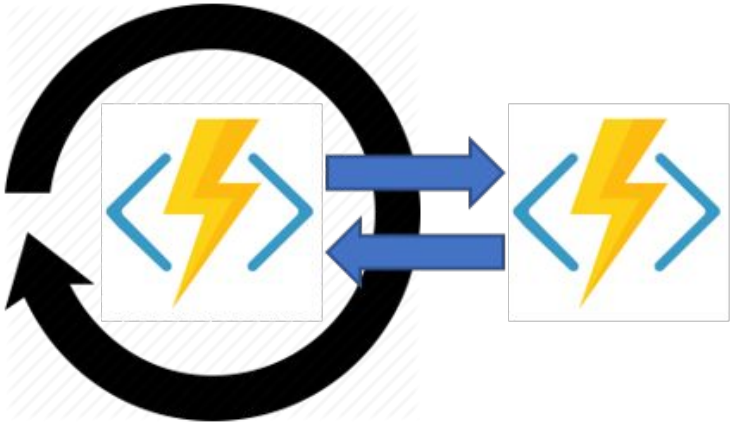


Serverless Orchestration with Azure Durable Functions

Hello!

I am Alec Trievel

You can find me on GitHub:

@atrievel

Agenda

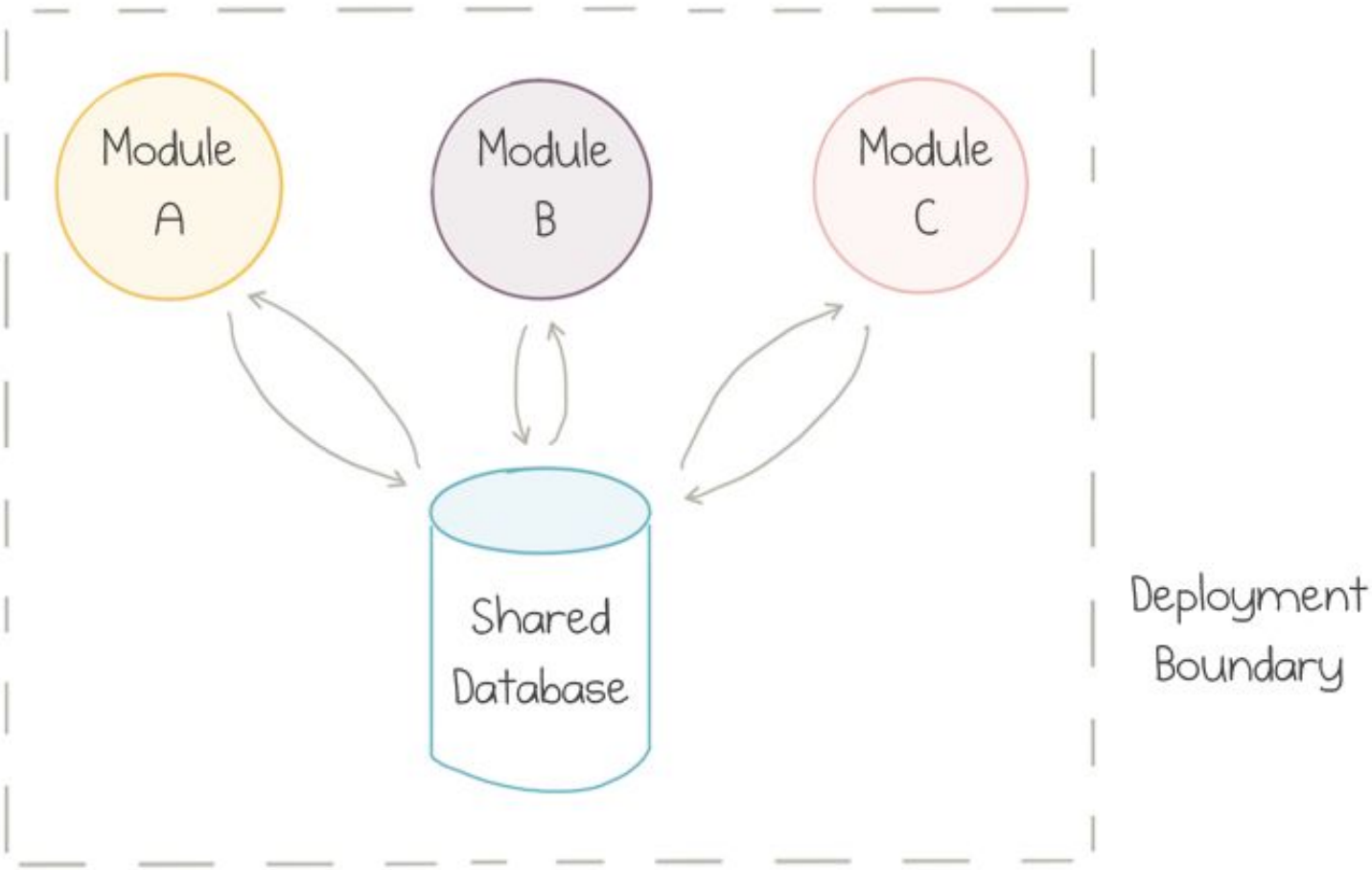
- Intro
- Why serverless?
- Challenges of serverless composition
- Why use Azure Durable Functions?
- Design patterns and concepts
- Pitfalls
- Demos
- Q & A

1.

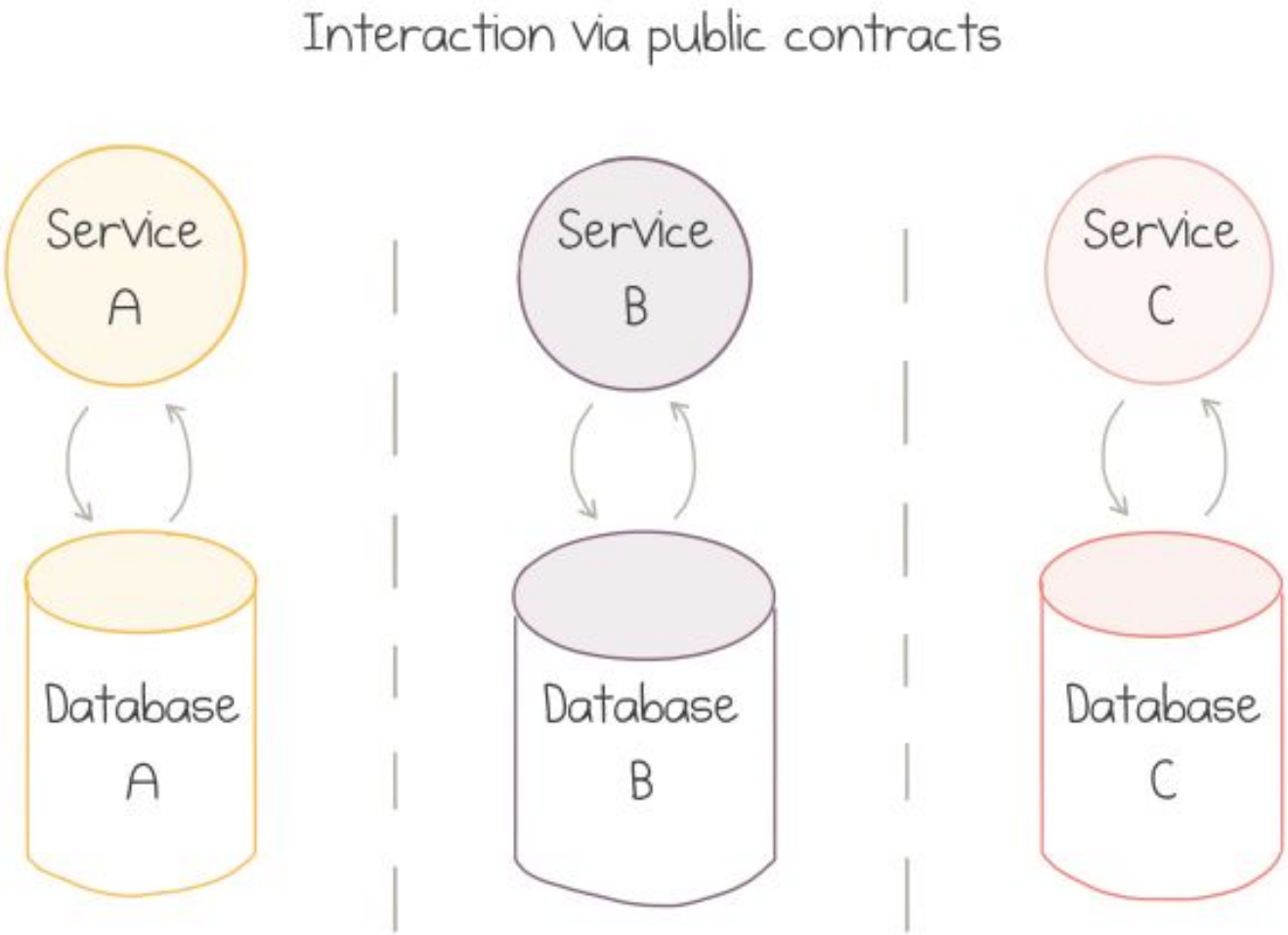
Why Serverless?

Let’s start with a brief history lesson

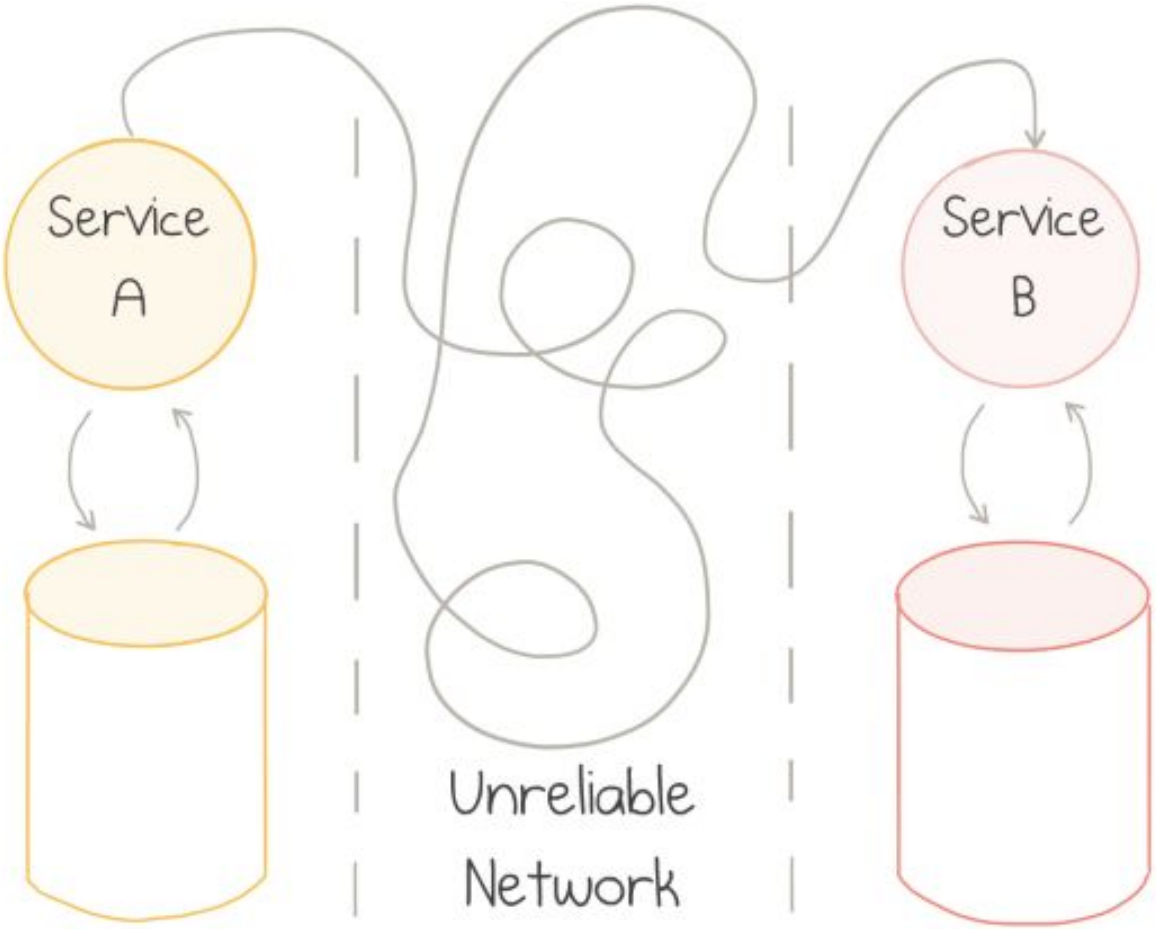
Monolithic


Module A
Module B
Module C
Shared Database
Deployment Boundary

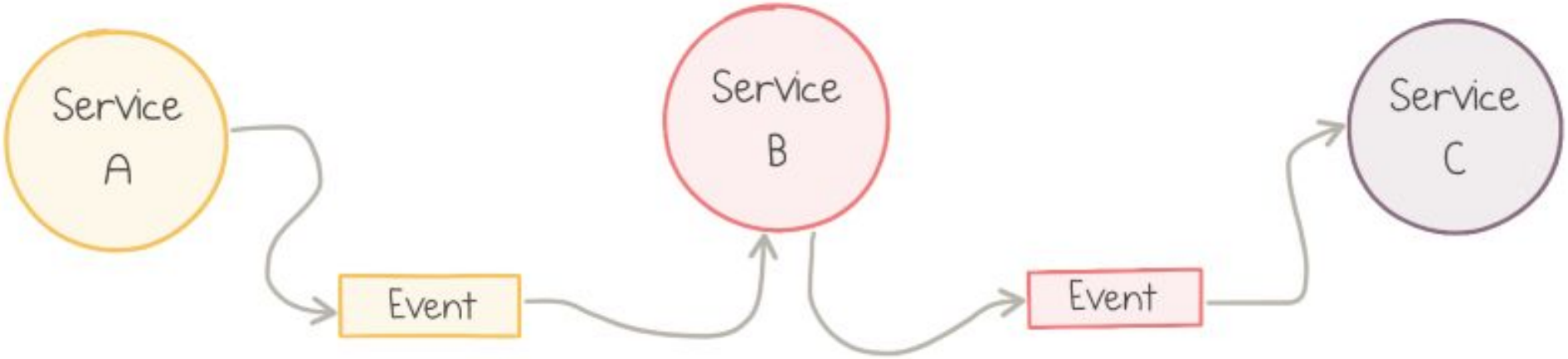
Microservices



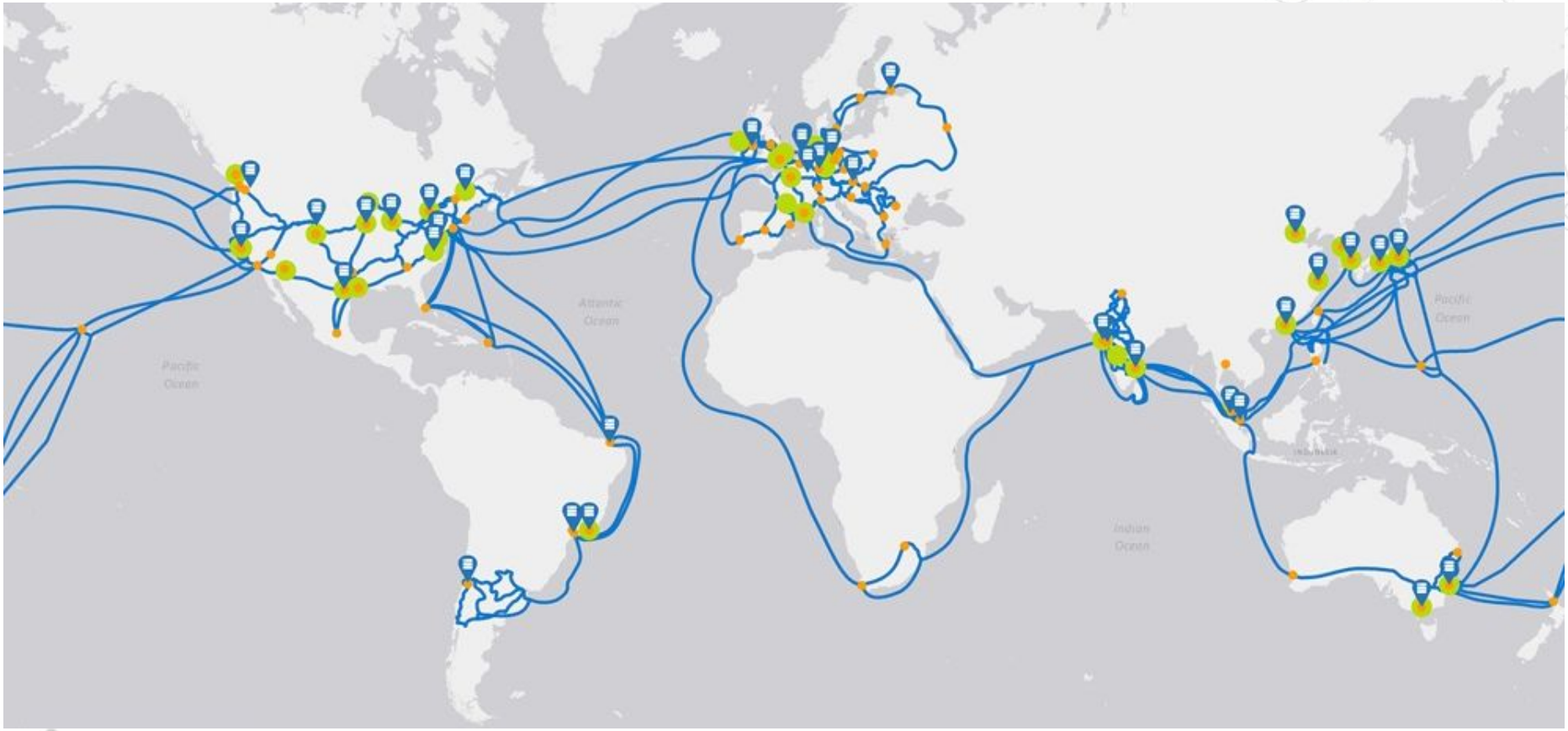
Microservices



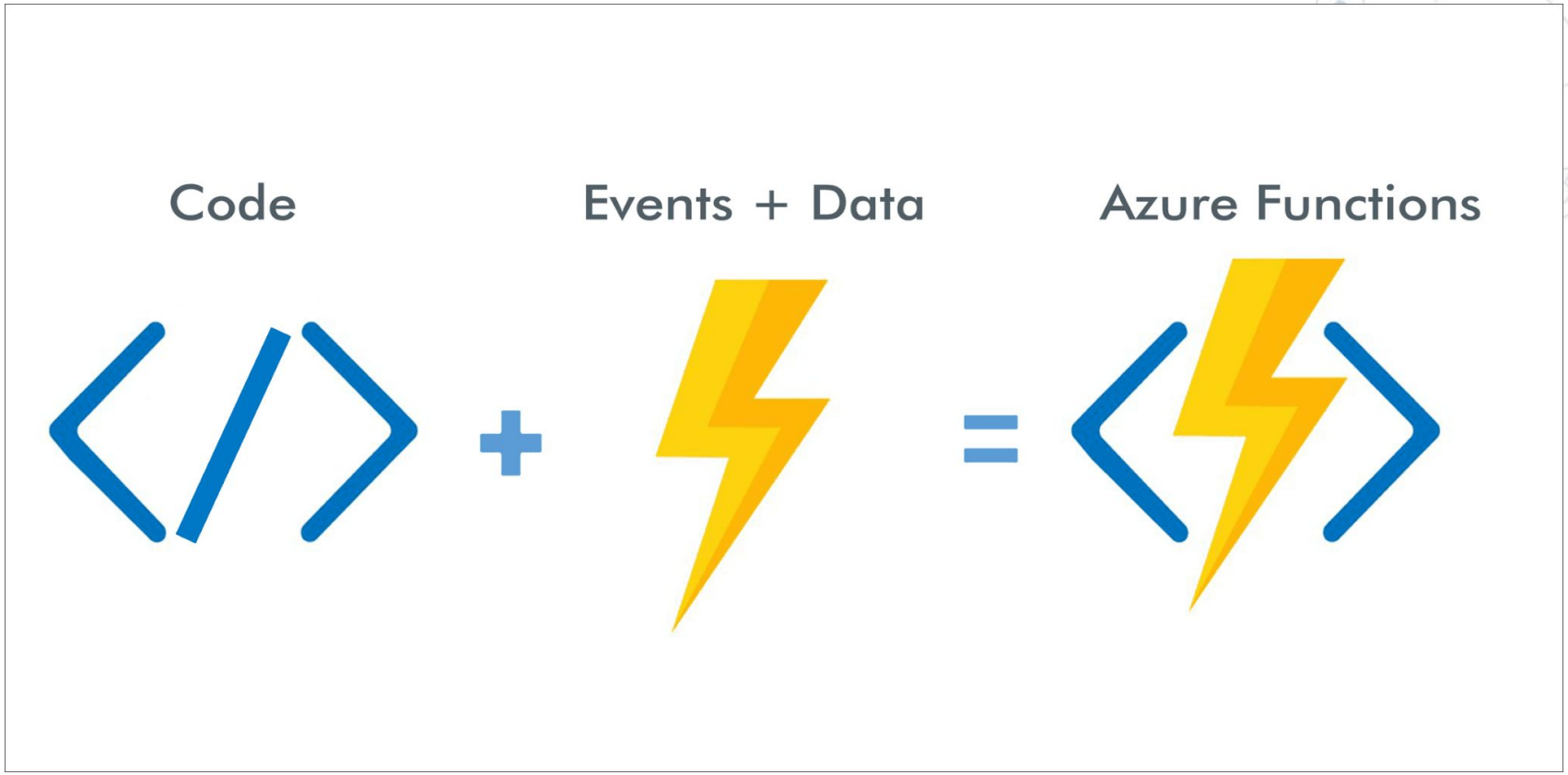
Event Driven



Cloud



Serverless



2.

Challenges with Serverless Design

The stuff that's still difficult

What's Still Difficult?

Challenges:

- Flexible sequencing
- Error handling
- Parallel actions
- Polling / monitoring
- External sources

Stateless needs to be:

- Event driven
- Stateless
- Short lived
- Scalable

We're missing something to manage all these issues...

3.

Why Azure Durable Functions?

Orchestrator to the rescue!

Stateful workflows on top of stateless serverless cloud functions

Durable Functions

- A library that brings workflow orchestration abstractions to Azure Functions
- Records the history of all actions in Azure Storage services
- Stateful workflows authored in code

Client Functions

- Entry point for creating an instance of a Durable Functions orchestration

```
[FunctionName("QueueStart")]
public static Task Run(
    [HttpTrigger("place-order-trigger-func")] string cart,
    [OrchestrationClient] DurableOrchestrationClient starter)
{
    return starter.StartNewAsync("ProcessOrder", cart);
}
```

Orchestrator Functions

- Sole purpose is to manage the flow of execution and data among several activity functions

```
[FunctionName("SequentialWorkflow")]
public static async Task SequentialWorkflow(
[OrchestrationTrigger] DurableOrchestrationContext context,
Cart cart)
{
    var products = await context
        .CallActivityAsync<Product>("GetProducts", cart);
    var order = await context
        .CallActivityAsync<Order>("PlaceOrder", products);
    await context.CallActivityAsync("ShipOrder", order);
}
```

Activity Functions

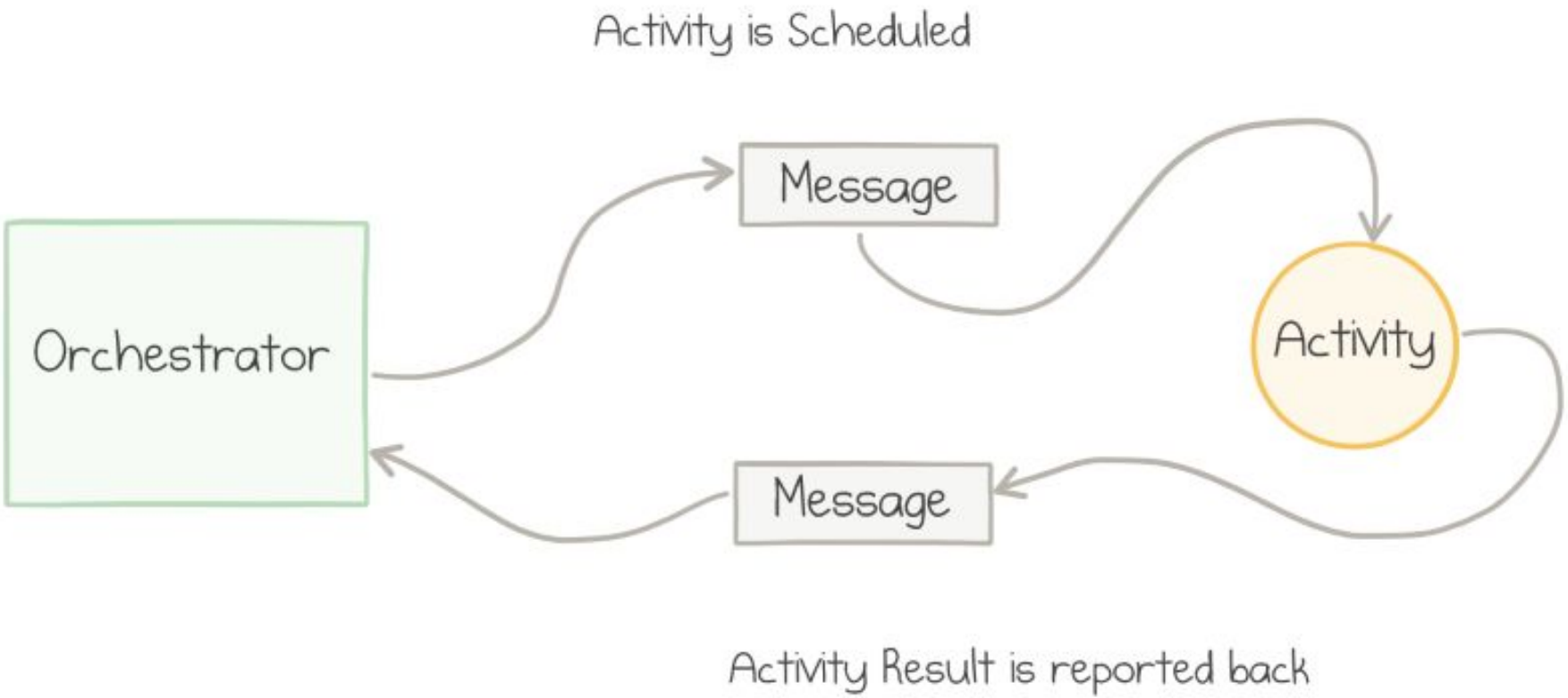
- Stateless single-purpose building blocks
- Basic unit of work
- Activity Functions can do pretty much anything

```
[FunctionName("PlaceOrder")]
public static Order PlaceOrder(
[ActivityTrigger] string productId)
{
    var order = OrderService.Create(productId);
    return new Order { OrderId = Guid.NewGuid()};
}
```

Sub-Orchestrations

```
[FunctionName("CombinedOrchestrator")]
public static async Task CombinedOrchestrator(
[OrchestrationTrigger] DurableOrchestrationContext context)
{
    var orders = await context
        .CallSubOrchestratorAsync("GetOrders", productID);
    await context
        .CallSubOrchestratorWithRetryAsync("ShipOrders", orders);
 }
```

Under the Hood



Error Handling and Retries

- Activity functions “notify” orchestrator on failure
- Developer is free to have backup logic
- Error could be transient and we want to retry

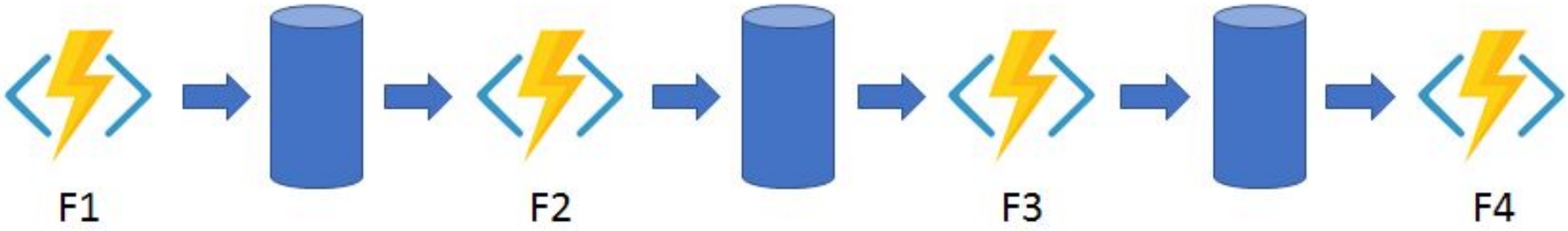
```
var options = new RetryOptions(
    firstRetryInterval: TimeSpan.FromMinutes(1),
    maxNumberOfAttempts: 3);
options.BackoffCoefficient = 2.0;

 await context
 .CallActivityWithRetryAsync("PlaceOrder", options, productId);
```

4.

Design Patterns

Templates for success

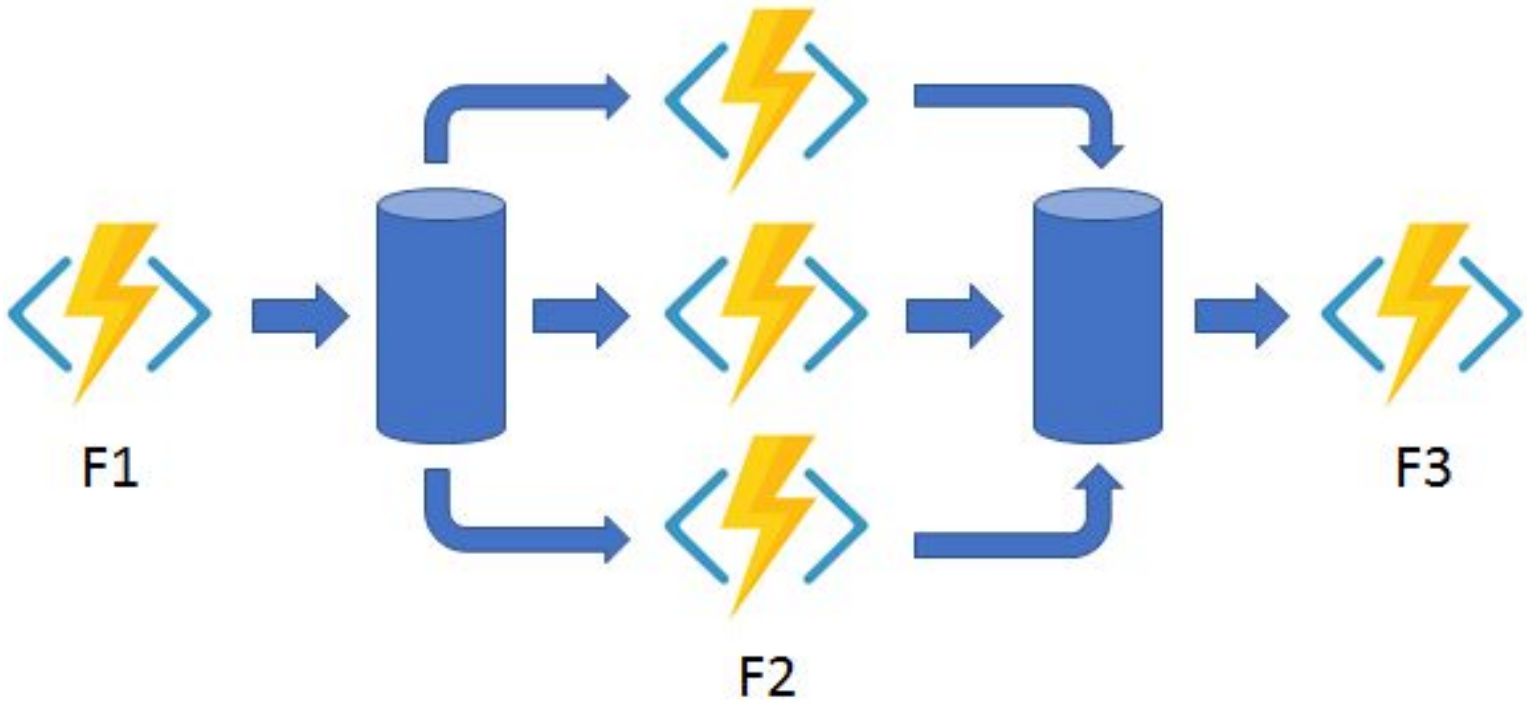
Function Chaining



Function Chaining

```
public static async Task<object> Run(
[OrchestrationTrigger] DurableOrchestrationContext context)
{
    try
    {
        var x = await context.CallActivityAsync<object>("F1");
        var y = await context.CallActivityAsync<object>("F2", x);
        var z = await context.CallActivityAsync<object>("F3", y);
        return  await context.CallActivityAsync<object>("F4", z);
    }
    catch (Exception)
    {
        // Error handling or compensation goes here.
    }
}
```

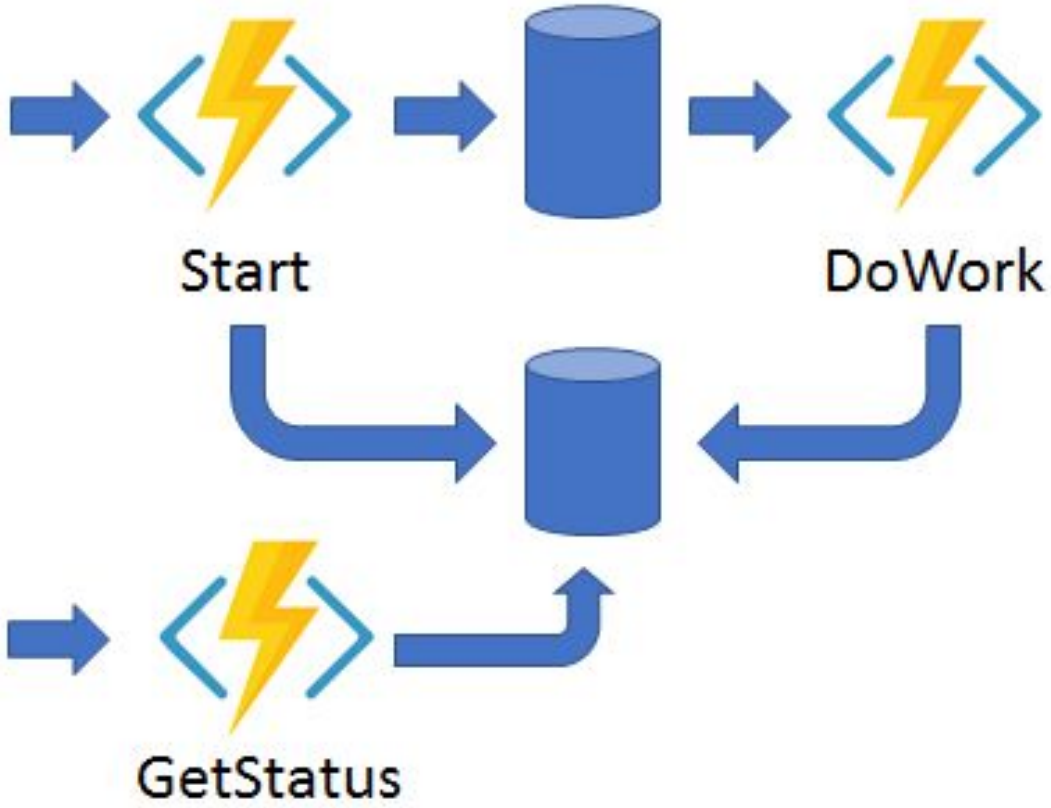
Fan-in / Fan-out



Fan-in / Fan-out

```
public static async Task Run(
[OrchestrationTrigger] DurableOrchestrationContext context)
{
    var shipmentEmails = new List<Task<Order>>();
    List<Order> orders = await context.CallActivityAsync<List<Order>>("GetOrders");

foreach (var order in orders)
    {
        Task<string> task = context
              .CallActivityAsync<int>("SendShipmentEmail", order);
        shipmentEmails.Add(task);
    }

    await Task.WhenAll(shipmentEmails);

    List<string> usaEmails= shipmentEmails.Where(t => t.Country == "USA");
    await context.CallActivityAsync("MarketingData", usaEmails);
 }
```

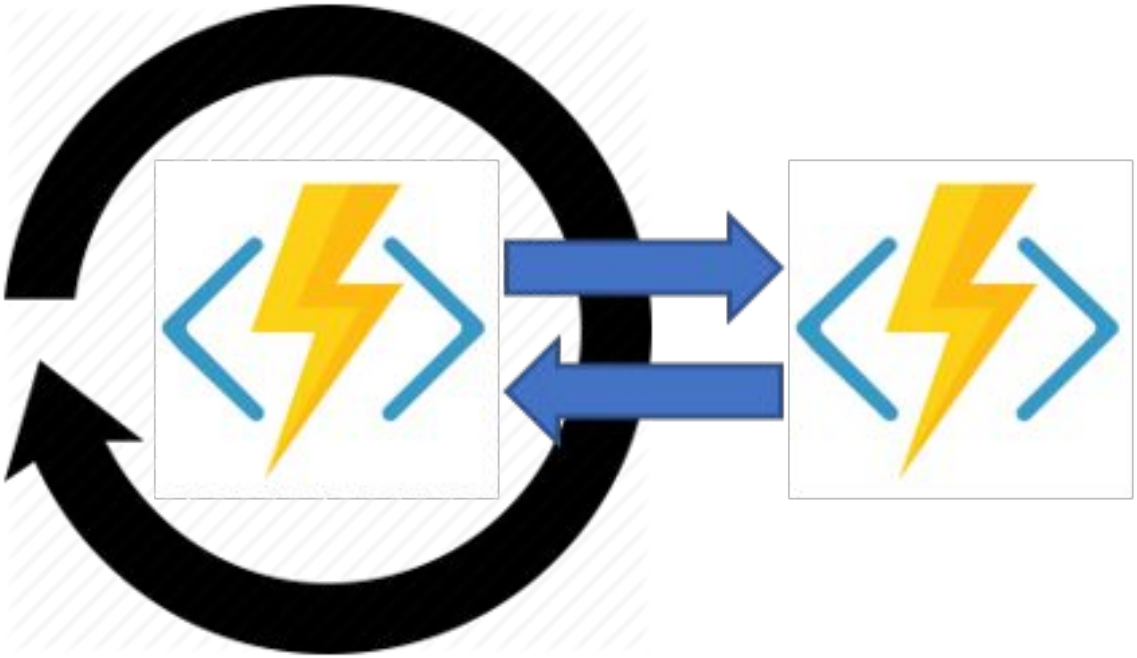
Async HTTP APIs



Async HTTP APIs

```
[FunctionName("StartNewOrchestration")]
public static async Task<HttpResponseMessage> Run(
    [HttpTrigger] HttpRequestMessage req,
    [OrchestrationClient] DurableOrchestrationClient starter,
    string functionName,
    ILogger log)
{
    dynamic data = await req.Content.ReadAsAsync<object>();
    string instanceId = await starter.StartNewAsync(functionName, data);

    log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

    return starter.CreateCheckStatusResponse(req, instanceId);
 }
```

Monitor



Monitor

```
[FunctionName("Orchestrator")]
public static async Task Run(
[OrchestrationTrigger] DurableOrchestrationContext context)
{
    float profit = context.GetInput<float>();
    int pollingInterval = GetPollingInterval();
    DateTime expiryTime = GetExpiryTime();

    while (context.CurrentUtcDateTime < expiryTime)
    {
        await context.CallActivityAsync("SendProfitToAccounting", profit);

        // Orchestration sleeps until this time.
        var nextCheck = context.CurrentUtcDateTime.AddSeconds(pollingInterval);
        await context.CreateTimer(nextCheck, CancellationToken.None);
    }
}
```

Human Interactions

5.
Pitfalls

Stuff to avoid

Pitfalls

- Orchestrator code must be deterministic
- Orchestrator should be non-blocking
- Avoid infinite loops and recursive fan-outs

Billing

- Same as Azure Functions
  - Free up to 1 million executions
  - Then $0.20 per million executions
- Slight overhead with Azure Storage
  - Queues and tables
- Beware of:
  - Eternal looping
  - Recursive fan-outs

Other Advice

- Use AppInsights for monitoring
- Function Apps have API endpoint for management
- Versioning your functions
  - Ostrich algorithm
  - Wait for orchestrator to finish and drain
  - Side-by-side development

6.

Demo

Putting a bow on it

7.

Q & A

Questions and (hopefully) answers