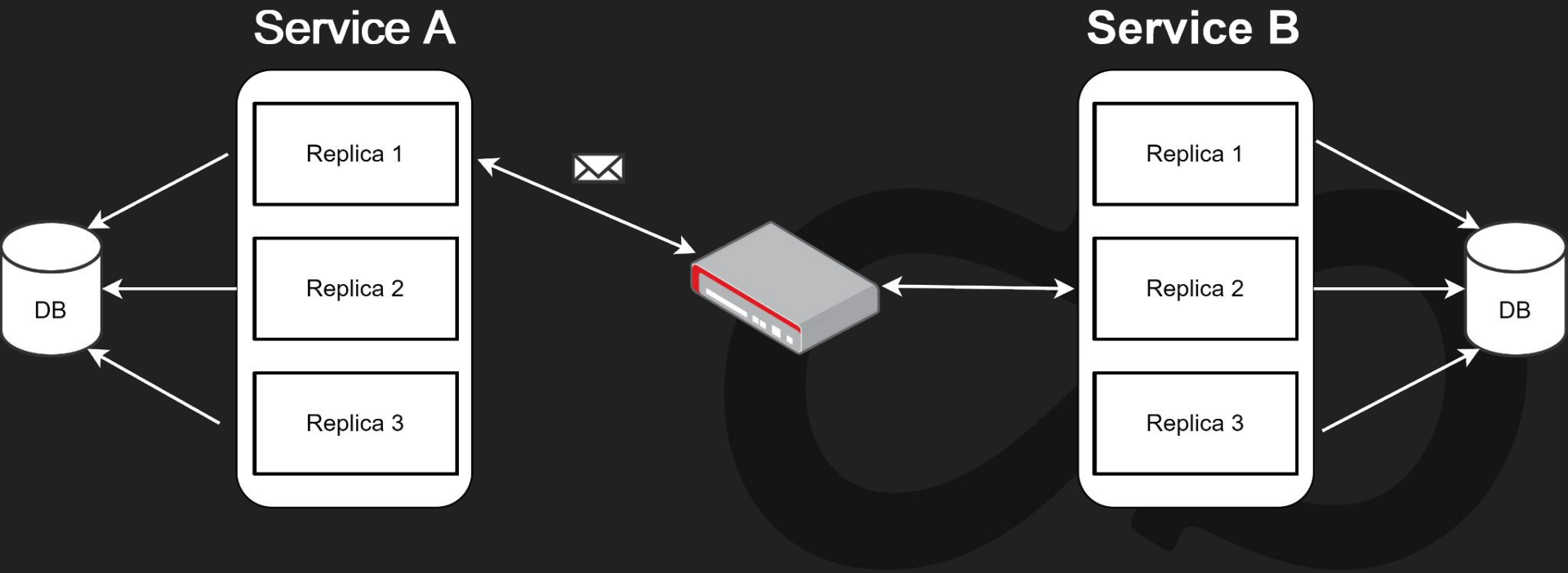
Communication Patterns

Point-to-Point

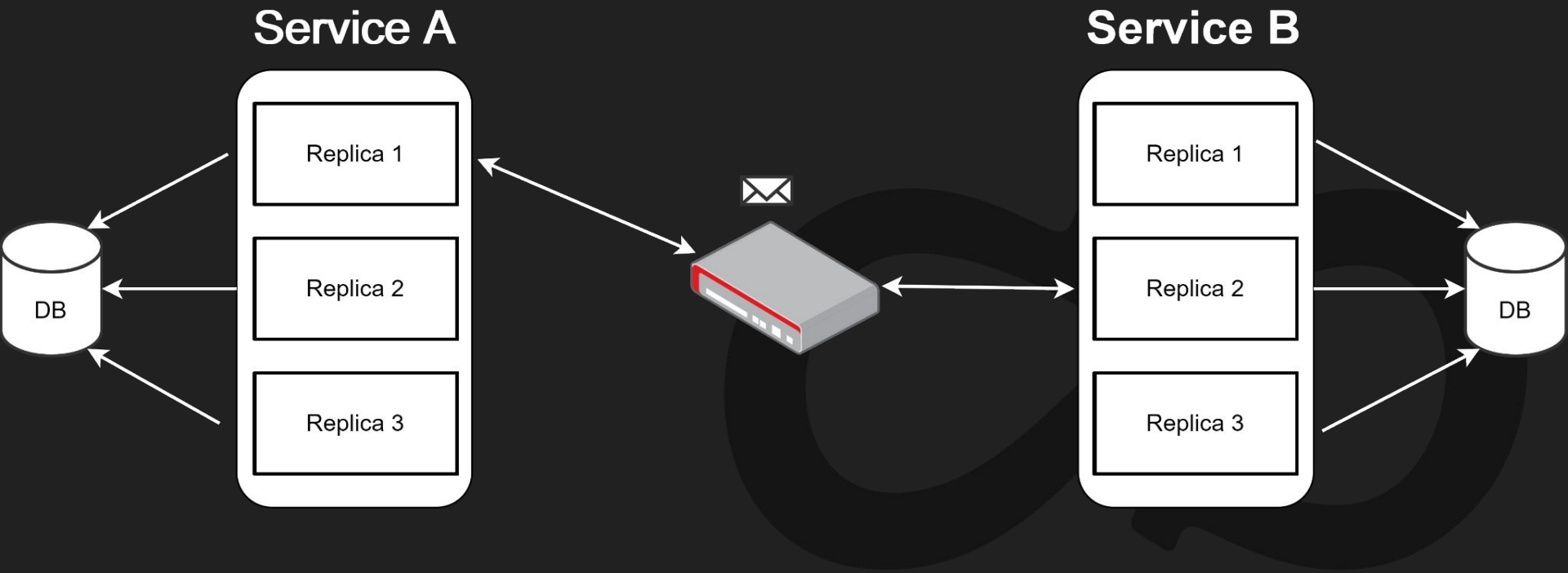
Communication Pattern - **Point-to-Point**



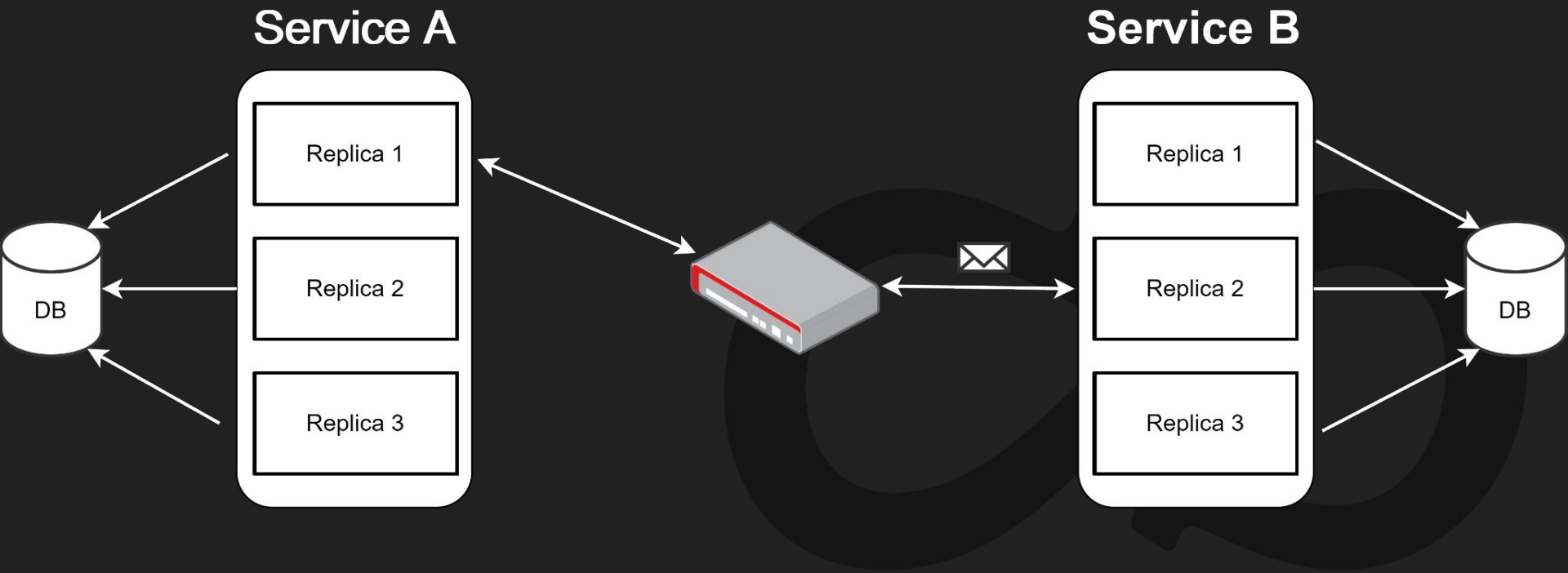
Communication Pattern - **Point-to-Point**



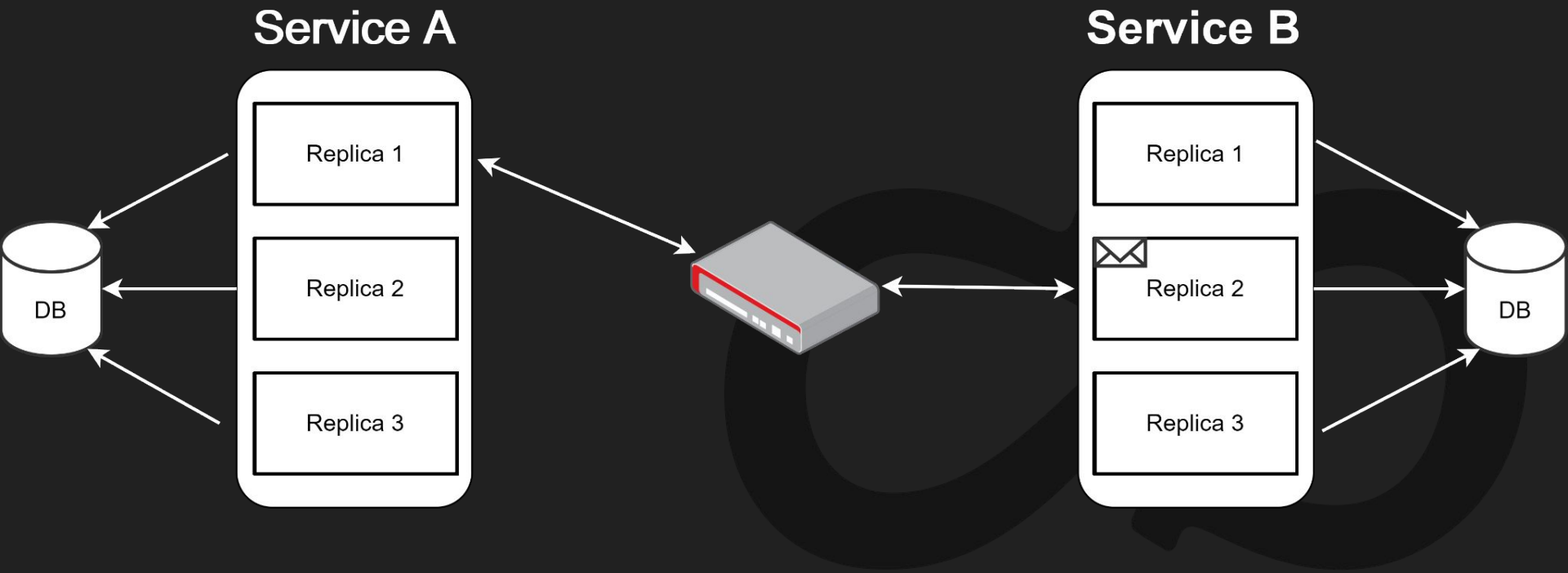
Communication Pattern - **Point-to-Point**



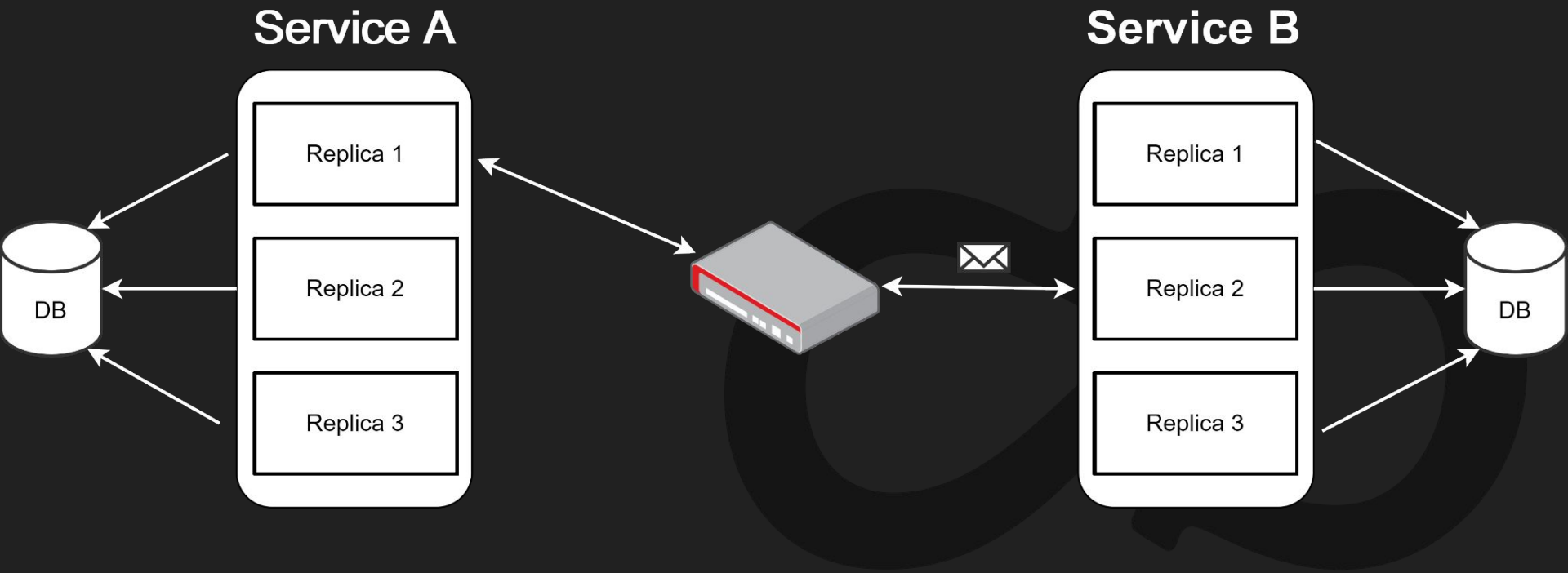
Communication Pattern - **Point-to-Point**



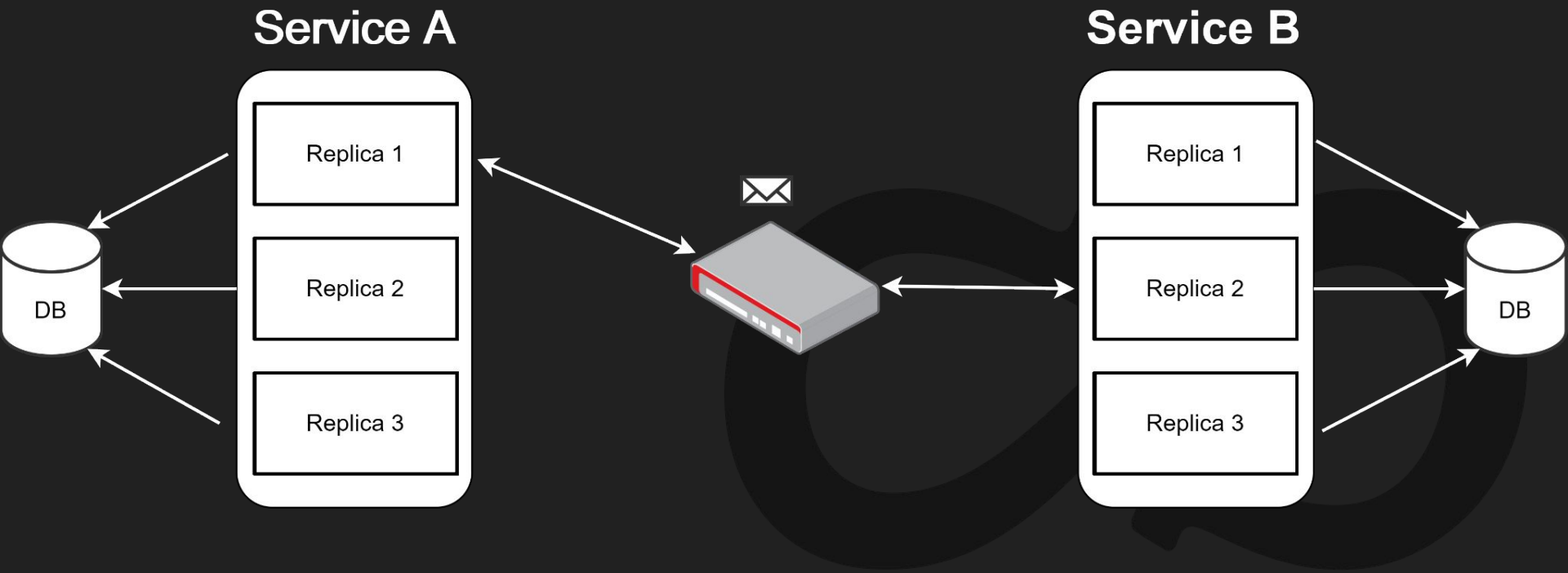
Communication Pattern - **Point-to-Point**



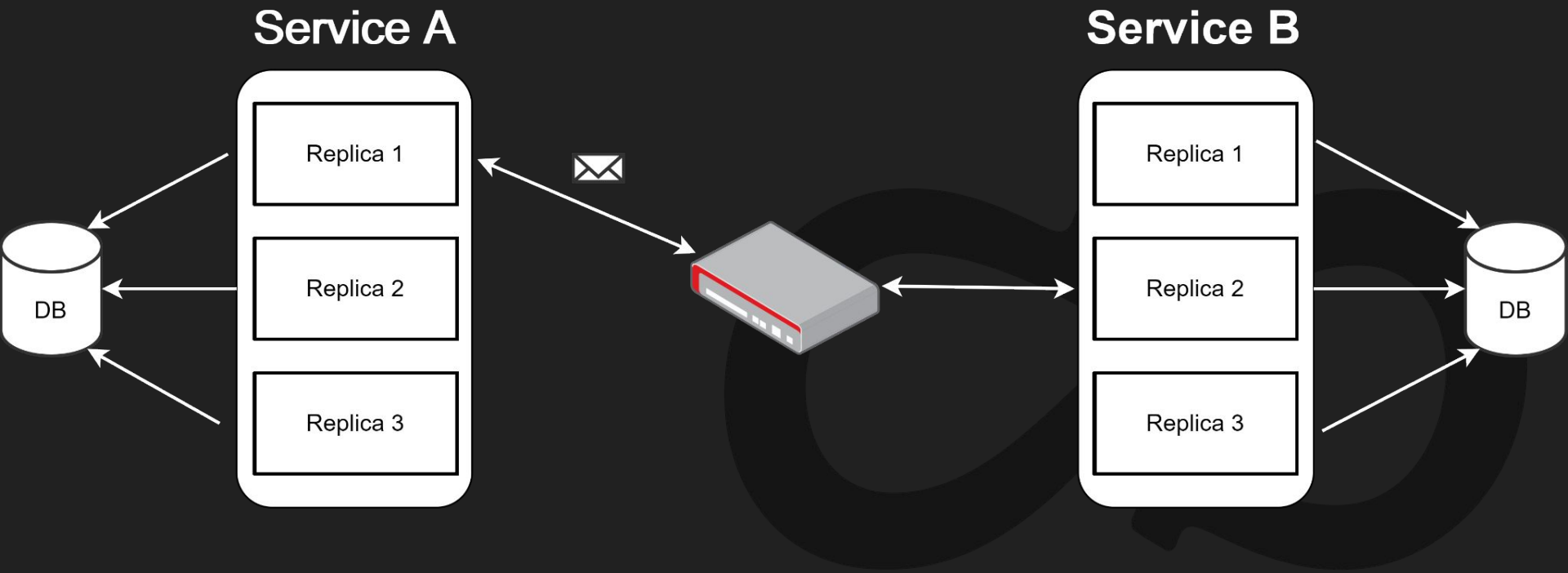
Communication Pattern - **Point-to-Point**



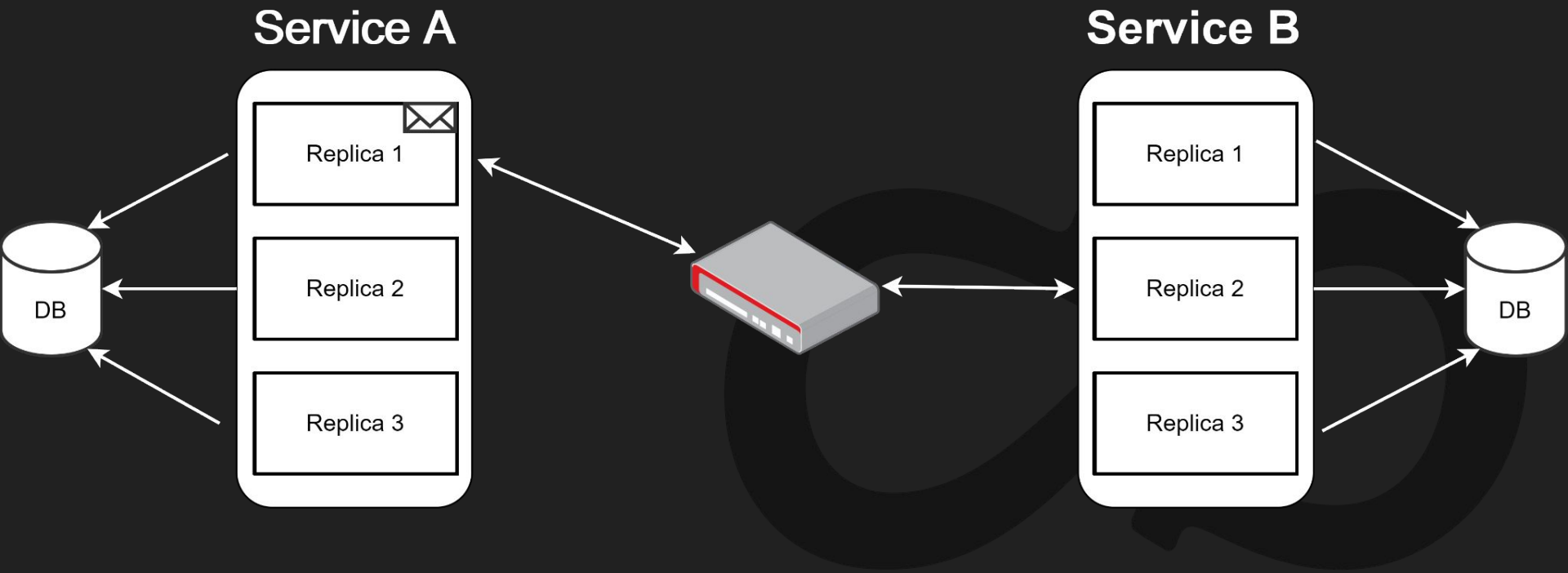
Communication Pattern - **Point-to-Point**



Communication Pattern - **Point-to-Point**

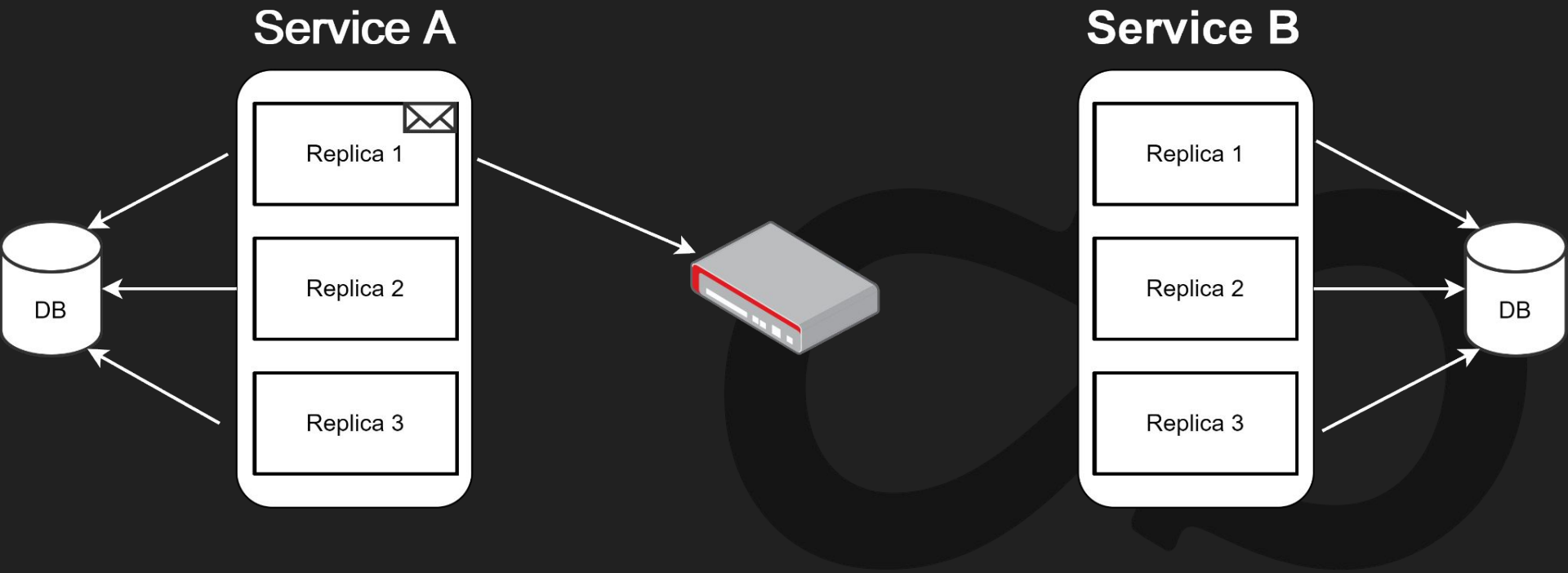


Communication Pattern - **Point-to-Point**

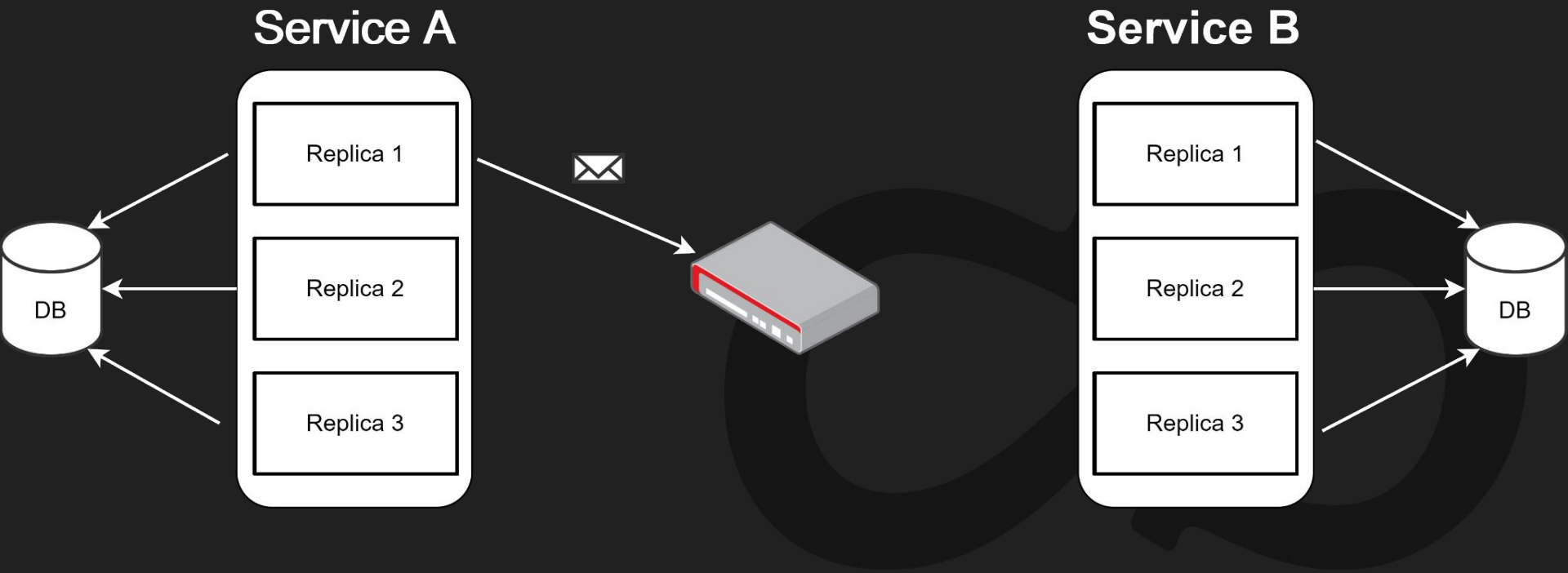


Brokered

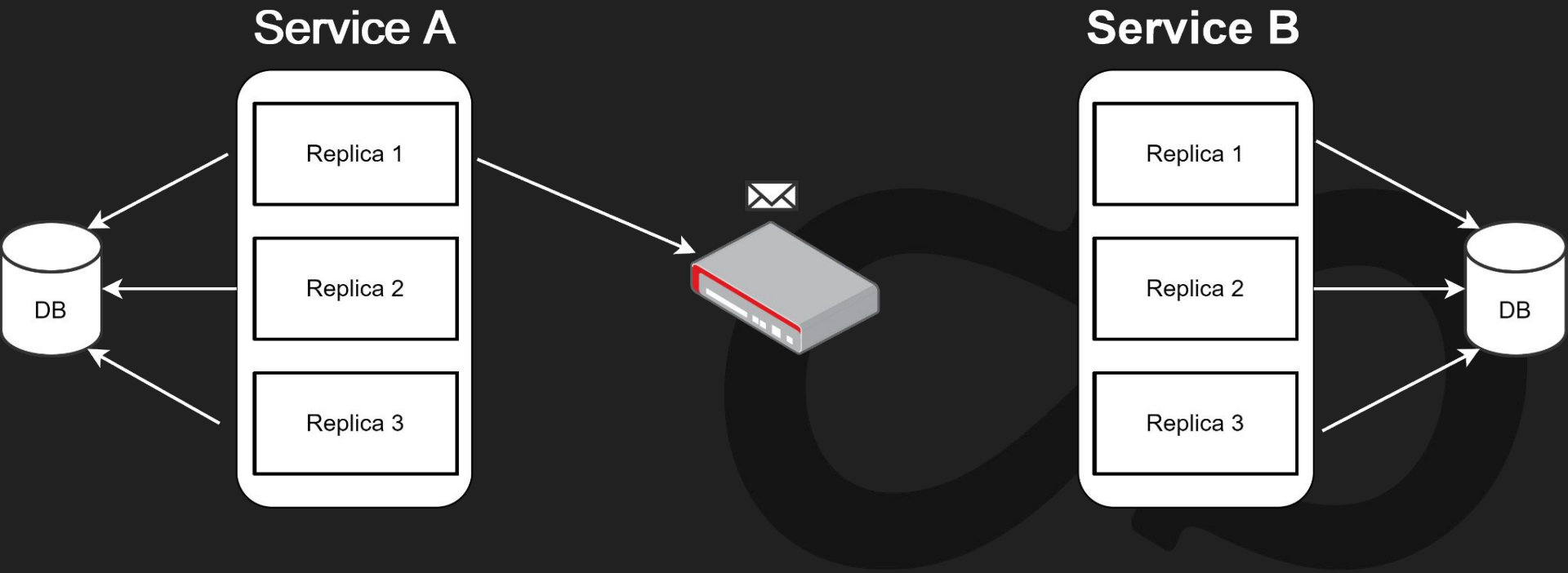
Communication Pattern - **Brokered**



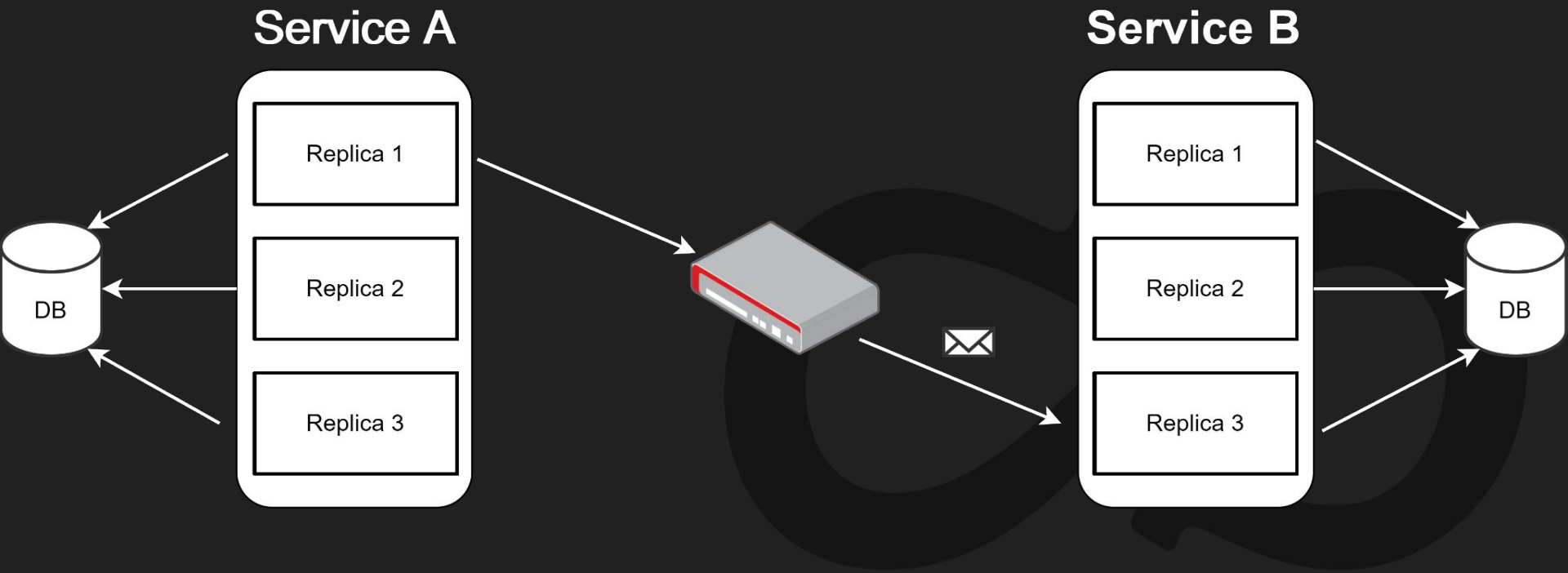
Communication Pattern - **Brokered**



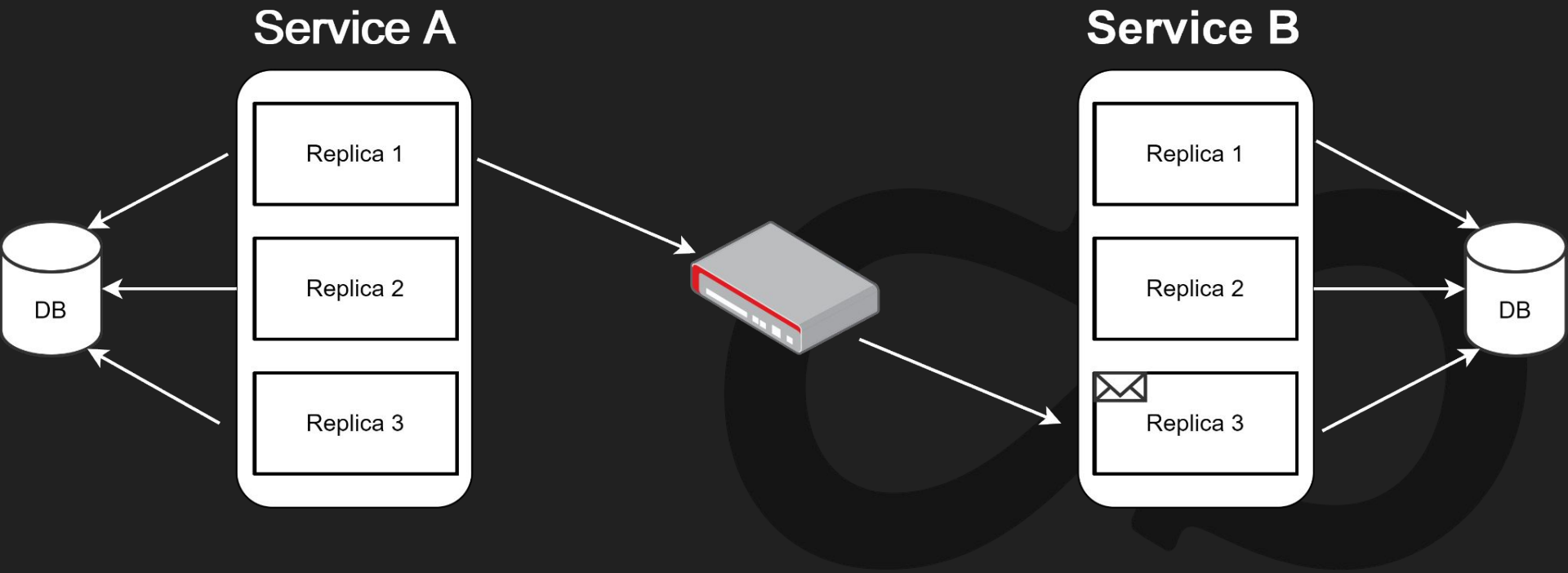
Communication Pattern - **Brokered**



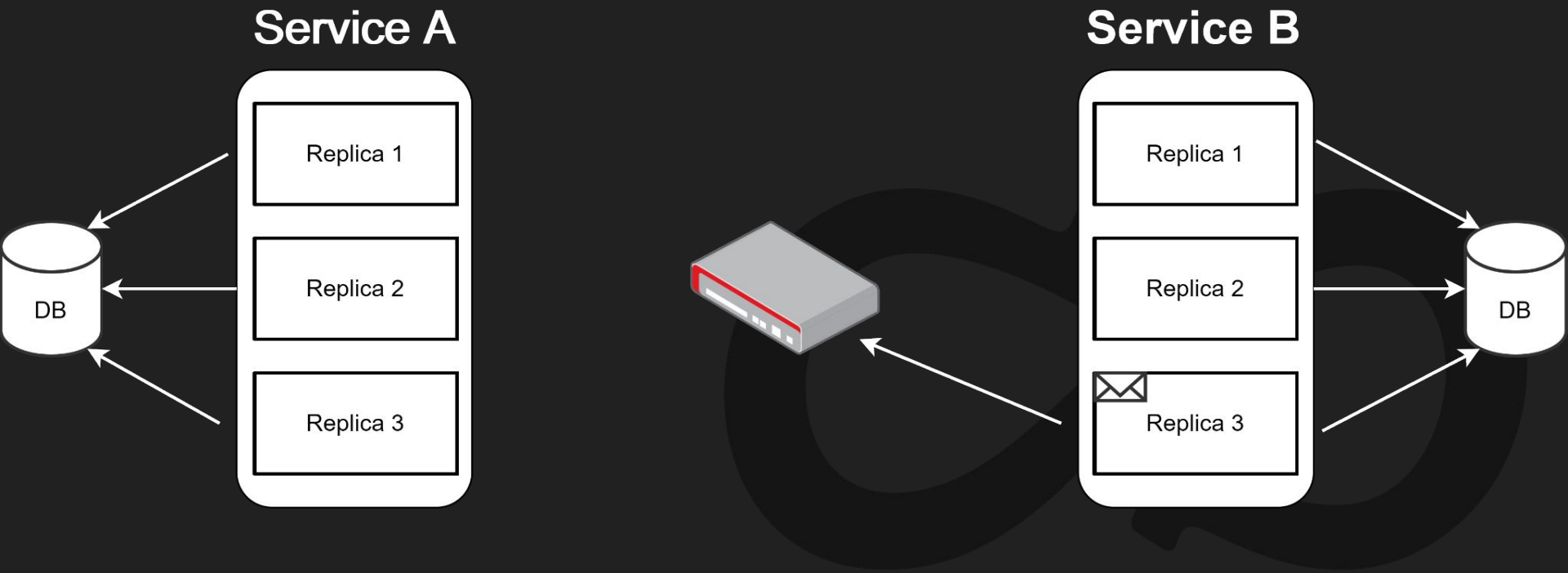
Communication Pattern - **Brokered**



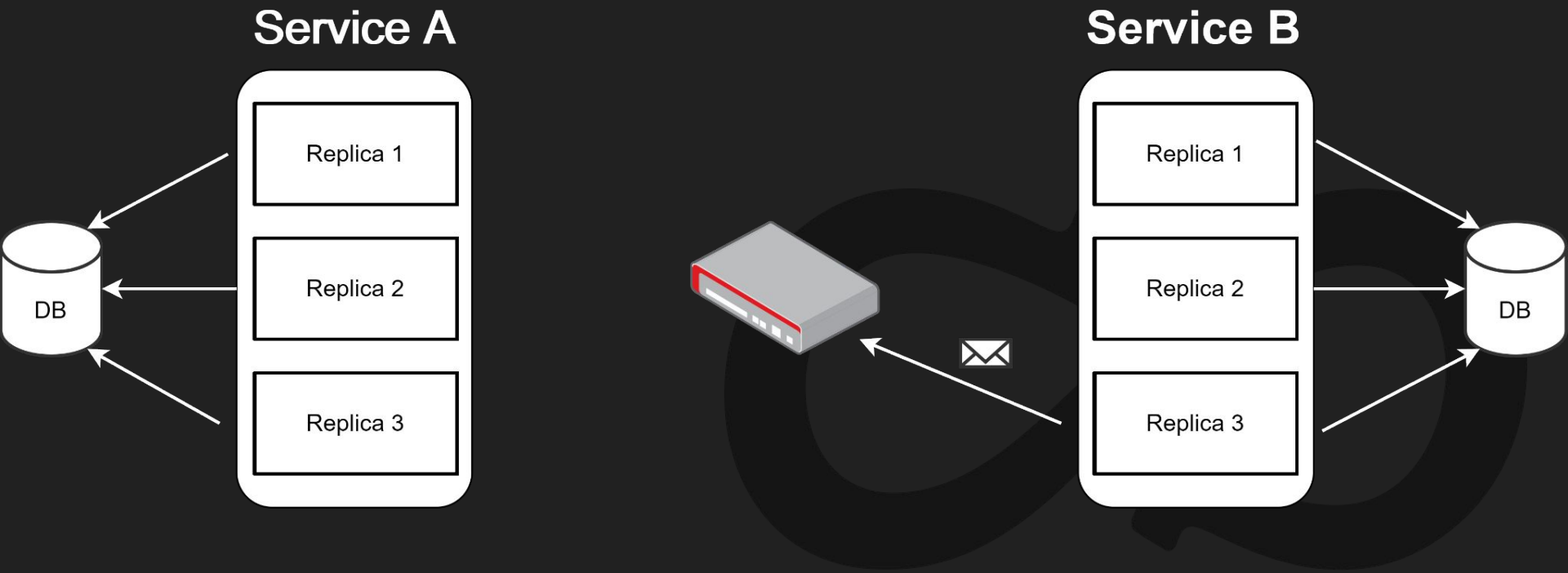
Communication Pattern - **Brokered**



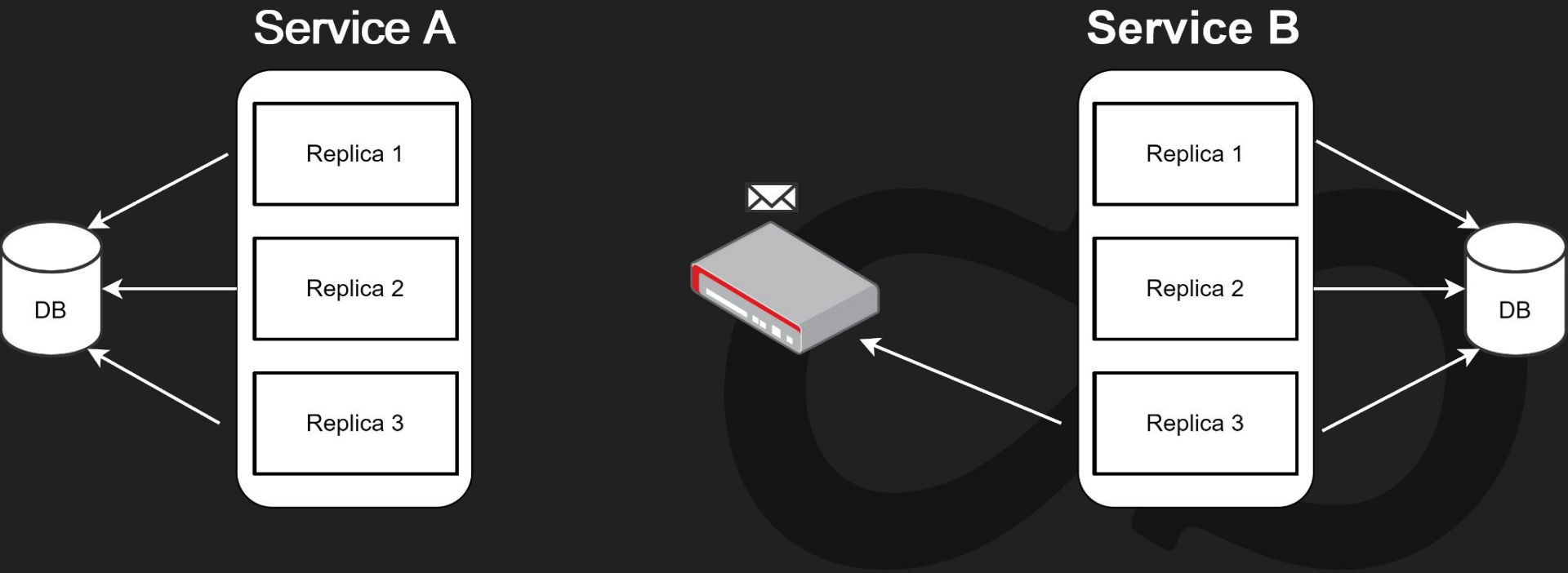
Communication Pattern - **Brokered**



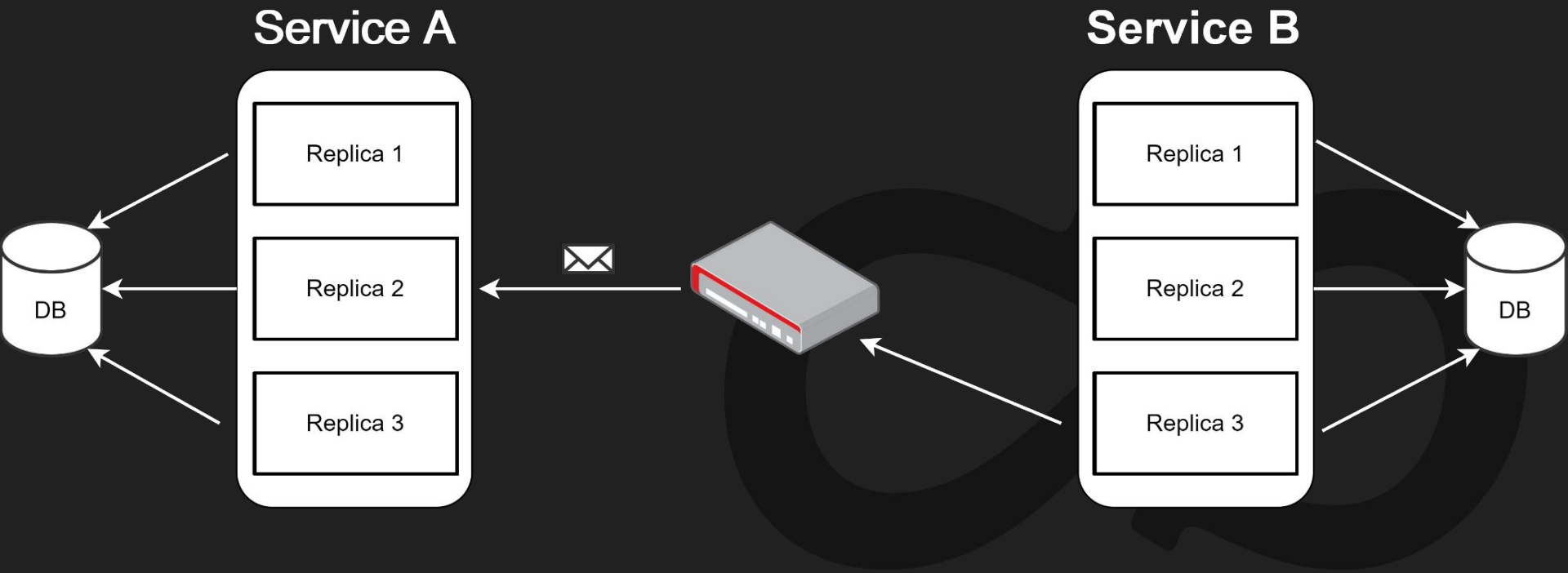
Communication Pattern - **Brokered**



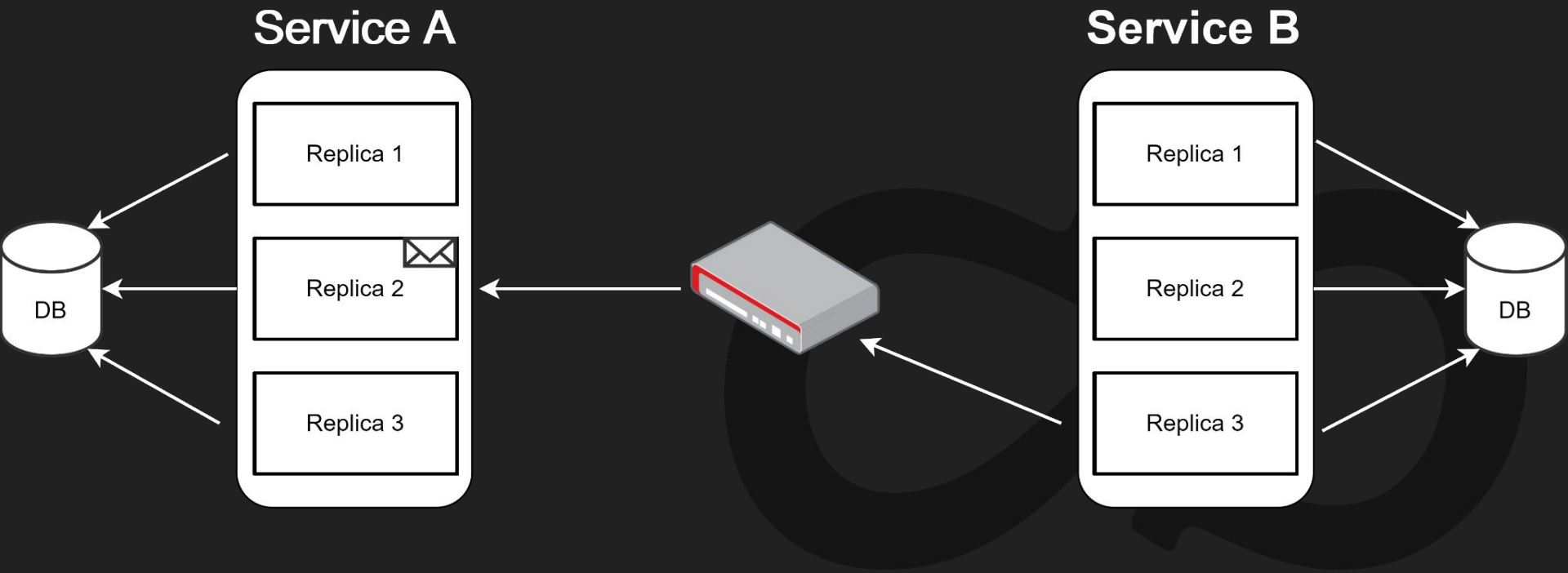
Communication Pattern - **Brokered**



Communication Pattern - **Brokered**



Communication Pattern - **Brokered**



Communication Patterns

How does this affect

the way we can implement

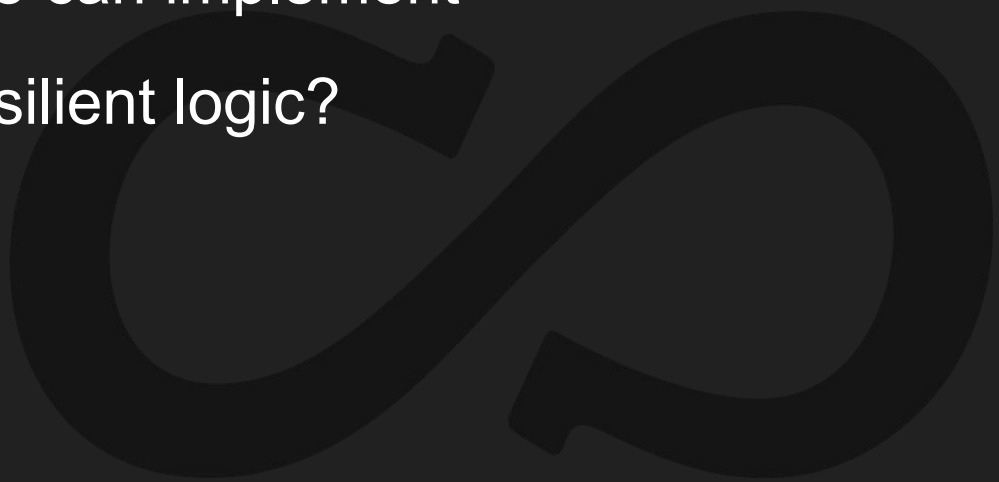
our code?

Communication Patterns

How does this affect

the way we can implement

our resilient logic?



Event-driven
Resilient Functions

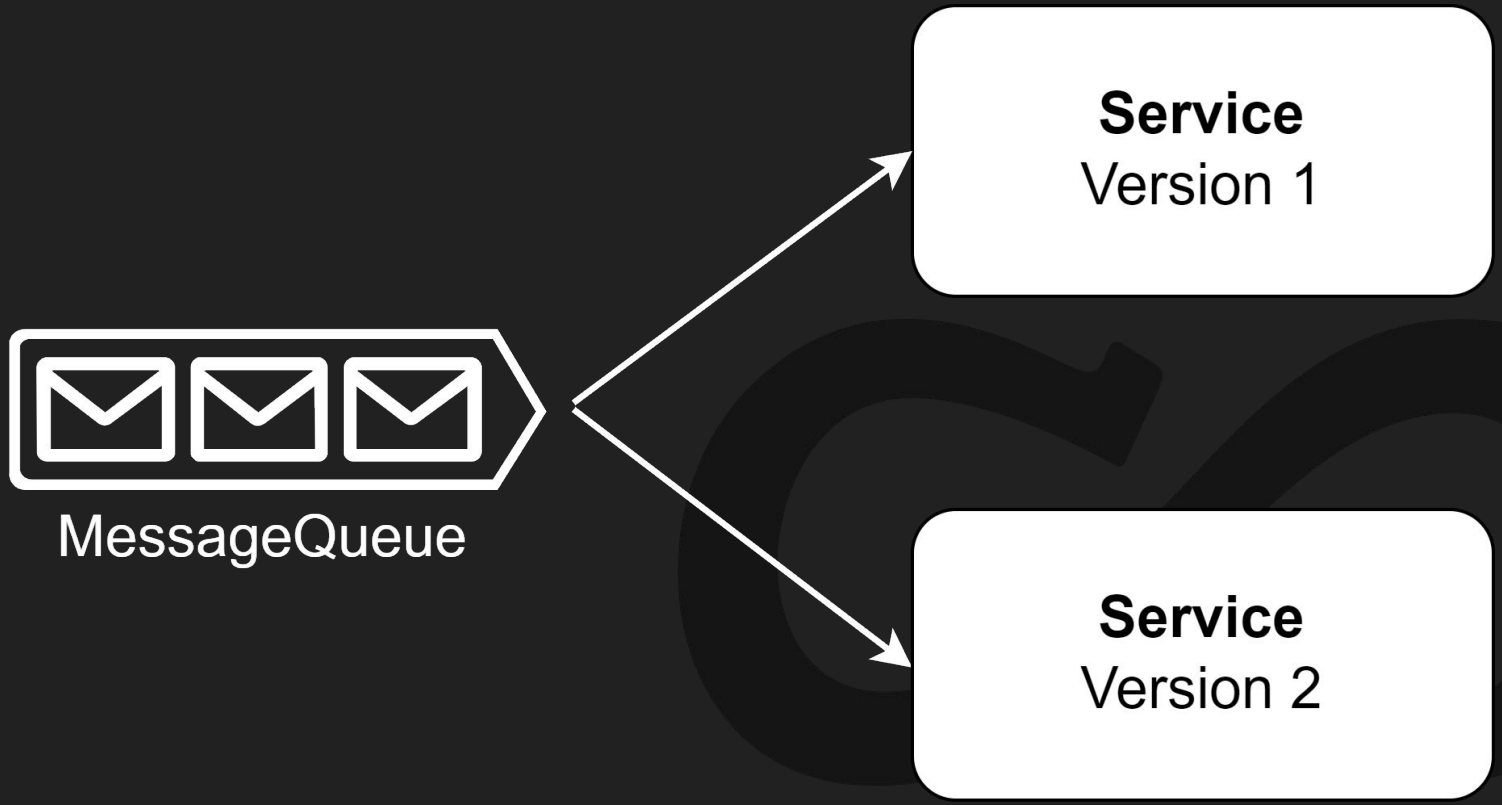
Message Queue **Panacea**

- **Message Queues** are often used for implementing *atomic business flows*
- At first glance this seems like a good idea

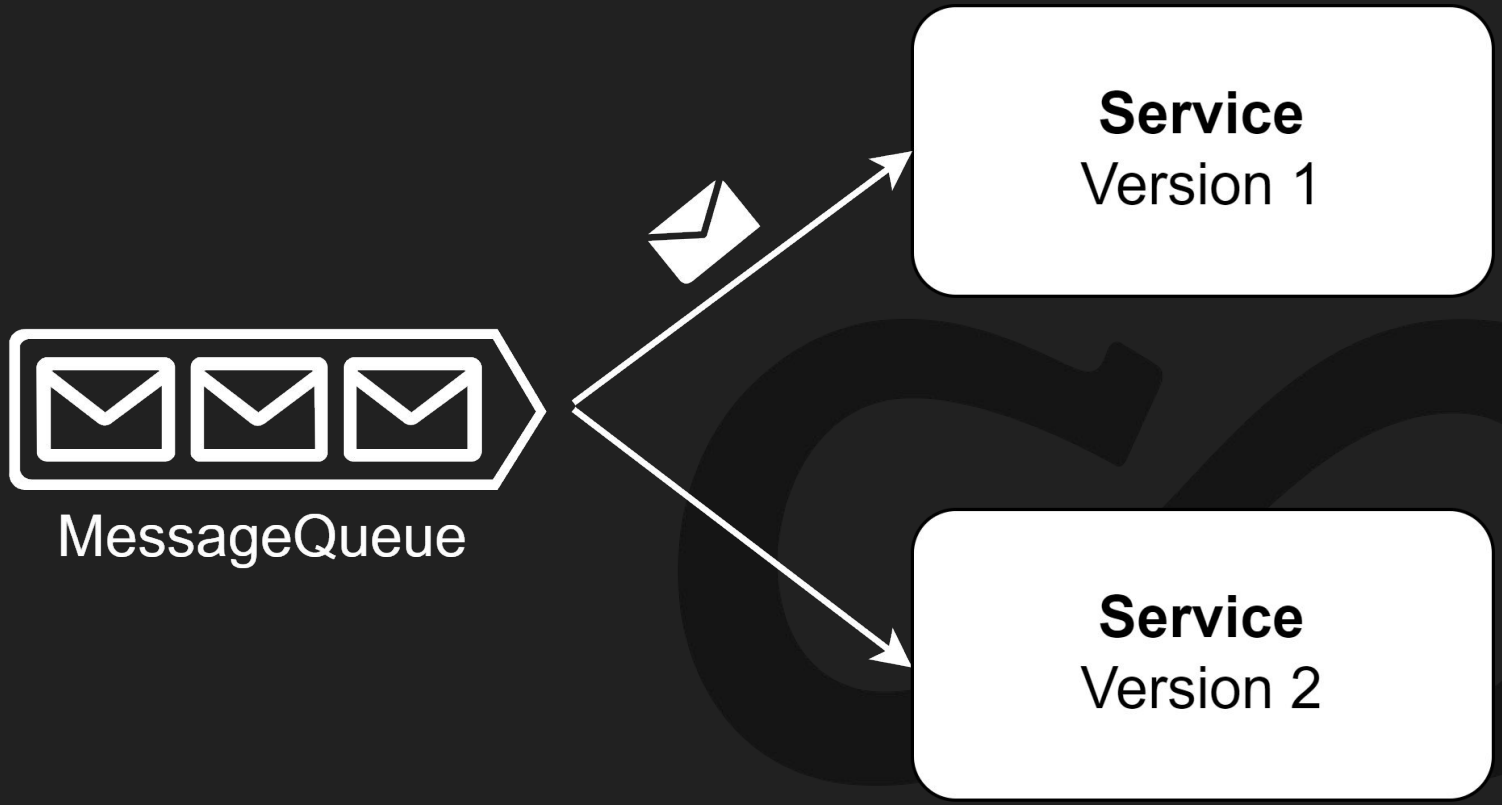
Message Queue **Panacea**

- **Message Queues** are often used for implementing *atomic business flows*
- At first glance this seems like a good idea
- However…

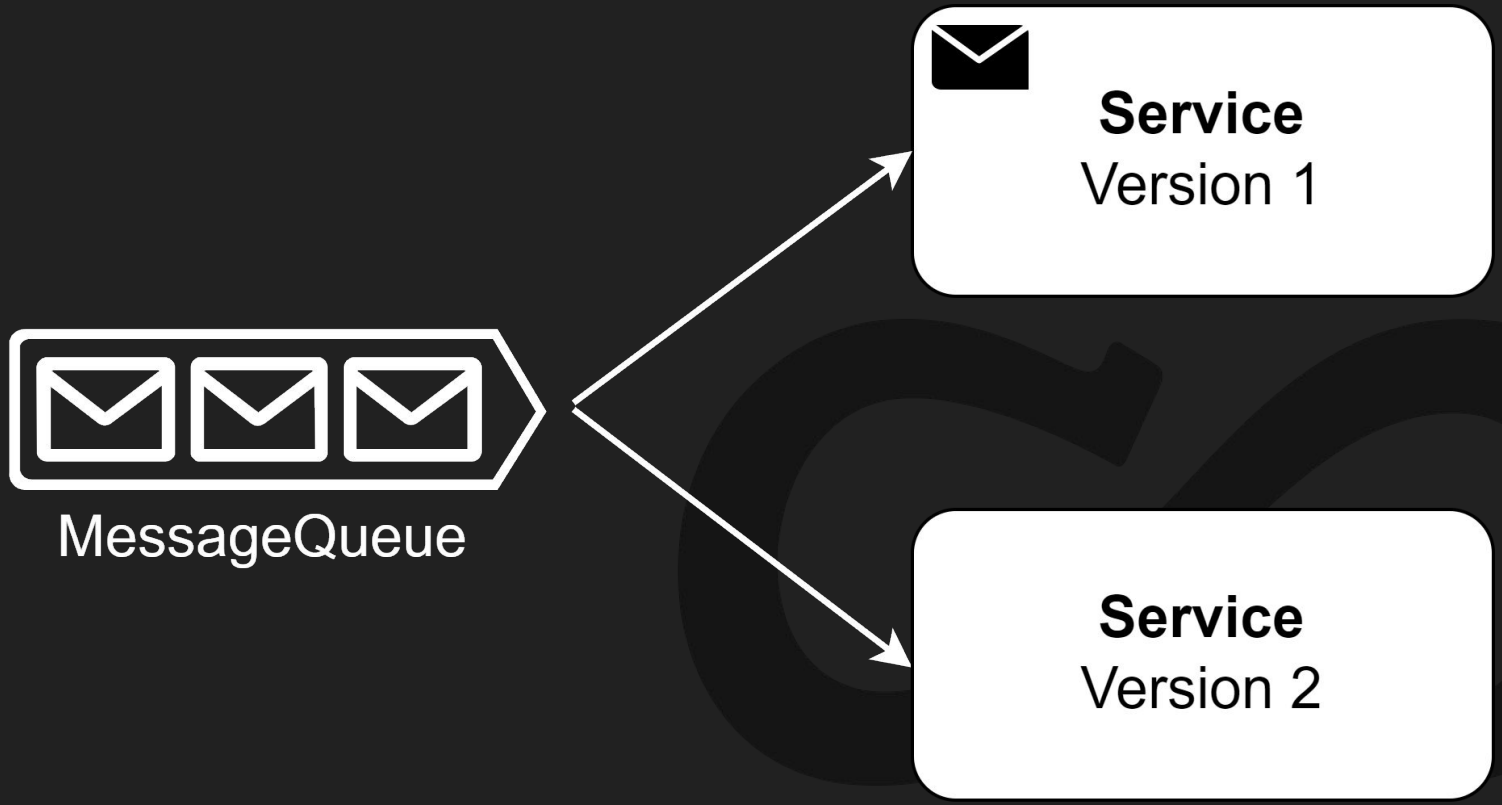
Resilient Functions - 2x **Processing Issue**



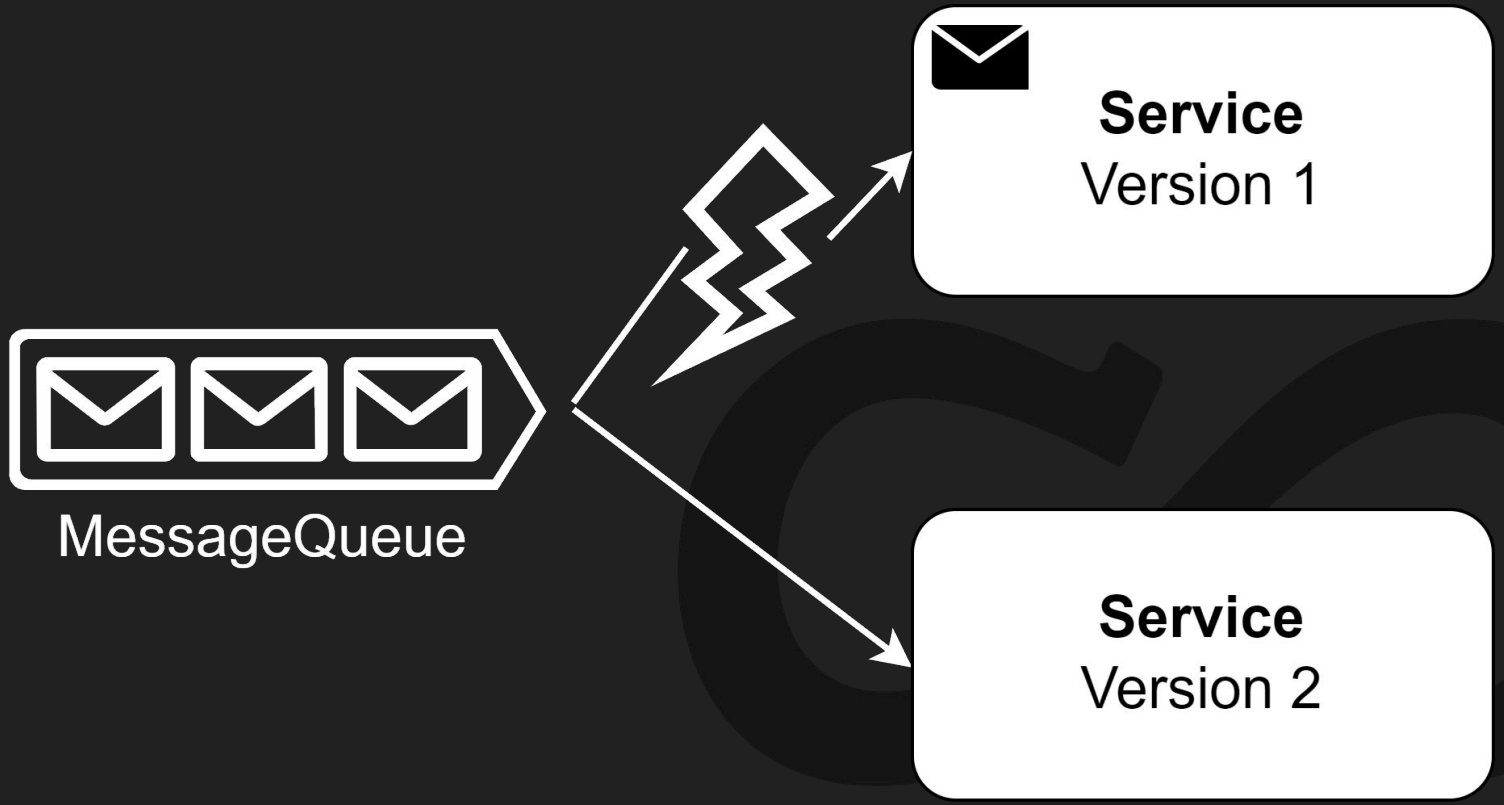
Resilient Functions - 2x **Processing Issue**



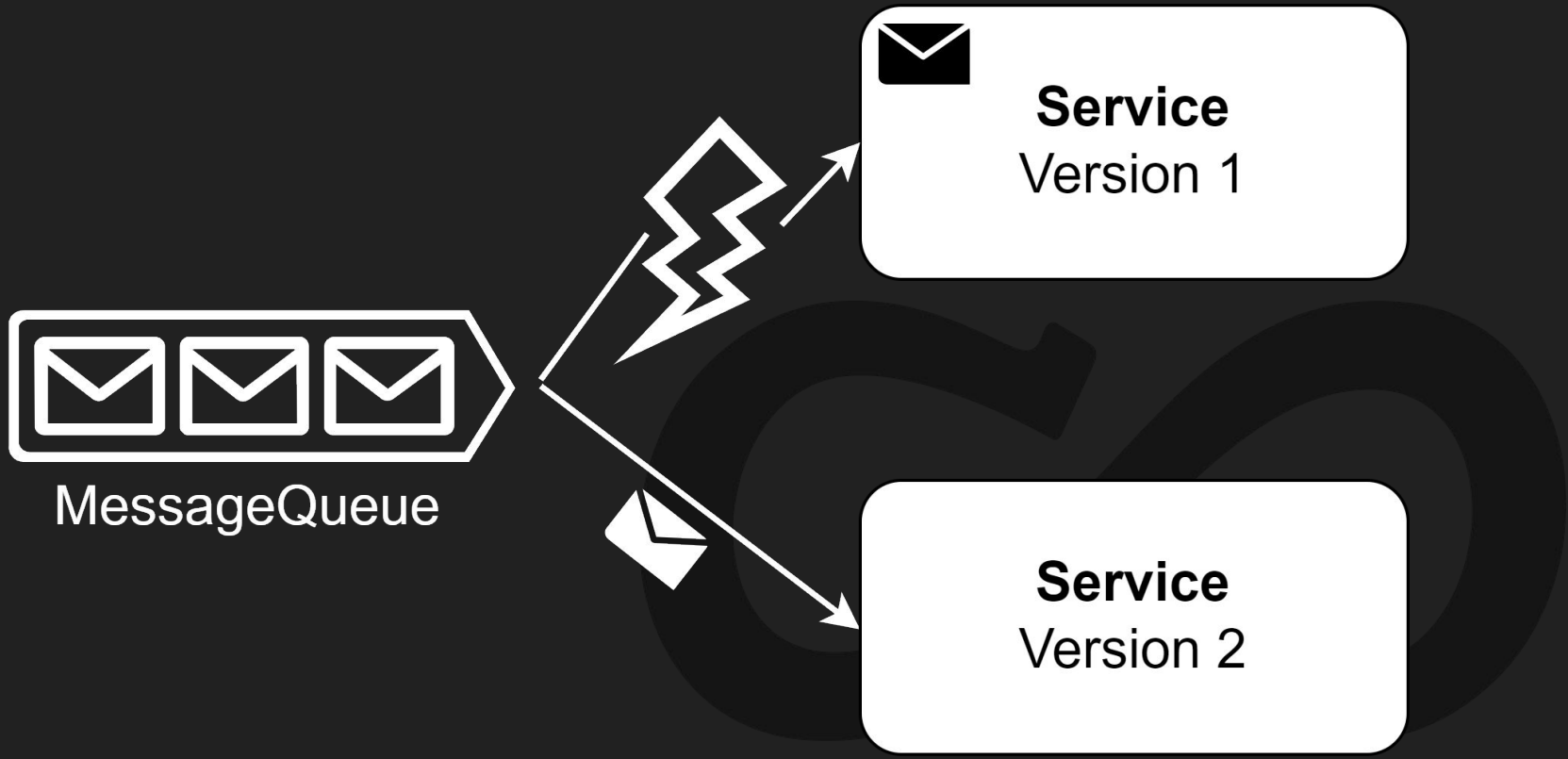
Resilient Functions - 2x **Processing Issue**



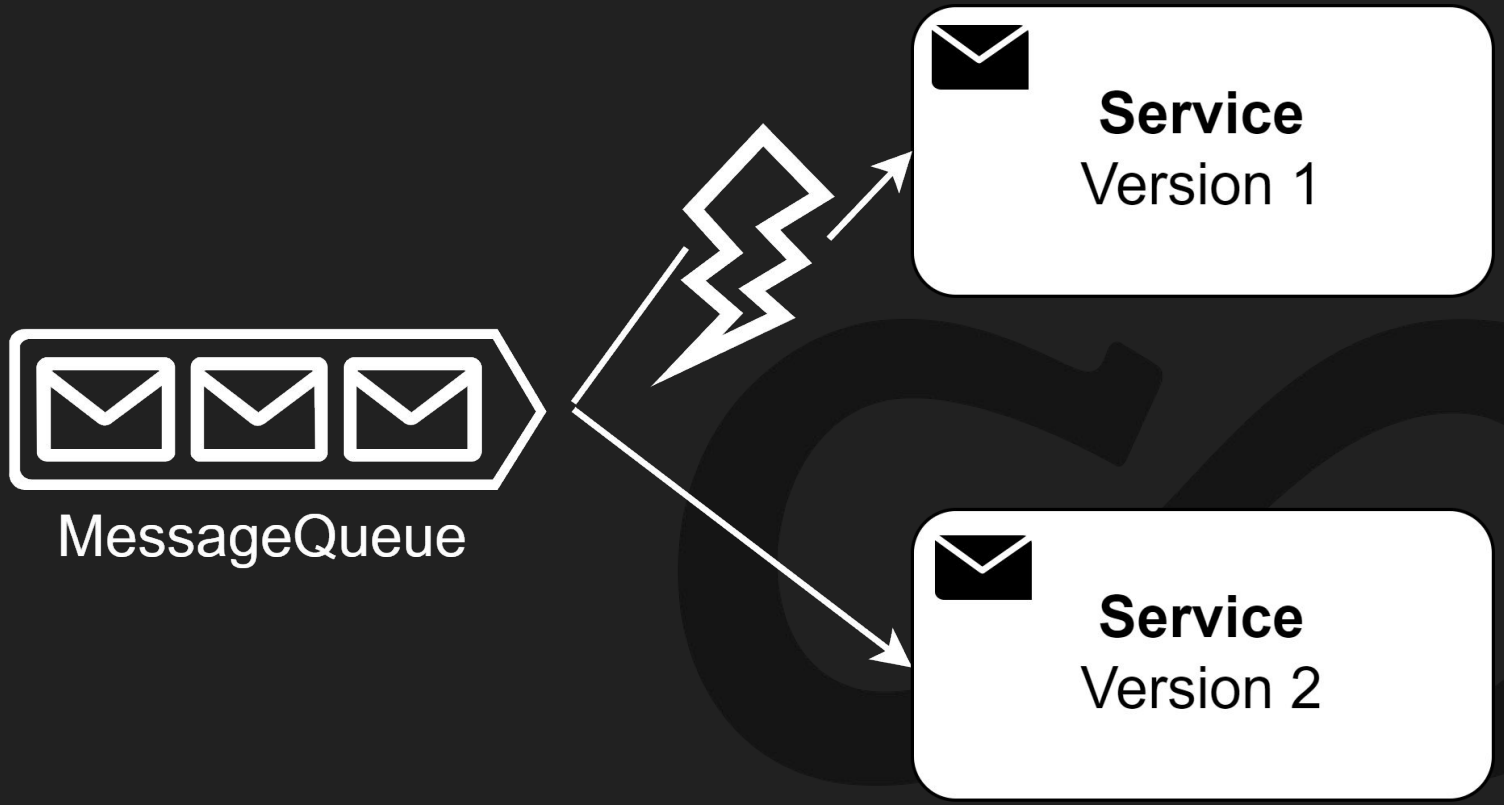
Resilient Functions - 2x **Processing Issue**



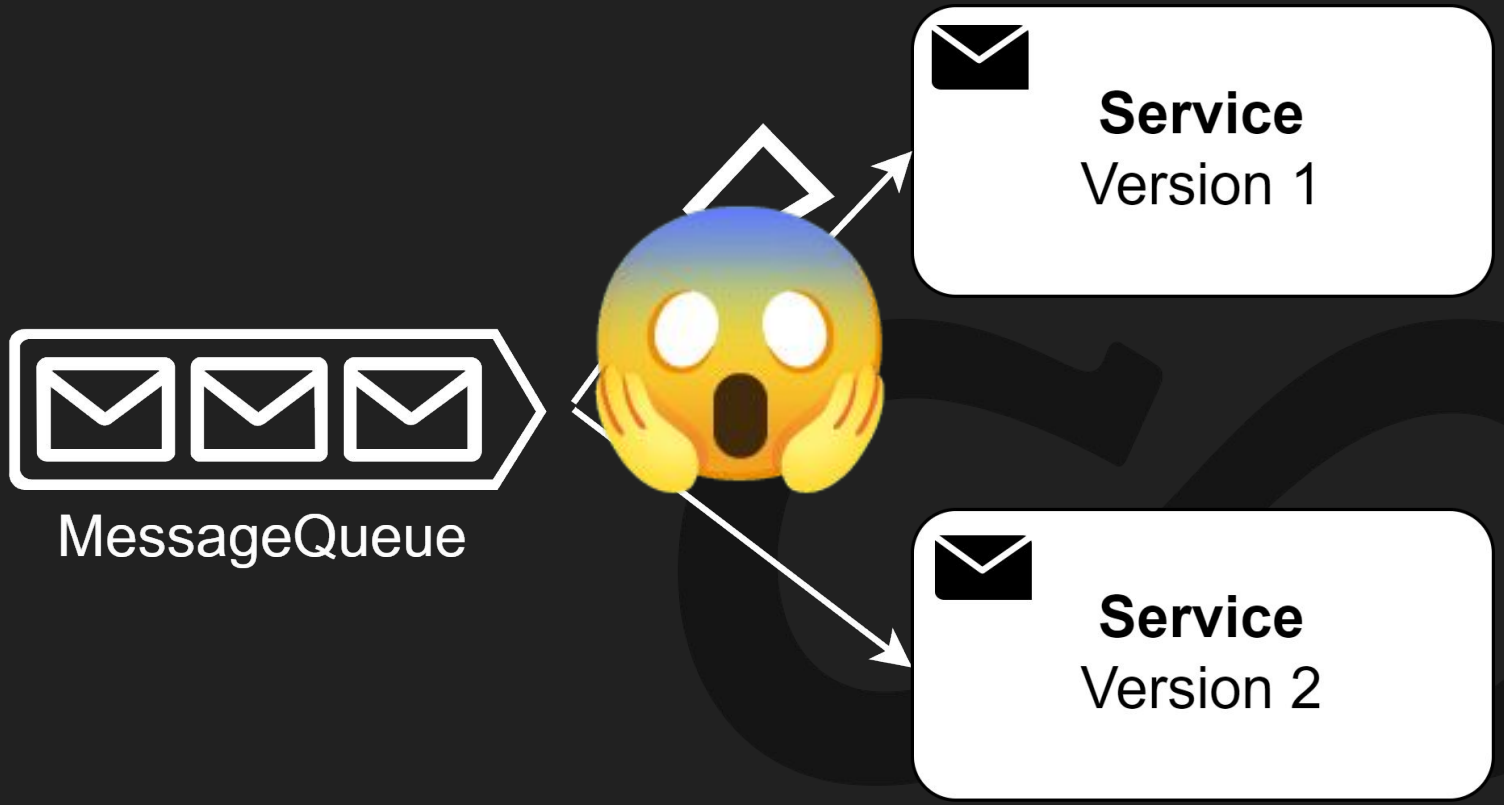
Resilient Functions - 2x **Processing Issue**



Resilient Functions - 2x **Processing Issue**



Resilient Functions - 2x **Processing Issue**



Message Queue **Panacea** - Problems

Difficult, because one still have to handle:

- **Re-deliveries** & **Out-of-order** messages
- **Poison**-messages and **Dead-Letter-Queues**
- **Synchronization** - Ensuring only one business flow is executioning at a time
- **Failures** - How to Retry or Postpone an invocation

Event-driven Flows - Classic Framework Solution

However, frameworks addressing this do exist:

- NServiceBus 
- MassTransit 

- OpenSleigh

Event-driven Flows - Classic Framework Issues

Required:

- Create one message handler per event type
- Explicitly holding progress-state field variable
- Promoting variables to instance fields
- Restricted programming model

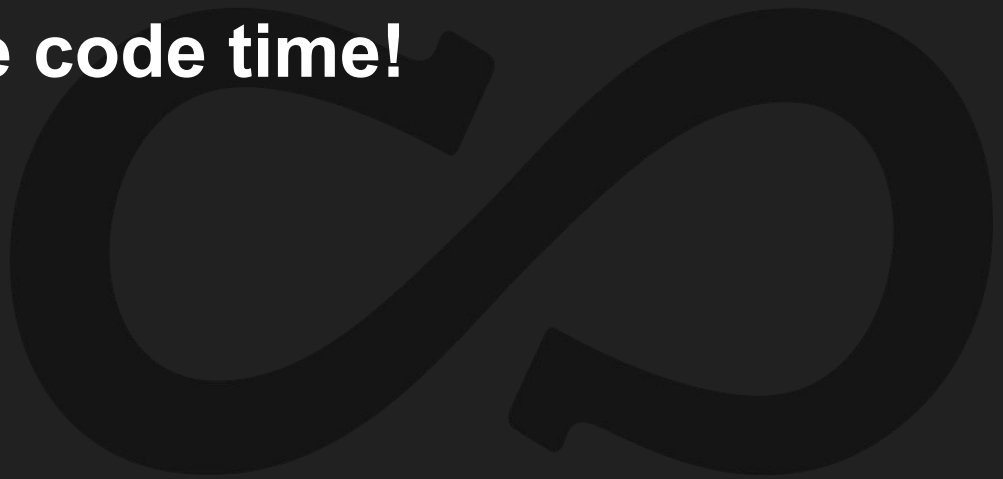
Consequences:

- Leads to fragmented and quite verbose code
- Affects understandability negatively

Event-driven Flows - Order Processing

Classic solution

Source code time!



Event-driven Flows - Classic Solution

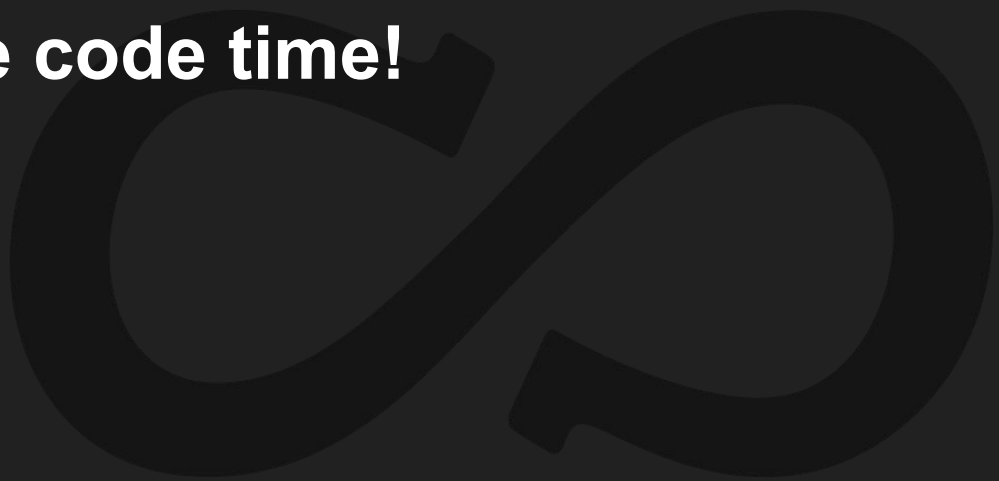
Challanges:

- Adding time-outs
- Exception handling / compensating actions
- Parallezing flows
- Changing state and **re-running** the flow

Event-driven Flows - Order Processing

Resilient Functions solution

Source code time!



Cross-cutting
Concerns

Cross-cutting Concerns - **Middleware**

Some functionality fits better inside middleware.
As opposed to, existing **explicitly** in our code.

Examples:

- **Retry-behavior** (i.e. exponential backoff)
- **Exception handling** (i.e. logging exception)
- **Publishing metrics**

Cross-cutting Concerns - **Retry Middleware**

Source code time!

Cross-cutting Concerns - **CorrelationId** (difficult)

Source code time!

Final Thoughts

He-Who-Must-Not-Be-Name

What is the really famous microservice pattern for coordinating business flows?

He-Who-Must-Not-Be-Name

What is the really famous microservice pattern for coordinating business flows?

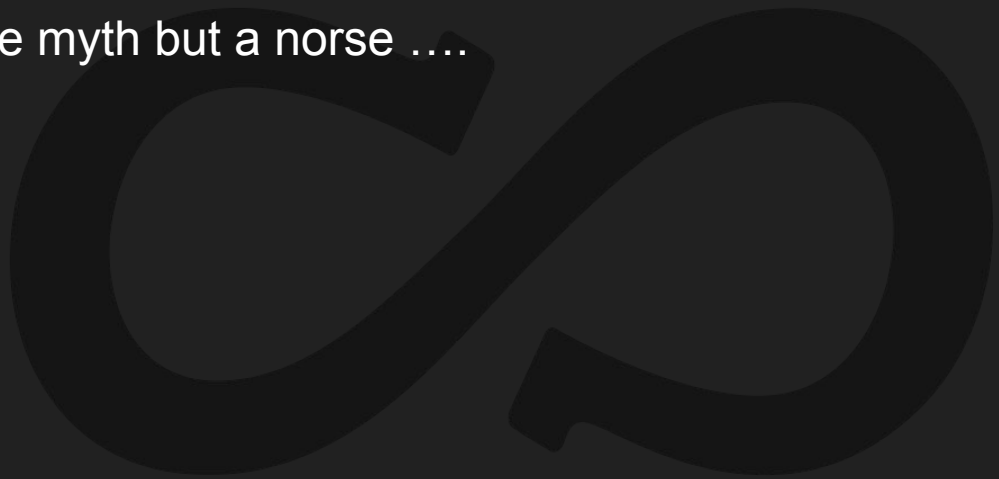
This framework is named after Odin’s horse ‘Sleipnir’

He-Who-Must-Not-Be-Name

What is the really famous microservice pattern for coordinating business flows?

This framework is named after Odin’s horse ‘Sleipnir’

It is not a norse myth but a norse ….



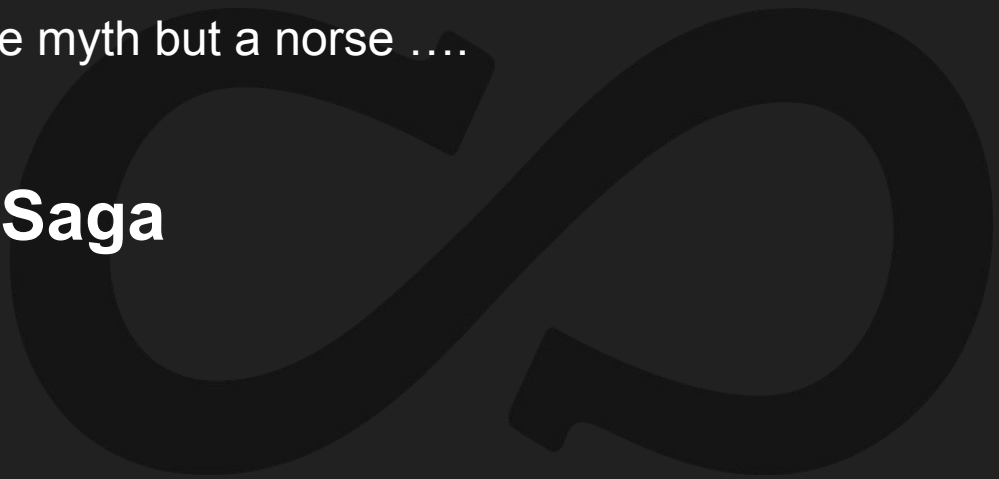
He-Who-Must-Not-Be-Name

What is the really famous microservice pattern for coordinating business flows?

This framework is named after Odin’s horse ‘Sleipnir’

It is not a norse myth but a norse ….

Saga



Resilient Functions & Sagas

We do not need to know saga terminology to solve the problem

Resilient Functions & Sagas

We do not need to know saga terminology to solve the problem

Just code…

Resilient Functions & Sagas

We do not need to know saga terminology to solve the problem

Just code…

The end; Questions?