Building **Resilient** Systems

Who am I?

- Software Developer with 11 years experience
  Currently Senior Software Developer @ Envidan 🤓
- MSc Computer Science from Aarhus University
- 2 years unfinished PhD in Distributed Systems at Stavanger University
- Open-source workflow-as-code .NET author (cleipnir.net) for 3 years
- Live in Randers with my wife and 3 children
- Like(d) to do cross fitness, hiking & play FIFA with my son in my spare time

‘Order Processing’

Example - **Order Processing**

```
public async Task ProcessOrder(Order order)

    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

🌅 Example - **Order Processing** 🌅

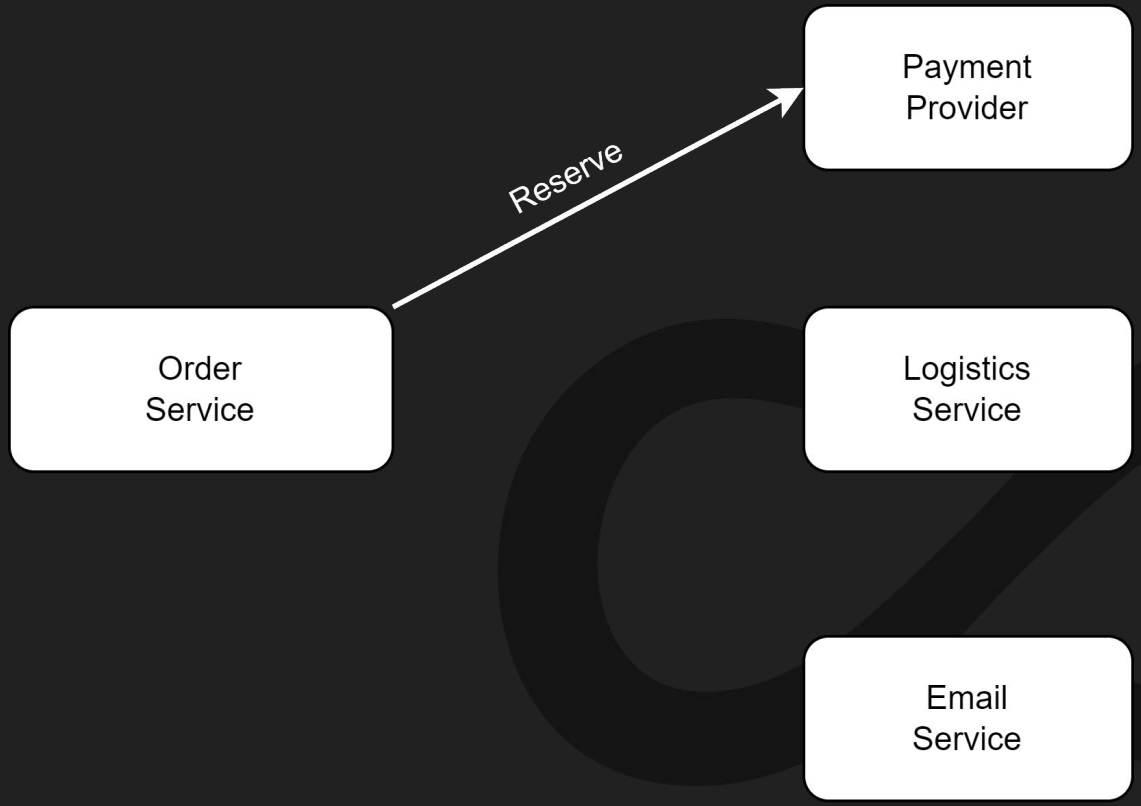
Payment Provider

Order Service

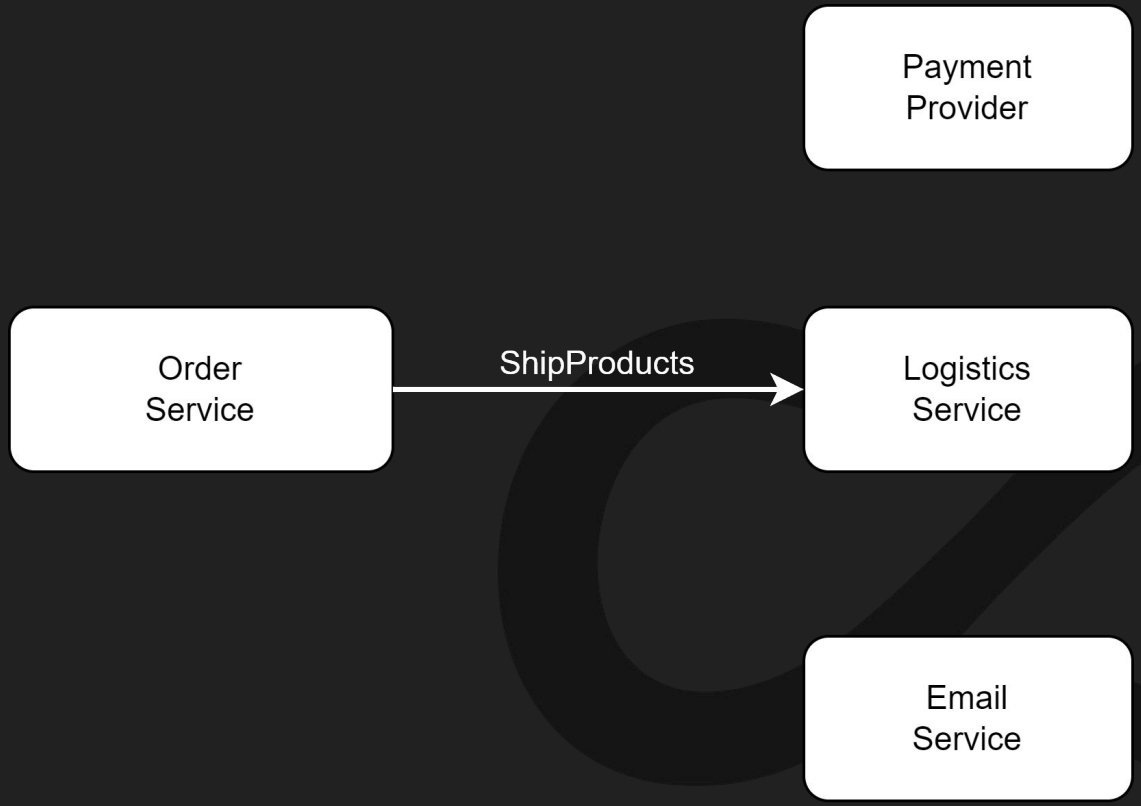
Logistics Service

Email Service

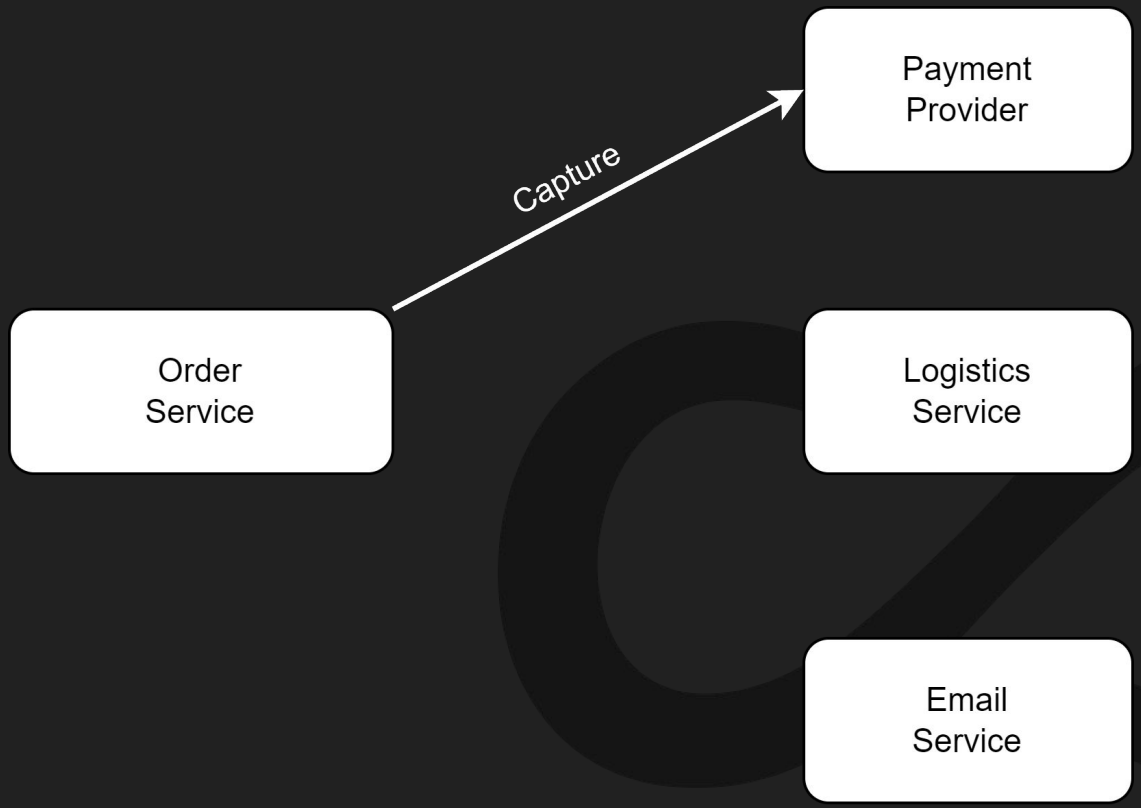
🌅 Example - **Order Processing** 🌅



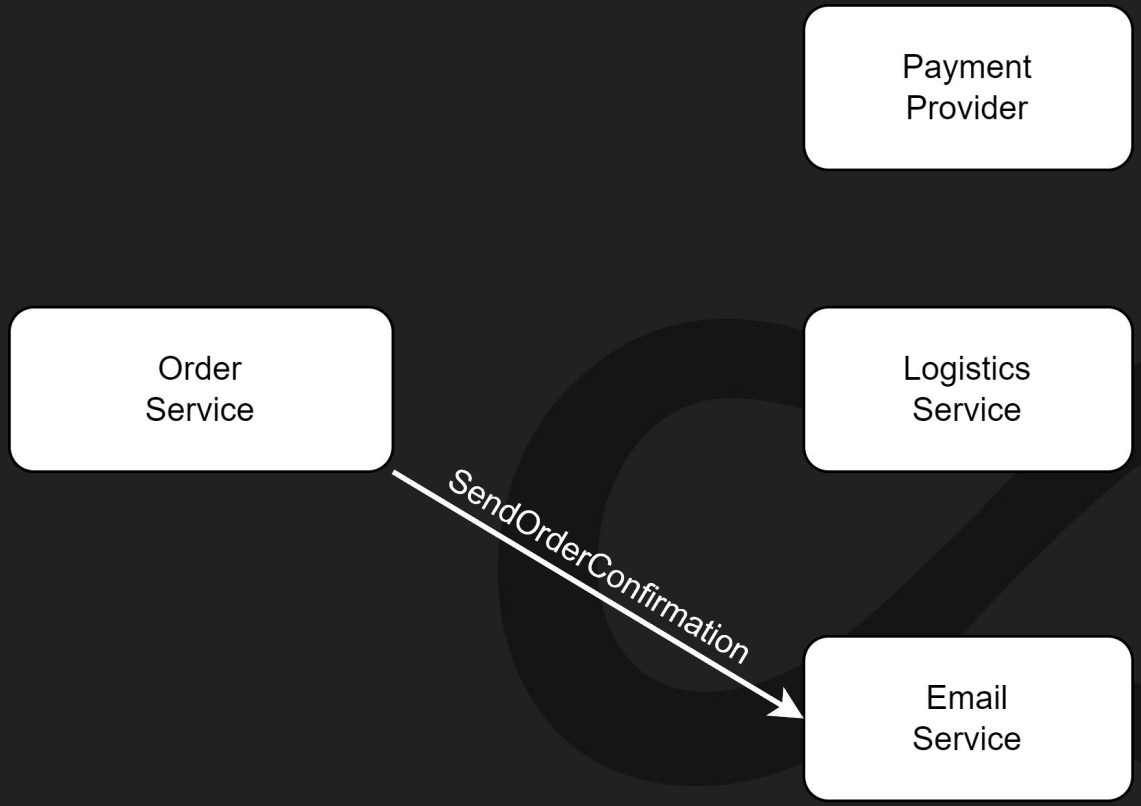
🌅 Example - **Order Processing** 🌅



🌅 Example - **Order Processing** 🌅



🌅 Example - **Order Processing** 🌅



🌅 Example - **Order Processing** 🌅

Payment Provider

Order Service

Logistics Service

Email Service

Example - **Order Processing -** *What can go wrong?*

```
public async Task ProcessOrder(Order order)

    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Example - **Order Processing -** *What can go wrong?*

```
public async Task ProcessOrder(Order order)

    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

<u>Question:</u>

What can go wrong if the process **crashes** at any point during the execution?

Example - **Order Processing -** *What can go wrong?*

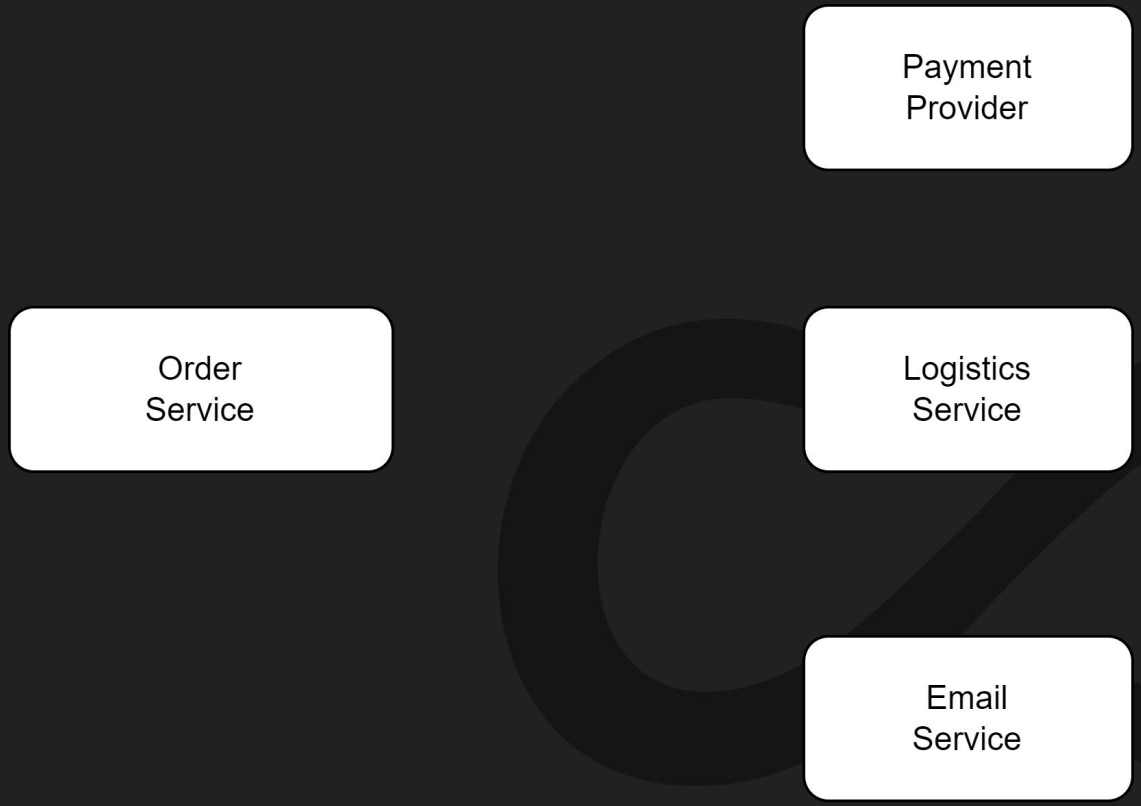
```
public async Task ProcessOrder(Order order)

    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

<u>Question:</u>

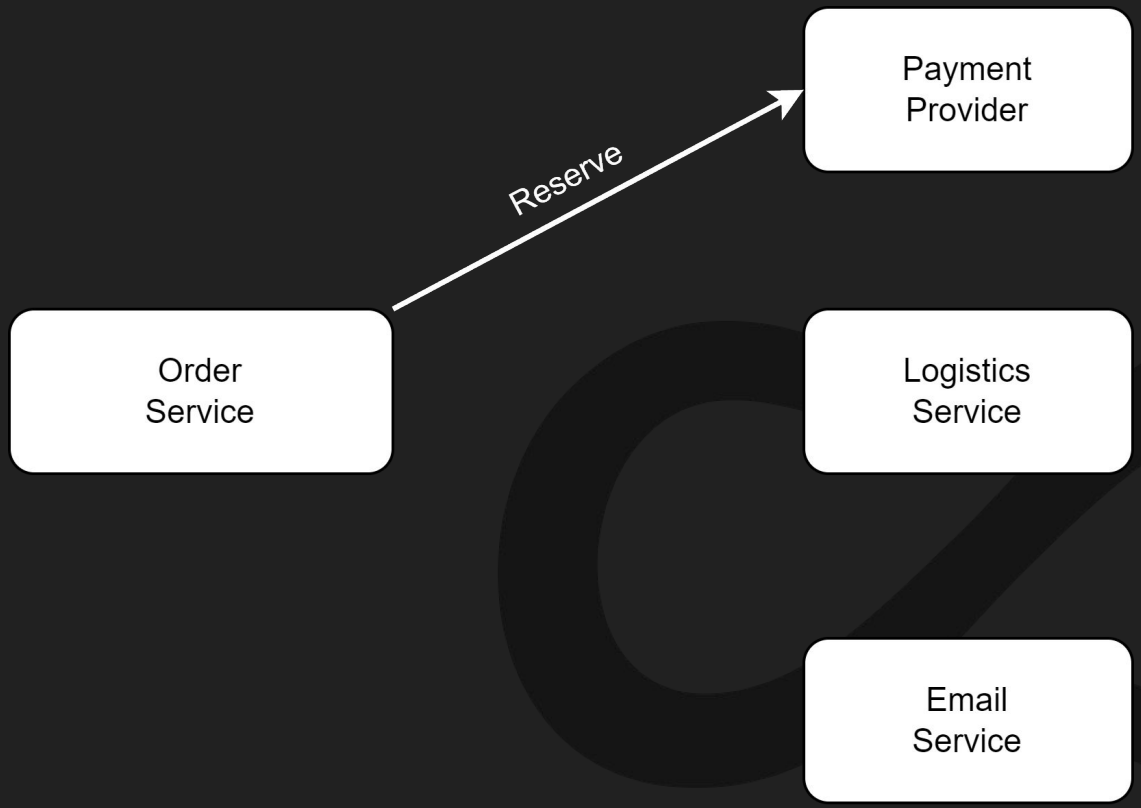
What can go wrong if the process **crashes** at any point during the execution?

How do we **observe** a crash?

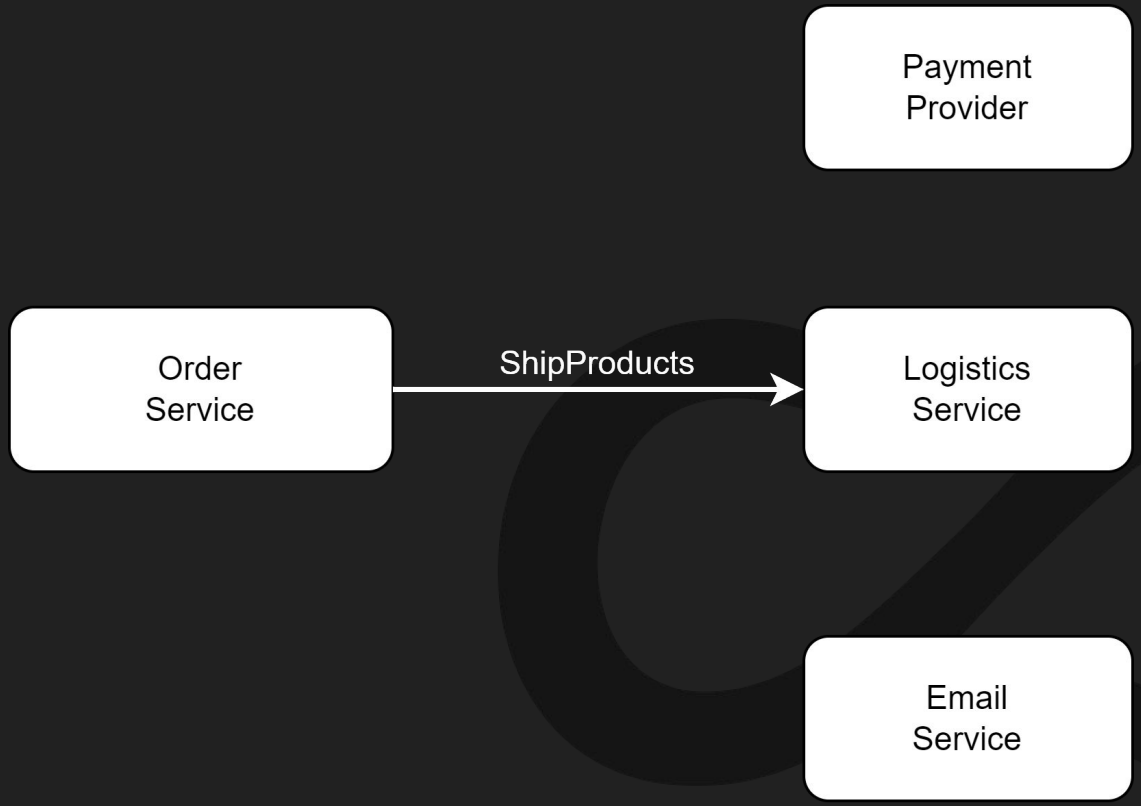
💀 Example - **Order Processing** 💀



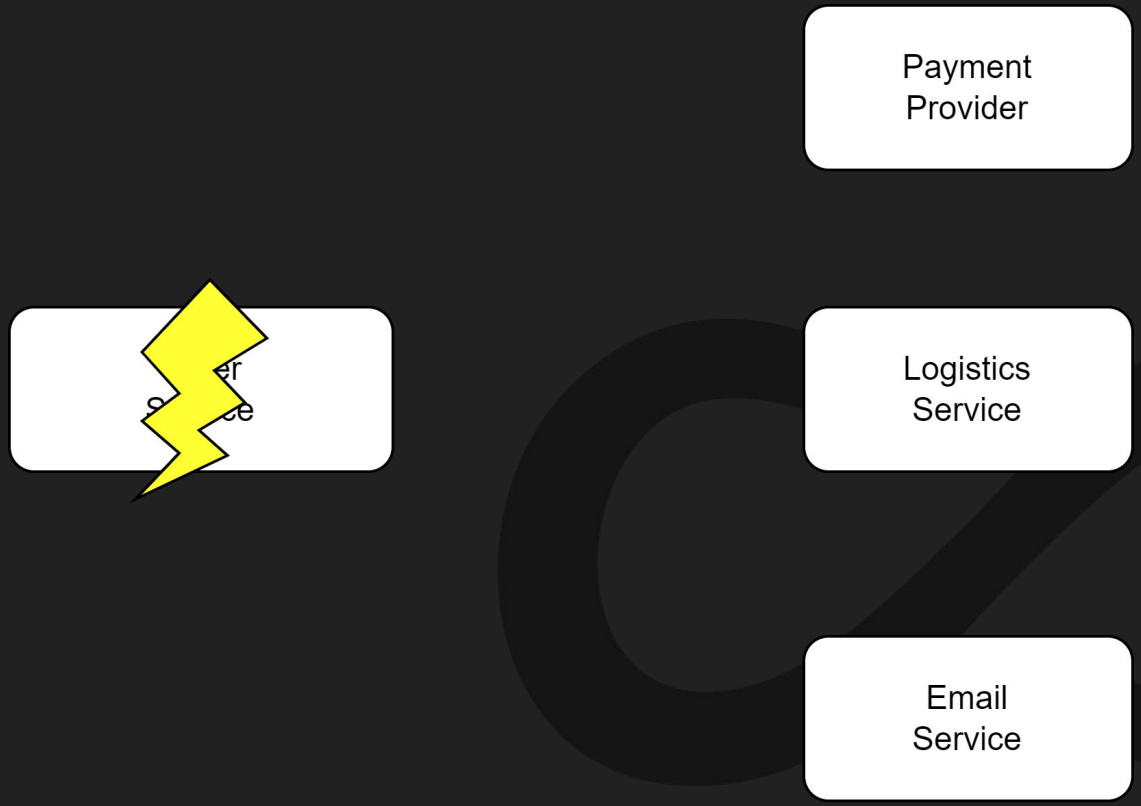
💀 Example - **Order Processing** 💀



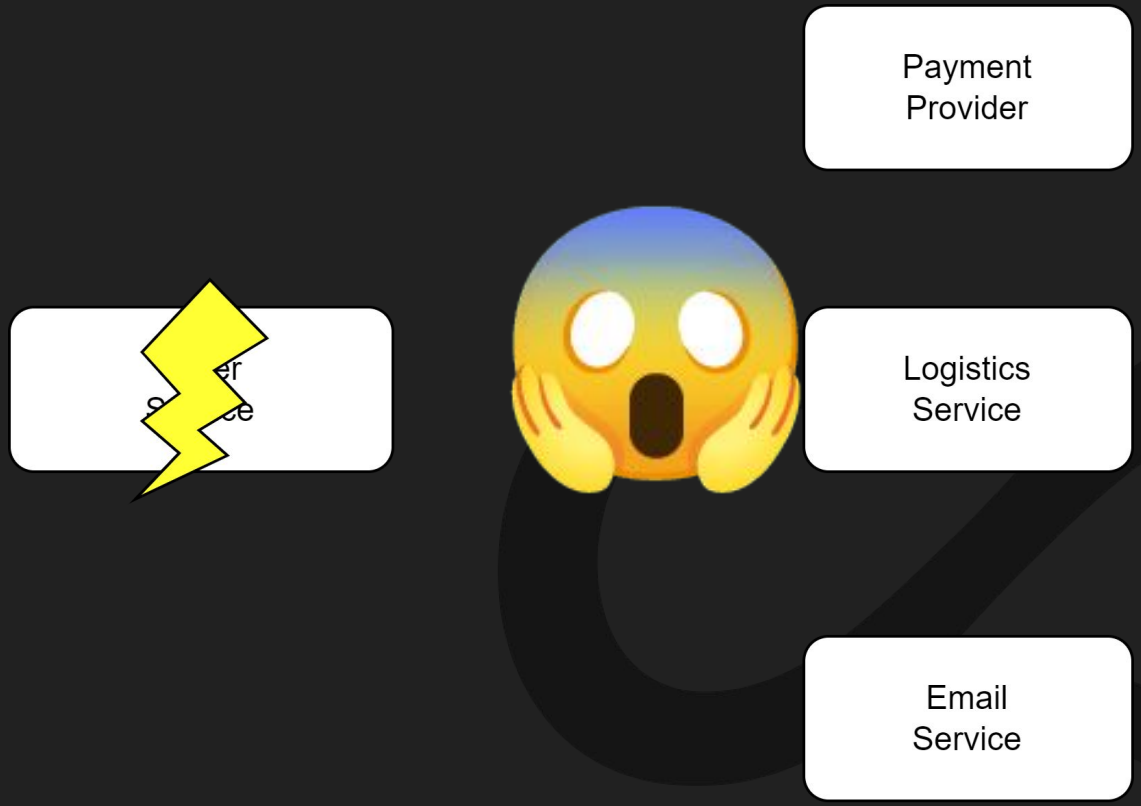
💀 Example - **Order Processing** 💀



💀 Example - **Order Processing** 💀



💀 Example - **Order Processing** 💀



Example - **Order Processing** - *Just restart?*

Can we just **restart** from the top after a crash?

```
public async Task ProcessOrder(Order order)

    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Example - **Order Processing** - *Just restart?*

Common **solutions**

```
public async Task ProcessOrder(Order order)

    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Example - **Order Processing** - *Just restart?*

Common **solutions**

```
public async Task ProcessOrder(Order order)
```

Message Queues (RabbitMQ, Kafka)

```
    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Example - **Order Processing** - *Just restart?*

Common **solutions**

```
public async Task ProcessOrder(Order order)
```

Message Queues (RabbitMQ, Kafka)

```
    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);
```

ServiceBus (Wolverine, MassTransit)

```
    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Example - **Order Processing** - *Just restart?*

Common **solutions**

```
public async Task ProcessOrder(Order order)
    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Message Queues (RabbitMQ, Kafka)

JobScheduler (Hangfire)

ServiceBus (Wolverine, MassTransit)

Example - **Order Processing** - *Just restart?*

Common **solutions**

```
public async Task ProcessOrder(Order order)
    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Message Queues (RabbitMQ, Kafka)

Outbox pattern

JobScheduler (Hangfire)

ServiceBus (Wolverine, MassTransit)

Example - **Order Processing** - *Just restart?*

Common **solutions**

```
public async Task ProcessOrder(Order order)
    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Message Queues (RabbitMQ, Kafka)

Outbox pattern

JobScheduler (Hangfire)

ServiceBus (Wolverine, MassTransit)

Listen-to-yourself pattern

Example - **Order Processing** - *Just restart?*

Common **solutions**

```
public async Task ProcessOrder(Order order)
    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Message Queues (RabbitMQ, Kafka)

Outbox pattern

Workflow-as-Code

JobScheduler (Hangfire)

ServiceBus (Wolverine, MassTransit)

Listen-to-yourself pattern

What can go **Wrong**?

Transient Failures 🤔

Transient failures can be handled in code using **backoff strategies** (i.e. with **Polly**)

- External service is down or overloaded
- Database deadlock

Transient Failures 🤔

Transient failures can be handled in code using **backoff strategies** (i.e. with **Polly**)

- External service is down or overloaded
- Database deadlock

<u>Question:</u> Any issues when waiting arbitrarily long for a service to become available (i.e. using exponential backoff)?

Fatal Failures 🤔

<u>Unrecoverable errors or benign restarts</u>:

- Process crash
  - Process killed by OS (out-of-memory, stack overflow)
  - Underlying physical machine loses power

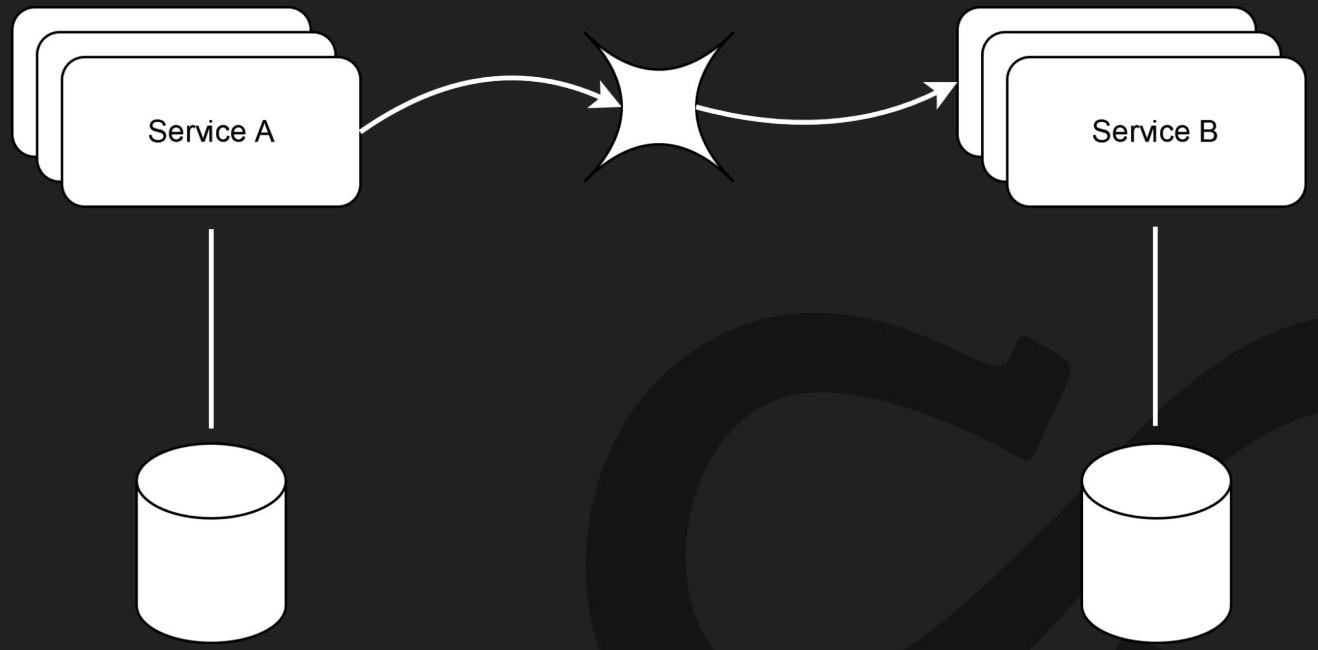
- Code is re-deployed (graceful shutdown is not enough)

The **Conundrum**… 🤔

- Thus, our code may **crash** and **stop** executing arbitrarily (either due to **hardware failures** or **deployments**)
- So, how do we ensure our *business flows* **complete** *correctly*?

Distributed Concepts 101

Micro-service Architecture



- **Zero-**downtime
- **Replicated** Services

Concepts - **Distributed Transactions**

We do <u>not</u> have distributed transaction support at our disposal.

Thus, the following becomes **tricky**:

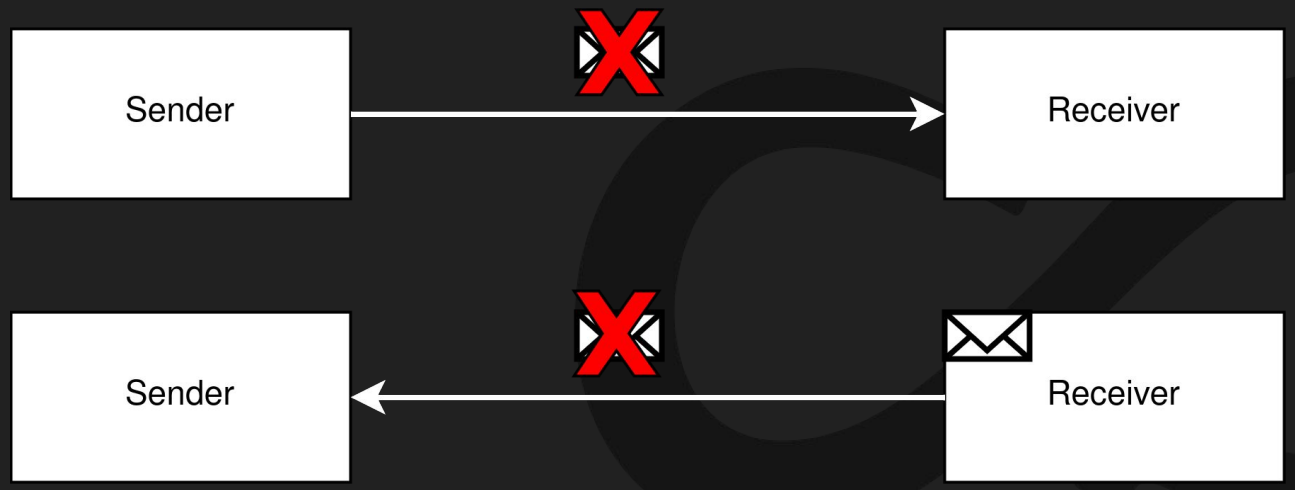
```
public void Handle(CreateOrder createOrder)

    var order = ValidateAndConvertToOrder(createOrder)

    SaveToDatabase(order)

    PublishMessage(order)
```

Concepts - **At-Most-Once & At-Least-Once**

When communicating over a unreliable network we **always** have to choose between *at-most-once* and *at-least-once* delivery. Why?

Concepts - **At-Most-Once & At-Least-Once**

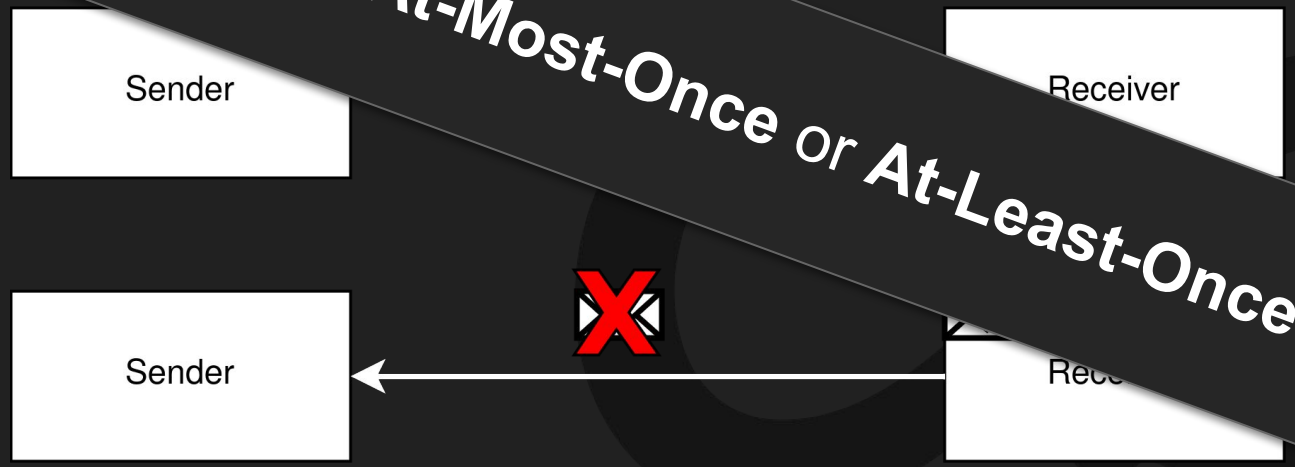
When communicating over a unreliable network we **always** have to choose between *at-most-once* and *at-least-once* delivery. Why?



Concepts - **At-Most-Once & At-Least-Once**

W ver a unreliable network we **always** have to choose
b *ast-once* delivery. Why?

Code is also **At-Most-Once** or **At-Least-Once!**



Concepts - **Exactly-Once** & **Idempotency**

We ‘nullify’ **At-Least-Once** effects (when we can) with **idempotency keys**.

Concepts - **Exactly-Once** & **Idempotency**

We ‘nullify’ **At-Least-Once** effects (when we can) with **idempotency keys**.

<u>Challenge:</u> *“For how long do we keep idempotency keys at the receiver?”*

Concepts - **Determinism**

Does a **restart** of an **crashed** business flow end up in the same state as previously or does it ***diverge***?

```
public async Task ProcessOrder(Order order)

  var transactionId = Guid.NewGuid();

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

Concepts - **Determinism**

There are many examples of **non-deterministic** operations:

- `Guid.NewGuid()`
- `DateTime.Now`
- `Random.Shared.Next(0, 1_000)`

Concepts - **Determinism**

End-points can also be **non-deterministic**:

```
RocketSender.FireRocket()
```

What do we **want?**

Resilient Programming Solution - Quality Attributes

- What are **aspirational** attributes for a resilient solution?

Resilient Programming - Quality Attributes

- **Synchronization**
  The same business flow instance must not be able execute concurrently

```
public async Task ProcessOrder(Order order)

   var transactionId = Guid.NewGuid();

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

   var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

   await _paymentProviderClient.Capture(transactionId);

   await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Resilient Programming - Quality Attributes

- **State Support**
  Intermediary state must be persistable in order to ensure correct re-execution after a crash/restart.

```
public async Task ProcessOrder(Order order)

   var transactionId = Guid.NewGuid();

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

   var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

   await _paymentProviderClient.Capture(transactionId);

   await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Resilient Programming - Quality Attributes

- **Programming Model**

  How awkward is it to implement business flow?

```
public async Task ProcessOrder(Order order)

   var transactionId = Guid.NewGuid();

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

   var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

   await _paymentProviderClient.Capture(transactionId);

   await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Resilient Programming - Quality Attributes

- **Programming Model**

  Can we suspend the current invocation to free up resources?

```
public async Task ProcessOrder(Order order)

   var transactionId = Guid.NewGuid();

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

   var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

   await _paymentProviderClient.Capture(transactionId);

   await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Resilient Programming - Quality Attributes

- **Programming Model**

  Can we control space consumption?

```
public async Task ProcessOrder(Order order)

   var transactionId = Guid.NewGuid();

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

   var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

   await _paymentProviderClient.Capture(transactionId);

   await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Resilient Programming - Quality Attributes

- **Programming Model**

  Can we communicate **externally** with an executing flow?

```
public async Task ProcessOrder(Order order)

   var transactionId = Guid.NewGuid();

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

   var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

   await _paymentProviderClient.Capture(transactionId);

   await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Resilient Programming - Quality Attributes

- **Locatability**

  Can we inspect the current state of a workflow instance?

- **Discoverability**

  If a flow has crashed can we detect it?

Resilient Programming - Quality Attributes

- **Versioning**
  Can we fix bugs and add new functionality to existing executing flows?

```
public async Task ProcessOrder(Order order)

    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Resilient Programming - Quality Attributes

- **Performance**

  Do we have the necessary flexibility to ensure good performance?

Resilient Programming - Quality Attributes

1. Synchronization
2. Intermediary State Support
3. Programming Model
4. Locatability & Discoverability
5. Versioning
6. Performance

Industry Approaches

Job Scheduler - **Hangfire & Quartz**

Job schedulers such as **Hangfire** and **Quartz** can be used to ensure a business process completes.

Job Scheduler - **Hangfire & Quartz**

They both ensure **synchronization** and **retry** failed executions.

Job Scheduler - **Hangfire & Quartz**

Unfortunately, out-of-the-box they only provide support for very simple business flow. **No support for**: (1) *Intermediary state*, (2) *suspension* (3) *external notifications*.

Job Scheduler - **Hangfire & Quartz**

Business flow must be **idempotent** and **deterministic**.

Job Scheduler - **Hangfire & Quartz**

Business flow must be **idempotent** and **deterministic**.

```
public async Task ProcessOrder(Order order)

    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Message **Queues**

A **Message Queue** provide a way to **decouple** and **reliability** communicate between **services**.

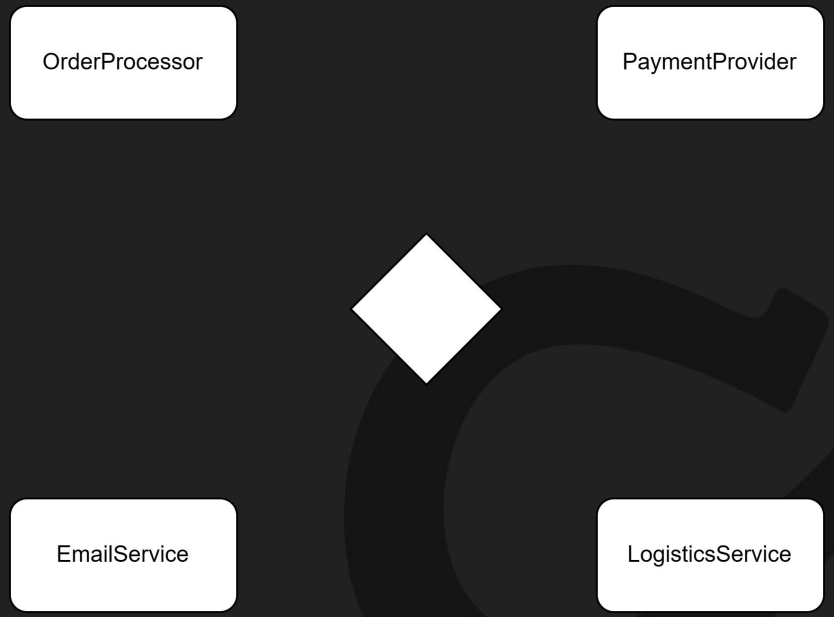
Message **Queues**

Out-of-the-box **message queues** provides **rudimentary support** for implementing **resilient** business flows.

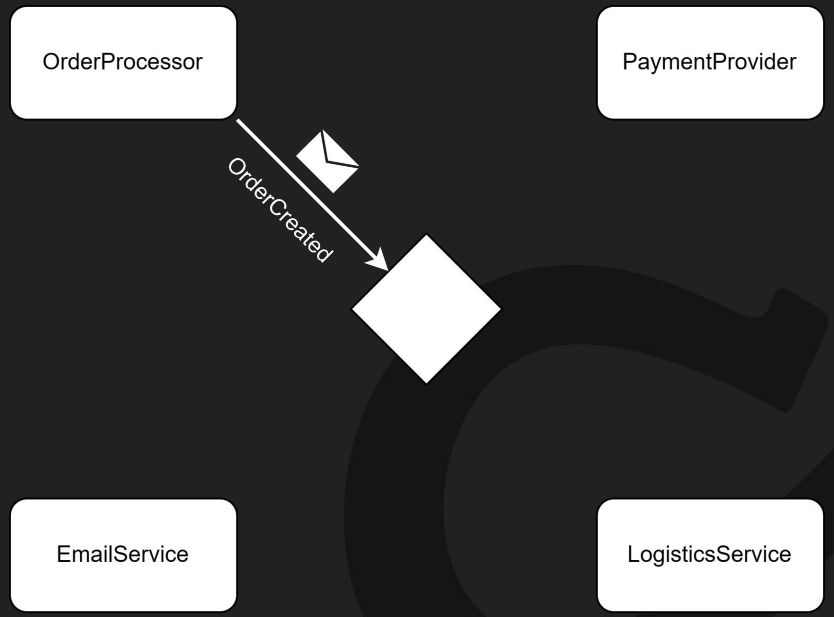
Message **Queues**

This is also called: **Choreography**

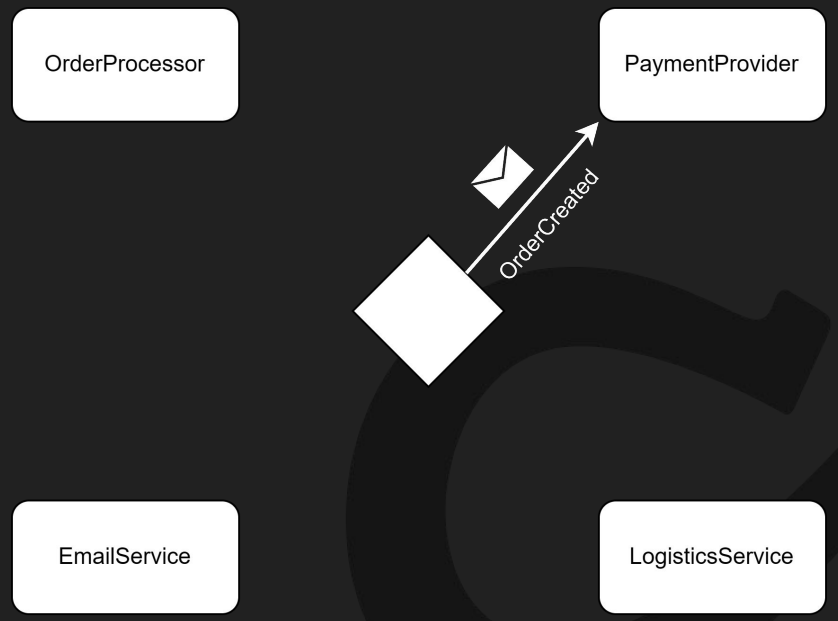
Choreography - Order Processing



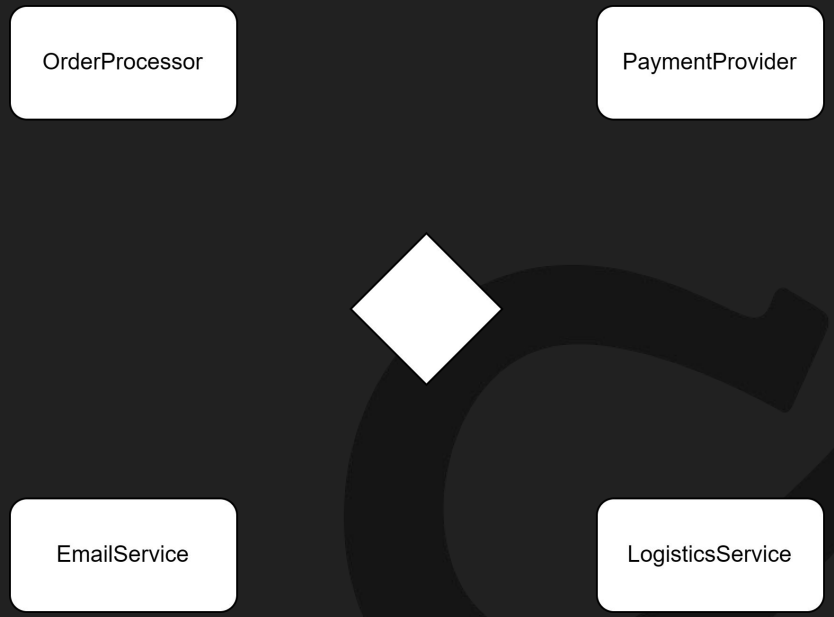
Choreography - Order Processing



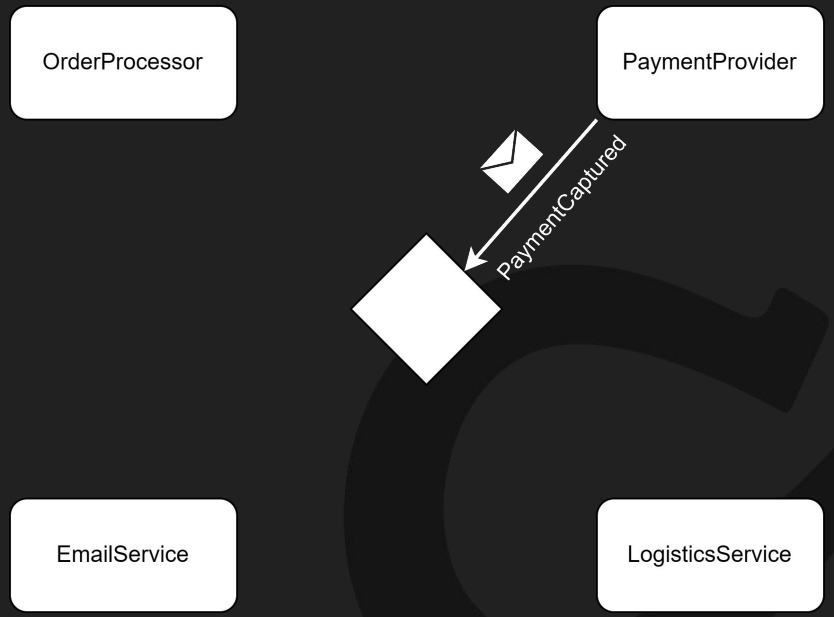
Choreography - Order Processing



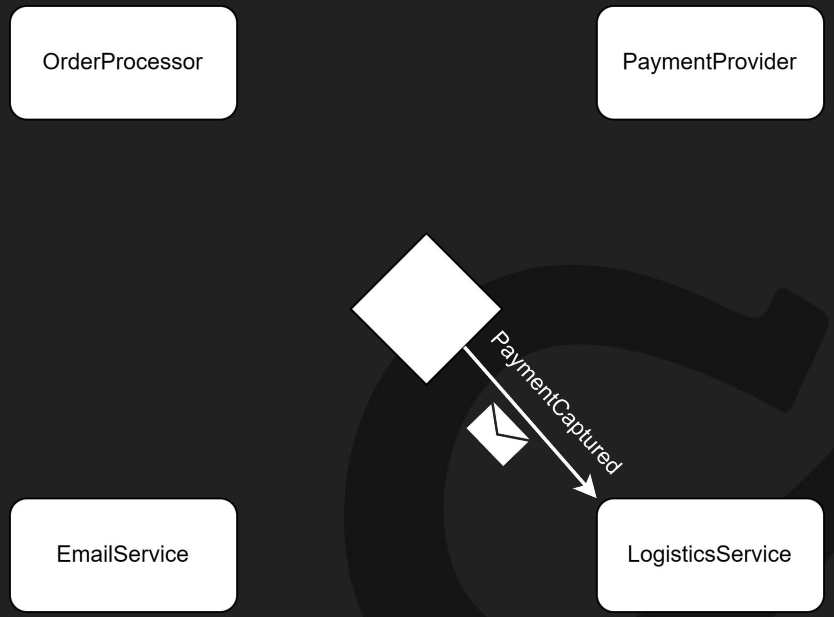
Choreography - Order Processing



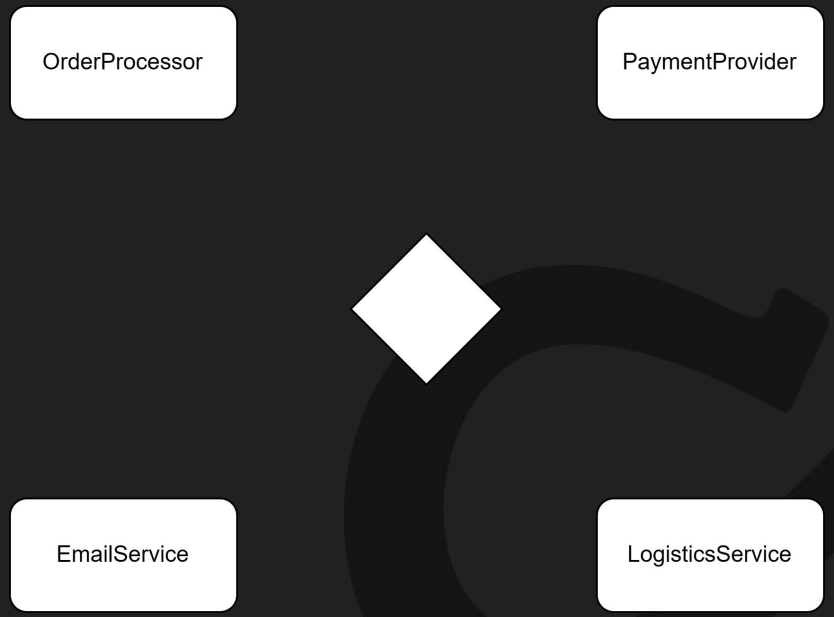
Choreography - Order Processing



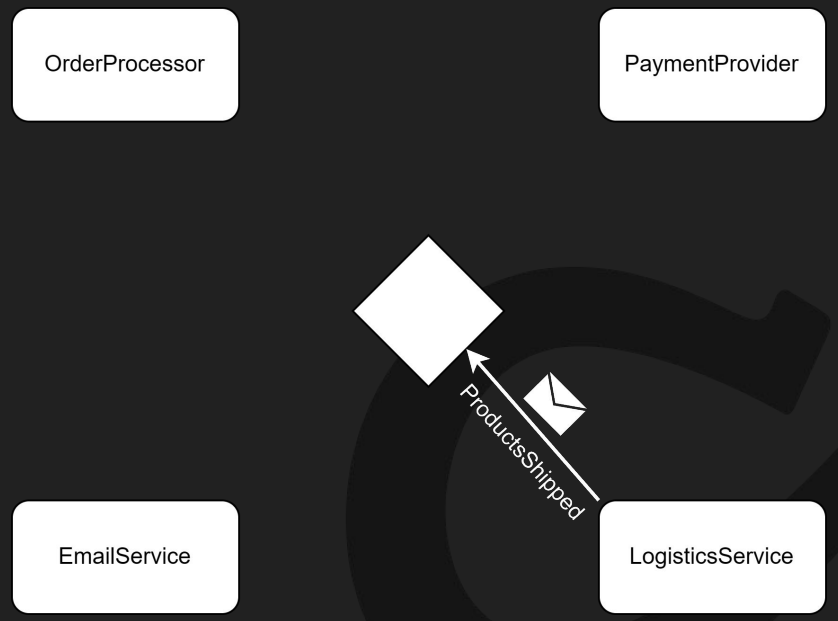
Choreography - Order Processing



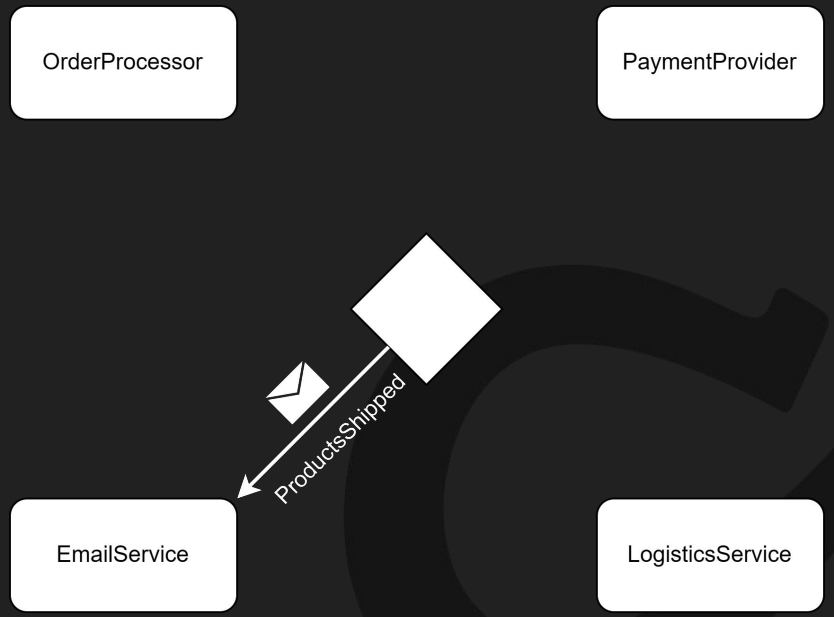
Choreography - Order Processing



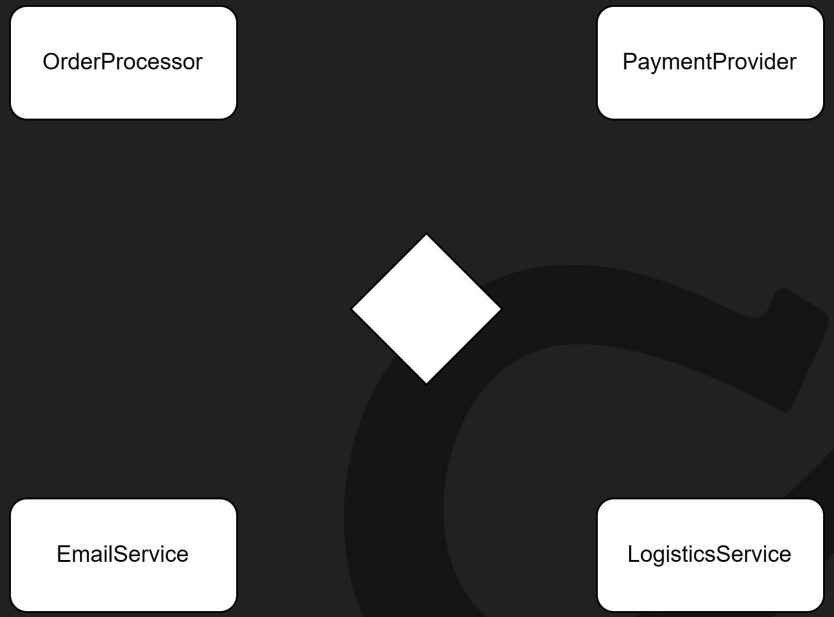
Choreography - Order Processing



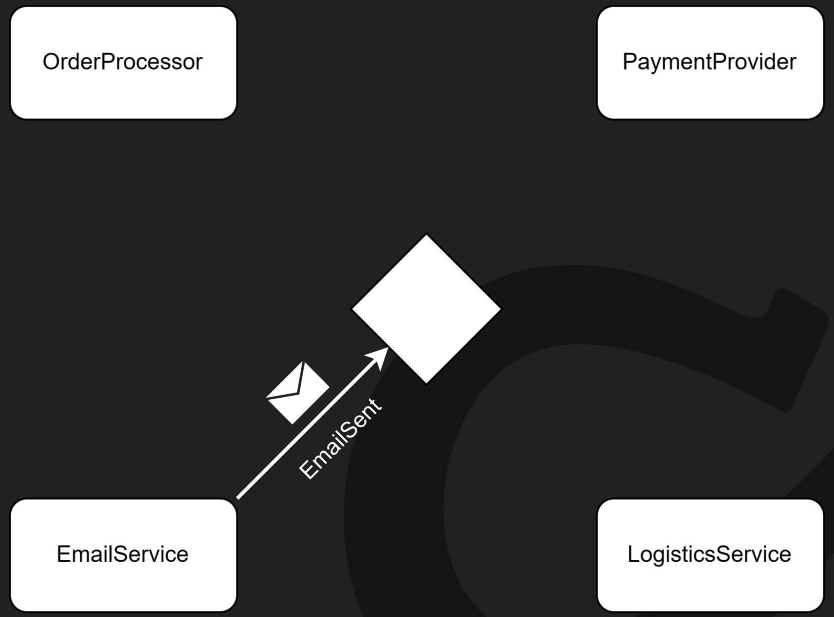
Choreography - Order Processing



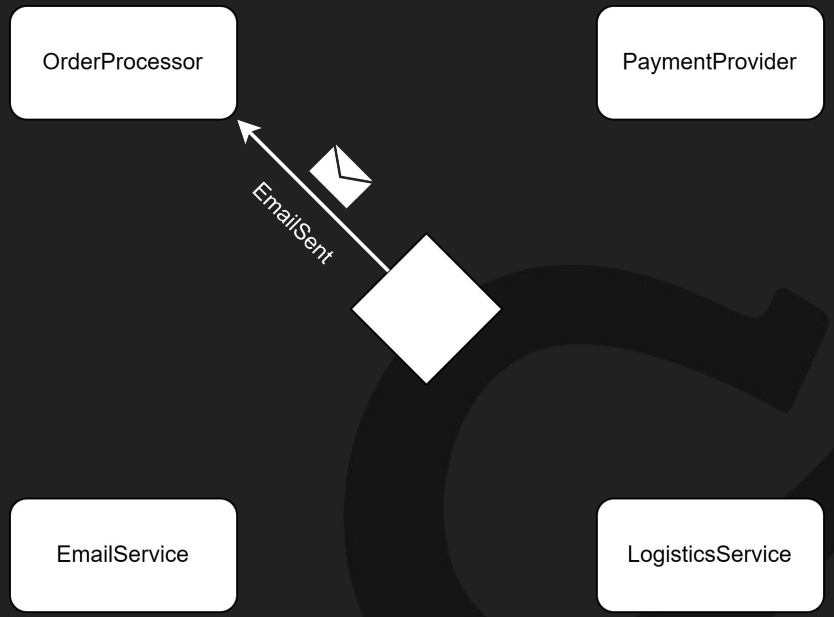
Choreography - Order Processing



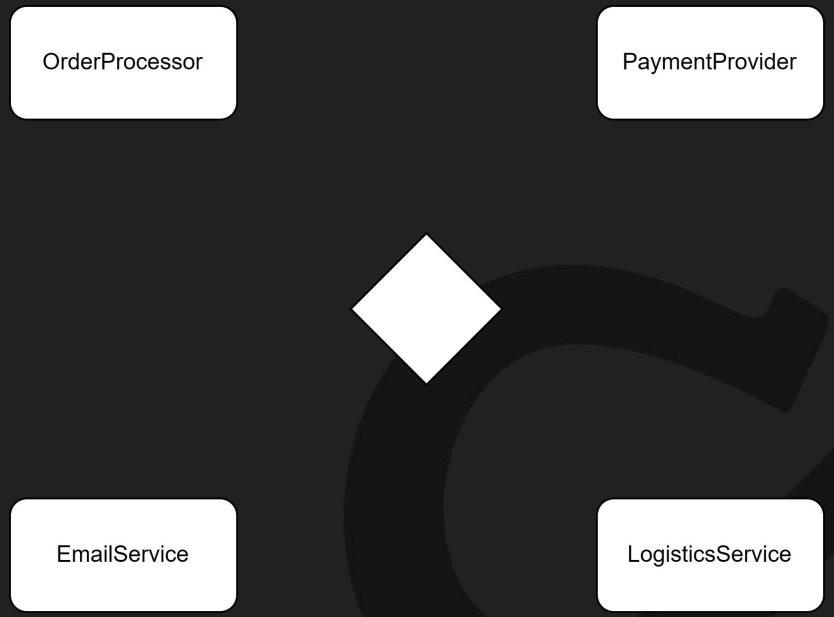
Choreography - Order Processing



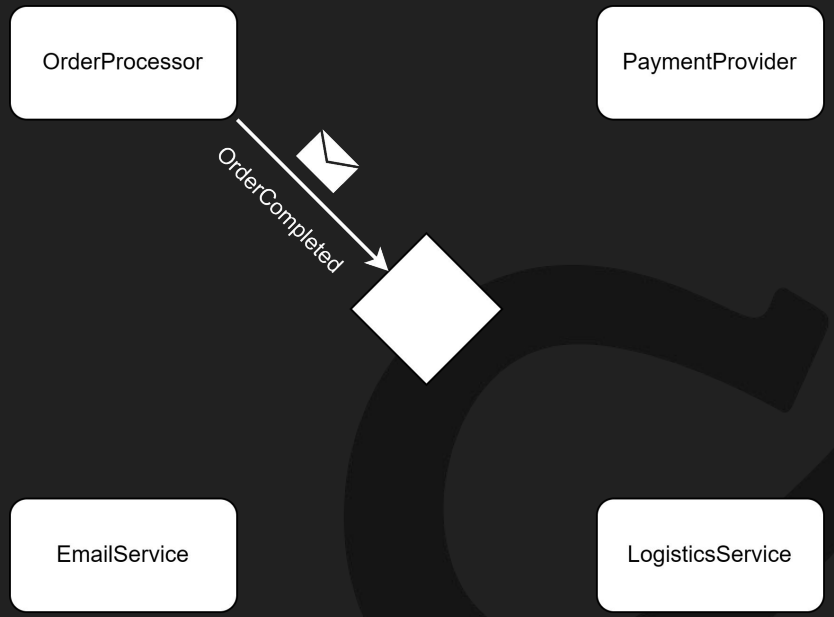
Choreography - Order Processing



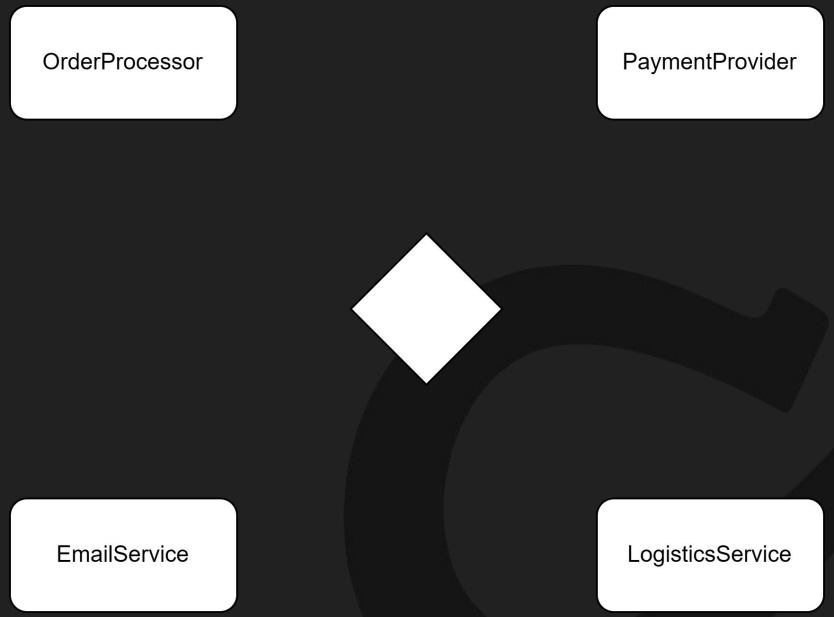
Choreography - Order Processing



Choreography - Order Processing

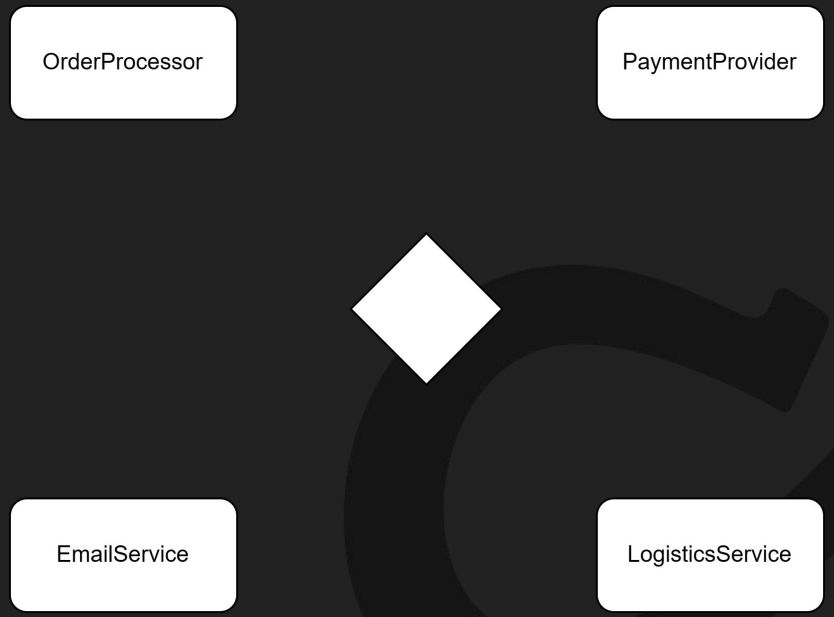


Choreography - Order Processing



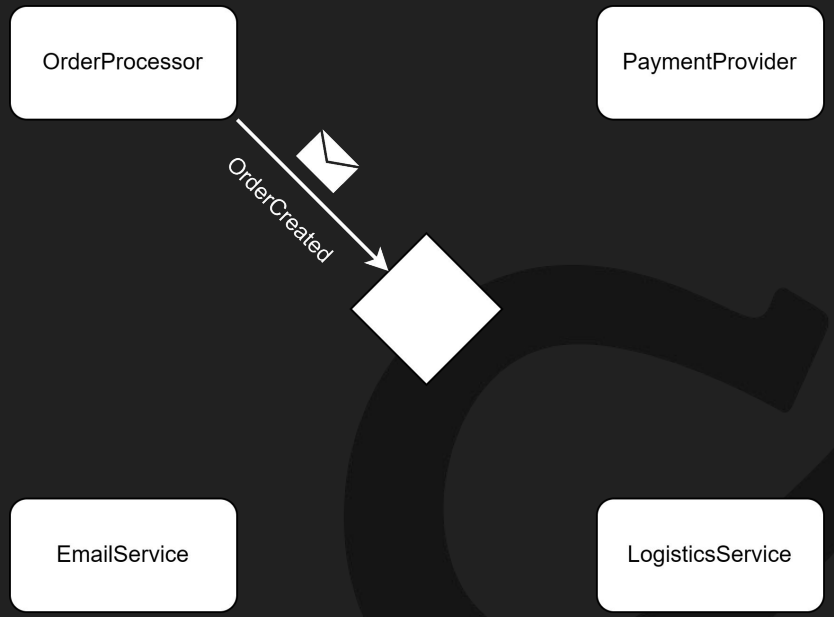
What are the **pros** and **cons** of this approach?

Choreography - Order Processing



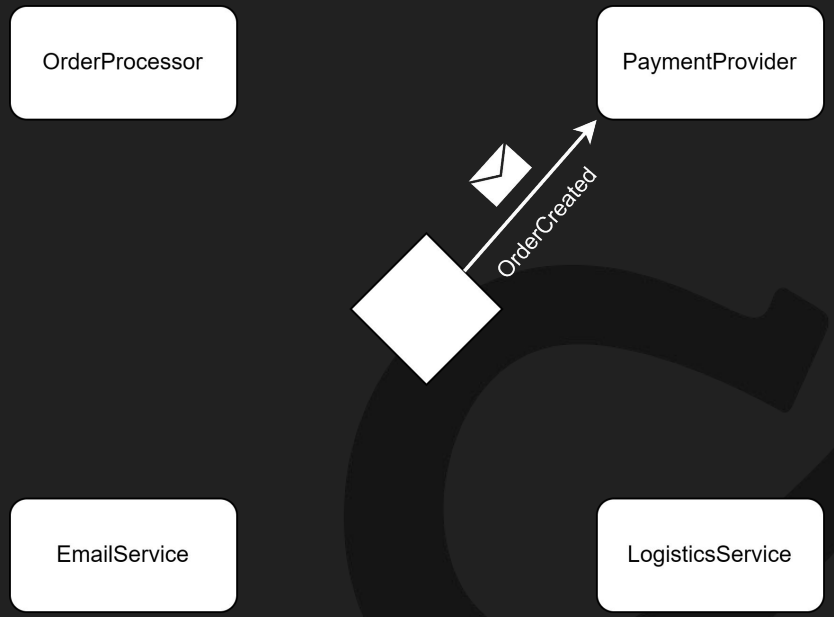
Did you notice? I **cheated** 😈

Choreography - Order Processing



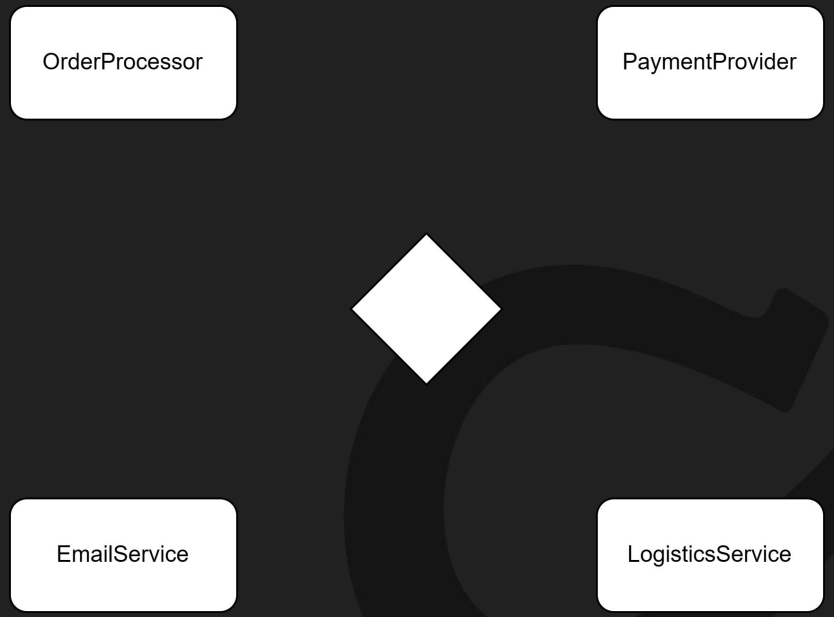
Did you notice? I **cheated** 😈

Choreography - Order Processing



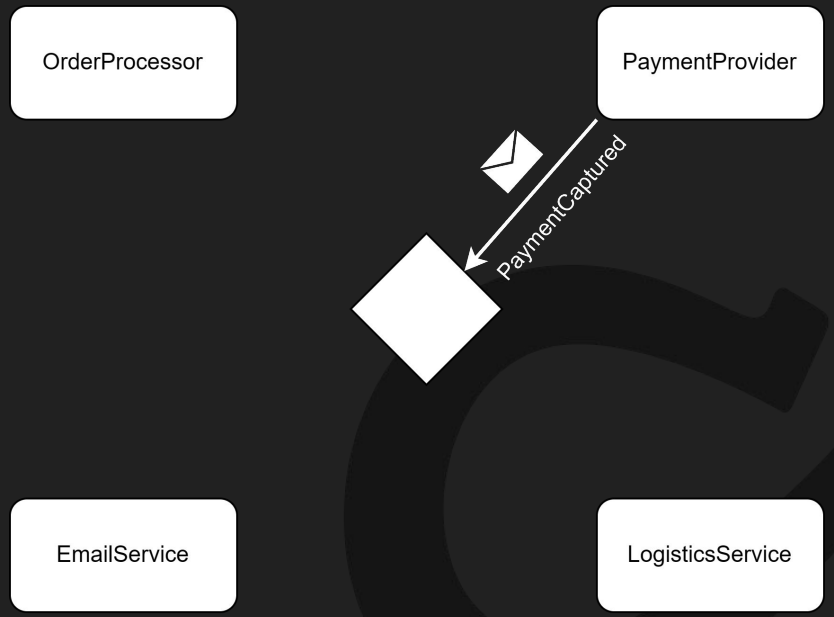
Did you notice? I **cheated** 😈

Choreography - Order Processing



Did you notice? I **cheated** 😈

Choreography - Order Processing



Did you notice? I **cheated** 😈

Choreography - Order Processing

```
public async Task ProcessOrder(Order order)

    var transactionId = Guid.NewGuid();

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

    var trackAndTrace = await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);

    await _paymentProviderClient.Capture(transactionId);

    await _emailClient.SendOrderConfirmation(order.CustomerId, trackAndTrace, order.OrderNumber);
```

Did you notice? I **cheated** 😈

Message **Queue** - The **Ack** Problem

When receiving a message an important decision is when to ack the message.

```
while (true)

    var msg = await FetchNextMessage();

    await HandleMessage(msg);

    await AckMessage(msg);
```

```
while (true)

    var msg = await FetchNextMessage();

    await AckMessage(msg);

    await HandleMessage(msg);
```

Message Queues - **Single Event Handling**

How do we ensure that an internal state change in the service and the accompanying event both happen (*despite failures*) when handling a **single** event?

Message-driven - **Outbox Pattern**

<u>The Problem:</u>

A workflow needs to both update its **own state** and publish related **events**.

```
public void Handle(OrderCreated orderCreated)

  var transactionId = Guid.NewGuid();

  PublishMessage(new PaymentReserved(orderNumber));

  SaveToDatabase(orderNumber, transactionId);
```

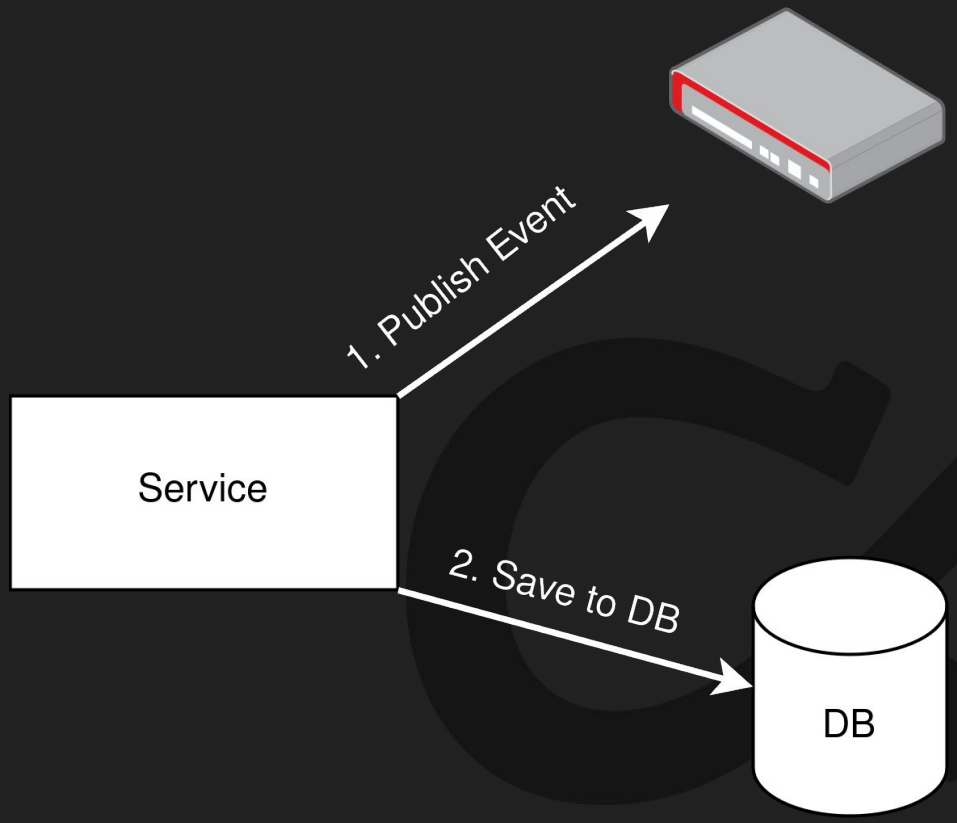
Message-driven - **Outbox Pattern**

<u>The Problem:</u>

A workflow needs to both update its **own state** and publish related **events**.

```
public void Handle(OrderCreated orderCreated)

  var transactionId = Guid.NewGuid();

  SaveToDatabase(orderNumber, transactionId);

  PublishMessage(new PaymentReserved(orderNumber));
```

Message-driven - **Outbox Pattern**



Message-driven - **Outbox Pattern**

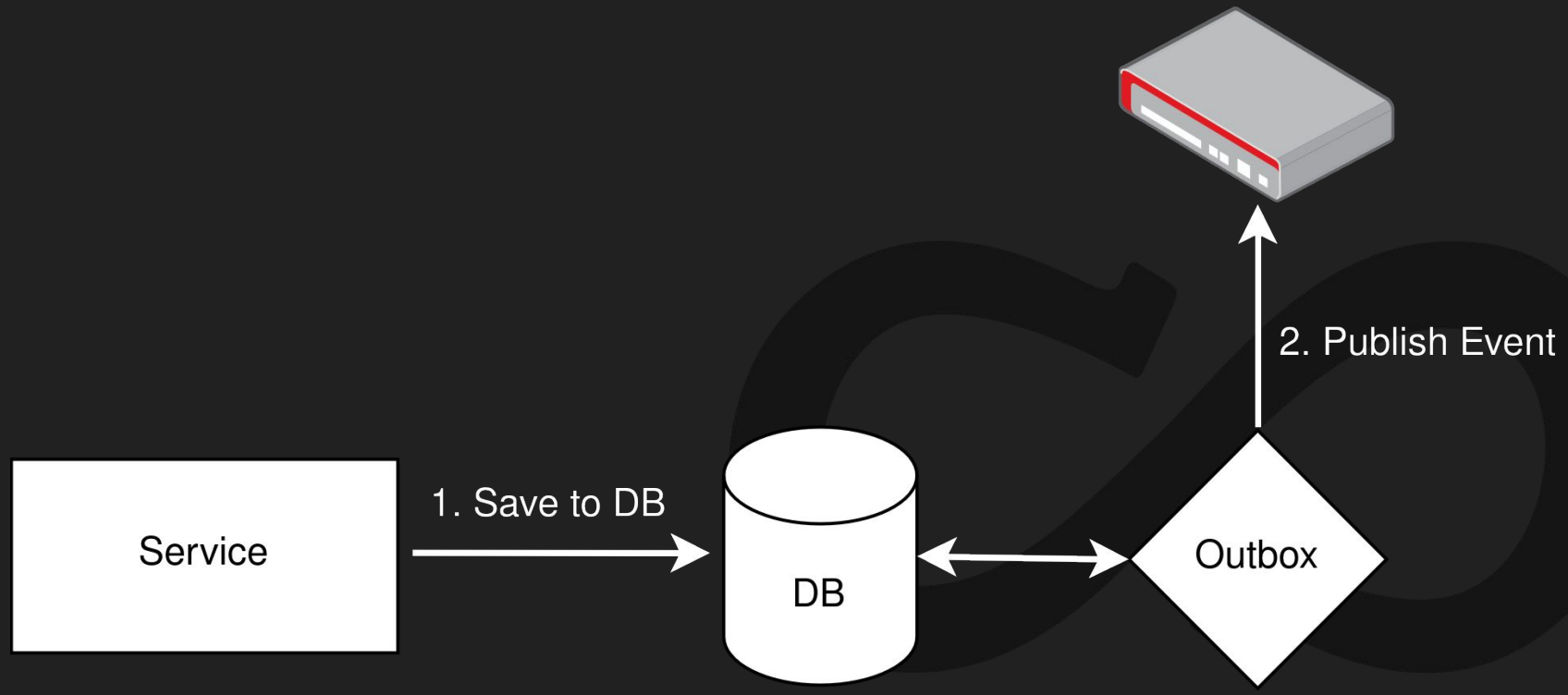
The Solution:

Wrap **both** the system's new state and events inside **one database transaction**. Afterwards, publish events in the background (i.e. using background service)

```
public static void Handle(OrderCreated order)

    var transactionId = Guid.NewGuid();

    var tran = BeginTransaction();

    SaveToOutboxTable(new PaymentReserved(orderNumber), tran);

    SaveToOrdersTable(order, tran);

    tran.Commit();
```

```
public static void OutboxLoop()

    while (true)

        var msg = FetchNext();

        PublishMessage(msg);

        RemoveFromOutbox(msg);
```

Message-driven - **Outbox Pattern**



Message-driven - **Outbox Pattern**

<u>The Challenge:</u>

How do we avoid re-publishing the same message multiple times?

```
public static void Handle(OrderCreated order)

    var transactionId = Guid.NewGuid();

    var tran = BeginTransaction();

    SaveToOutboxTable(new PaymentReserved(orderNumber), tran);

    SaveToOrdersTable(order, tran);

    tran.Commit();
```

```
public static void OutboxLoop()

    while (true)

        var msg = FetchNext();

        PublishMessage(msg);

        RemoveFromOutbox(msg);
```

Message-driven - **Outbox Pattern**

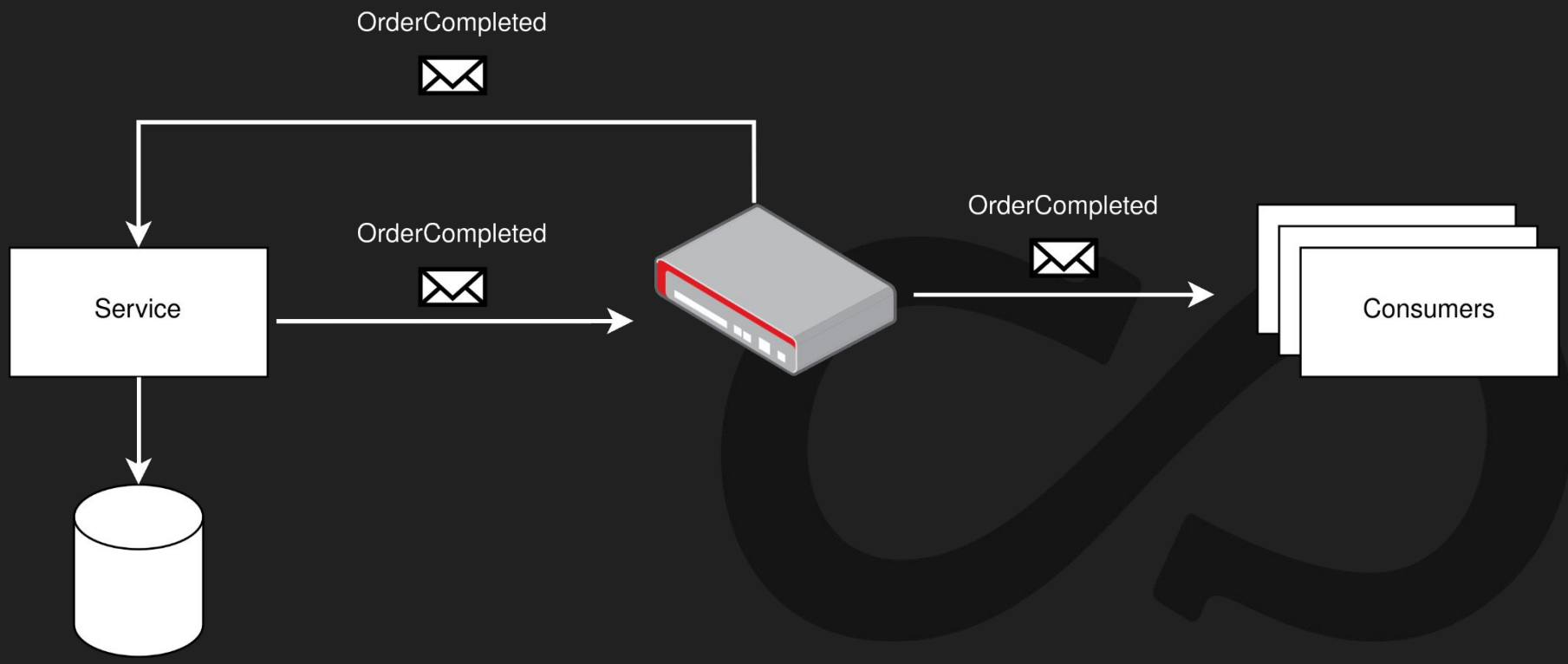
<u>The Challenge:</u>

How do we avoid re-publishing the same message multiple times?

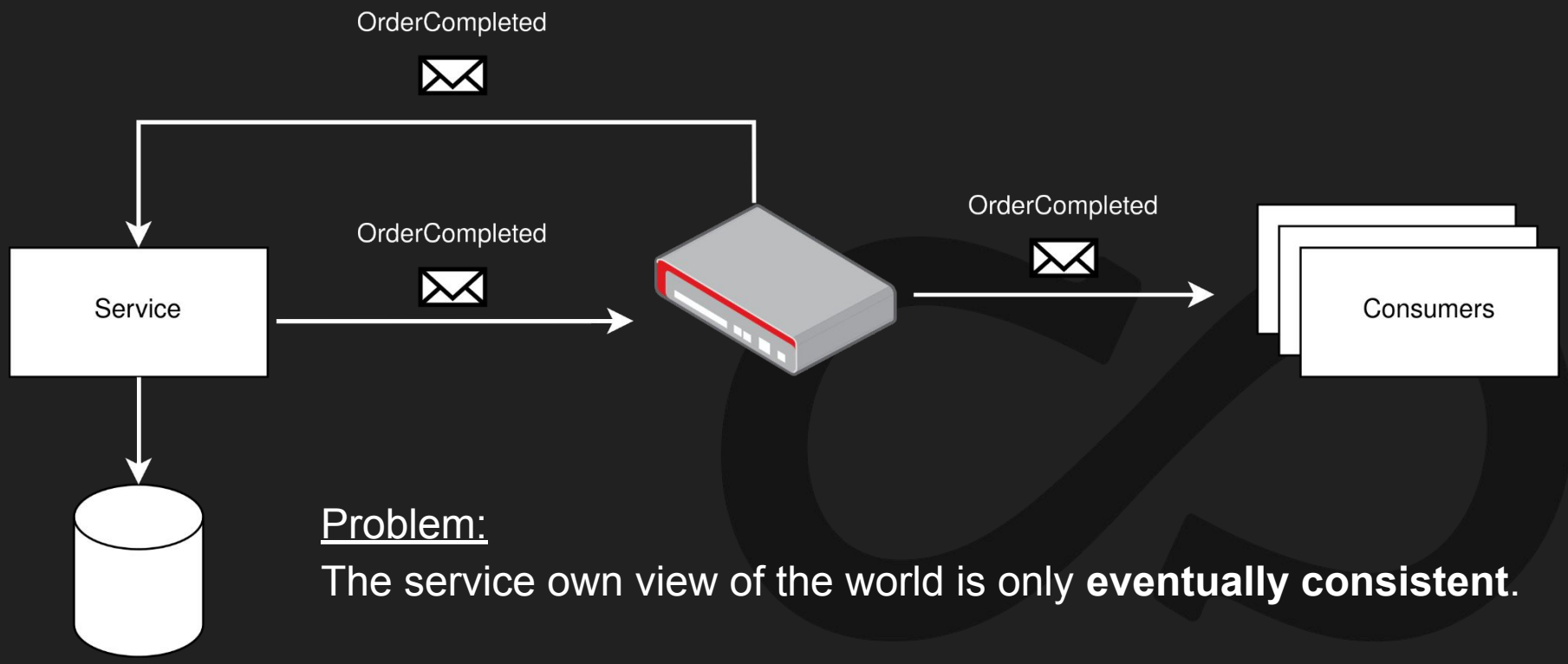
```
public static void Handle(OrderCreated order)

    var transactionId = Guid.NewGuid();

    var tran = BeginTransaction();

    SaveToOutboxTable(new PaymentReserved(orderNumber), tran);

    SaveToOrdersTable(order, tran);

    tran.Commit();
```

```
public static void OutboxLoop()

    while (true)

        var msg = FetchNext();

        PublishMessage(msg);

        RemoveFromOutbox(msg);
```

Message-driven - **Listen-to-Yourself Pattern**



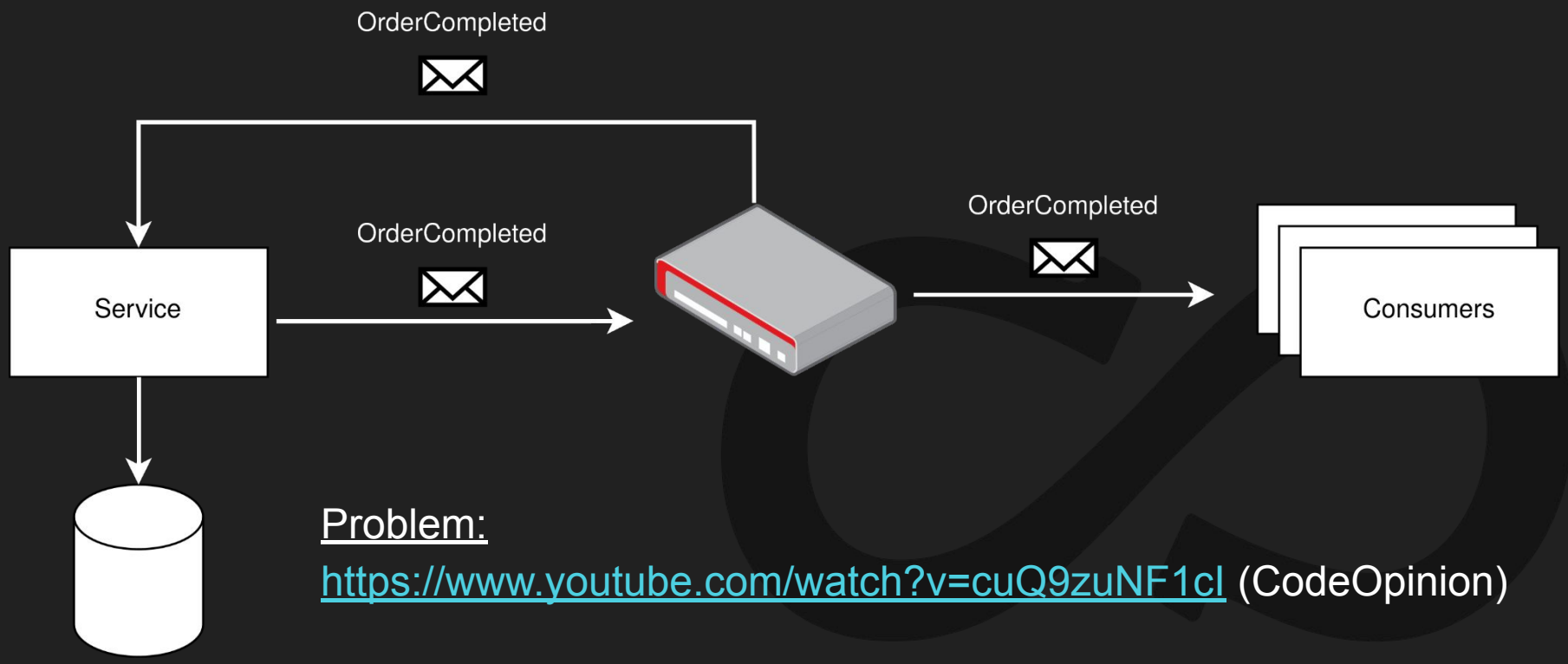
Message-driven - **Listen-to-Yourself Pattern**



<u>Problem:</u>

The service own view of the world is only **eventually consistent**.

Message-driven - **Listen-to-Yourself Pattern**



Problem:

https://www.youtube.com/watch?v=cuQ9zuNF1cI (CodeOpinion)

Message-driven - **ServiceBus**

What are ServiceBus frameworks?

Message-driven - **ServiceBus**

What are ServiceBus frameworks?

— ChatGBT:

~~*Service-buses, provide a set of tools for building applications that communicate through asynchronous messaging.*~~

~~*They manage the sending, receiving, and processing of messages, ensuring reliable communication between different parts of a system or between different systems.*~~

Message-driven - **ServiceBus**

What are ServiceBus frameworks?

— TLDR:

They simplify using message queues!

Message-driven - **ServiceBus**

Examples:

- NServiceBus
- MassTransit
- Wolverine
- Rebus

Message-driven - **ServiceBus & Sagas**

A ServiceBus often have **saga** support. Thereby, addressing the *shortcomings* of the previously shown solutions.

A **saga** is a **centralized coordinator**, that handles all messages related to a specific business flow.

Message-driven - **ServiceBus & Sagas**

A ServiceBus often have **saga** support. Thereby, addressing the *shortcomings* of the previously shown solutions.

A **saga** is a **centralized coordinator**, that handles all messages related to a specific business flow.

—

N.B. the term saga is overloaded. Not to be confused with the famous academic pattern about the saga-pattern.

Message-driven - **ServiceBus**

<u>Approaches:</u>

1. One **message handler** per **message type** approach:

```
public Task Handle(FundsReserved fundsReserved) => ...

public Task Handle(ProductsShipped productsShipped) => ...

public Task Handle(FundsCaptured fundsCaptured) => ...

public Task Handle(OrderConfirmationEmailSent emailSent) => ...
```

Message-driven - **ServiceBus**

<u>Challenge:</u>

Cumbersome to implement non-trivial flows

```
public Task Handle(FundsReserved fundsReserved) => ...

public Task Handle(ProductsShipped productsShipped) => ...

public Task Handle(FundsCaptured fundsCaptured) => ...

public Task Handle(OrderConfirmationEmailSent emailSent) => ...
```

Message-driven - **ServiceBus**

<u>Approaches:</u>

2. Declaratively specifying a state machine

```
public class OrderStateMachine :
    MassTransitStateMachine<OrderState>
{
    public OrderStateMachine()
    {
        Initially(
            When(SubmitOrder)
                .TransitionTo(Submitted),
            When(OrderAccepted)
                .TransitionTo(Accepted));

        During(Submitted,
            When(OrderAccepted)
                .TransitionTo(Accepted));

        During(Accepted,
            Ignore(SubmitOrder));
    }
}
```

Message-driven - **ServiceBus**

<u>Challenge:</u>

Cumbersome to implement non-trivial flows

```
public class OrderStateMachine :
    MassTransitStateMachine<OrderState>
{
    public OrderStateMachine()
    {
        Initially(
            When(SubmitOrder)
                .TransitionTo(Submitted),
            When(OrderAccepted)
                .TransitionTo(Accepted));

        During(Submitted,
            When(OrderAccepted)
                .TransitionTo(Accepted));

        During(Accepted,
            Ignore(SubmitOrder));
    }
}
```

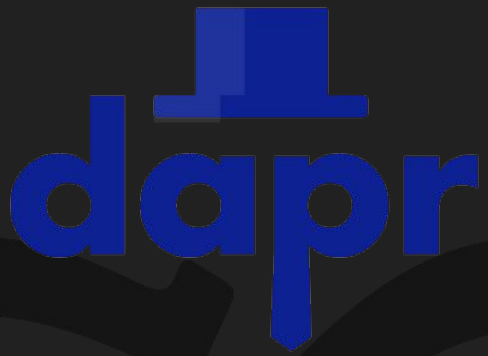
Workflow-as-Code

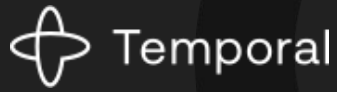
Ordinary-looking code with **resiliency**

```
[Activity]
public async Task DoPurchaseAsync(Purchase purchase)
{
    using var resp = await client.PostAsJsonAsync(
      "https://api.example.com/purchase",
      purchase,
      ActivityExecutionContext.Current.CancellationToken);

    // Make sure we succeeded
    try
    {
        resp.EnsureSuccessStatusCode();
    }
    catch (HttpRequestException e) when (resp.StatusCode < HttpStatusCode
    {
        // We don't want to retry 4xx status codes, only 5xx status codes
        throw new ApplicationFailureException("API returned error", e, no
    }
}
```

```
[Workflow]
public class OneClickBuyWorkflow
{
    private PurchaseStatus currentStatus = PurchaseStatus.Pending;
    private Purchase? currentPurchase;

    [WorkflowRun]
    public async Task<PurchaseStatus> RunAsync(Purchase purchase)
    {
        currentPurchase = purchase;

        // Give user 10 seconds to cancel or update before we send it through
        try
        {
            await Workflow.DelayAsync(TimeSpan.FromSeconds(10));
        }
        catch (TaskCanceledException)
        {
            currentStatus = PurchaseStatus.Cancelled;
            return currentStatus;
        }

        // Update the status, perform the purchase, update the status again
        currentStatus = PurchaseStatus.Confirmed;
        await Workflow.ExecuteActivityAsync(
            (PurchaseActivities act) => act.DoPurchaseAsync(currentPurchase!),
            new() { ScheduleToCloseTimeout = TimeSpan.FromMinutes(2) });
        currentStatus = PurchaseStatus.Completed;
        return currentStatus;
    }
```

Workflow-as-Code - dotnet Frameworks





Workflow-as-Code

<u>Trick:</u>

At the core the trick of archiving ordinary looking code with resiliency is:

*“Remembering the result of previously executed **work** using **event sourcing**.“*

Workflow-as-Code

Trick:

At the core the trick of archiving ordinary looking code with resiliency is:

*“Remembering the result of previously executed **work** using **event sourcing**.“*

Challenge:

Space-consumption increases over time

Cleipnir.**NET**

What is Cleipnir.NET?

A .NET framework

simplifying code

which needs to be executed in its

entirety

despite **crashes**

Cleipnir.**NET** - Framework Concepts

- **Effects**

  Taming non-determinism and ensuring efficient re-execution

- **Messages**

  Handling external messages

- **Utilities**

  Handy tools for cross-flow challenges

Framework Concepts - **Flow**

Just inherit and implement the flow logic:

```
public class OrderFlow : Flow<Order>

  public override async Task Run(Order order)
```

```
public class OrderFlow : Flow<Order, Guid>

  public override async Task<Guid> Run(Order order)
```

Framework Concepts - **Flow**

Start a flow using the source-generated entry-point:

```
public class OrderController : ControllerBase

   private readonly OrderFlows _orderFlows;

   public OrderController(OrderFlows orderFlows) => _orderFlows = orderFlows;


    [HttpPost]

   public async Task<ActionResult> Post(Order order)

       await _orderFlows.Run(order.OrderId, order);

       return Ok();
```

Framework Concepts - **Effects**

Robustifying **external** communication:

```
await Effect.Capture(
    "SendMail",
    work: () => _mailClient.SendMail(customerMail)
);
```

Framework Concepts - **Effects** & **At-Least-Once**

Robustifying **external** communication:

```
await Effect.Capture(
    "SendMail",
    work: () => _mailClient.SendMail(customerMail)
    ResiliencyLevel.AtLeastOnce
);
```

Framework Concepts - **Effects** & **At-Most-Once**

Robustifying **external** communication:

```
await Effect.Capture(
    "SendMail",
    work: () => _mailClient.SendMail(customerMail)
    ResiliencyLevel.AtMostOnce
);
```

Framework Concepts - **Effects** & **Determinism**

Taming **non-determinism**:

```
var requestId = await Effect.Capture("RequestId", Guid.NewGuid);
```

Alternatively:

```
var requestId = await Effect.CreateOrGet("RequestId", Guid.NewGuid());
```

Framework Concepts - **Effects** & **State**

Space-efficient track-keeping of the **invocation's progress**:

```
var i = await effect.CreateOrGet("i", 0);
while (i < elms.Count)
    var elm = elms[i];
    Console.WriteLine(elm);
    await effect.Upsert("i", i++);
```

Framework Concepts - **Messages**

Each flow has its *own* associated messages-instance (event-sourced) which allows the workflow to wait for **external events** before continuing:

```
await Messages.FirstOfType<FundsReserved>()
```

Framework Concepts - **Messages**

Reactive operators can also be chained together to form more elaborate ‘event selection’-logic:

```
var externalEvents = await Messages
    .OfType<SomeEvent>()
    .Take(3)
    .Completion()
```

Framework Concepts - **Messages**

Reactive operators can also be chained together to form more elaborate ‘event selection’-logic:

```
var externalEvents = await Messages
    .OfType<SomeEvent>()
    .Take(3)
    .SuspendUntilCompletion()
```

Framework Concepts - **Messages**

Reactive operators can also be chained together to form more elaborate ‘event selection’-logic:

```
var externalEvents = await Messages
    .OfType<SomeEvent>()
    .Take(3)
    .TakeUntilTimeout("TimeoutId", expiresAt: DateTime.Now.AddMinutes(5))
    .SuspendUntilCompletion()
```

Framework Concepts - **Messages**

A workflow instance can also communicate with other workflows:

```
await Workflow.PublishMessage(
  workflowId,
  message: $"Hello from {Workflow.FunctionId}",
  idempotencyKey: workflowId.ToString()
);
```

Framework Concepts - **ControlPanel**

A flow can be (1) **inspected**, (2) **altered** and (3) **retried** using the flow's associated control-panel:

```
var controlPanel = await flows.ControlPanel("2023-10");

await controlPanel!.Effects.Remove("i");


await controlPanel.ReInvoke();
```

Framework Concepts - **Postpone**

A flow’s execution can be postponed:

```
await Workflow.Delay("Delay", TimeSpan.FromDays(1));
```

Framework Concepts - **Utilities**

When several flows needs to be synchronized **Utilities**-instance can be used:

```
await using var @lock = await Utilities.Monitor.Acquire(
    group: nameof(MonitorExample),
    name: "monitor",
    lockId,
    maxWait: TimeSpan.FromSeconds(10)
);
```

Order-flow
Revisited

Order Flow **- Challenge #1**

For the **Payment Provider** API:

- Ensure the same *transaction id* is used if the flow is restarted

Order Flow **- Challenge #2**

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

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

<u>Thus:</u>

- It must be called *at-most-once*
- **Fail** subsequent invocation and investigate…

Order Flow **- Challenge #3**

If **Logistics Service** call fails make a refund.

Order Flow **- Event-based**

Implement the Order-flow using an **event-based** solution

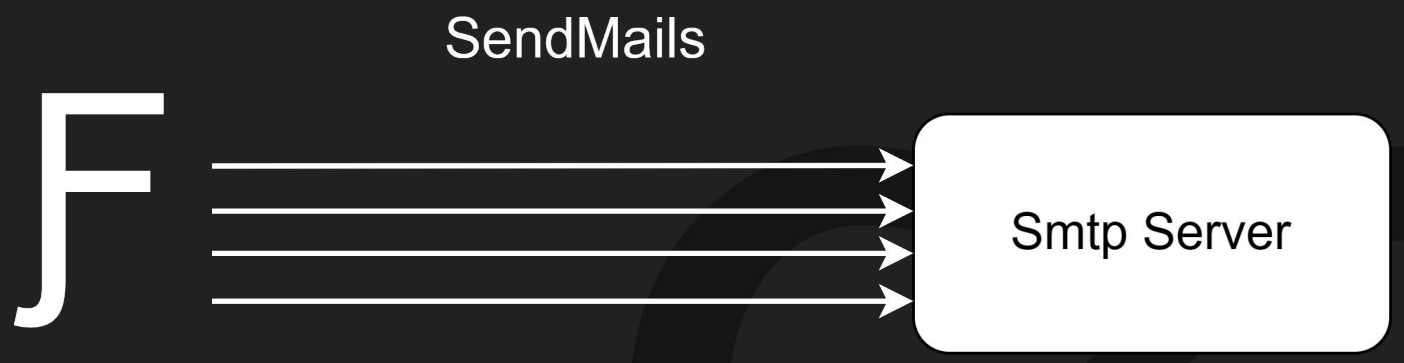
MOB Programming

Time

Newsletter

Sender

Newsletter - Example:



Newsletter - Example:

Solution Type: RPC, brief execution

A monthly **newsletter** needs to be sent out to all subscribers.

Flow:

Given: A list of subscriber email addresses

Then: Send newsletter to all subscribers

Customer Sign-up

Customer Signup - Example:

<u>Solution Type:</u> RPC & Event-based, long execution

Remind a user to activate their account after sign-up

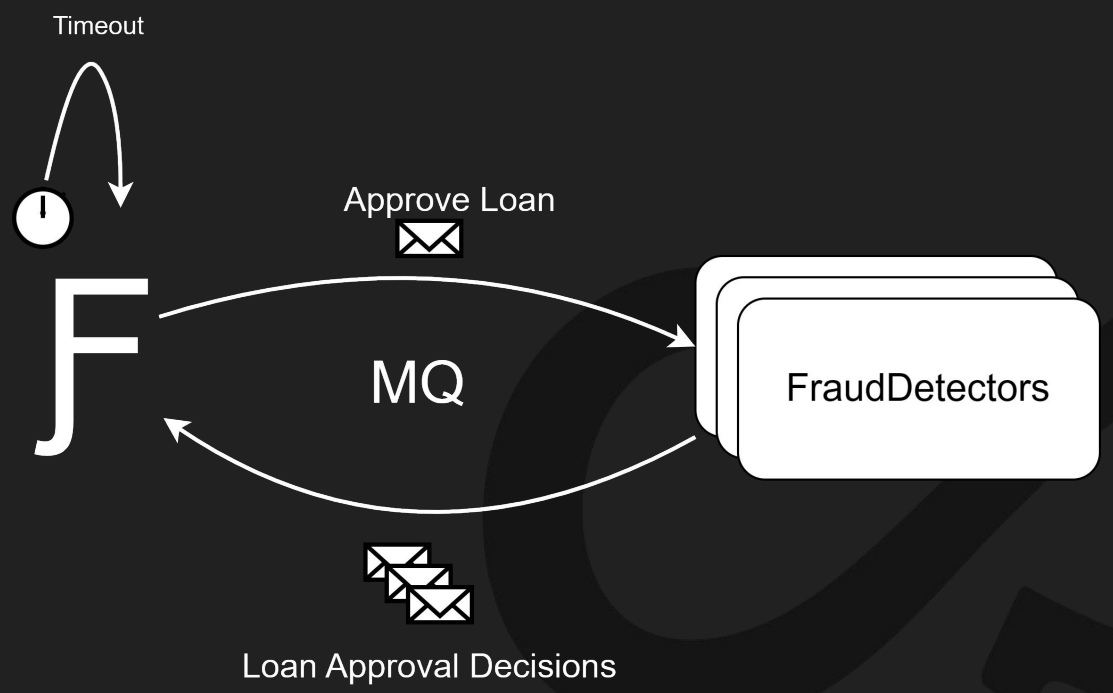
Flow:

- Send **activation-mail** to newly signed-up user with activation link
- If user has not been activated then send daily reminder (max 3 days)
- Finally, send **welcome-mail** or fail flow

Loan

Application

Loan Application - Example:



Loan Application - Example:

<u>Solution Type:</u> Event-based, (semi) long-running

Implement the following **loan application** flow

- In parallelle broadcast a loan application message
- Wait for at least 2/3 replies within 15 minutes
- If any replies are reject then reject the application. Otherwise, accept it.

Loan Application - Example:

<u>Solution Type:</u> Event-based, (semi) long-running

Implement the following **loan application** flow

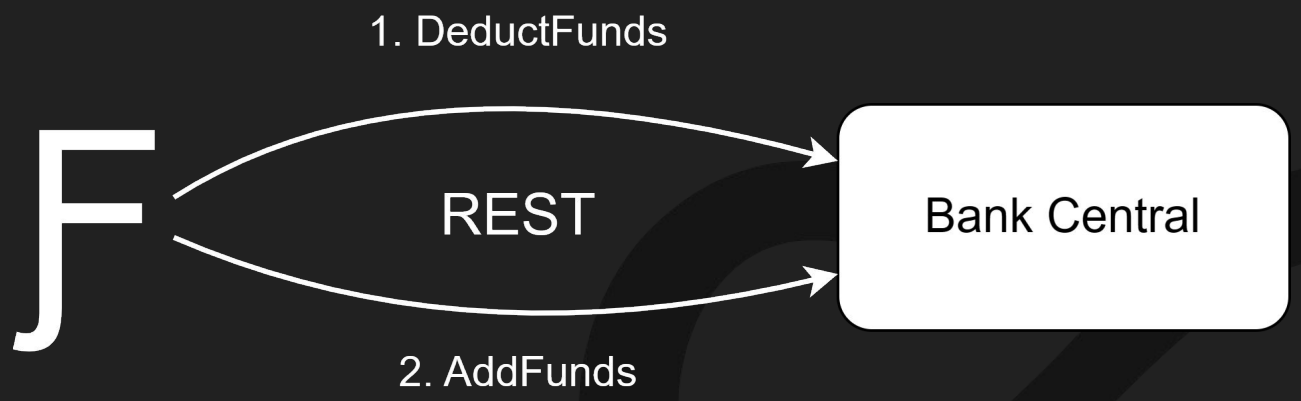
- In parallelle broadcast a loan application message
- Wait for at least 2/3 replies within 15 minutes
- If any replies are reject then reject the application. Otherwise, accept it.

Should we suspend the execution? What is the difference in code?

Bank

Transfer

Bank Transfer - Example:



Bank Transfer - Example:

Solution Type: RPC, brief execution

A bank transfer request needs to be processed by the bank's back-end system.

Steps:

1. Ensure that there are enough funds on the sender account
2. Deduct funds from the sender account
3. Add funds to the receiver account

Bank Transfer - Example:

Solution Type: RPC, brief execution

A bank transfer request needs to be processed by the bank's back-end system.

Steps:

1. Ensure that there are enough funds on the sender account
2. Deduct funds from the sender account
3. Add funds to the receiver account

Is there a lurking **race-condition**?

Outbox

Pattern

Workflow-as-Code & The **Outbox Pattern**

Do we need the **outbox pattern** when using **workflow-as-code**?

```
public void Handle(CreateOrder createOrder)

    var order = ValidateAndConvertToOrder(createOrder)

    SaveToDatabase(order)

    PublishMessage(order)
```

Bulk
Orders Batch

Bulk Order Processing - Example:

Solution Type: Work-distribution, long execution, parallelization

Flow:

When receiving a batch of orders message process all orders efficiently.