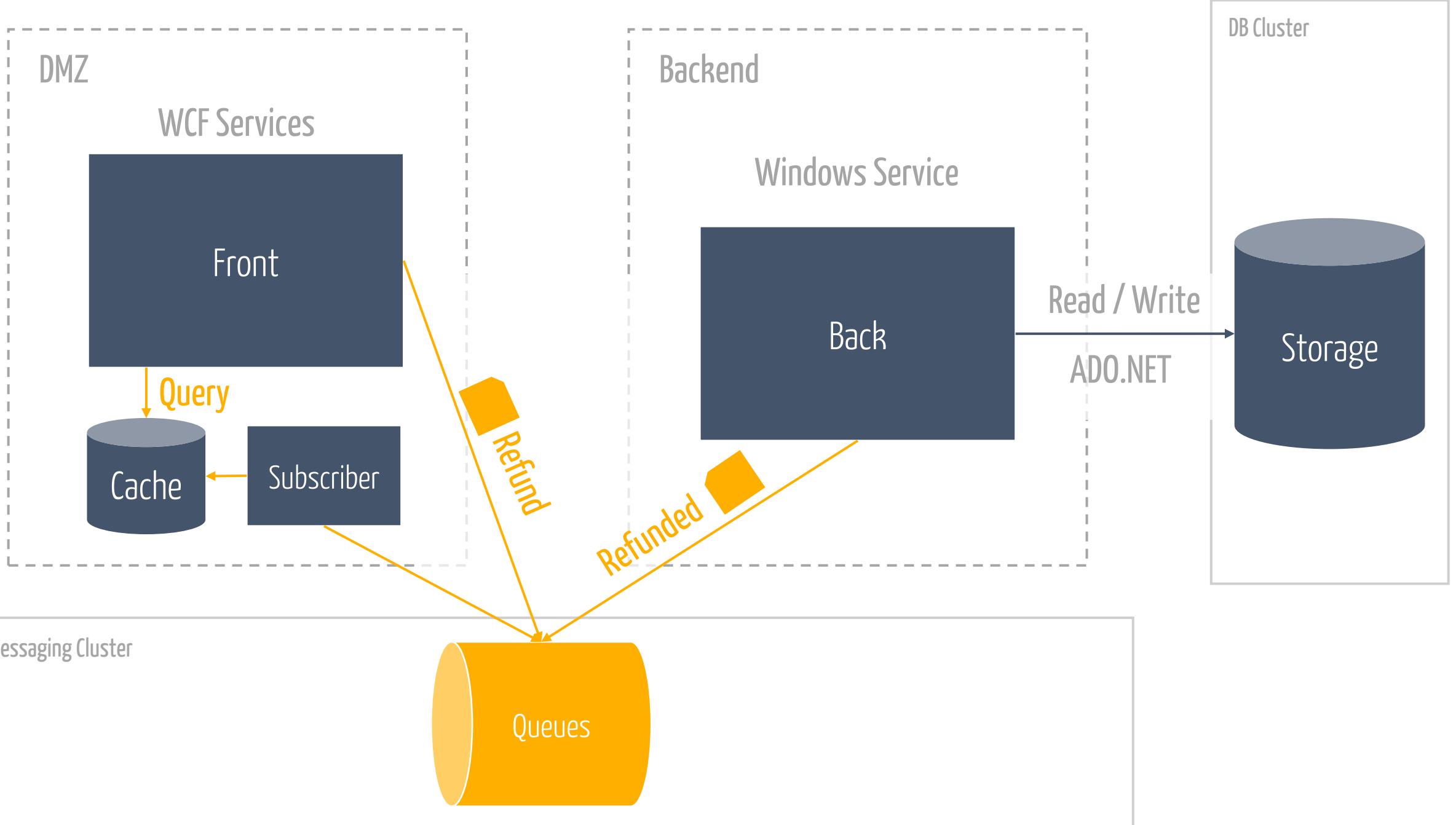


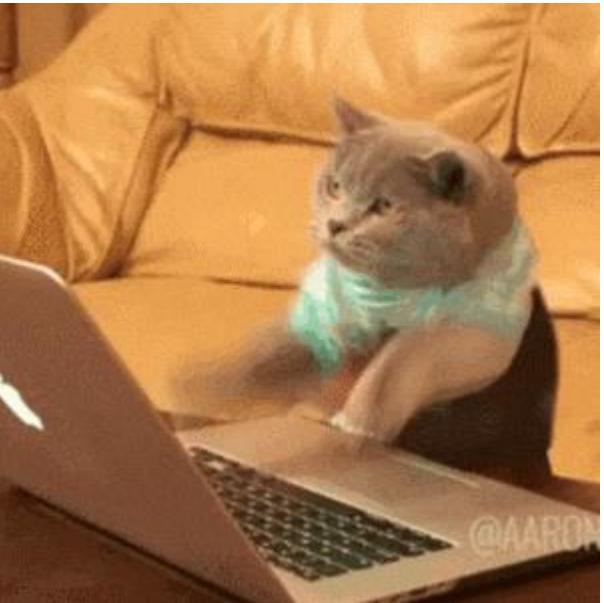


DIY Async Message Pump

Lessons from the trenches



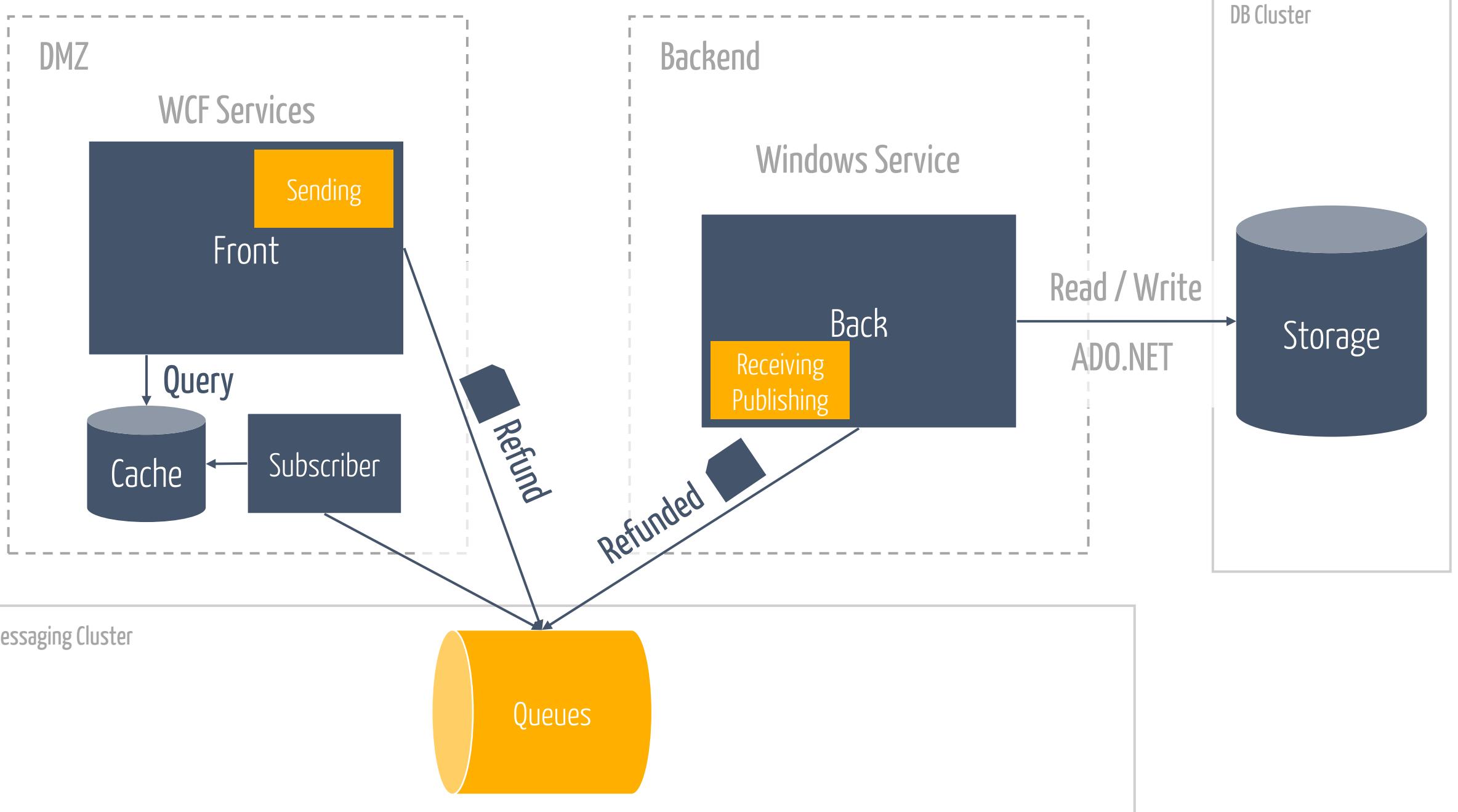
Build



or



Buy



Building the

pump

```
class ThePump : NotInteresting
{
    2 references
    Task pumpTask;
    4 references
    CancellationTokenSource tokenSource;

    public void Start() {
        tokenSource = new CancellationTokenSource();
        var token = tokenSource.Token;
        public async Task Stop() {
            tokenSource.CancelAfter(TimeSpan.FromSeconds(5));
            await pumpTask.ConfigureAwait(false);
        }
        tokenSource.Dispose();
    }
    })());
}

    await pumpTask.ConfigureAwait(false);
    tokenSource.Dispose();
}
```



```
var (payload, headers) = Serialize(message);  
await queue.Send(payload, headers);
```

Easy!

TPL handwaving

Cooporative cancellation 101

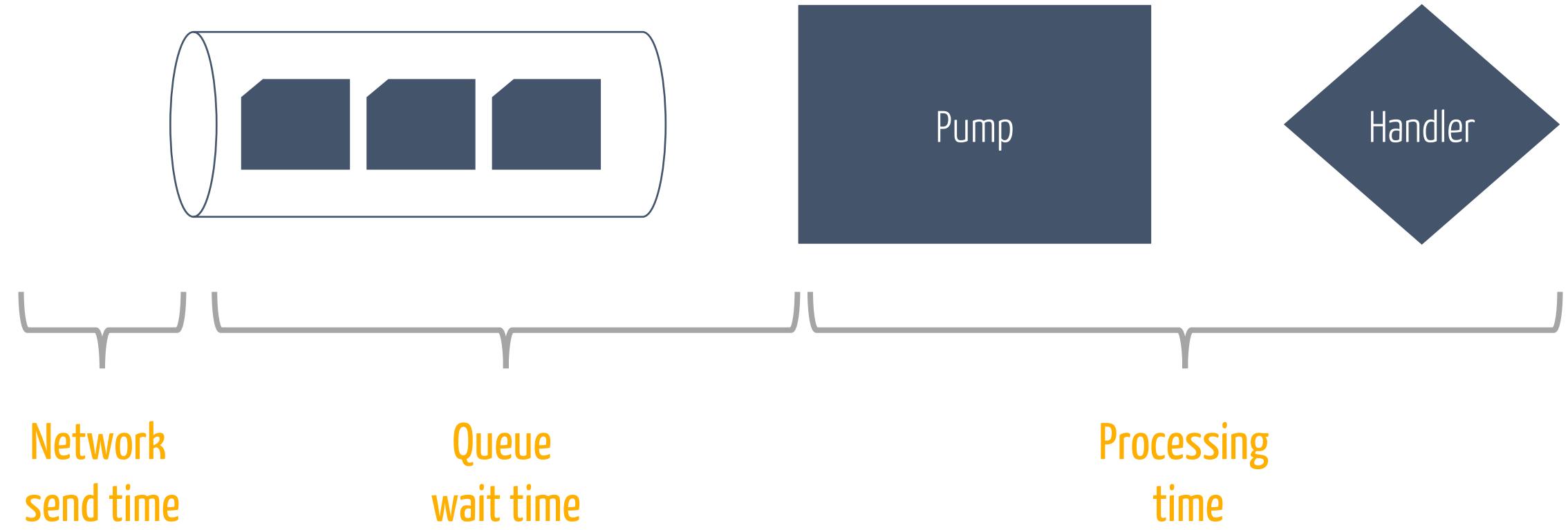
Async / Await

Ship it!

It worked until



Rush hour



C:\p\MessagePump

> dotnet run

Let's throw in some

concurrency

```
class TheConcurrencyPump : NotInteresting
{
    2 references
    Task pumpTask;
    4 references
    CancellationTokenSource tokenSource;

    1 reference
    public void Start() {
        tokenSource = new CancellationTokenSource();

    public void Start() {
        tokenSource = new CancellationTokenSource();
    }

    asvnc Task FetchAndHandleMessage(CancellationToken token = default) {
    public async Task Stop() {
        tokenSource.CancelAfter(TimeSpan.FromMilliseconds(100));

        await pumpTask.ConfigureAwait(false);

        WriteLine("Are we done yet?");

        tokenSource.Dispose();
    }

}

    public async Task Stop() {
        tokenSource.CancelAfter(TimeSpan.FromMilliseconds(100));

        await pumpTask.ConfigureAwait(false);

        WriteLine("Are we done yet?");

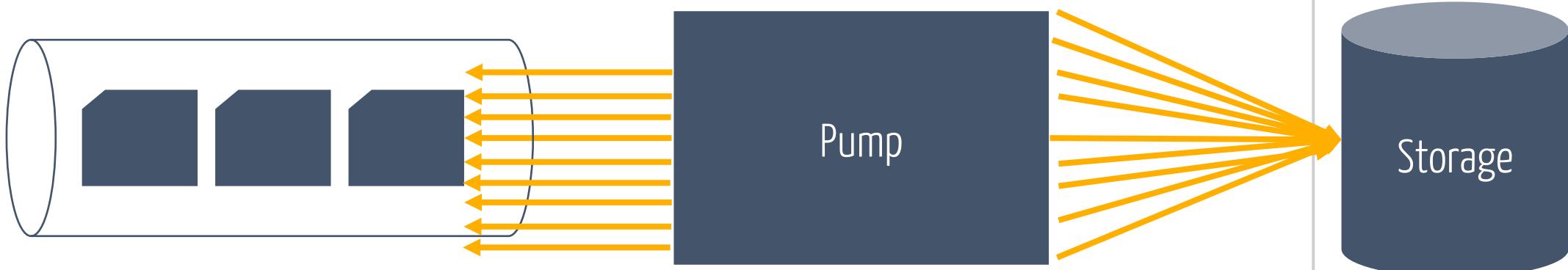
        tokenSource.Dispose();
    }
}
```

Just a tiny change...
Introduce Fire & Forget
Ship it!

It worked until



Rush hour



DB Cluster

Storage

C:\p\MessagePump

> dotnet run

ThePump executing

Pumping 1...

Pumping 2...

Pumping 3...

Pumping 4...

Pumping 5...

Press to continue...

Better limit

concurrency

```
class TheLimitingConcurrencyPump : NotInteresting
{
    #region Hide ...
    1 reference
    public void Start()
    {
        tokenSource = new CancellationTokenSource();
        semaphore = new SemaphoreSlim(MaxConcurrency);

        public void Start()
    }

    async Task FetchAndHandleAndRelease(SemaphoreSlim semaphore, CancellationToken token = default) {
        public async Task Stop() {
            tokenSource.CancelAfter(TimeSpan.FromSeconds(5));

            await Graceful(pumpTask).ConfigureAwait(false);

            while (semaphore.CurrentCount != MaxConcurrency) {
                await Task.Delay(50).ConfigureAwait(false);
            }
        }

        as
    }

    tokenSource.Dispose();

    }

}

await Graceful(pumpTask).ConfigureAwait(false);

while (semaphore.CurrentCount != MaxConcurrency) {
    await Task.Delay(50).ConfigureAwait(false);
}

tokenSource.Dispose();
}
```

Pumping 62577...
Pumping 62576...
Pumping 62575...
Pumping 62574...
Pumping 62573...
Pumping 62572...
Pumping 62571...
Pumping 62570...
Pumping 62569...
Pumping 62568...
Pumping 62567...
Pumping 62566...
Pumping 54400...
Pumping 54476...
Pumping 54519...
Pumping 54518...
Pumping 54391...
Pumping 54401...

Semaphore controls floodgate
Ship it!

It worked





As time passes

Make it

flextensible

```
public class ThePumpHowItProbablyLooksLikeInReality : NotInteresting
{
    2 references
    | private const int MaxConcurrency = 3;
    4 references
    | private CancellationTokenSource tokenSource;
    4 references
    | private SemaphoreSlim semaphore;
    2 references
    | private Task pumpTask;

    0 references
    public void Start() {
        tokenSource = new CancellationTokenSource();

        var token = tokenSource.Token;

        semaphore = new SemaphoreSlim(MaxConcurrency);

        pumpTask = ((Func<Task>)(async () => {
            while (!token.IsCancellationRequested)
            {
                await semaphore.WaitAsync(token);

                FireAndForget(FetchAndHandleAndReleaseWithMiddleware(semaphore, token));
            }
        }))();
    }

    0 references
    public async Task Stop() {
        tokenSource.CancelAfter(TimeSpan.FromSeconds(5));
        await Graceful(pumpTask).ConfigureAwait(false);

        while (semaphore.CurrentCount != MaxConcurrency) {
            await Task.Delay(50).ConfigureAwait(false);
        }

        tokenSource.Dispose();
    }
}
```



```
1 reference
async Task FetchAndHandleAndReleaseWithMiddleware(SemaphoreSlim semaphore, CancellationToken token = default) {
    using (var transaction = CreateTransaction()) {
        var (payload, headers) = await ReadFromQueue(token, transaction).ConfigureAwait(false);
        var message = Deserialize(payload, headers);

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

                await middleware(message).ConfigureAwait(false);

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

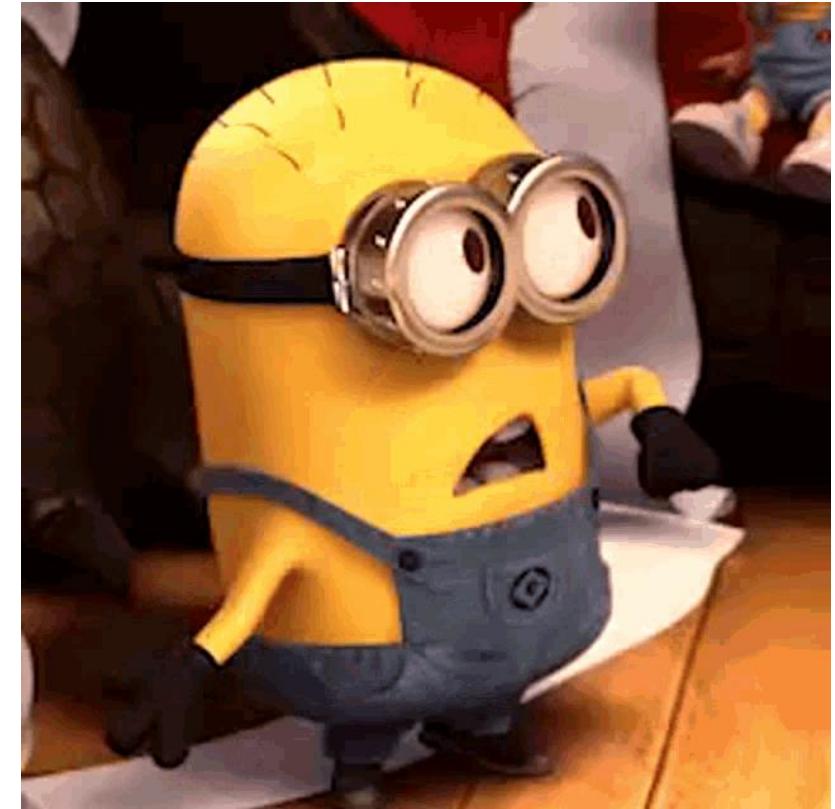
```
1 reference
async Task Middleware1(HandlerContext context, Func<HandlerContext, CancellationToken, Task> next, CancellationToken token = default) {
    await next(context, token);
}
```

```
1 reference
async Task Middleware2(HandlerContext context, Func<HandlerContext, CancellationToken, Task> next, CancellationToken token = default) {
    var handlers = context.Provider.Resolve<IHandleMessage<Message>>();

    foreach (var handler in handlers) {
        await handler.Handle(context.Message, context, token).ConfigureAwait(false);
    }

    await next(context, token).ConfigureAwait(false);
}
```

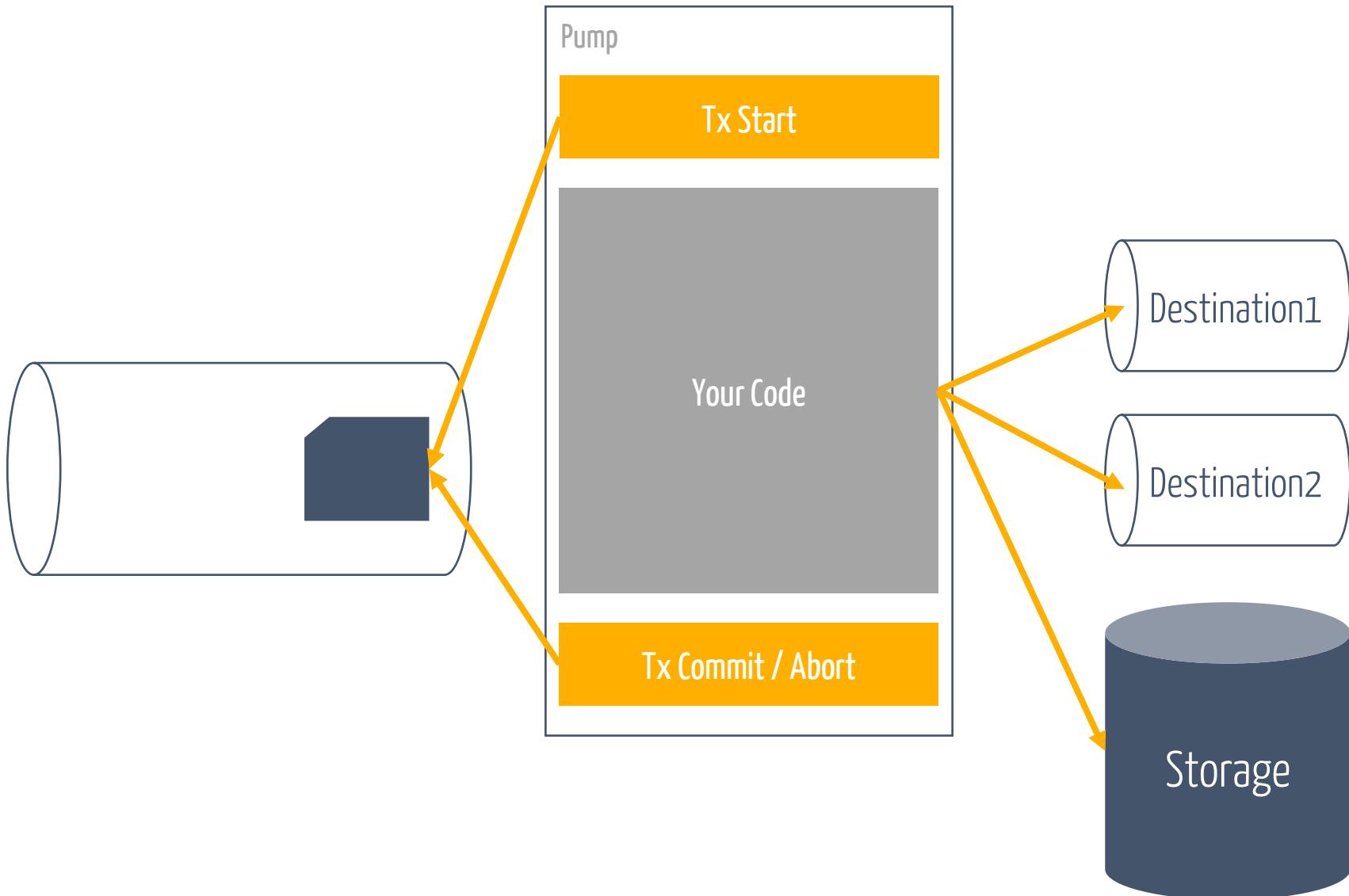
```
0 references
class MessageHandler : IHandleMessage<Message>
{
    1 reference
    public async Task Handle(Message message, HandlerContext context, CancellationToken token = default)
    {
        await Task.Delay(1000).ConfigureAwait(false);
        Pumping(message);
        await context.Send(new Message()).ConfigureAwait(false);
    }
}
```

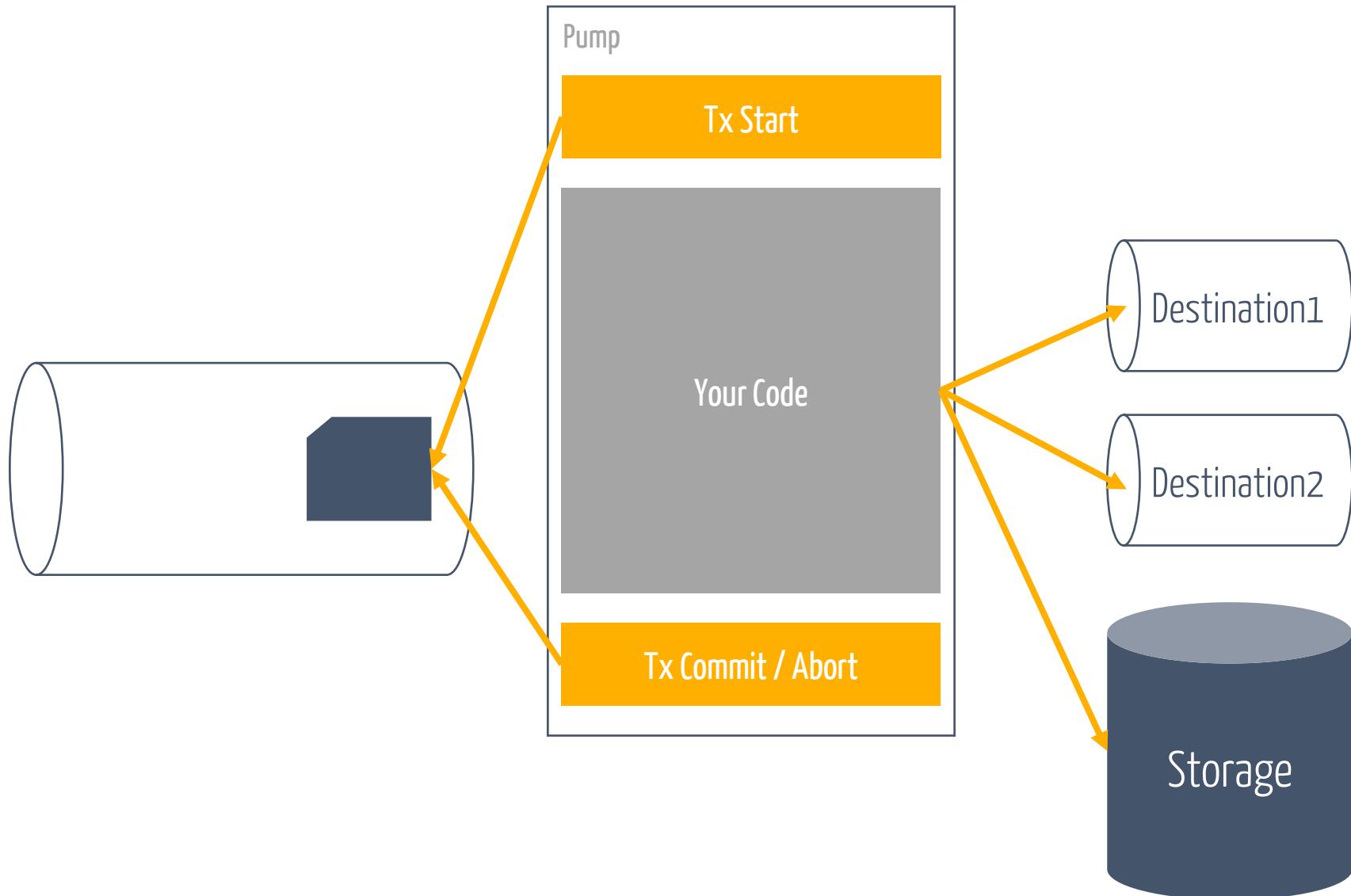


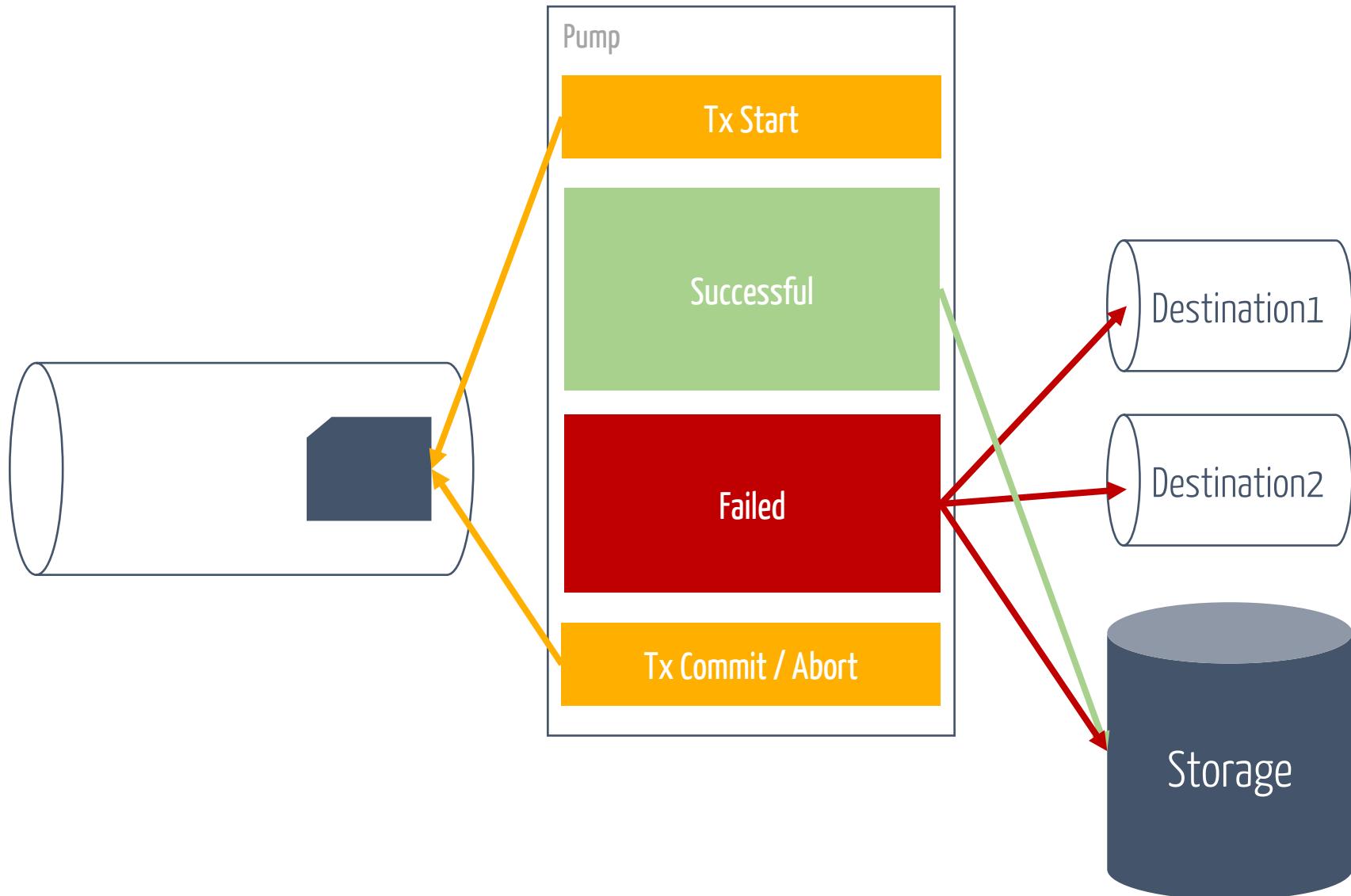


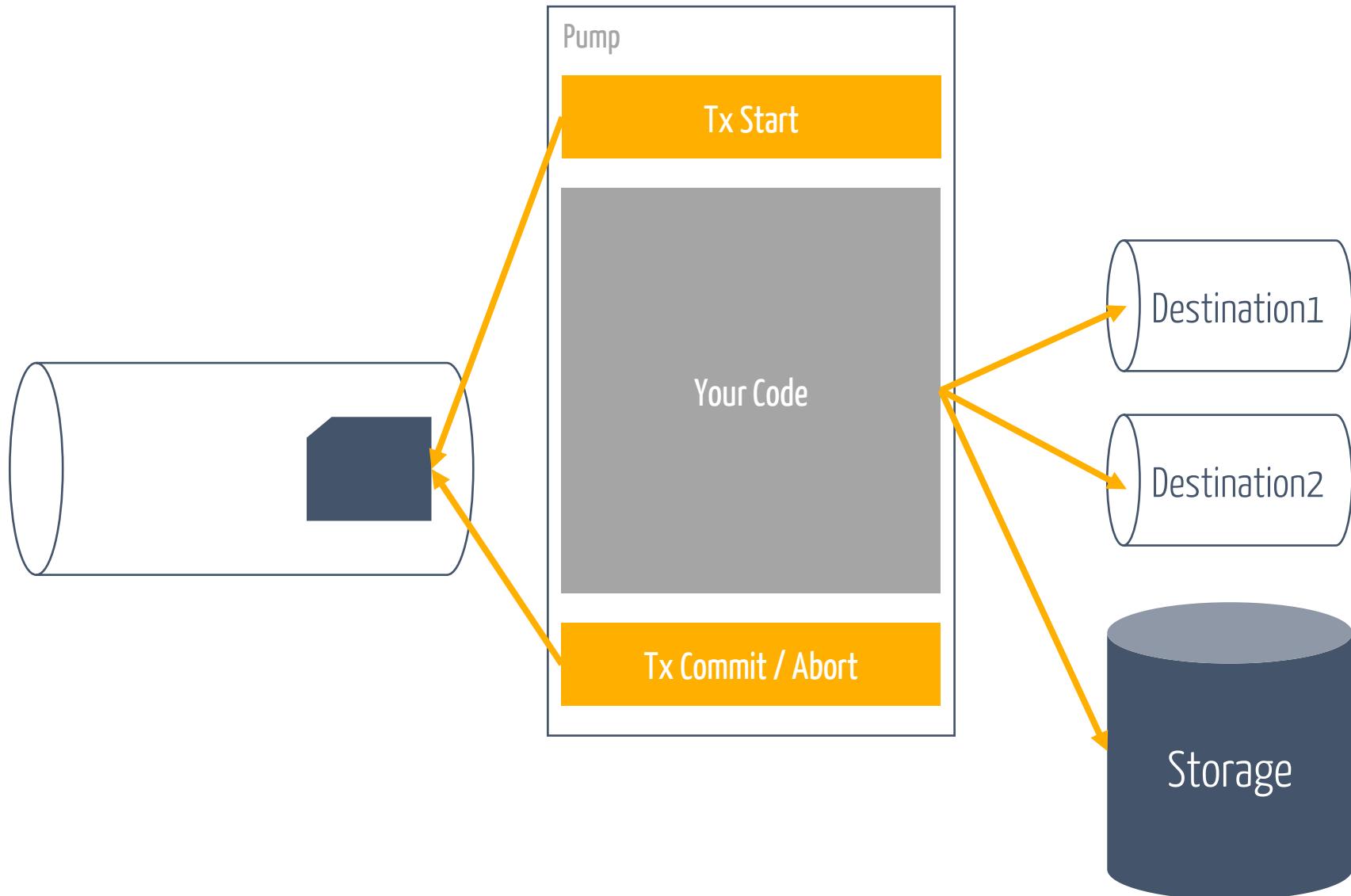
Life beyond

transactions



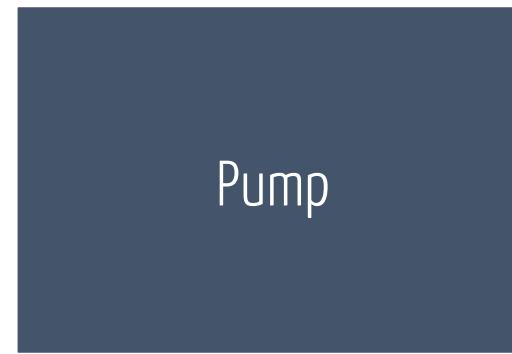


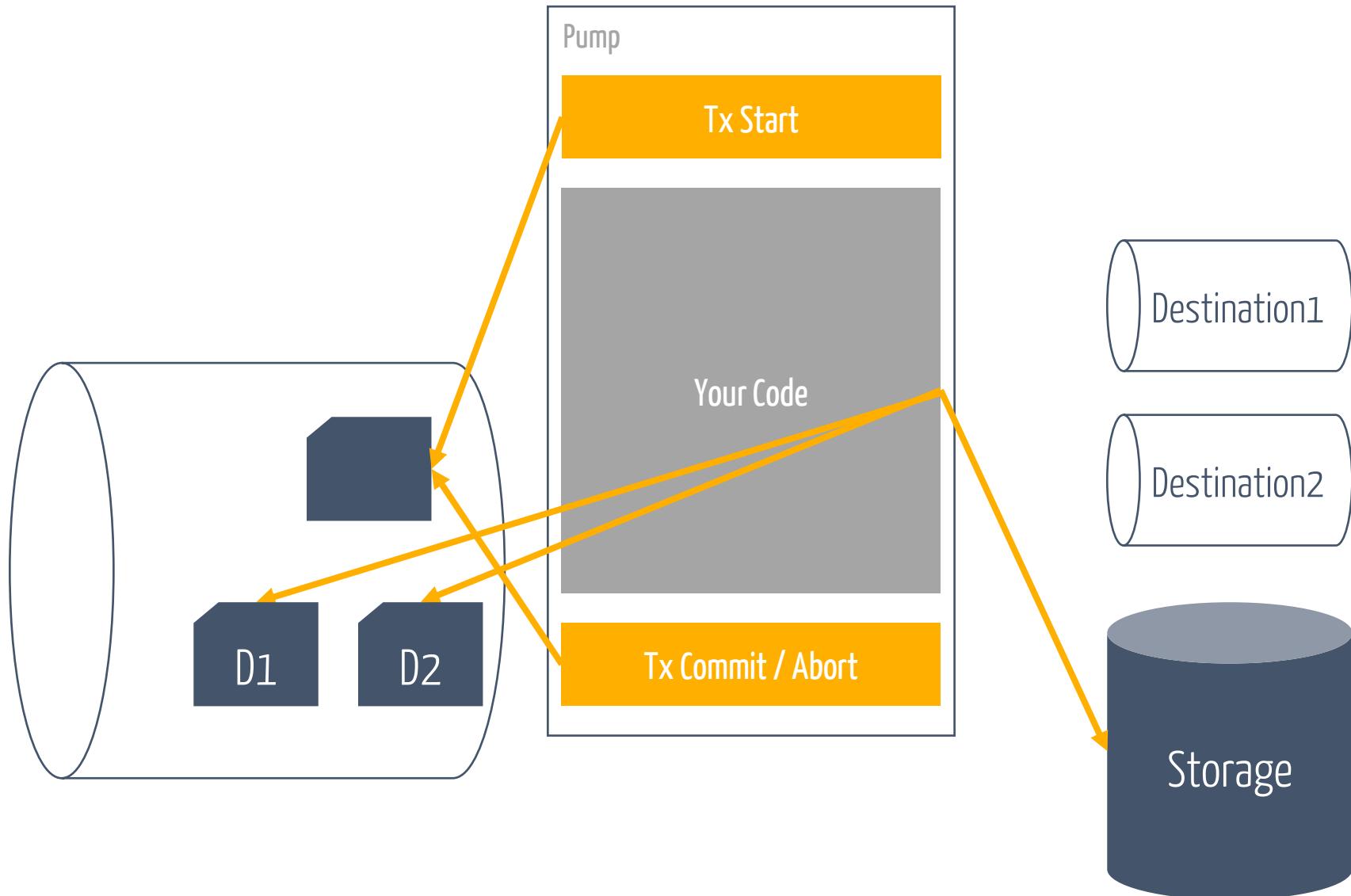




Cloudy with a chance of

failura



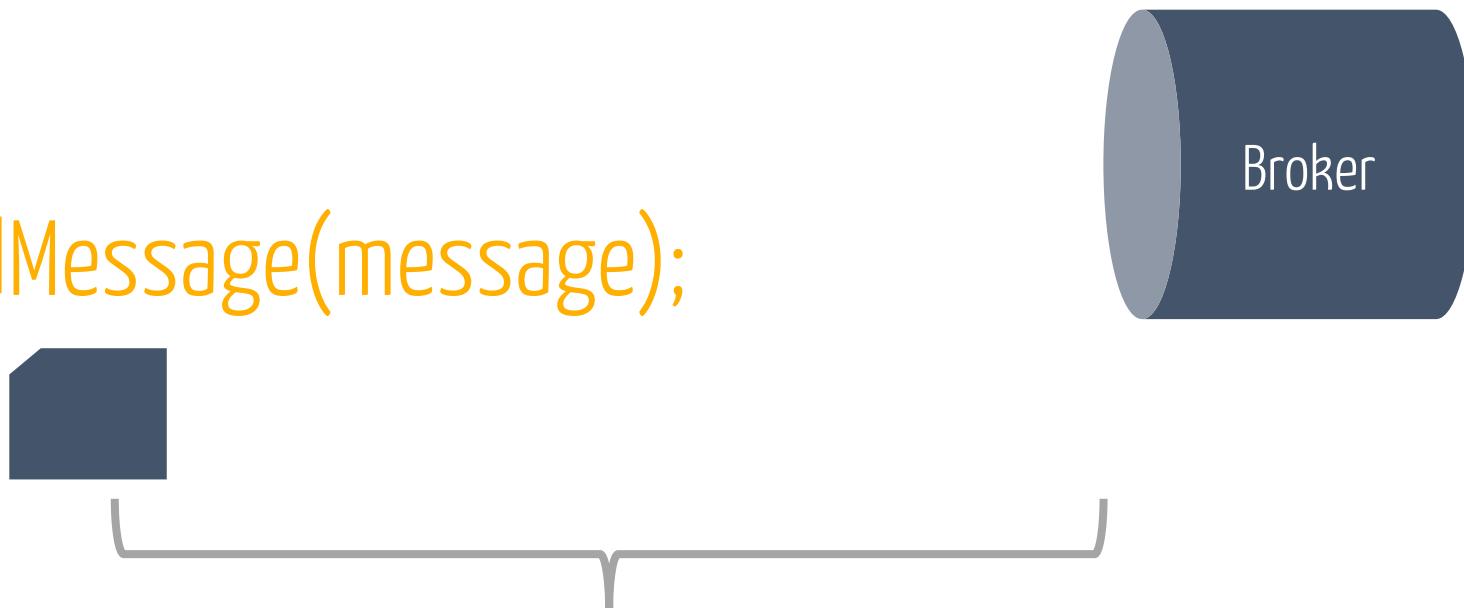


Penny Pinching

```
foreach(var message in messsagesToBeSent) {
```

```
    await SendMessage(message);
```

```
}
```

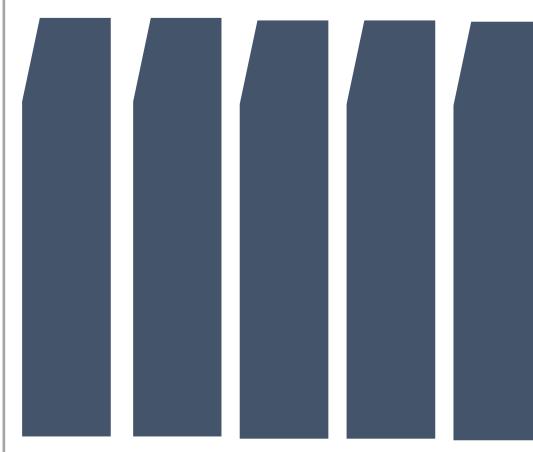
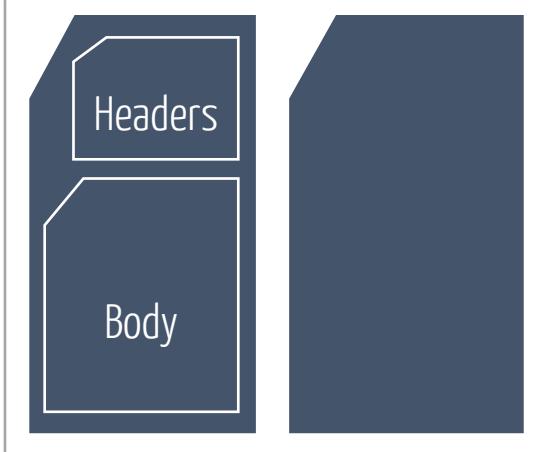


Latency / Operation costs

Destination 1

Destination 2

256 KB

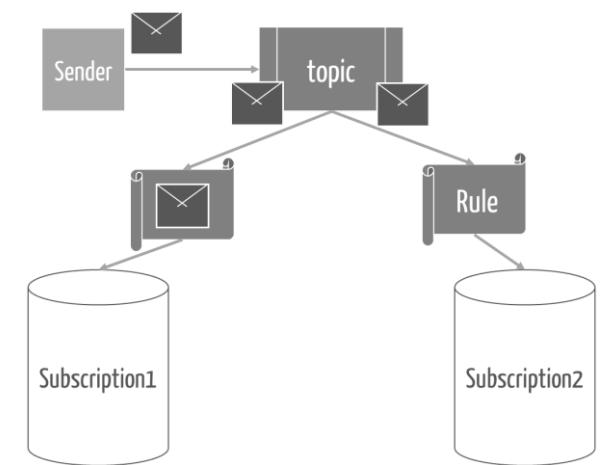
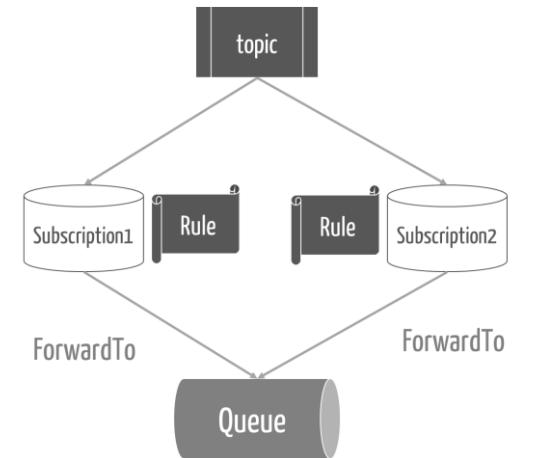


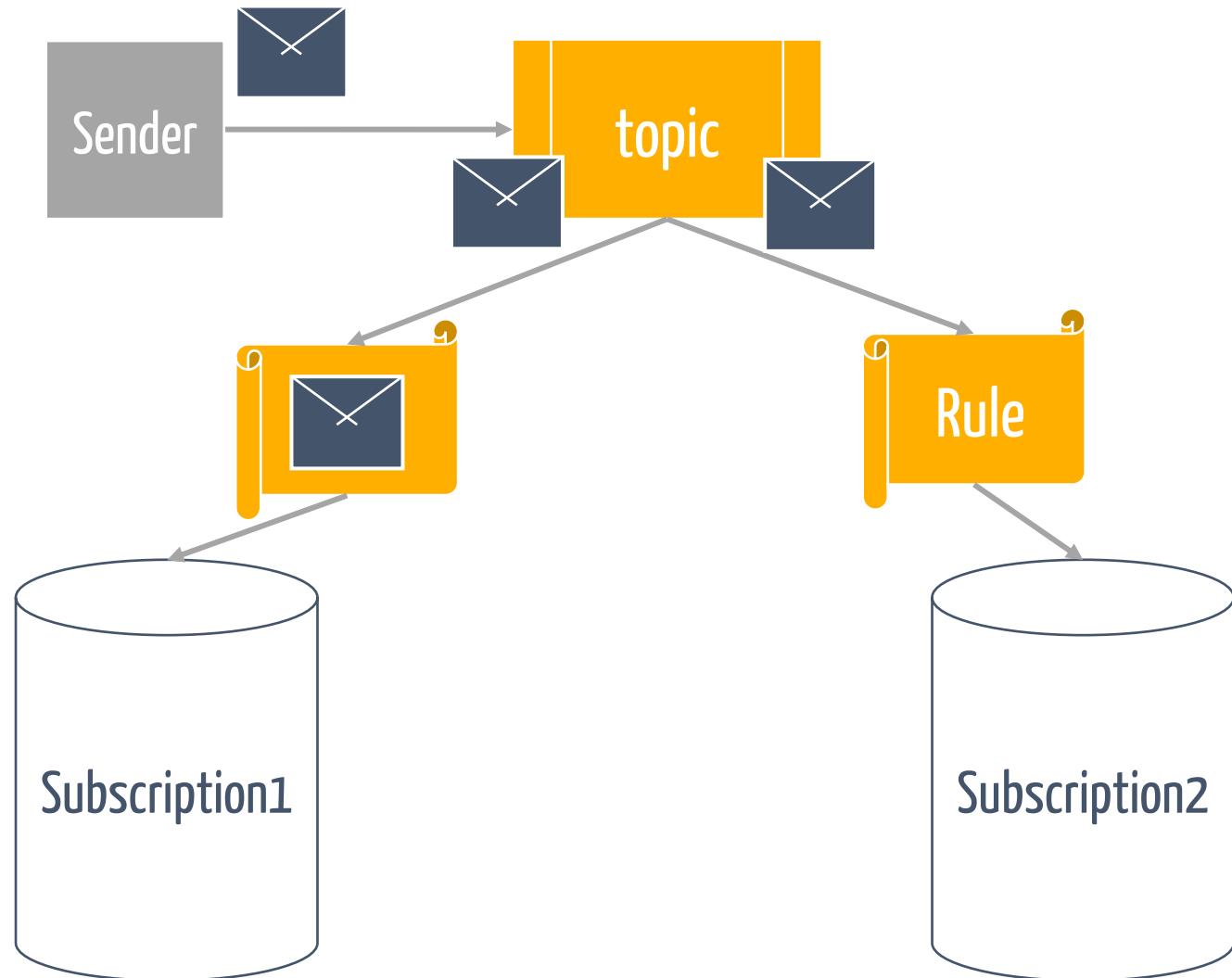
All the messages to be sent

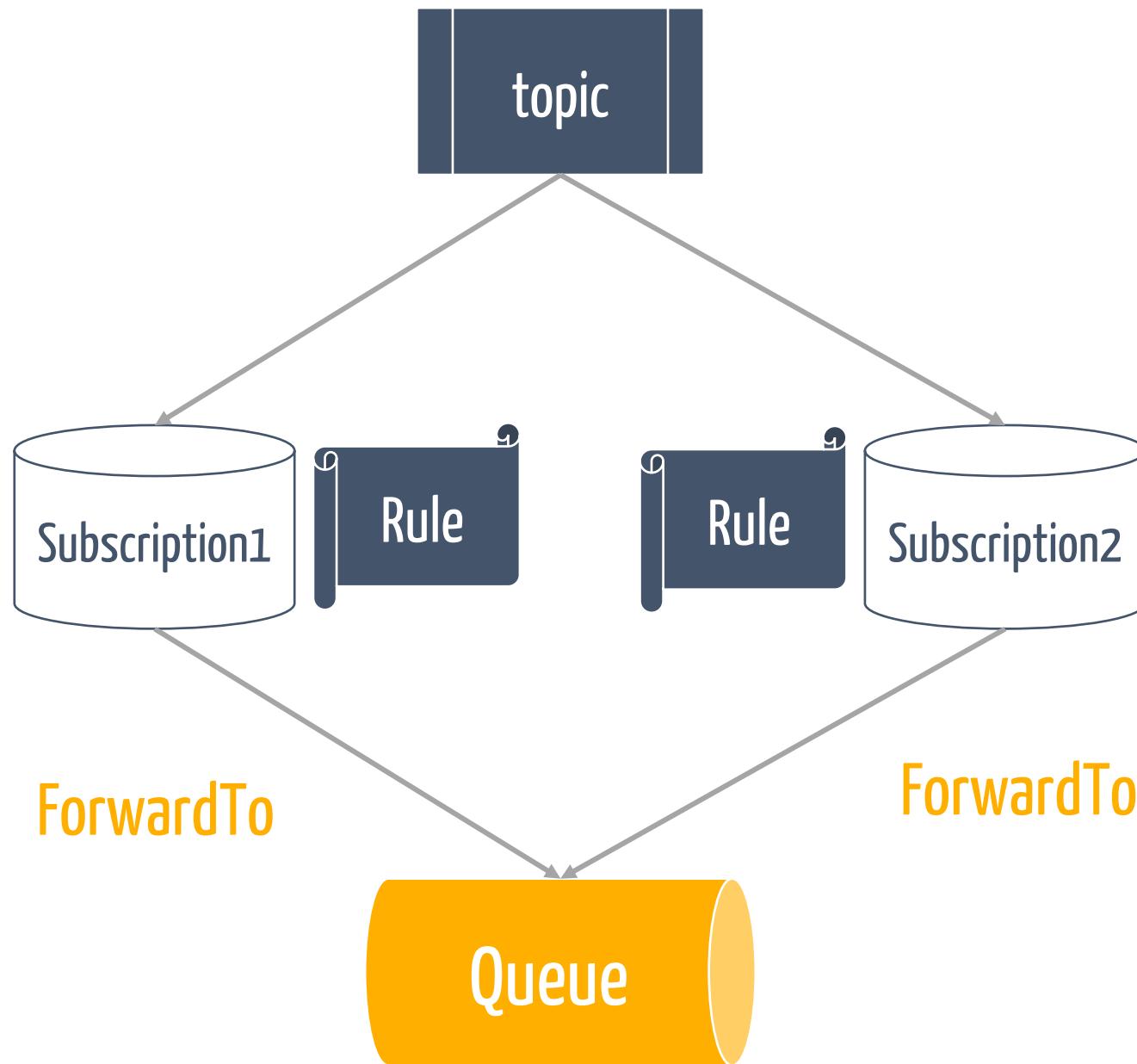
Now we have the

basic bits

Pub/Sub



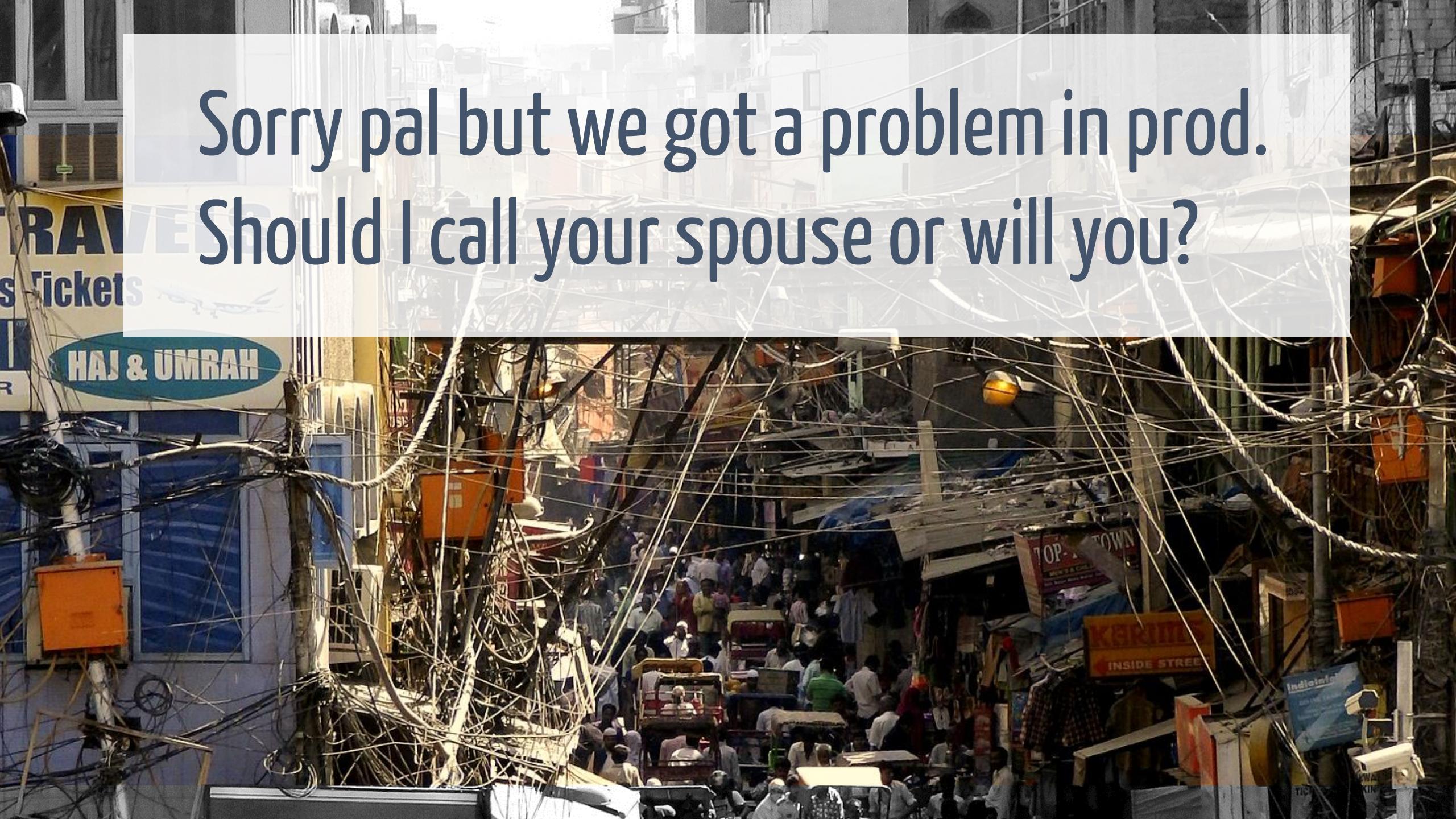


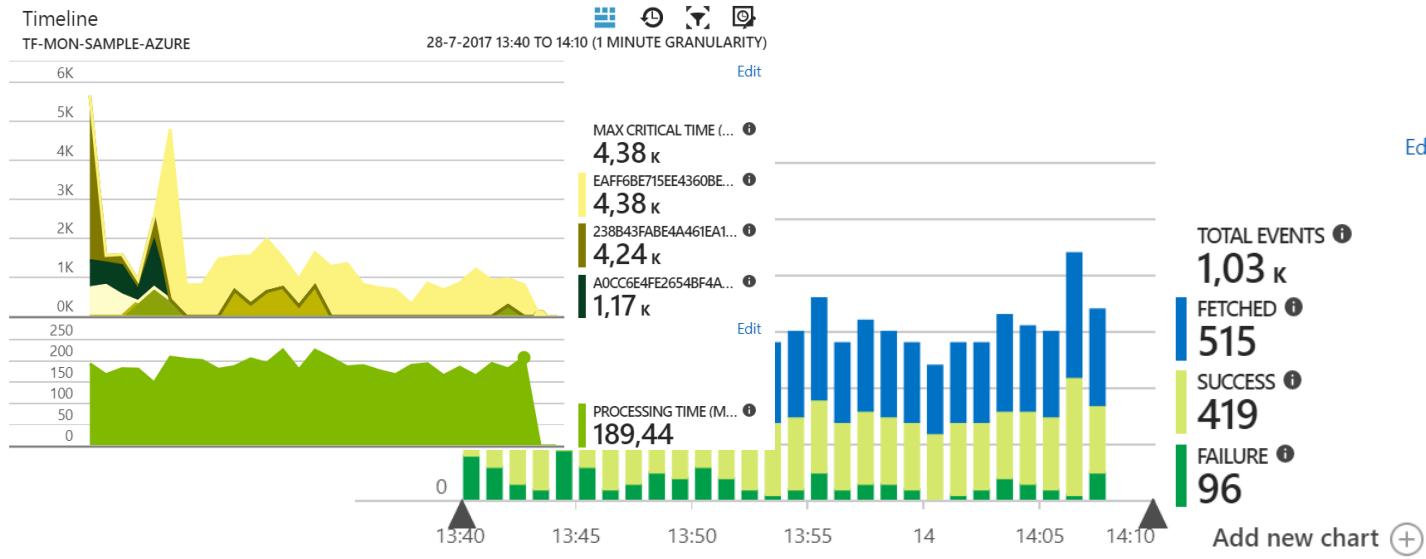
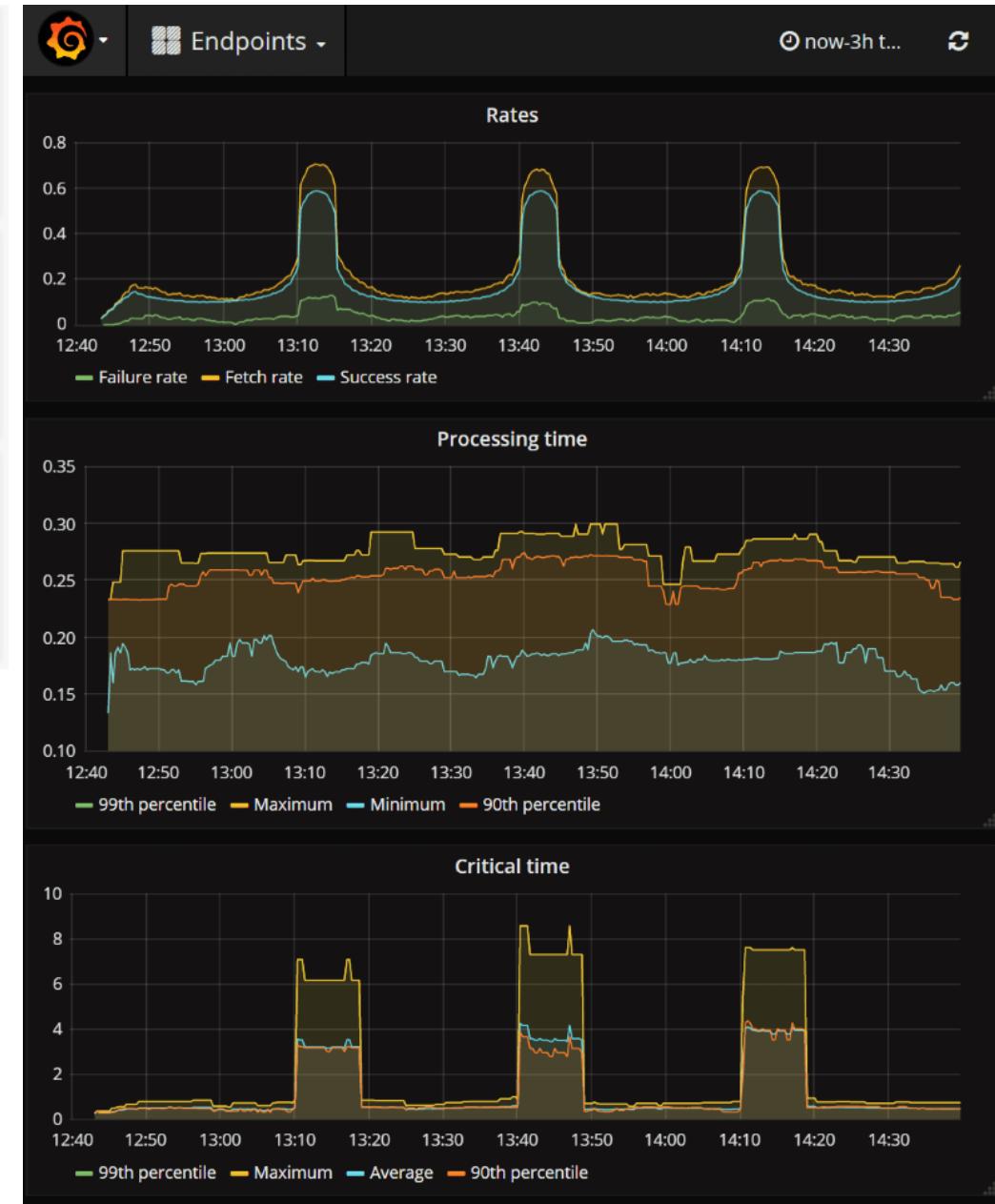
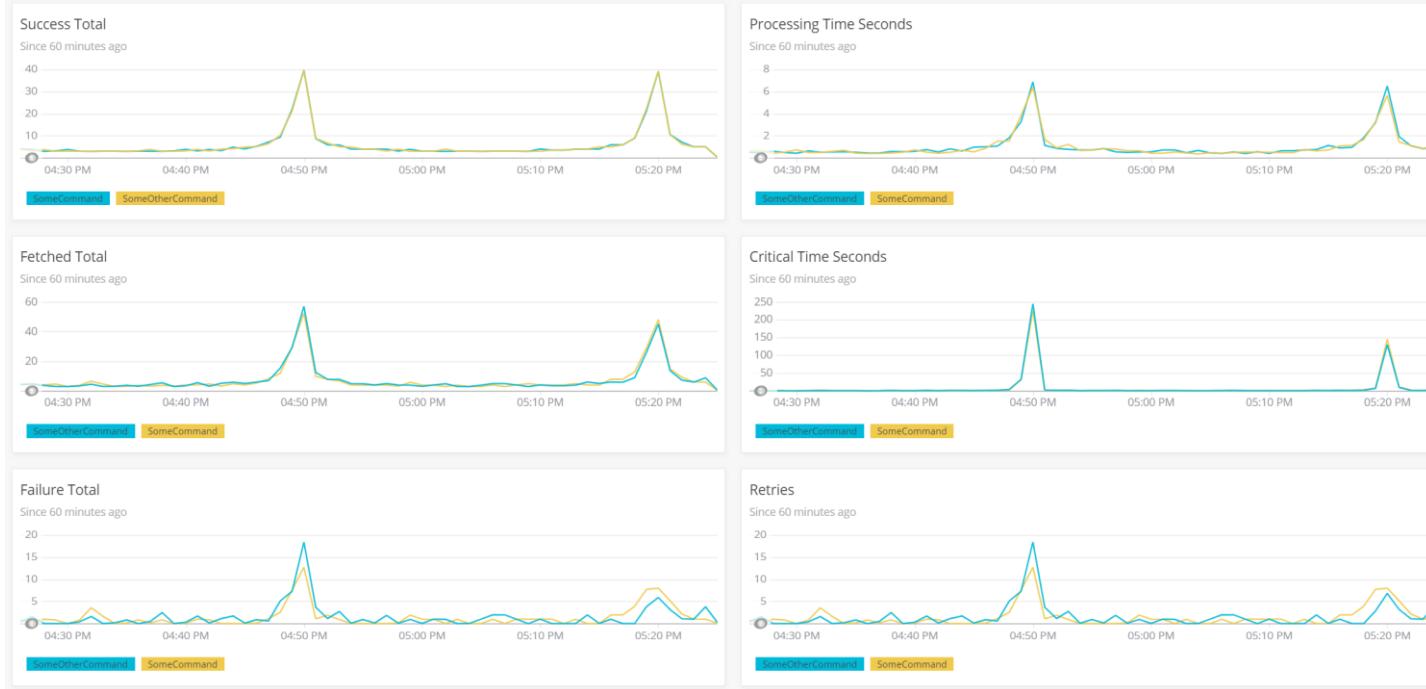


Living on the edge of sanity



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







Try it yourself



Thanks

Slides, Links...

github.com/danielmarbach/MessagePump



Software Engineer
Microsoft MVP



@danielmarbach
particular.net/blog
planetgeek.ch



Thanks