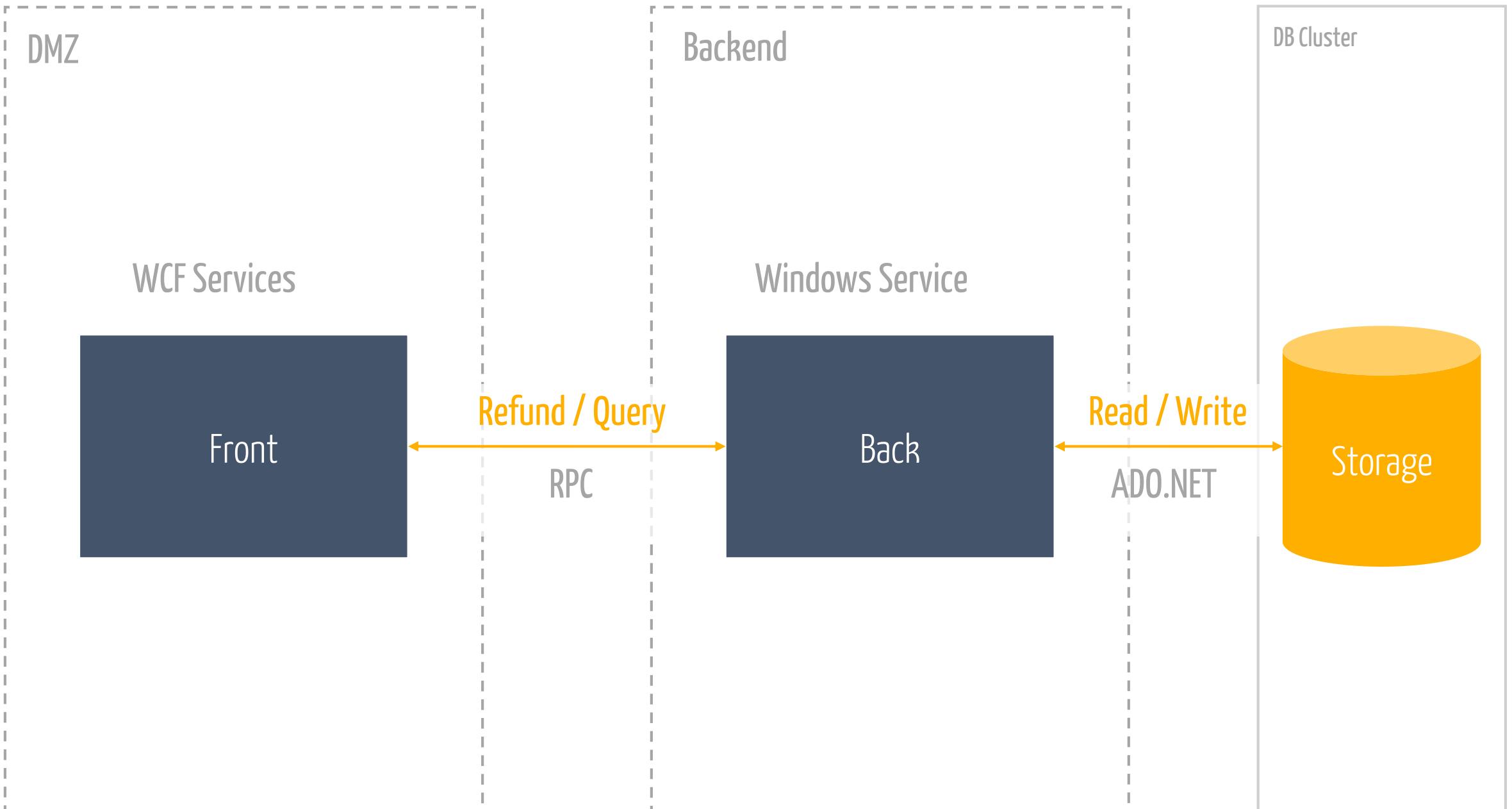




Do it yourself. A message pump
that kicks ass

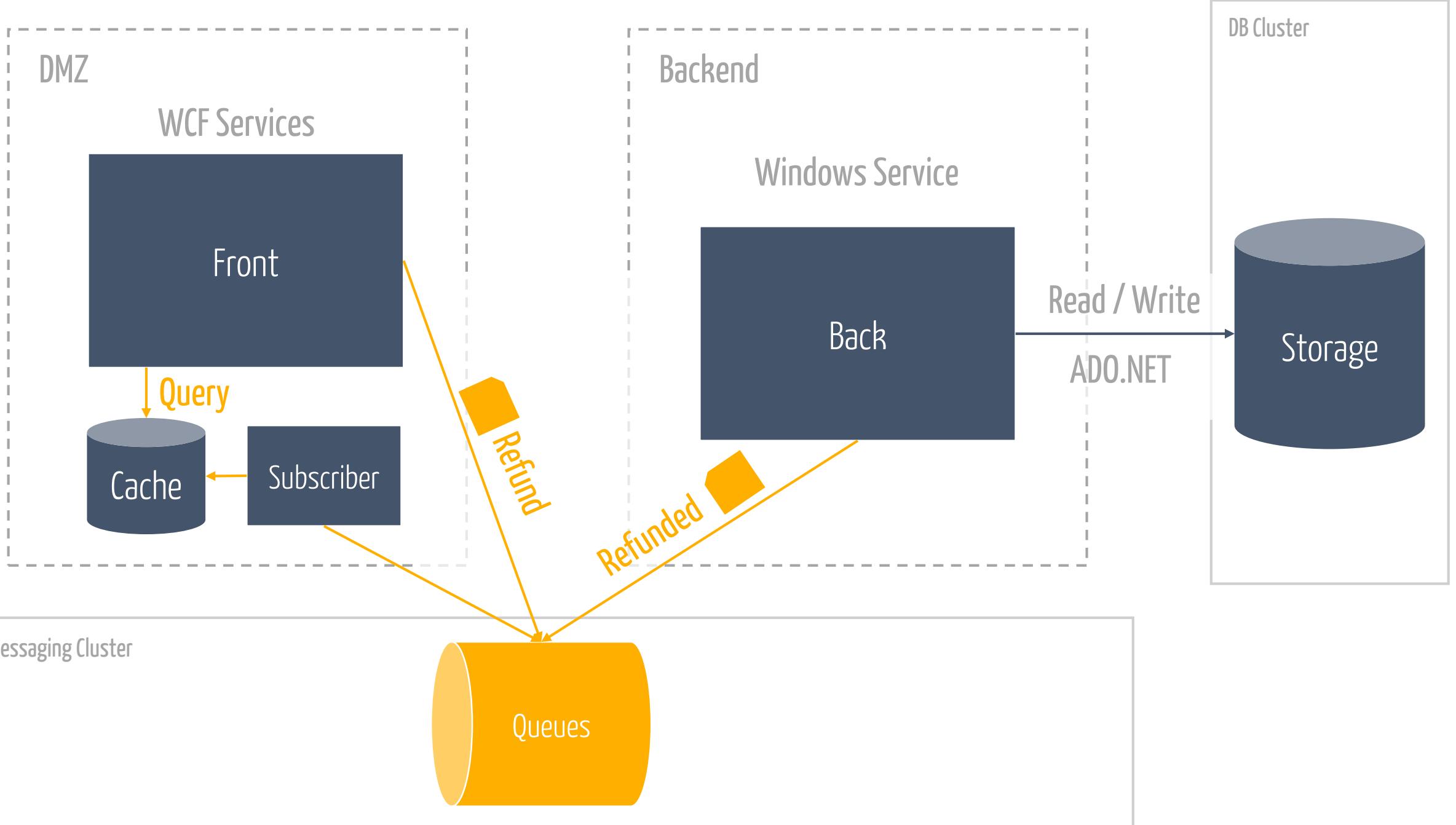


The RPC callstack of

doom

praise the lords of

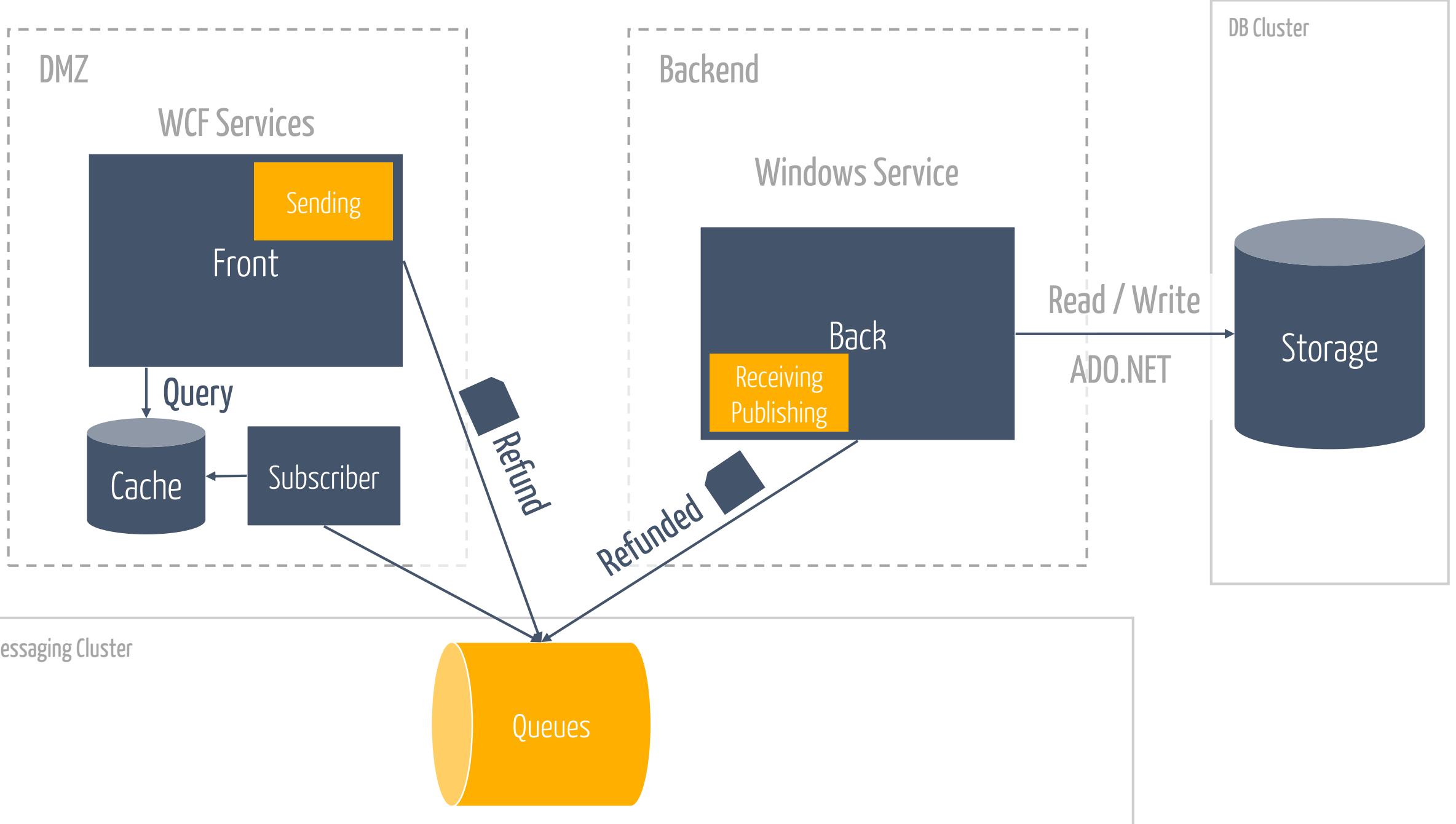
messaging



Build

or

Buy



Building the

pump



TPL handwaving

Cooporative cancellation 101

Async / Await

Ship it!

It worked until



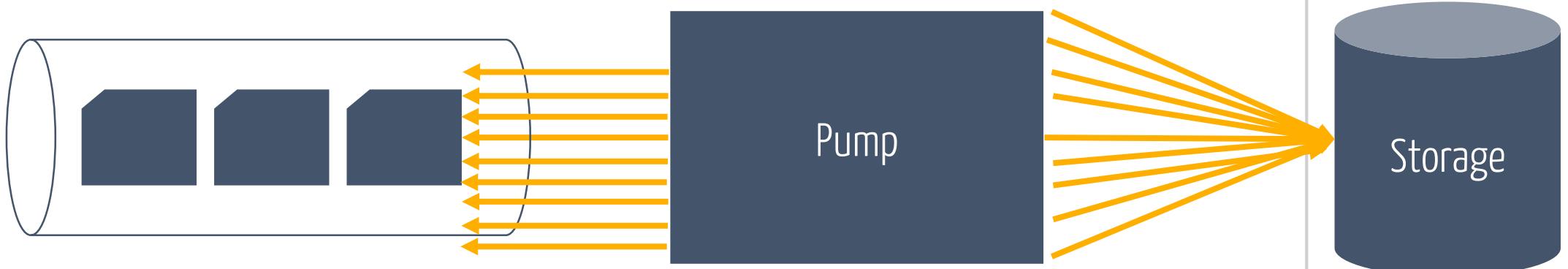
Rush hour



Let's throw in some

concurrency

Rush hour



DB Cluster

Storage

Better limit

concurrency

Semaphore controls floodgate
Ship it!

It worked



A black and white photograph of a white car parked on a curb. The car is positioned at an angle, showing its front right side. It has a multi-spoke alloy wheel and a small plant growing through the cracks in the concrete curb. In the background, there is a dense hedge or bush. Overlaid on the image is the text "As time passes" in a large, bold, orange sans-serif font.

As time passes

1 reference

```
static async Task FetchAndHandleAndReleaseWithMiddleware(SemaphoreSlim semaphore)
{
    using (var transaction = CreateTransaction())
    {
        var (payload, headers) = await ReadFromQueue(transaction);
        var message = Deserialize(payload, headers);

        using (var childServiceProvider = CreateChildServiceProvider())
        {
            try
            {
                var middlewareFuncs = new Func<HandlerContext, Func<HandlerContext, Task>, Task>[] { Middleware1, Middleware2 };
                var middleware = FlextensibleMiddleware(childServiceProvider, middlewareFuncs);

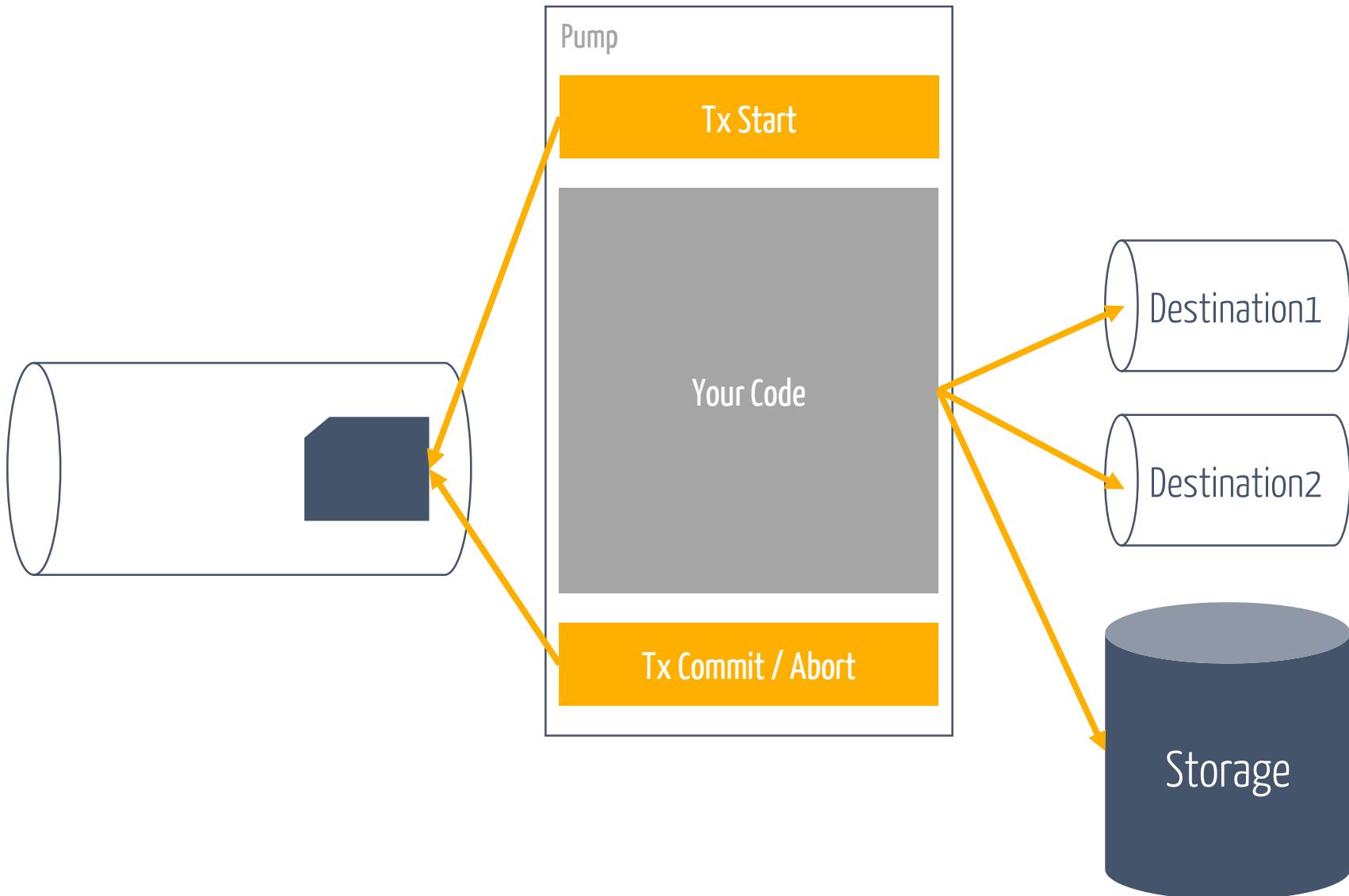
                await middleware(message).ConfigureAwait(false);

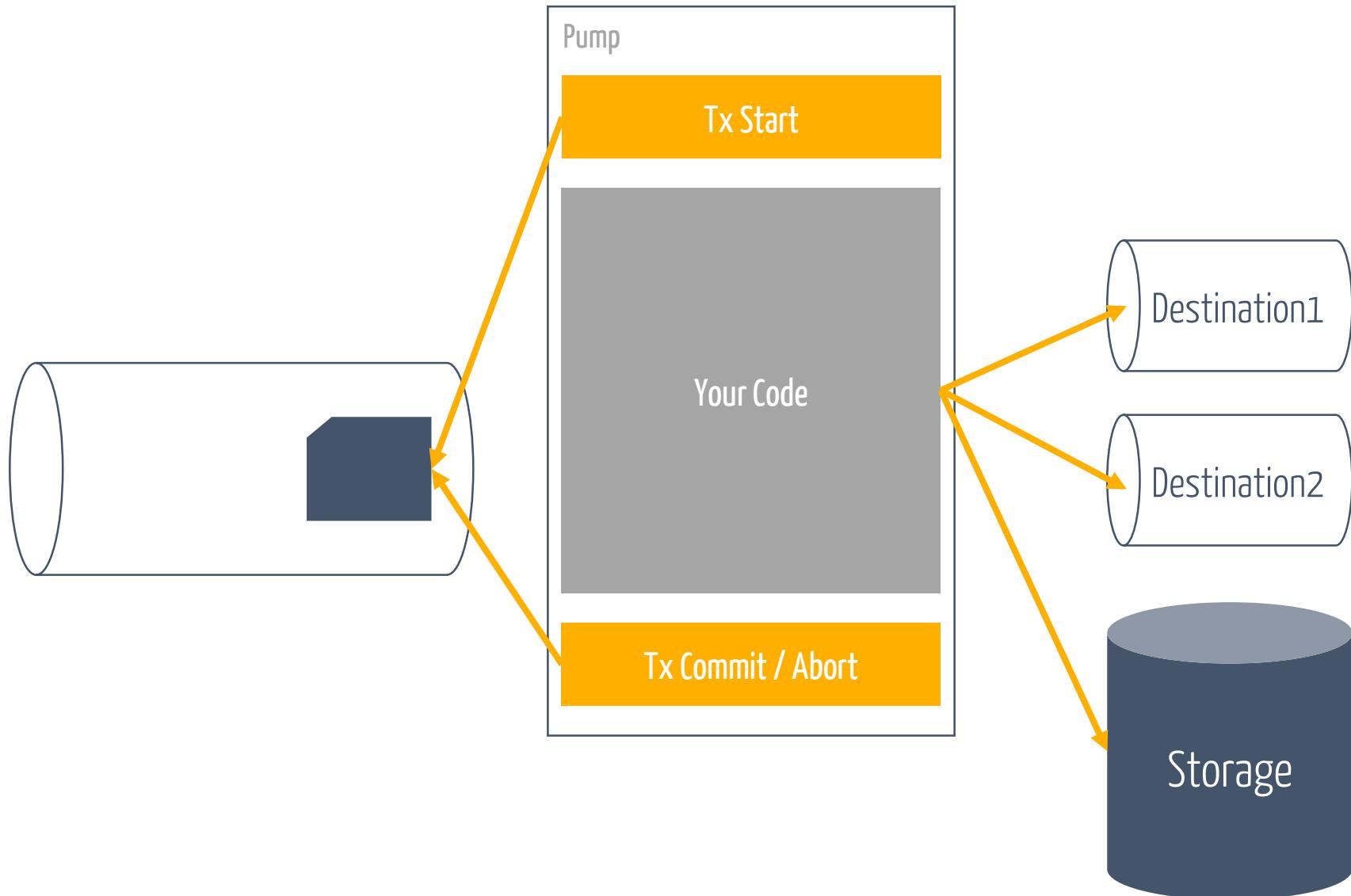
                transaction.Complete();
            }
            catch (Exception)
            {
                // Just log?
            }
            finally
            {
                semaphore.Release();
            }
        }
    }
}
```

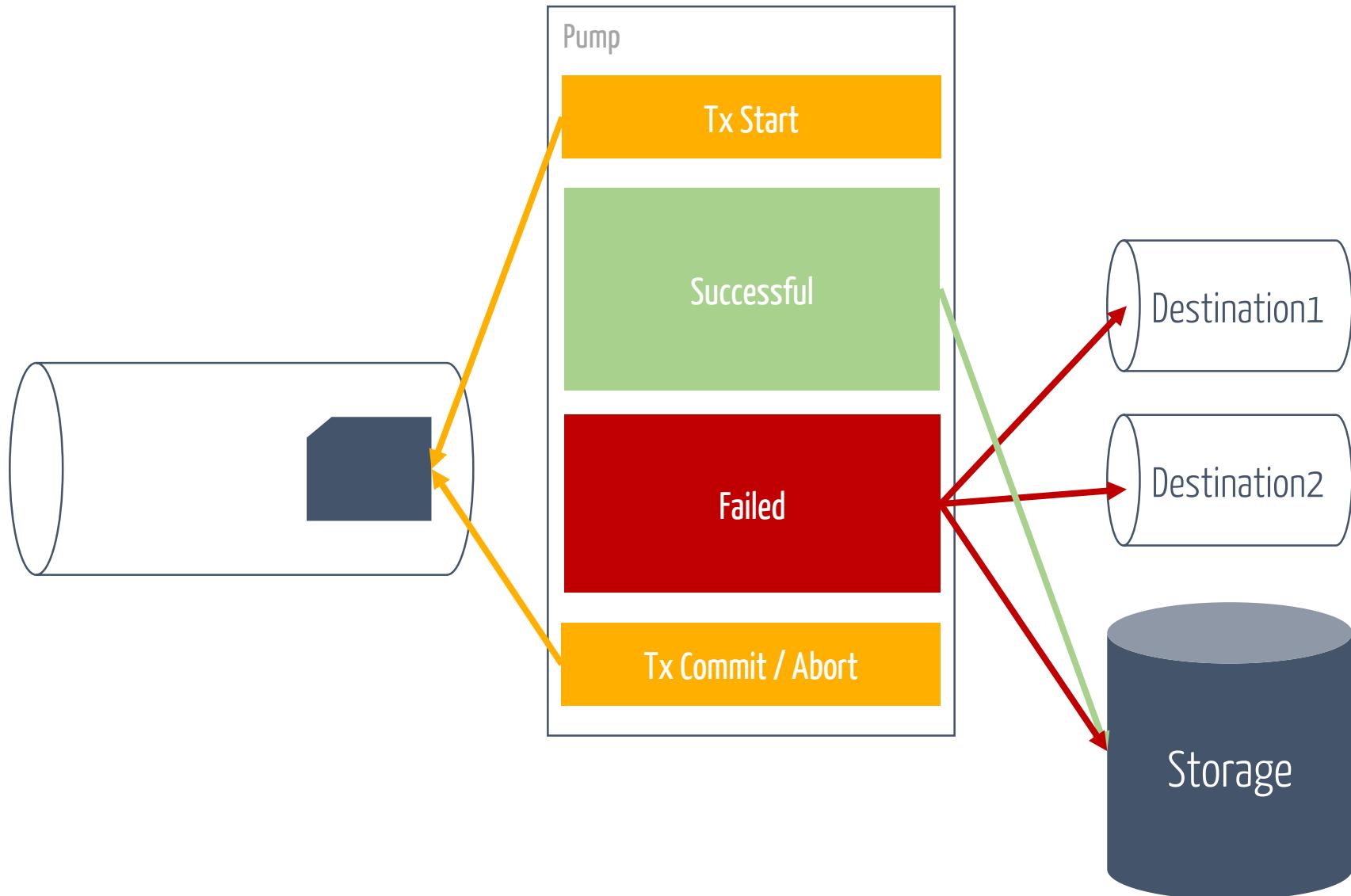
flextensible

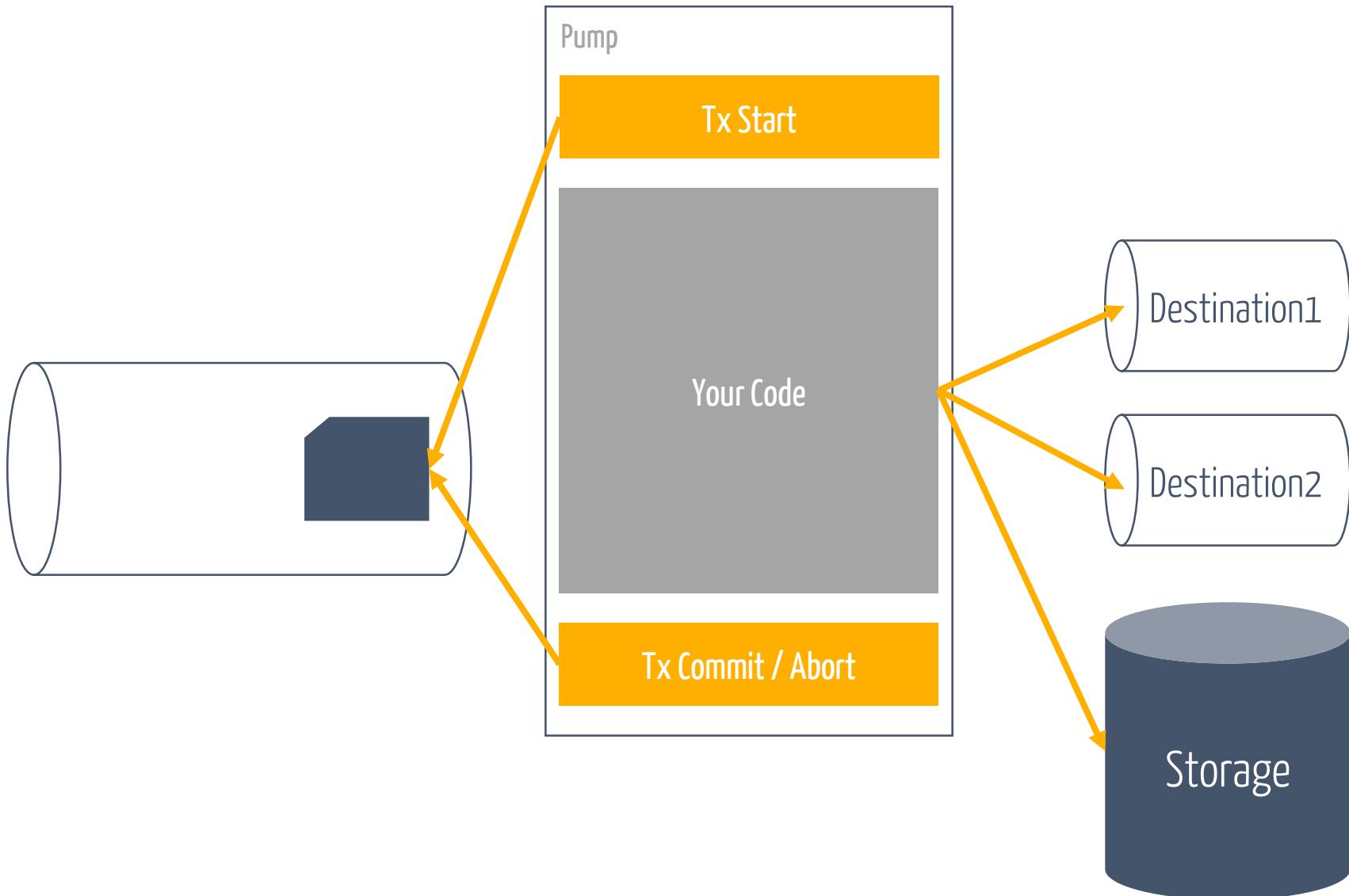
Life beyond

transactions



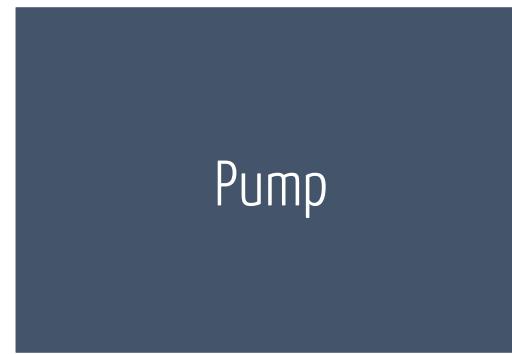


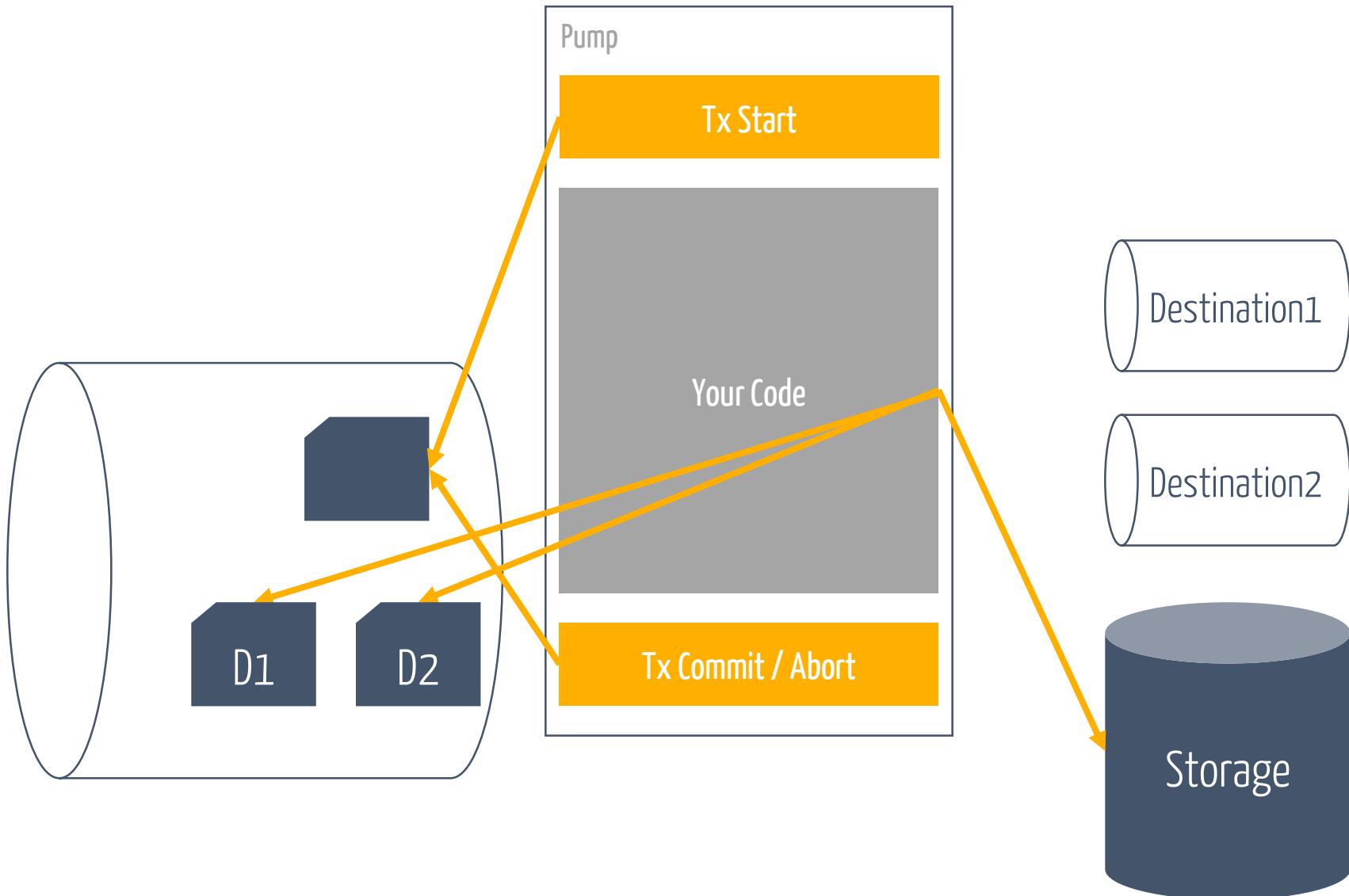




Cloudy with a chance of

failura





Penny Pinching

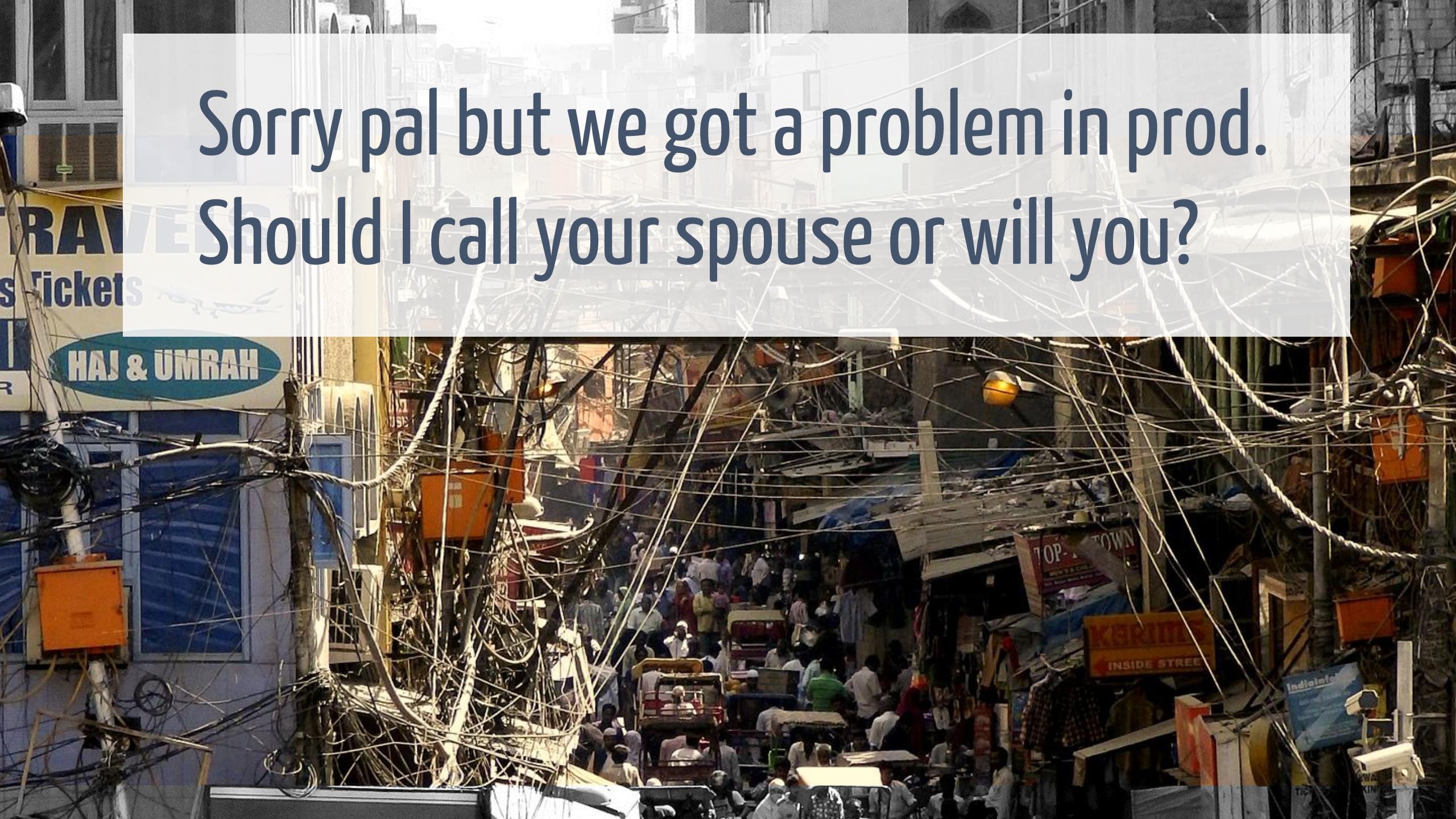
Now we have the

basic bits

Living on the edge of sanity



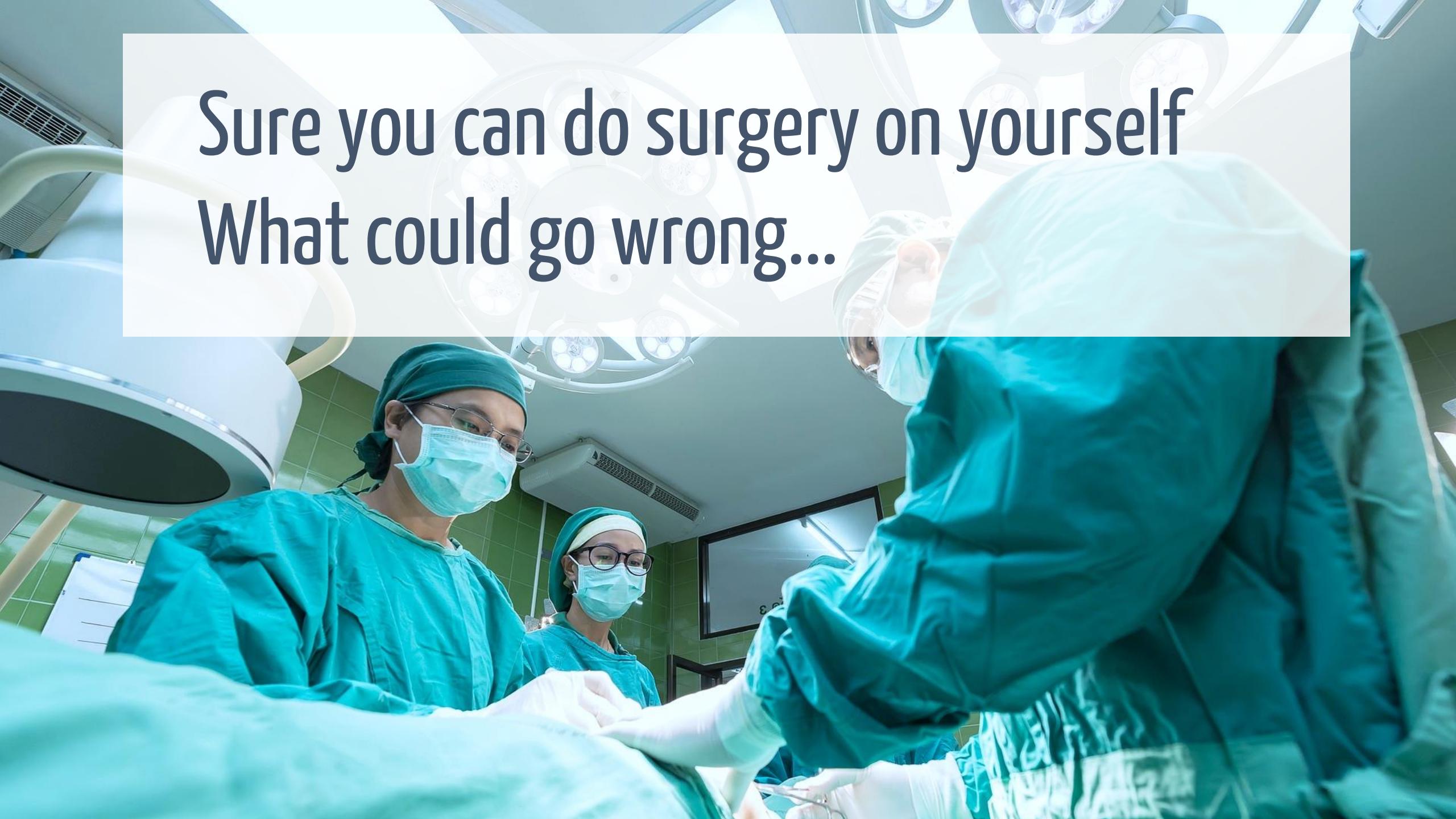
Sorry pal but we got a problem in prod.
Should I call your spouse or will you?



A pair of binoculars is positioned diagonally across the frame, serving as a background for the text. The lenses are clear and reflecting some light, while the body and eyecups are a dark grey or black color.

I want
insights

Sure you can do surgery on yourself
What could go wrong...





Thanks

NServiceBus Quick Start

In this tutorial, we'll see why software systems built on asynchronous messaging using NServiceBus are superior to traditional synchronous HTTP-based web services. We'll also show how NServiceBus guarantees reliability and extensibility that can't be achieved with REST.

This tutorial skips over some concepts and implementation details in order to get up and running quickly. If you'd prefer to go more in-depth, check out our [Introduction to NServiceBus](#) tutorial. It will teach you the NServiceBus API and important concepts you need to learn to build successful message-based software systems.

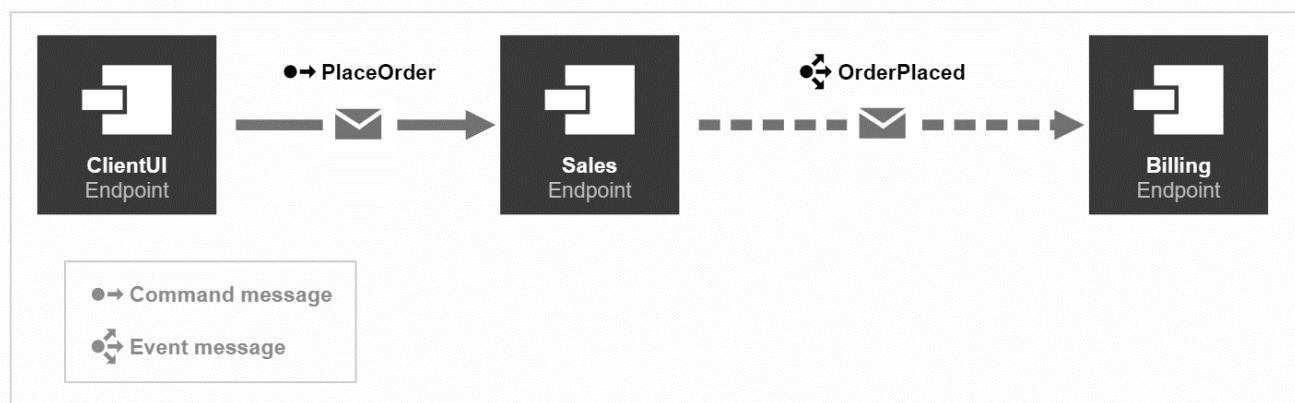
To get started, download the solution, extract the archive, and then open the **RetailDemo.sln** file with Visual Studio 2017[©].

[Download the solution now](#)

Project structure

The solution contains four projects. The **ClientUI**, **Sales**, and **Billing** projects are endpoints that communicate with each other using NServiceBus messages. The **ClientUI** endpoint mimics a web application and is an entry point in our system. The **Sales** and **Billing** endpoints contain business logic related to processing and fulfilling orders. Each endpoint references the **Messages** assembly, which contains the definitions of messages as POCO class files.

As shown in the diagram, the **ClientUI** endpoint sends a **PlaceOrder** command to the **Sales** endpoint. As a result, the **Sales** endpoint will publish an **OrderPlaced** event using the publish/subscribe pattern, which will be received by the **Billing** endpoint.



The solution mimics a real-life retail system, where the command to place an order is sent as a result of a customer interaction, and the actual processing occurs in the background. Publishing an event allows us to isolate the code to bill the credit card from the code to place the order, reducing coupling and making the system easier to maintain over the long term. Later in this tutorial, we'll see how to add a second subscriber in the **Shipping** endpoint which would begin the process of shipping the order.

NServiceBus

- Getting Started
- Service Platform
- Quick Start Tutorial**
- Messaging Basics Tutorial
 - Getting Started
 - Sending a command
 - Multiple endpoints
 - Publishing events
 - Retrying errors
 - Message Replay Tutorial
 - Step by Step Sample
 - Configuration choices Sample
 - Concepts
 - Platform Installer
 - License
 - Extensions
 - Contributing
 - Architectural Principles
 - Bus vs. Broker

Upgrade Guides

Messaging

Hosting

Handlers and Sagas

Testing

Recoverability

Pipeline

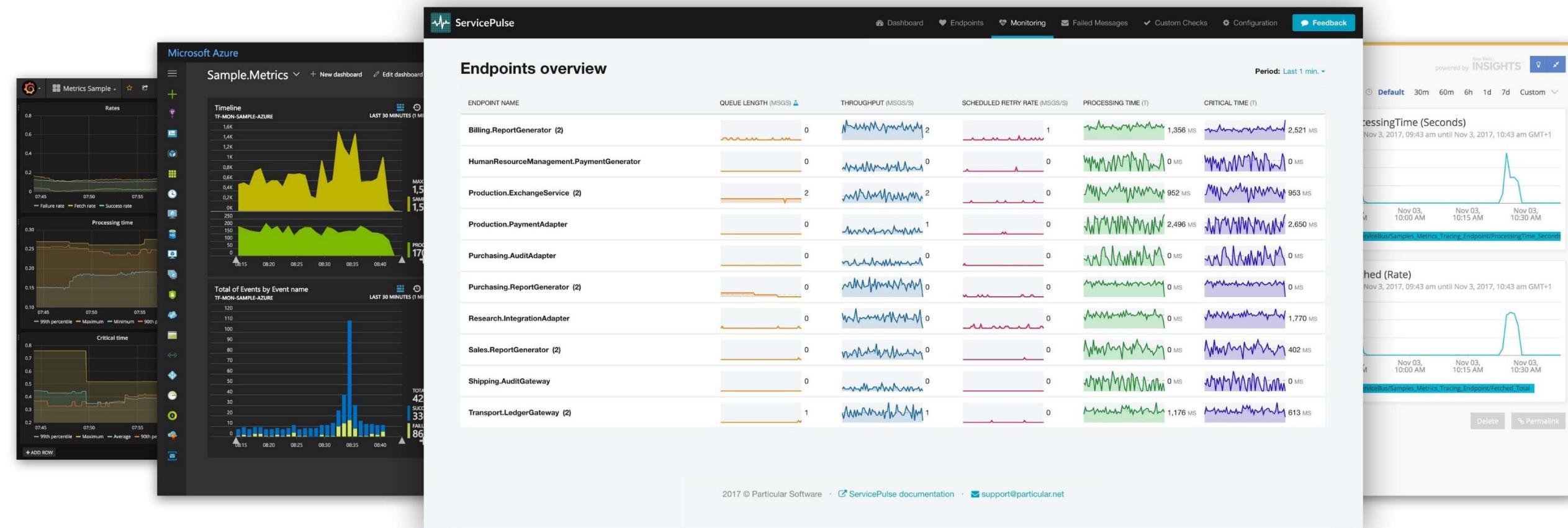
Serialization

Containers

Logging

Security

Operations



go.particular.net/demo

Slides, Links...

github.com/danielmarbach/MessagePump



Q & A



Software Engineer
Enthusiastic Software Engineer
Microsoft MVP

@danielmarbach
particular.net/blog
planetgeek.ch



Thanks