

Durable Executions in the Face of (some) Failures

Andrew Fitz Gibbon (aka "Fitz")
Staff Developer Advocate
@ Temporal Technologies

✉ fitz@temporal.io
🐦 [@fitzface](https://twitter.com/fitzface)

Hi! My name's Fitz and I'm a Developer Advocate with Temporal Technologies.

Who here is familiar with the 8 fallacies of distributed systems?

The 8 Fallacies

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

This talk is not about that, but here they are as a reminder, just in case. ... What if they were actually true? Bear with me...

Simple Program

```
func OrderItem(o OrderRequest) error {  
    c, err := api_client.New()  
    defer c.Close()  
  
    order, err := c.InitOrder(o) ← Initiate  
    status, err := c.FulfillOrder(order) ← Fulfill  
  
    db, err := archival.NewClient()  
    defer db.Close()  
  
    err = db.Persist(order, status) ← Archive/Log  
}
```

So, let's look at a simple go-esque program for placing an order for something.

It's got three main steps:

- Initiate the order, say, to call a payment provider.
- Actually fulfill the order
- And then when done, log it somewhere so that our business teams can do whatever they need to do with that data.

Unreliable Services

```
func OrderItem(o OrderRequest) error {
    // err := api_client.Create()
    defer c.Close()

    order, err := c.CreateOrder(o)
    status, err := c.FulfillOrder(order)

    db, err := archive.NewClient()
    defer db.Close()

    err = db.Persist(order, status)
}
```

And the problem of course is that things are unreliable.

It could fail in initiating the order...
<click>

... fulfillment
<click>

Archiving
<click>

even in creating the API clients.

<click>

Really the whole thing is unreliable.

"Proper" Error Handling?

```
func OrderItem(o OrderRequest) error {
    c, err := api_client.New()
    if err != nil { log.Fatal(err) }
    defer c.Close()

    order, err := c.InitOrder(o)
    if err != nil { log.Fatal(err) }

    status, err := c.FulfillOrder(order)
    if err != nil { log.Fatal(err) }

    db, err := archival.NewClient()
    if err != nil { log.Fatal(err) }
    defer db.Close()
    err = db.Persist(order, status)
    if err != nil { log.Fatal(err) }
}
```

So of course we add in some error handling. And this is great but now our whole program crashes if anything bad happens at all.

"Better"?

Initiate

Fulfill

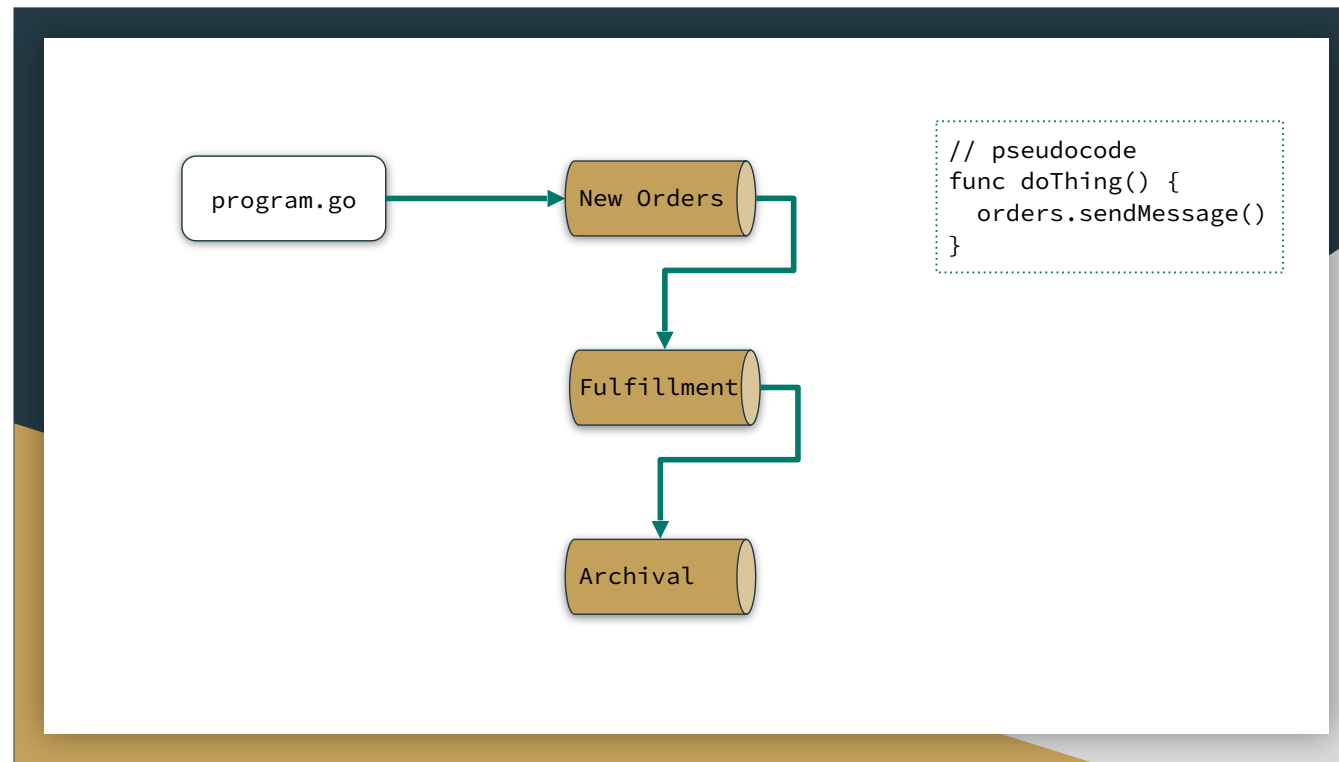
Archive/Log

```
1 // ...
2 // ...
3 // ...
4 // ...
5 // ...
6 // ...
7 // ...
8 // ...
9 // ...
10 // ...
11 // ...
12 // ...
13 // ...
14 // ...
15 // ...
16 // ...
17 // ...
18 // ...
19 // ...
20 // ...
21 // ...
22 // ...
23 // ...
24 // ...
25 // ...
26 // ...
27 // ...
28 // ...
29 // ...
30 // ...
31 // ...
32 // ...
33 // ...
34 // ...
35 // ...
36 // ...
37 // ...
38 // ...
39 // ...
40 // ...
41 // ...
42 // ...
43 // ...
44 // ...
45 // ...
46 // ...
47 // ...
48 // ...
49 // ...
50 // ...
51 // ...
52 // ...
53 // ...
54 // ...
55 // ...
56 // ...
57 // ...
58 // ...
59 // ...
60 // ...
61 // ...
62 // ...
63 // ...
64 // ...
65 // ...
66 // ...
67 // ...
68 // ...
69 // ...
70 // ...
71 // ...
72 // ...
73 // ...
74 // ...
75 // ...
76 // ...
77 // ...
78 // ...
79 // ...
80 // ...
81 // ...
82 // ...
83 // ...
84 // ...
85 // ...
86 // ...
87 // ...
88 // ...
89 // ...
90 // ...
91 // ...
92 // ...
93 // ...
94 // ...
95 // ...
96 // ...
97 // ...
98 // ...
99 // ...
100 // ...
```

So instead of crashing outright, we add retries, timers, distributed key stores...

<click>
and those three API calls exist,

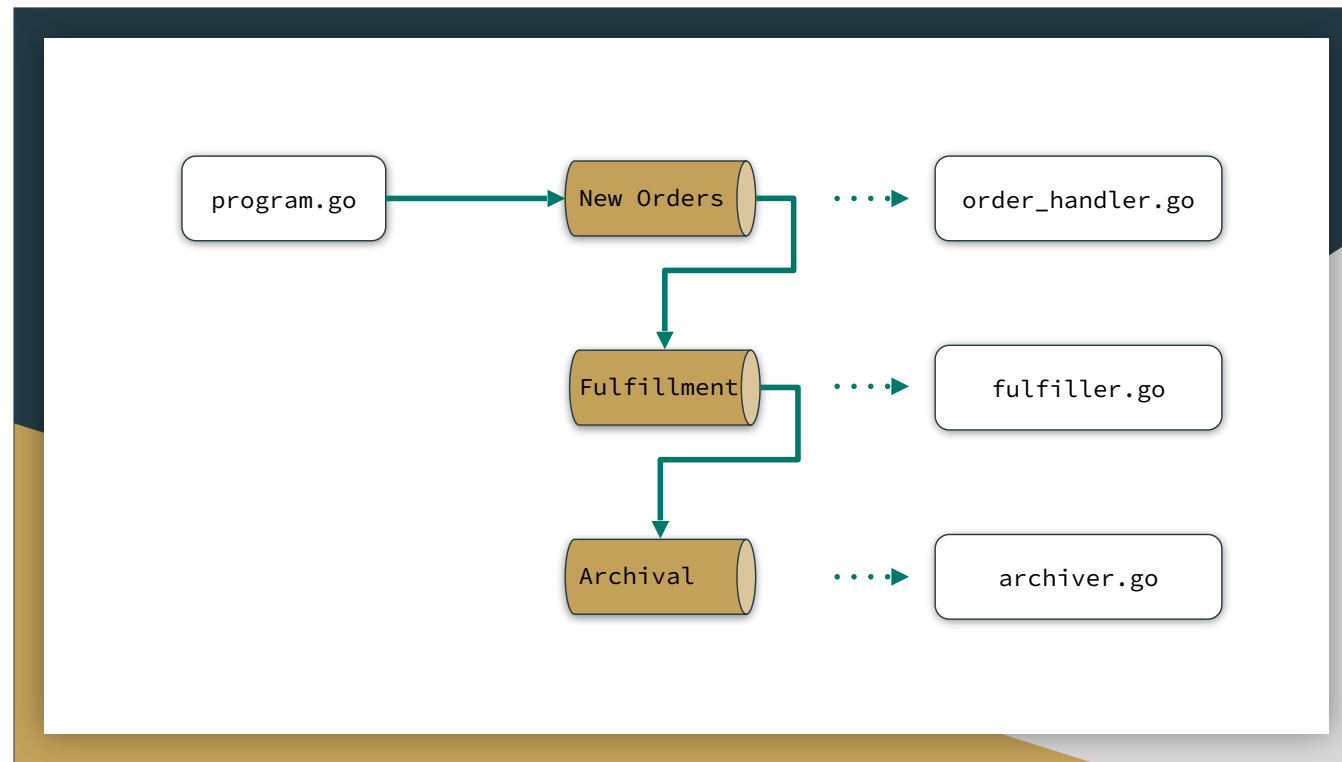
... they're just kinda buried.



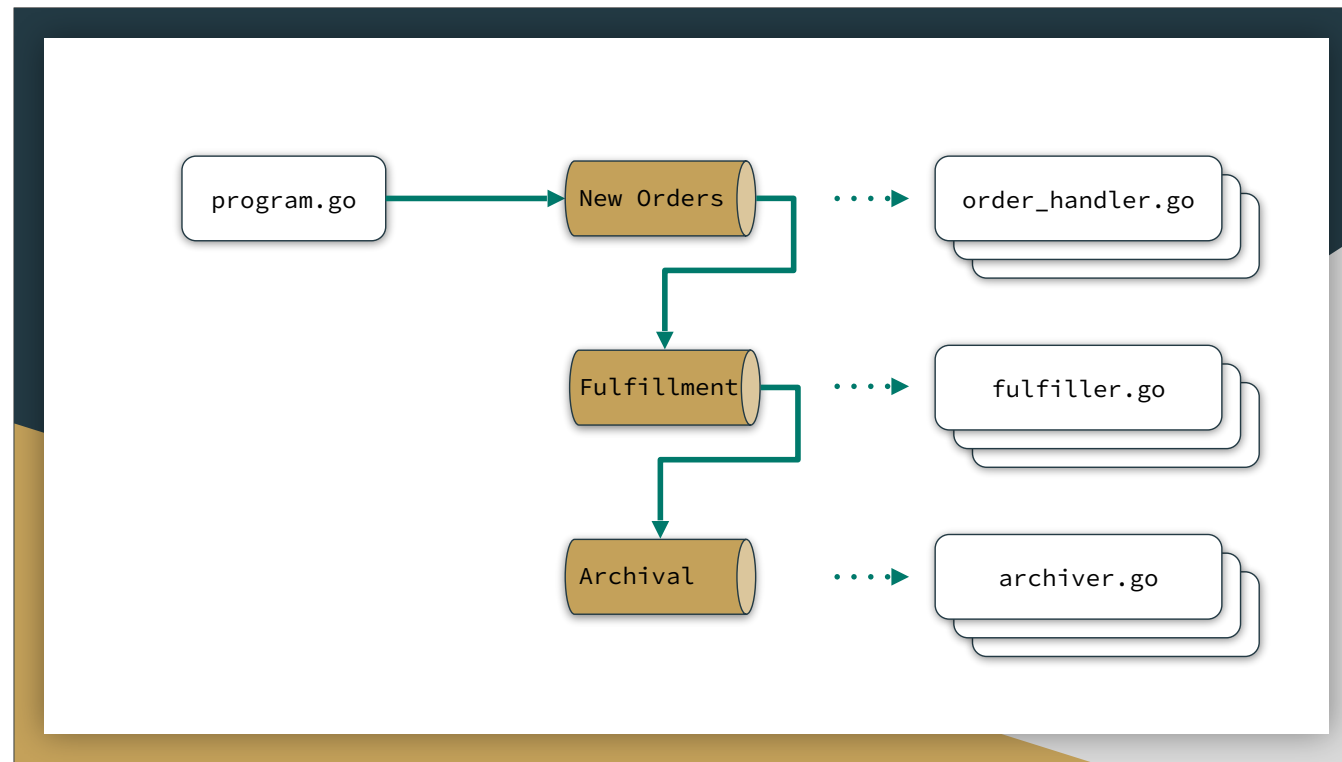
And well, if we're going to modify the program anyways, let's rearchitect the whole thing, and move to message queues for each major step.

<click>

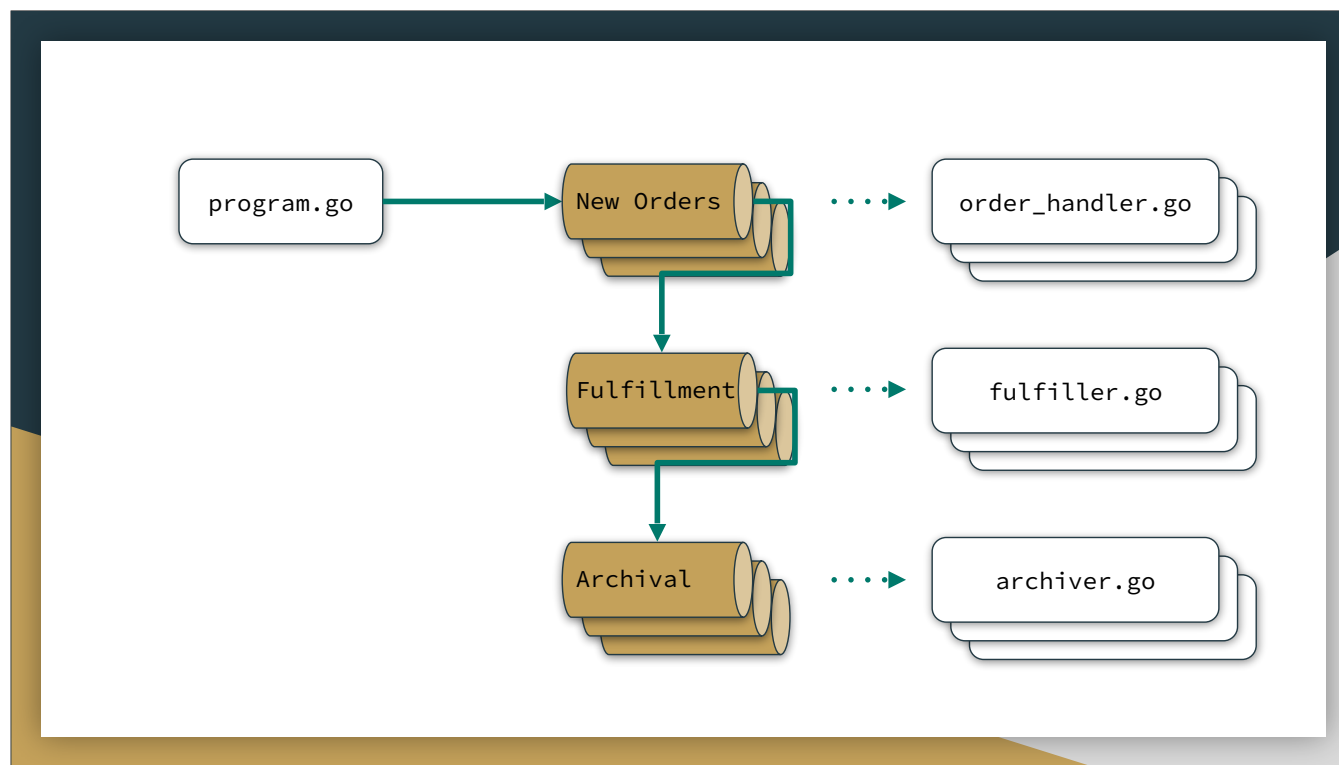
That way, our program simplifies to just kicking off the pipeline.



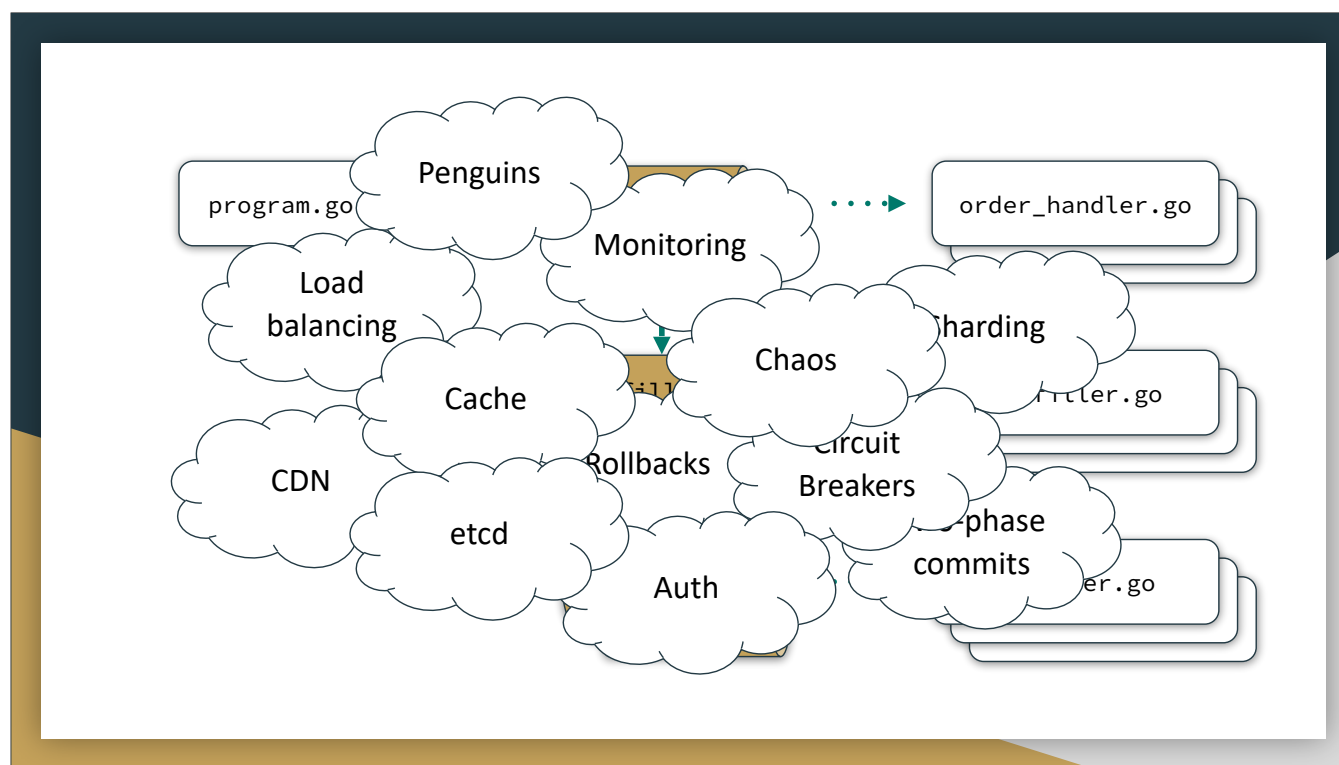
But it's not actually that simple because we need handlers for each of those queues so that we can actually still call the relevant APIs



And then we replicate them to handle load and single-node failures.



And then we replicate the queues themselves, to distribute load.



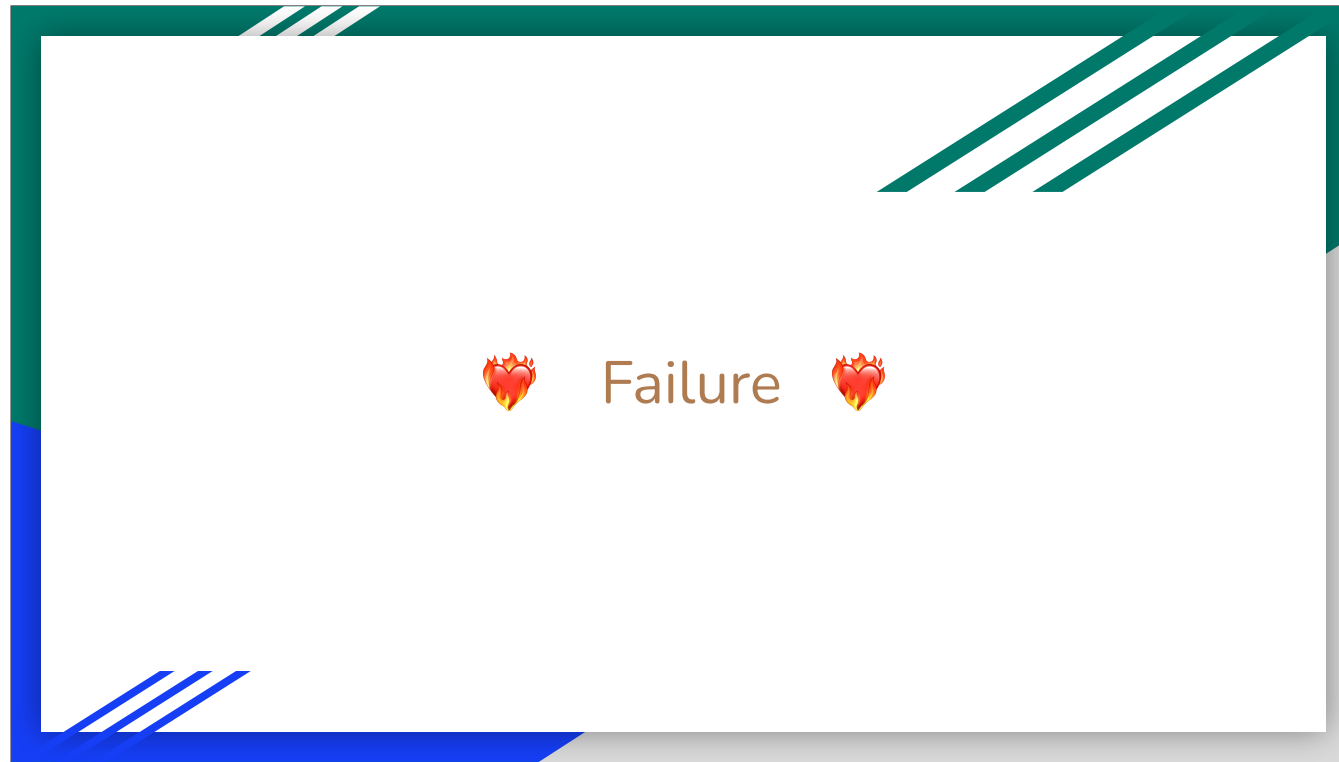
And then we need to add monitoring

<click>

load balancers

<click>

and more and more and more until this simple program isn't so simple anymore.



(Time check: 2m +/- 10s)

Breathe

All of this is put in place to handle failure.

Simple Program (pseudo-go)

```
func OrderItem(o OrderRequest) error {  
    c, err := api_client.New()  
    defer c.Close()  
  
    order, err := c.InitOrder(o)  
  
    status, err := c.FulfillOrder(order)  
  
    db, err := archival.NewClient()  
    defer db.Close()  
  
    err = db.Persist(order, status)  
}
```

Caveats:

- Absolutely NO error handling
- Process or node dies?
 - Restart from beginning
- API call takes too long?
 - Probably also restart


Looking again at the "simple" program from the beginning, the big thing it was missing was any kind of failure handling.

If the process or node running the program dies, we just have to restart from the beginning. Which limits its retrievability.

If one of the API calls takes too long, the likelihood that the process or node dies increases and, well, we probably also have to restart from the beginning.

We end up writing a ton of code and implementing various architectural patterns to mitigate these issues.

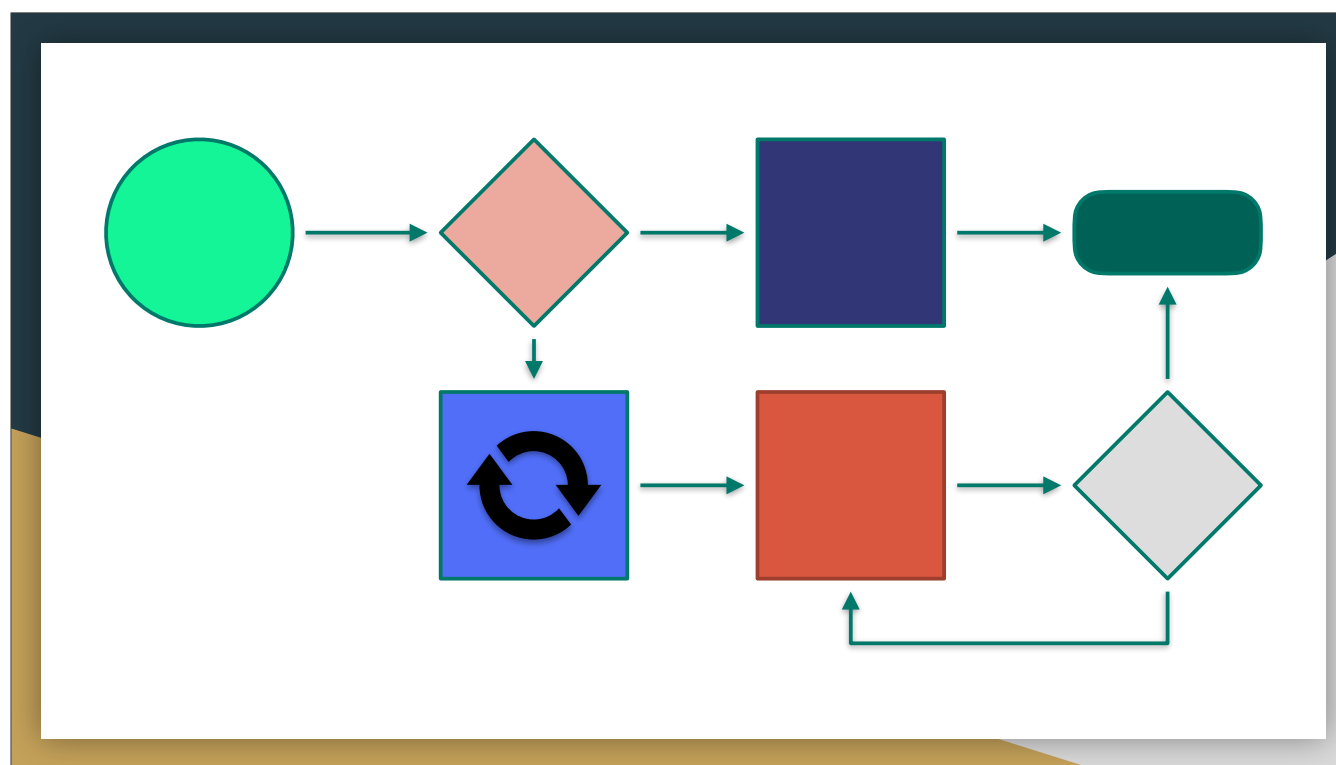
Durable Execution

... with  Temporal

<https://github.com/temporalio/temporal>

Regardless of how many times we have to retry, resume, or wait ... what we really want is for the execution of those steps to reliably happen to completion at least once.

And with the Open Source project Temporal, we call that Durable Execution.



One of the core concepts in Temporal is a Workflow.

I wouldn't be surprised if most if not all of you have either been handed a diagram like this or made one yourself, representing the idealized business or user flow.

<click>

And even if one of those steps takes a really really long time -- long enough that you start to worry if you have enough nines -- you just want it to keep going without having to add a bunch of infra to recover from failure.

<shift-down if spin hasn't finished yet>

Simple Program (Temporal Workflow)

Initiate

Fulfill

Archive

```
func OrderItem(ctx workflow.Context, o Order) error {
    retryPolicy := &temporal.RetryPolicy{
        InitialInterval:    time.Second,
        BackoffCoefficient:  2,
        MaximumInterval:    10 * time.Second,
        MaximumAttempts:    100,
        NonRetryableErrorTypes: []string{"PaymentFailed"},
    }
    ao := workflow.ActivityOptions{
        StartToCloseTimeout: 10 * time.Second,
        RetryPolicy:          retryPolicy,
    }
    ctx = workflow.WithActivityOptions(ctx, ao)
    log := workflow.GetLogger(ctx)

    err := workflow.ExecuteActivity(ctx,
        InitOrder, o).Get(ctx, &o)
    if err != nil {
        log.Error("CreateOrder failed", "Err", err)
        return err
    }

    var status OrderStatus
    err = workflow.ExecuteActivity(ctx,
        FulfillOrder).Get(ctx, &status)
    if err != nil {
        log.Error("FulfillOrder failed", "Err", err)
        return err
    }

    err = workflow.ExecuteActivity(ctx,
        ArchiveOrder).Get(ctx, nil)
    if err != nil {
        log.Error("ArchiveOrder failed", "Err", err)
        return err
    }
}
```

And so here's the same program written as a Temporal workflow, with the same three steps highlighted.

You might note that these look like they're just wrapped in an "execute activity" -- am I hiding something?

Simple Program (Temporal Workflow)

Initiate

Fulfill

Archive

```
func InitOrder(ctx context.Context, o Order) (Order, error) {  
    c, err := api_client.New()  
    if err != nil {  
        log.Error("Could not create api client", err)  
        return err  
    }  
    defer c.Close()  
  
    order, err := c.InitOrder(o)  
    return order, err  
}
```

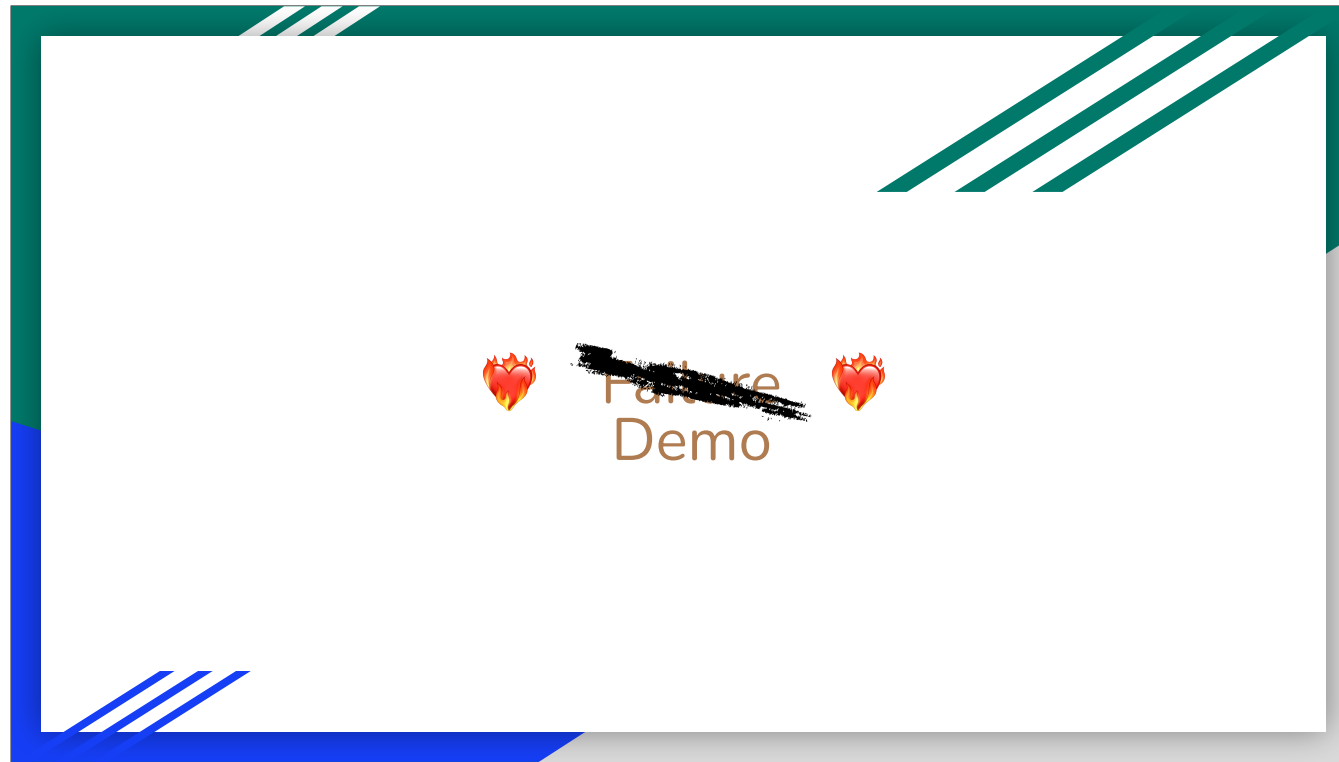
```
func FulfillOrder(ctx context.Context, o Order) (string, error) {  
    c, err := api_client.New()  
    if err != nil {  
        log.Error("Could not create api client", err)  
        return err  
    }  
    defer c.Close()  
  
    status, err := c.FulfillOrder(o)  
    return status, err  
}
```

```
func ArchiveOrder(ctx context.Context, o Order, s string) error {  
    db, err := archiver.NewClient()  
    if err != nil {  
        log.Error("Could not create api client", err)  
        return err  
    }  
    defer db.Close()  
  
    err = db.Persist(o, s)  
    return err  
}
```

Here's what's inside those wrappers... basically just an API client instantiation plus an API call and returning unrecoverable errors, like the customer not having a card on file -- since retrying with the same input won't fix that problem.



As a brief reminder, this was the ton of code that I had written earlier to mitigate various issues by hand.



(Note: Don't click!!)

Let's go see this run real quick. I've already got a Temporal server running, plus the execution process -- aka a worker.

```
`vim workflow.go`
```

Here's the workflow itself. THERE'S NO TIME, so trust me when I say it's the same as what was in the slides.

I also have a script to artificially kick off a workflow -- normally you'd have this tied to something like a customer clicking checkout.

```
`go run starter/main.go`
```

To avoid the internet at a conference, these activities just randomly sleep to simulate processing delays.

<show worker output>

In the Temporal web UI, it shows that the workflow's done, too. Clicking in... there's entries for each one of the activities.

I'm going to simulate failure by killing the worker.

Let's run it again, but kill the worker after the middle step starts.

```
`go run starter/main.go`  
<switch to worker>  
<ctrl-c>
```

Back in the UI, we've got a running workflow here <click on workflow> that's just chilling on the first step.

With the worker stopped, I'm going to go in and make a change to the first activity: let's have it print out something.

And restarting the worker...

```
`go run worker/main.go`
```

... it picks up from the last uncompleted activity, rather than from the beginning, and then ...

<back to UI>
<refresh>

... runs to completion

- 1. Bring up terminals: one's running a development server (open source), one's running a Temporal worker.
- 2. Open workflow code: It's exactly what was on the slide (I just updated the copy-paste moments before getting on stage)
- 3. Show the activities: these are stubbed out for how you might imagine the actual operations working:
 - Initiating an order takes in a unique order identifier plus the order information and then calls the payment processor -- this is the api call that could fail.
 - Fulfilling the order is potentially a very long running operation.



Temporal:

- temporal.io
- github.com/temporalio

This demo (including slide 5's "better?" code):

- github.com/afitz0/gophercon2022-lightning-talk

Andrew Fitz Gibbon (aka "Fitz")
Staff Developer Advocate
@ Temporal Technologies

 fitz@temporal.io
 [@fitzface](https://twitter.com/fitzface)