

BIZTALK360

SERVERLESS360

ATOMIC SCOPE

TECHNOLOGY PARTNER



INTEGRATE 2019

Adventures of building a multi-tenant PaaS on Microsoft Azure

Tom Kerkhove

Azure Architect at Cudit, Microsoft Azure MVP, Creator of Promitor



Twitter: @TomKerkhove

GitHub: @TomKerkhove

blog.tomkerkhove.be
cudit.eu

PLATINUM SPONSORS



GOLD SPONSORS



Disclaimer

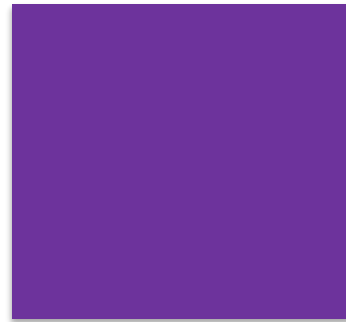
You'll learn about my adventures & findings, not about silver bullets

Scale

Scale up/down

Scale

- | Easiest way of scaling is to get a bigger box



- | The only trade-off is that it means your app will be unavailable for a while
- | At some point you'll run out of "bigger boxes"

Scale out / in

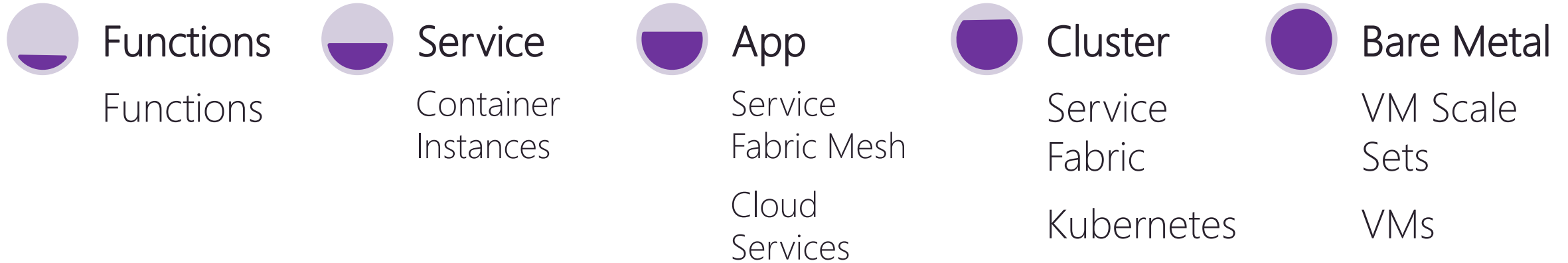
Scale

- | Provide multiple copies of your application based on your workload



- | No impact on your uptime, but more complex
- | My preferred way of scaling, but your application needs to be designed for it

Choose the right compute infrastructure

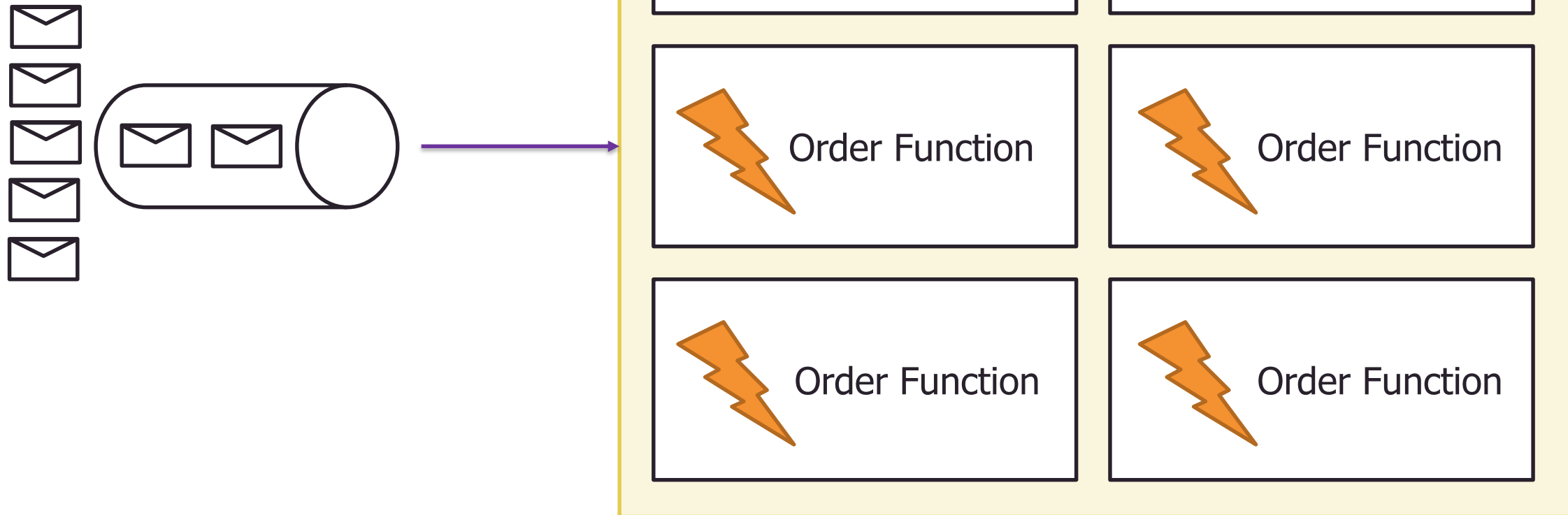


- | As control increases, so does complexity
- | Every service has it's own characteristics
 - | How you run your application
 - | How you package your application
 - | How you scale your application

Designing for scale

Azure Functions

Scale



Designing for scale with serverless

Scale

| The good

- | The service handles scaling for you

| The bad

- | The service handles scaling for you
- | Does not provide a lot of awareness

| The ugly

- | Dangerous to burn a lot of money

Beware “RunOnStartup” in Azure Functions – a serverless horror story

By Tom in Code

6th September 2018 2 Comments



Everyone loves a good technology horror story, right? The gorier the better in my experience. Well, over the last weekend we had an, um, “incident”. To my mind, it’s got all the ingredients of an excellent example of the genre: a small mistake, complicated conditions, and a big financial hit (mercifully averted). Sound like your kind of thing? Read on. I hope I do it justice...

What happened?

I started my Monday morning with the usual routine: having a quick nosey through the analytics and logs. Naturally, I tend to start with production, but (thankfully!) it was when I got to the stats for our development environment that I was filled with dread.

Our functions app was out of control and had scaled to 50x the usual number of servers. Huge amounts of data were flying about and the memory footprint was miles higher than normal.



Egress of our storage account

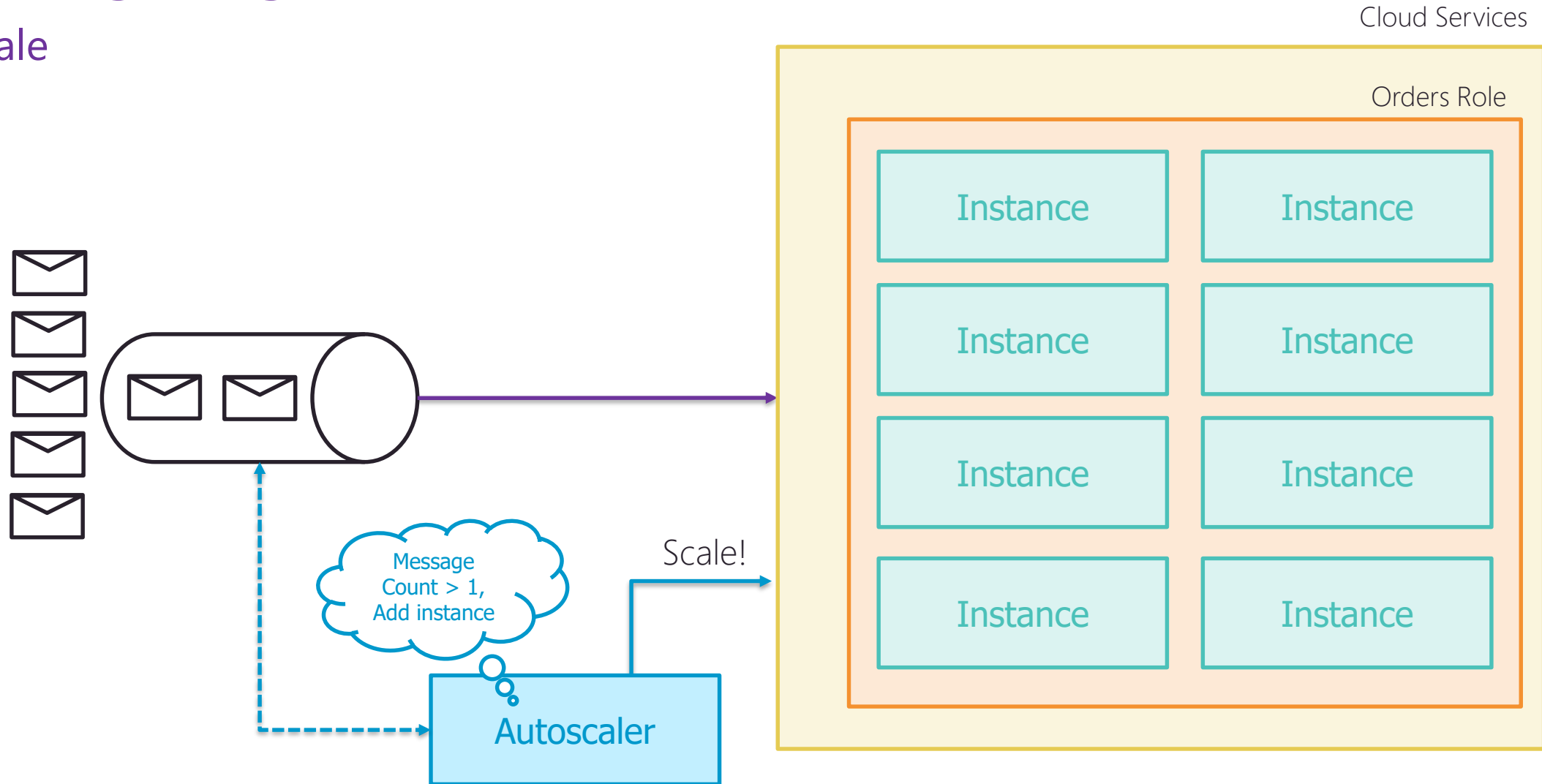


Data in and out of our app service

Source: <http://blog.tdwright.co.uk/2018/09/06/beware-runonstartup-in-azure-functions-a-serverless-horror-story/>

Designing for scale with PaaS

Scale



Designing for scale with PaaS

Scale

| The good

- | You need to define how it scales
- | Provides you with scaling awareness

| The bad

- | You need to define how it scales
- | Hard to determine the perfect scaling rules

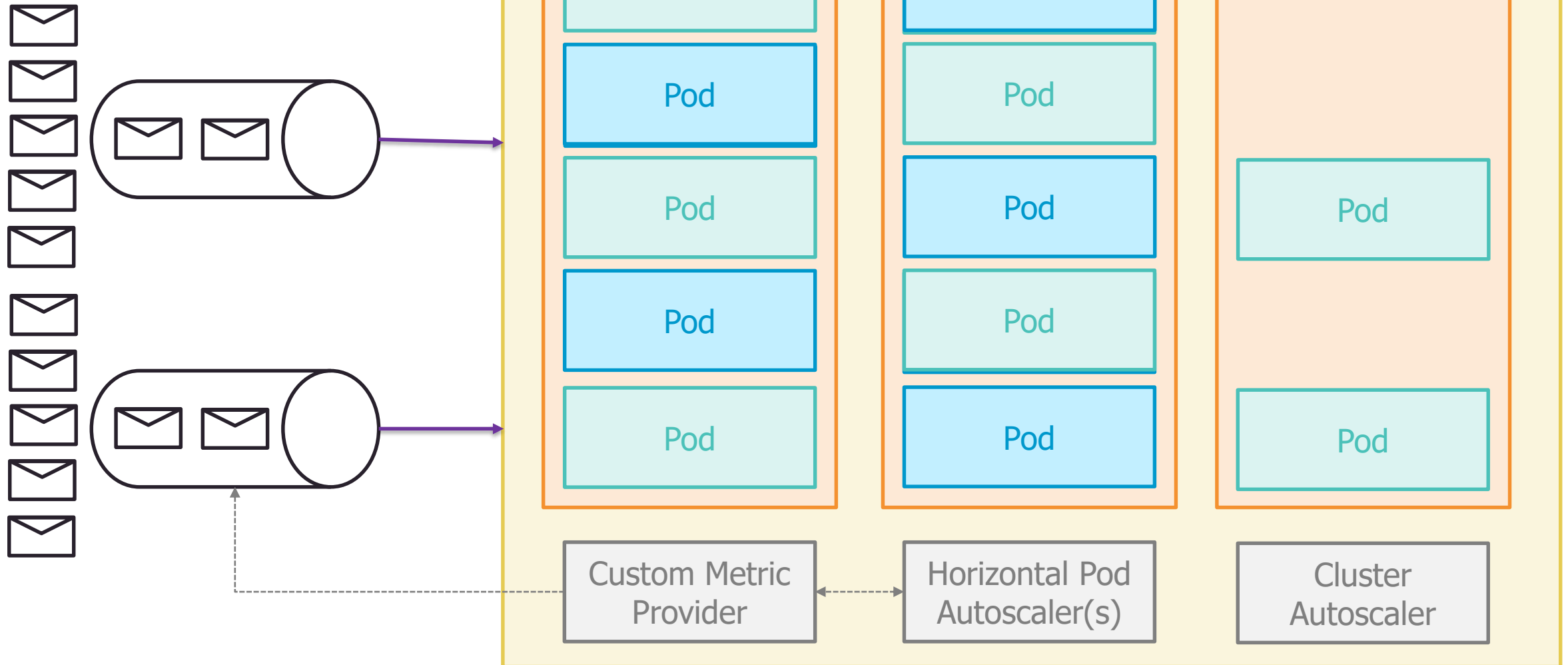
| The ugly

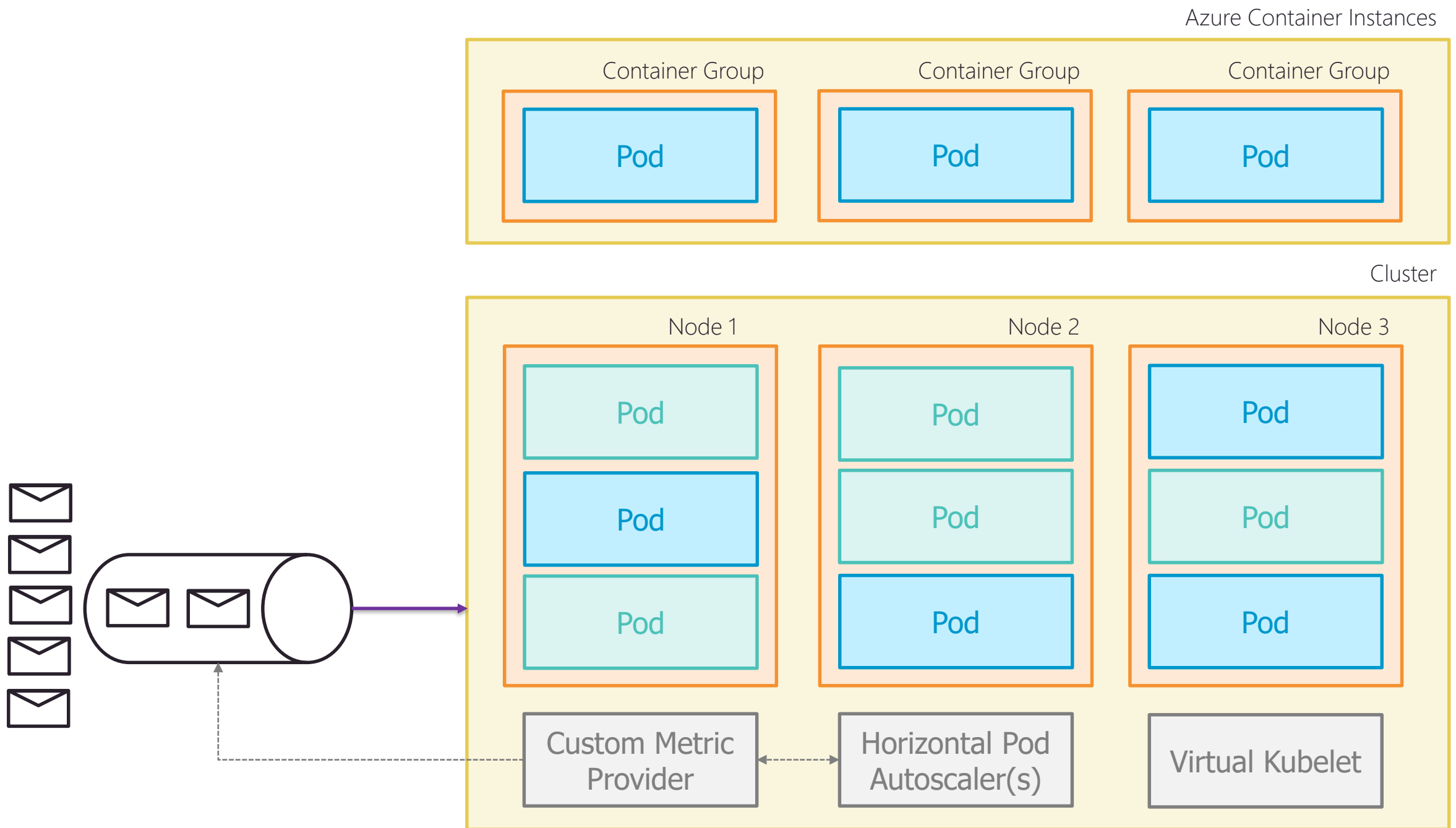
- | Be aware of “flapping” (<http://bit.ly/monitor-autoscale-best-practices>)
- | Be aware of infinite scaling loops



Designing for scale with CPaaS

Scale





Designing for scale with CPaaS

Scale

| The good

- | Share resources across different teams
- | Serverless scaling capabilities are available with Virtual Kubelet & Virtual Nodes

| The bad

- | You are in charge of providing enough resources
- | With great power, comes great responsibilities
 - | No autoscaling out-of-the-box
 - | Scaling on different levels
- | Scaling can become complex(er)

| The ugly

- | Takes a lot of effort to ramp up on how to scale
- | There's a lot to manage

Use the tool that fits your needs

Don't use a service because you know it, evaluate your options

Every technology has its trade-offs, learn them

Don't overengineer, "because we'll need it later"

You don't need hyper scale from day 1

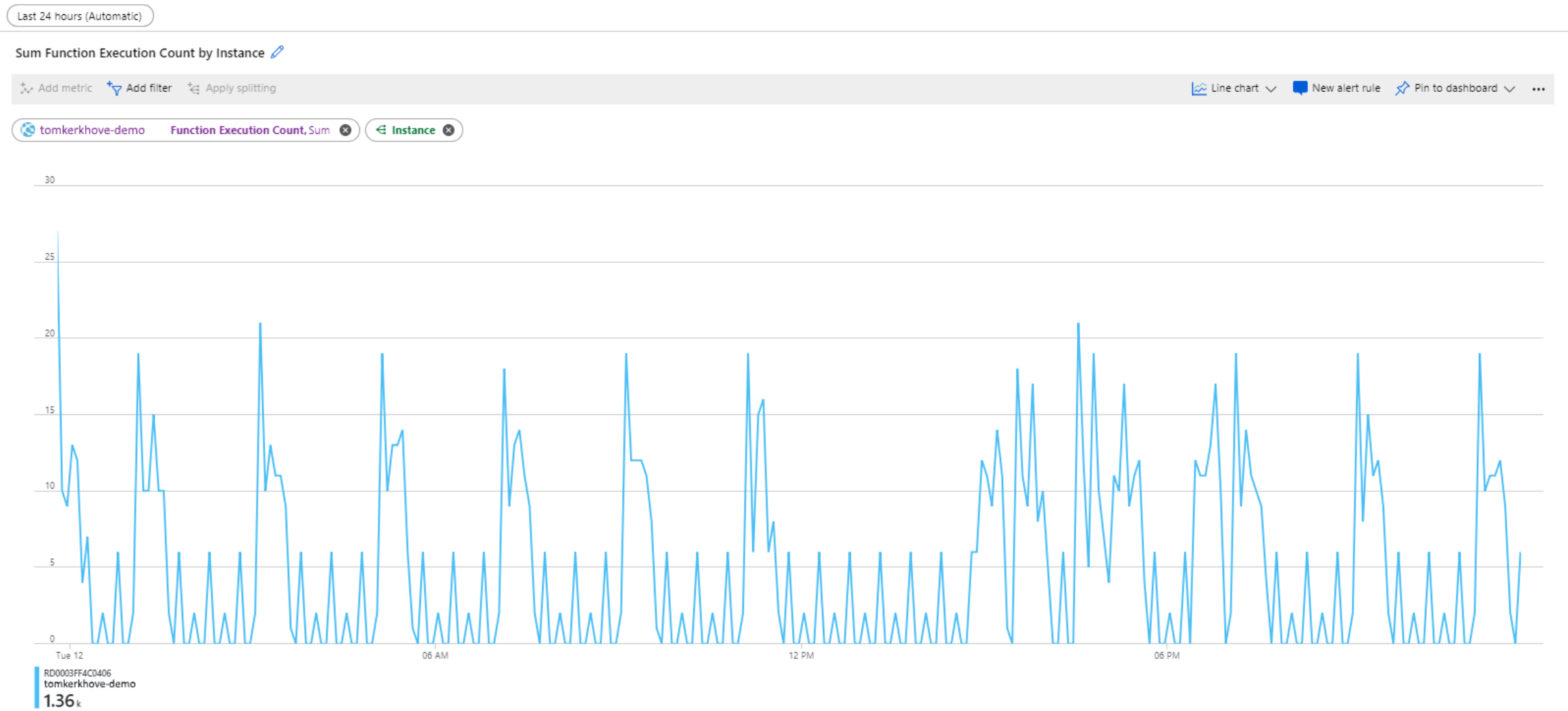
Create awareness around your autoscaling

Scale

- | Avoid burning money, get notified before it's too late!
- | Gain insights in your autoscaling rules
 - | Either configured by you or managed by Azure (*ie. Azure Functions*)
 - | Learn from them and tweak them
 - | Detect autoscaling loops in TEST instead of during live-site issue
- | Choose the approach that fits your needs
 - | Configure Azure Monitor notifications
 - | Use built-in metrics to visualize and alert on
 - | Provide your own tooling around it

Create awareness around your autoscaling

Scale



Tips

Scale

- | Resource consolidation pattern does not play nice with autoscaling
- | Configure maximum instance count for your autoscaling
- | Provide representable metrics of your remaining work
- | Azure Monitor Autoscale is a hidden gem in Azure, use it!
 - | Does all the great things an autoscaler should do
- | Use budget alerts, if feasible

Tenancy

What is our pricing model?
Do we need to reflect this in our tenancy?

How much isolation does it require between tenants?

How much customization will we allow?

Multi-tenancy is a nightmare about data sharding

Do our tenants need access to their data?

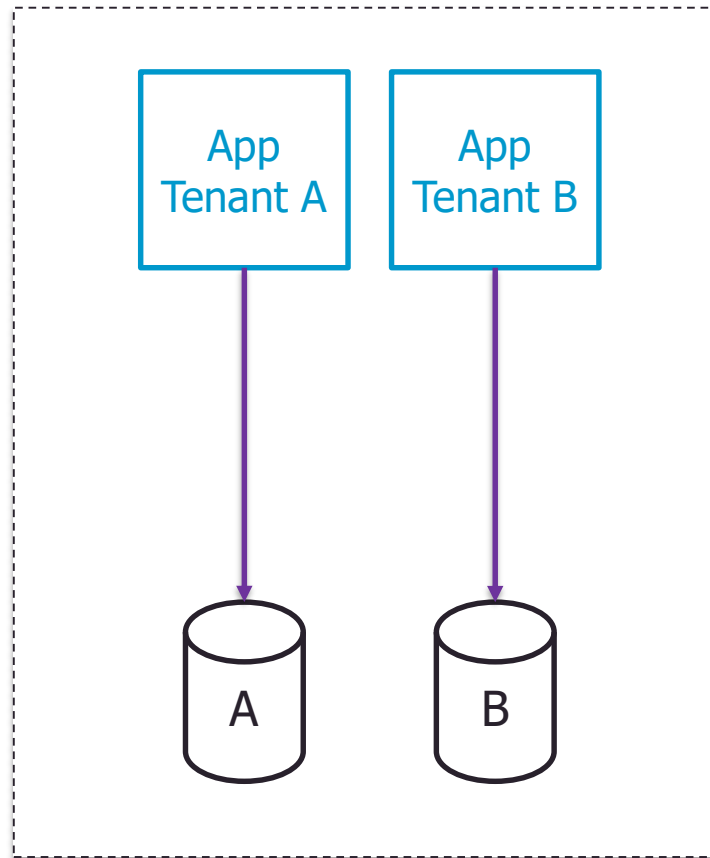
How will you deploy your application?

Will it run in multiple regions?

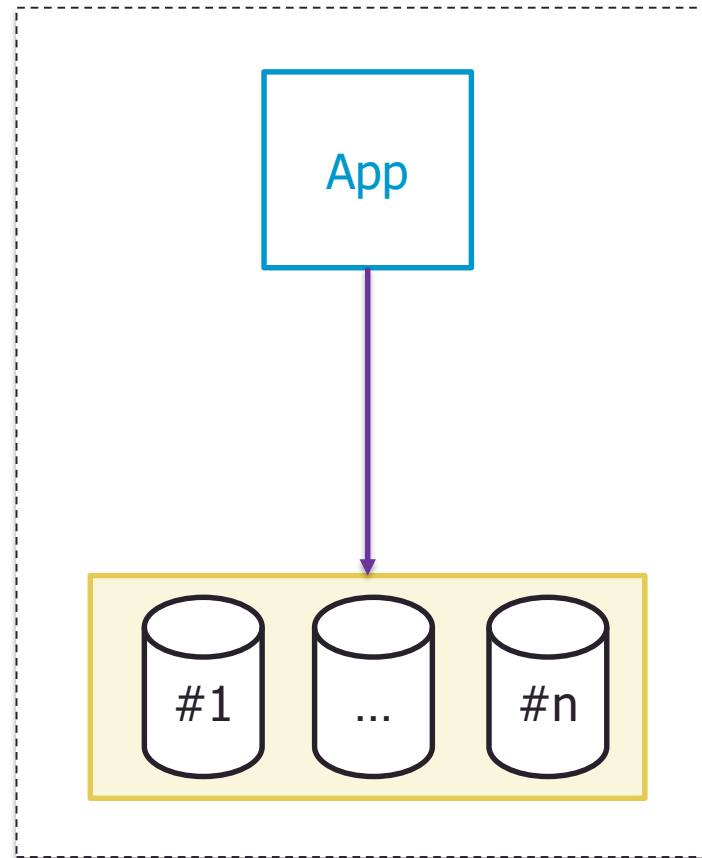
Will one region require multiple deployments?

Choosing a tenancy model

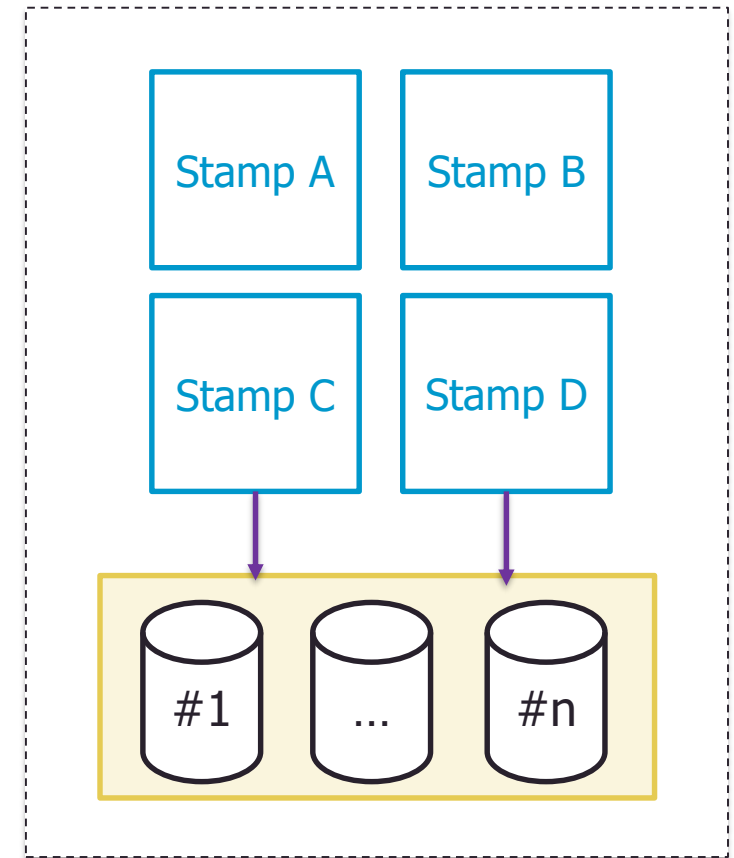
Tenancy



Full isolation between tenants by deploying everything for every tenant



Run a multi-tenant application, but use sharded data layer



App deployed in multiple stamps & geographies with sharded data layer

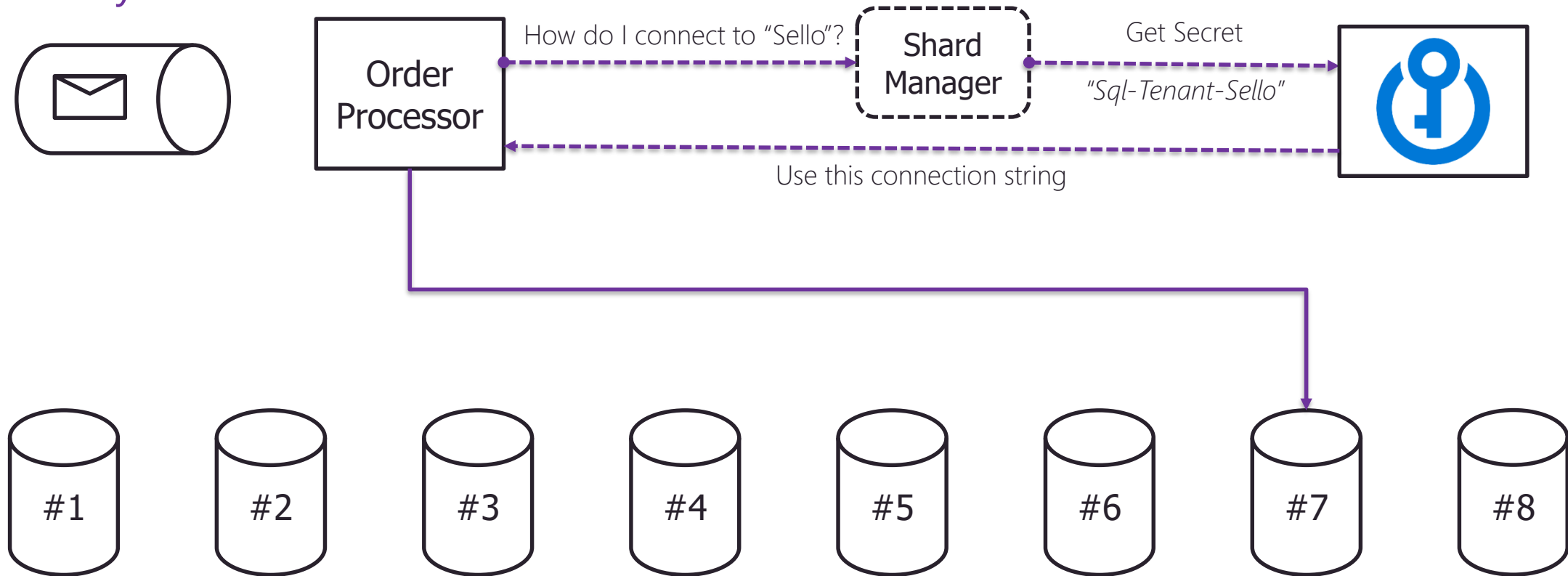
Choosing a sharding strategy

Tenancy

- | Spread all your data across multiple smaller databases instead of one big one
 - | Good example of scaling out to handle load
- | A shard key is used to determine the shard based on the chosen strategy
- | Choose your strategy wisely and think about your query patterns
 - | Does your customer need access to it? Then you should shard per tenant!
 - | You cannot easily change your strategy later on
- | More information: <http://bit.ly/sharding-pattern>

Locating shards

Tenancy



Using shard managers

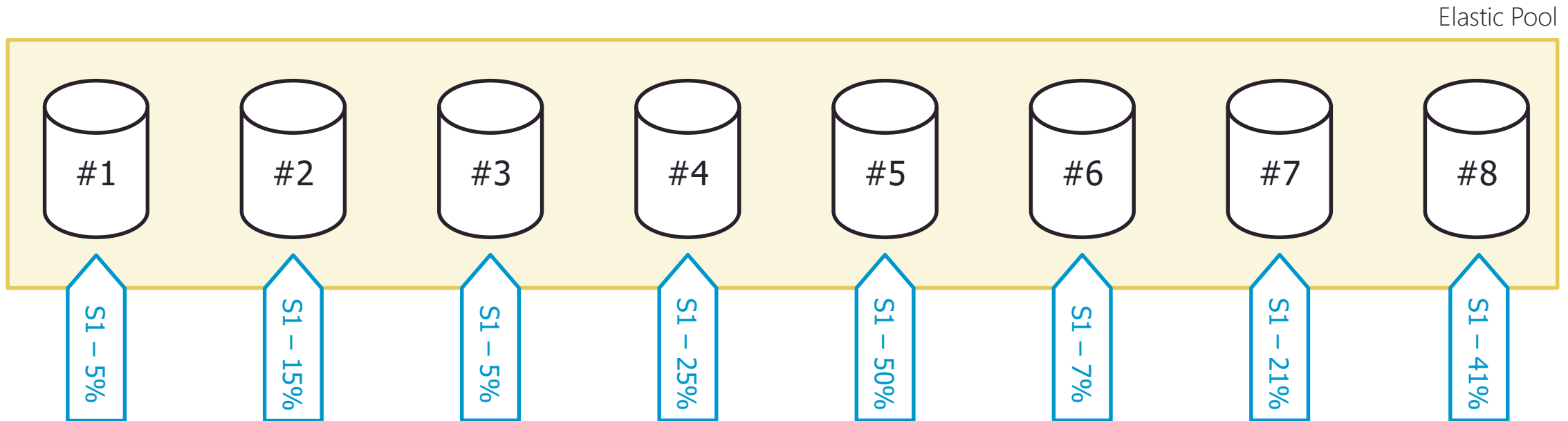
Tenancy

- | Provide catalog of all shard in the platform
- | Determine current shard based on shard key & chosen approach
- | Metadata is stored in a store of choice
 - | Be careful where you store your secrets
- | Choosing a good shard manager
 - | They should handle secrets in a secure manner
 - | Build your own, ie on top of Azure Key Vault
 - | Use existing tool, ie Azure SQL Database Elastic Tools

Cost-efficient sharding

Tenancy

| In a PaaS world you need to pay for every data store instance you have.



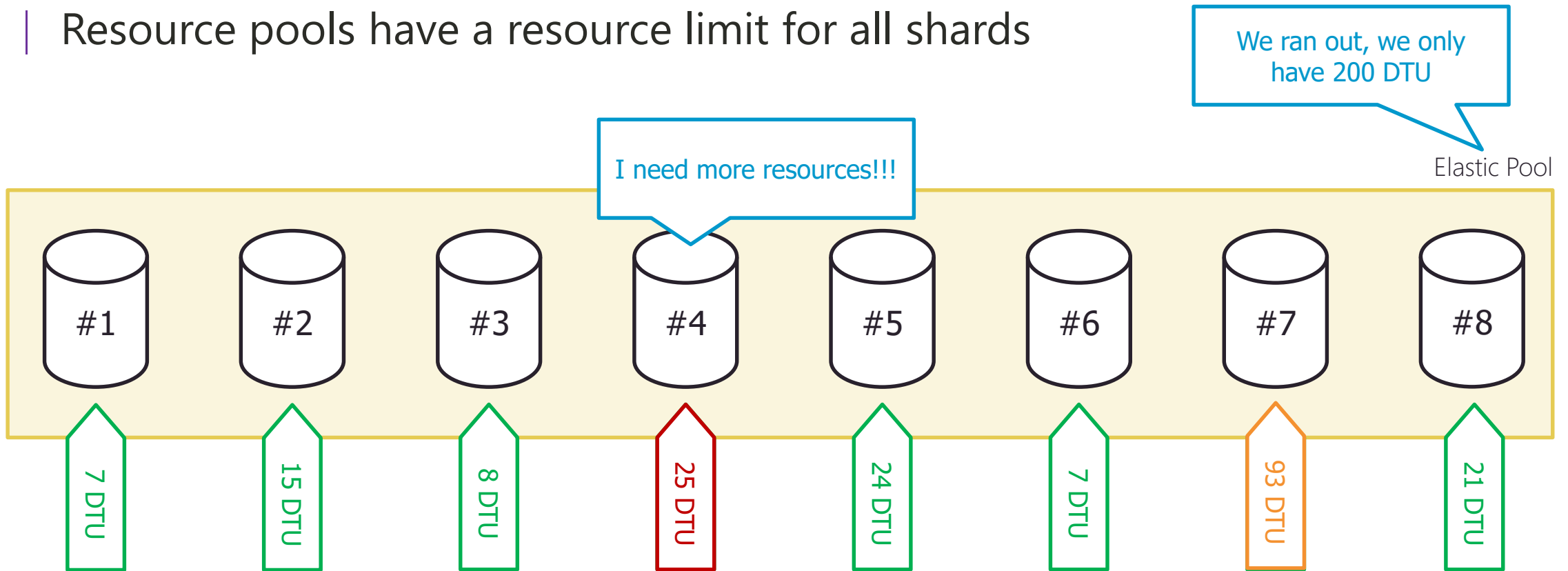
| We pay ~€200 for 160 DTU, but only use ~20 % of it

Use an Azure SQL Elastic Pools

Cost-efficient sharding

Tenancy

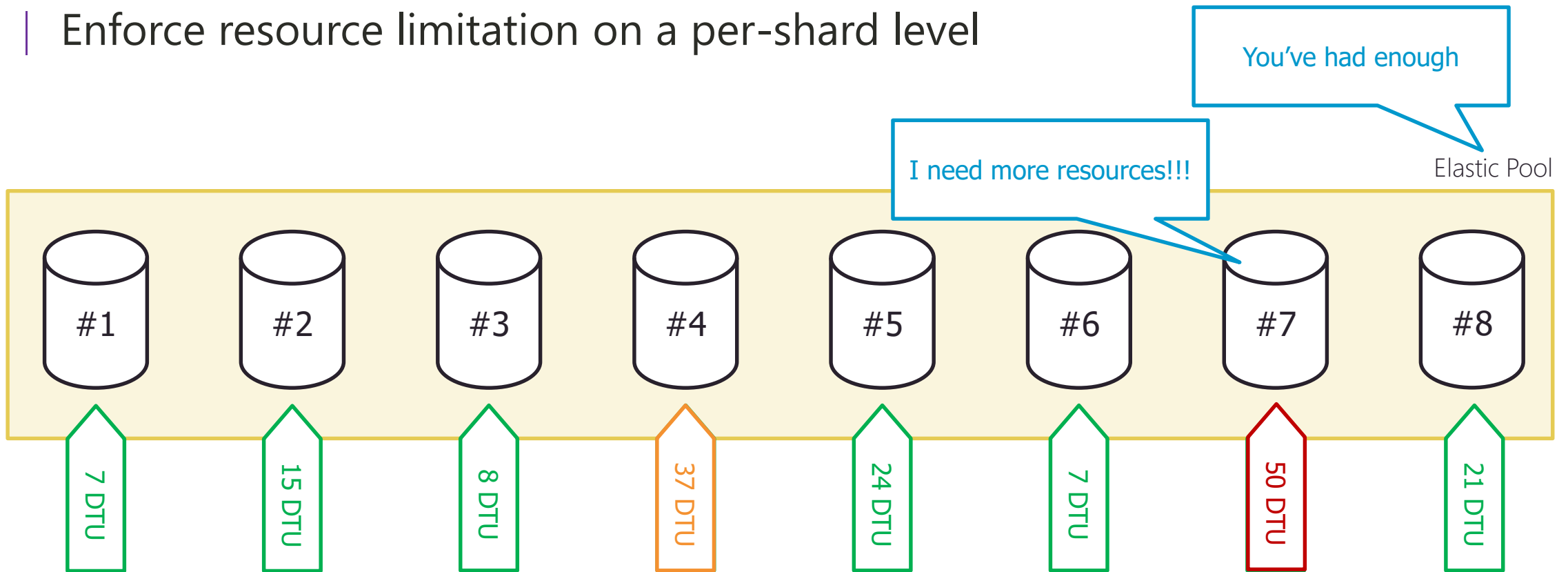
| Resource pools have a resource limit for all shards



Cost-efficient sharding

Tenancy

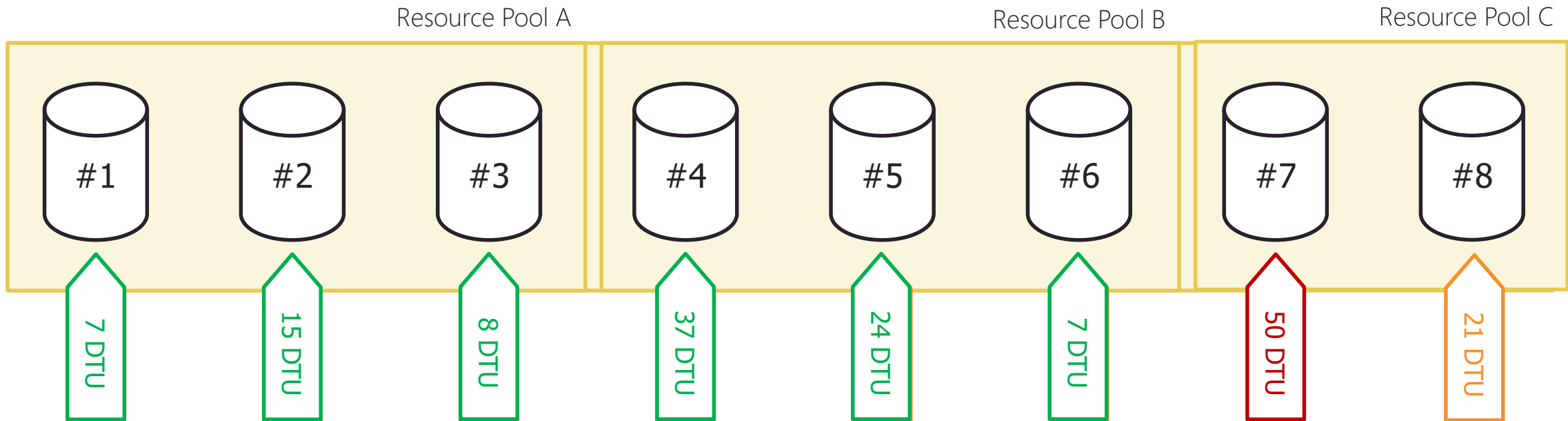
| Enforce resource limitation on a per-shard level



Cost-efficient sharding

Tenancy

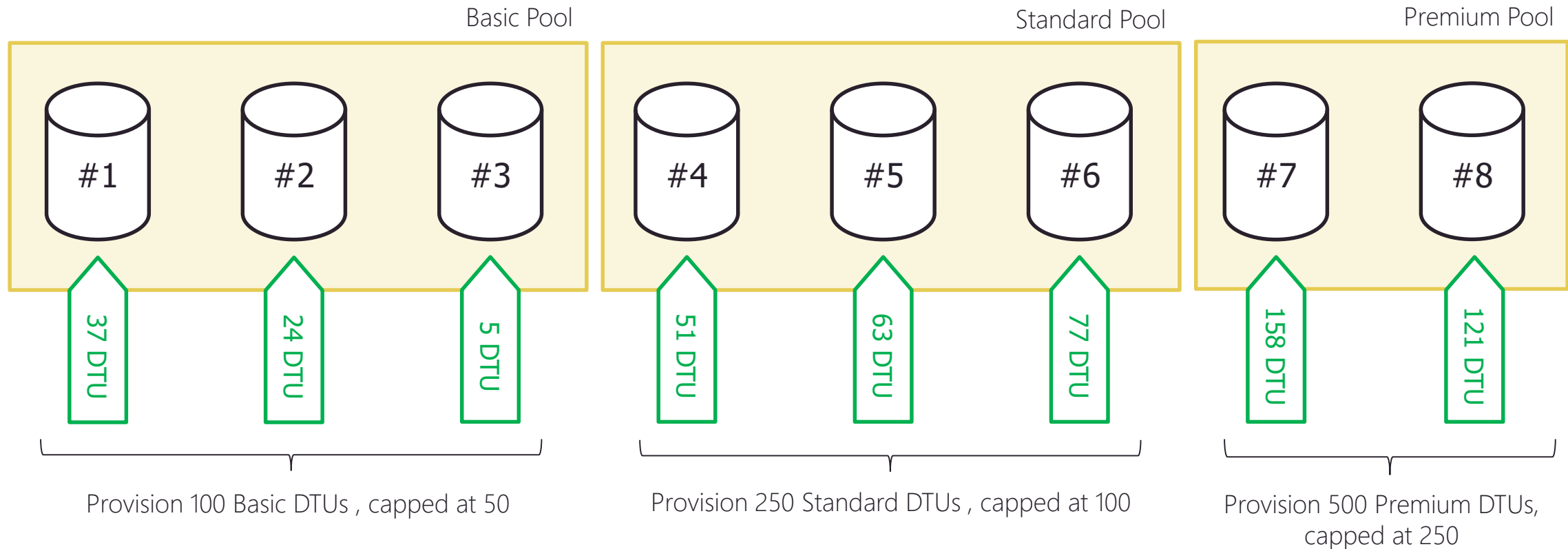
| Provide multiple resource pools to reduce impact of noisy neighbors



Cost-efficient sharding

Tenancy

| Reflect your pricing model in your resource pooling



Cost-efficient sharding

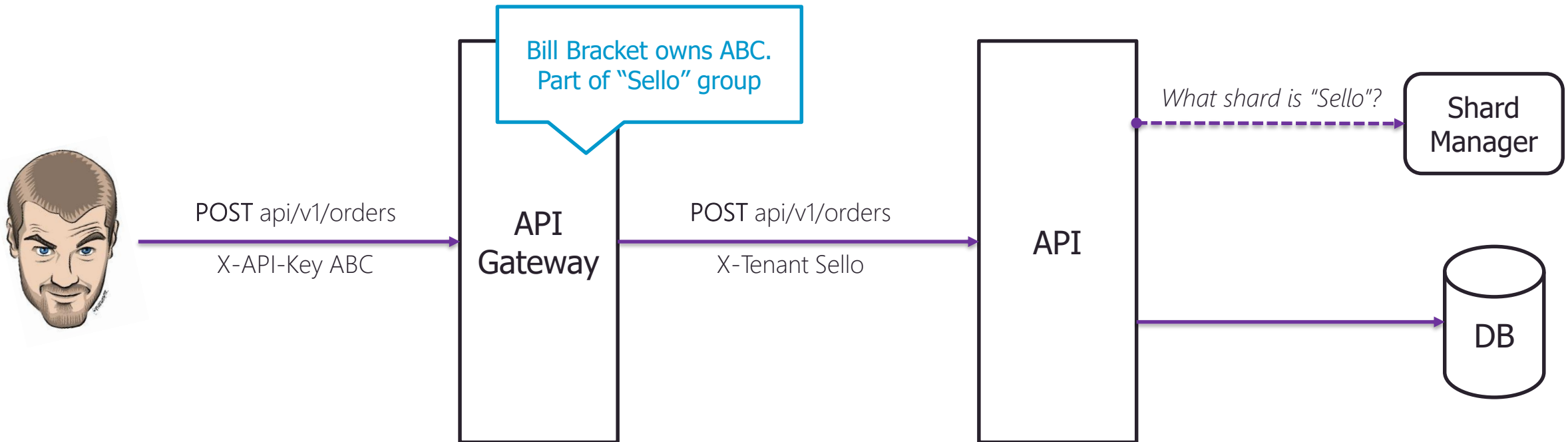
Tenancy

- | Consider moving all shards in a resource pool
- | Configure maximum consumption per database
- | Consider using multiple resource pools to reduce impact of noisy neighbor
- | Resource pools are a great way to reflect your pricing model
- | Monitor your pools as you would do for individual databases

Determining tenants

Tenancy

| Determining the tenant that is consuming your service via your API gateway



| Map the authentication key to the registered application and use its context

Determining tenants

Tenancy

```
<policies>
  <inbound>
    <base/>
    <set-header name="X-Tenant" exists-action="override">
      <value>
        @(string.Join(";", (from item in context.User.Groups where
item.Name.ToLower().Contains("sello -") select item.Name.Replace("Sello - ", String.Empty).Trim()))))
      </value>
    </set-header>
  </inbound>
  <backend>
    <base/>
  </backend>
  <outbound>
    <base/>
  </outbound>
  <on-error>
    <base/>
  </on-error>
</policies>
```


Monitoring





Monitoring is a shared responsibility

You only value good monitoring, if you've been on the other side of the fence

Train your developers to use their own toolchain,
use automated tests on live infrastructure

Enrich your telemetry

Correlated all your telemetry to provide a logical flow, not just traces

Provide app-specific contextual information to all telemetry

Always return your correlation ids to your consumers

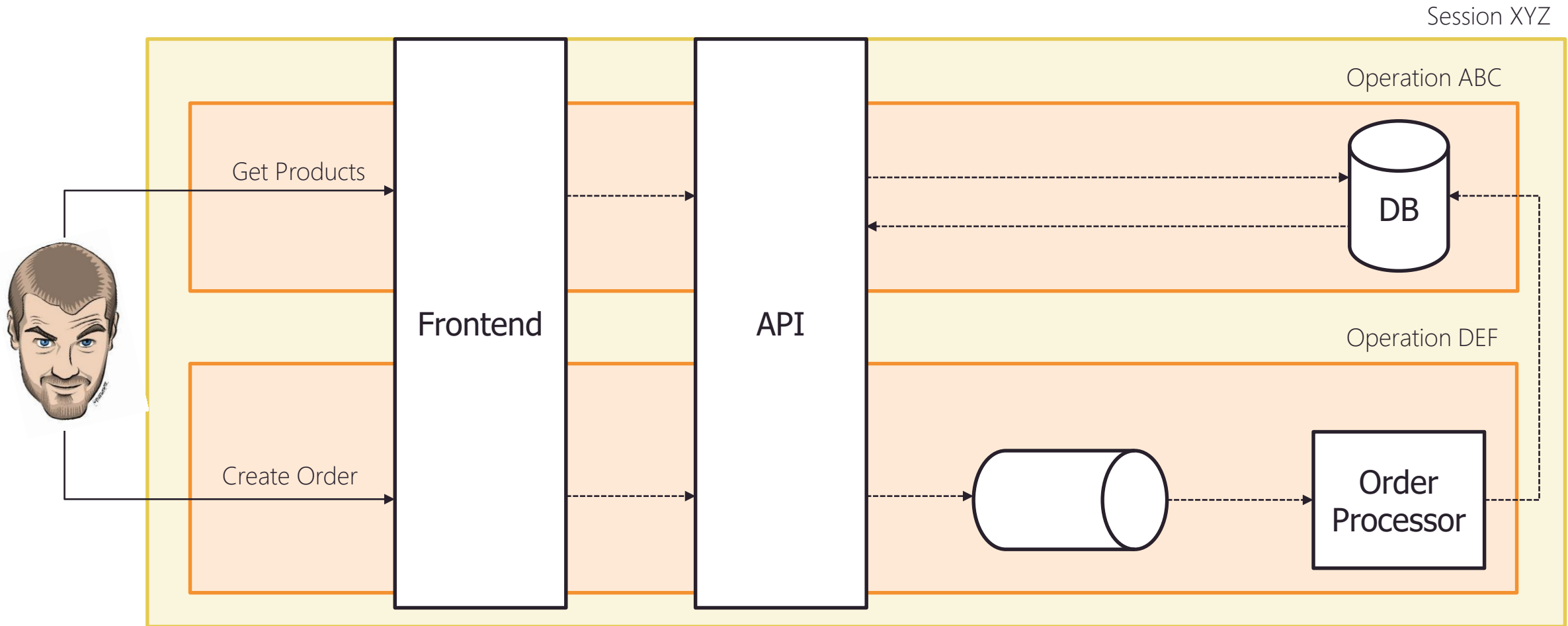
Never track personal identifiable information

Use different layers of correlation ids

Use consistent terminology

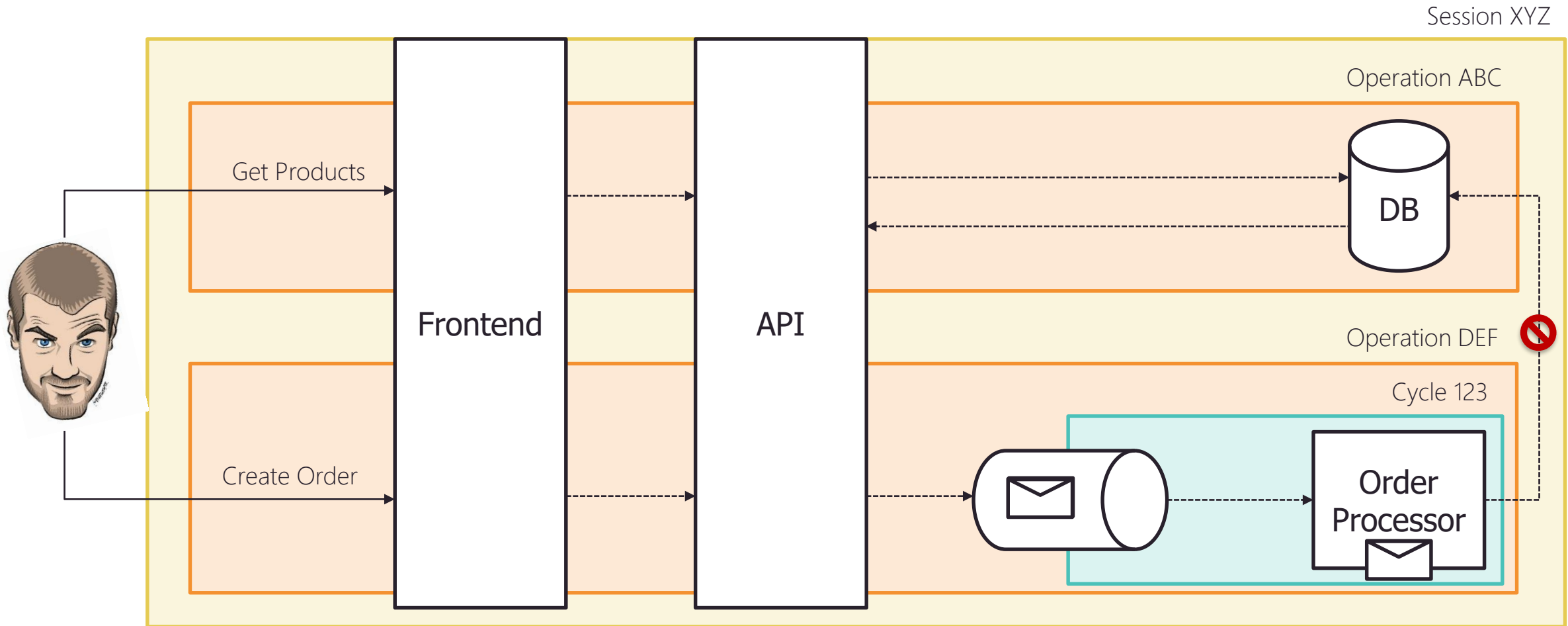
Correlate your telemetry

Monitoring



Correlate your telemetry

Monitoring



Health checks

Report status of your application - Is my application healthy? Is it ready?

Use them to verify deployments, measure latency, up time, cold start, ...

Always provide throttling to block noisy consumers

Think about your connection management

Go as far as you want

No direct business value, until it's too late

Health checks

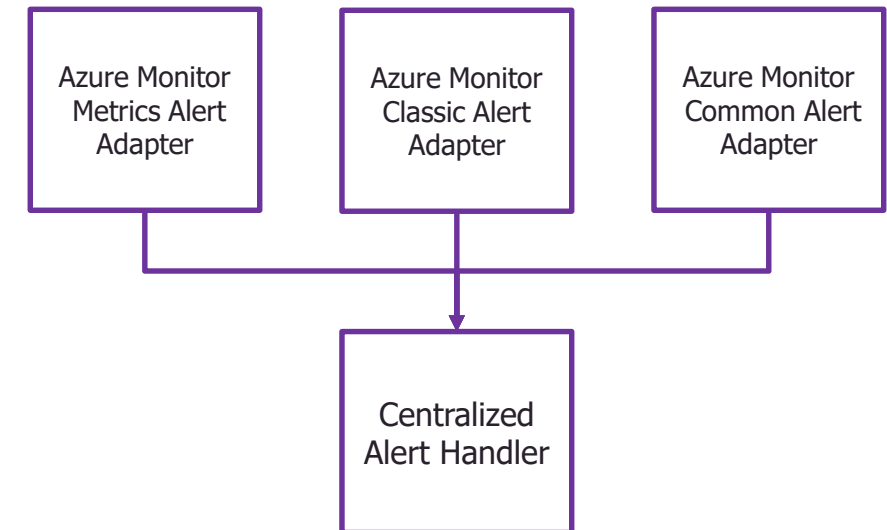
```
using System.Net;
using Microsoft.AspNetCore.Mvc;
using Swashbuckle.AspNetCore.Annotations;

namespace Promitor.Scraper.Host.Controllers.v1
{
    [Route("api/v1/health")]
    public class HealthController : Controller
    {
        /// <summary>
        ///     Get Health
        /// </summary>
        /// <remarks>Provides an indication about the health of the scraper</remarks>
        [HttpGet]
        [SwaggerOperation(OperationId = "Health_Get")]
        [SwaggerResponse((int)HttpStatusCode.OK, Description = "Scraper is healthy")]
        [SwaggerResponse((int)HttpStatusCode.ServiceUnavailable, Description = "Scraper is not healthy")]
        public IActionResult Get()
        {
            return Ok();
        }
    }
}
```

Handling alerts

Monitoring

- | Always automate alert creation, they are part of your infrastructure as well
- | Build a centralized alert handling process
 - | Azure Logic Apps is a good fit for this
- | Different alerts have different contracts
 - | Use adapters to receive notifications
 - | Map to internal metric contract
 - | Handle via centralized alert handler



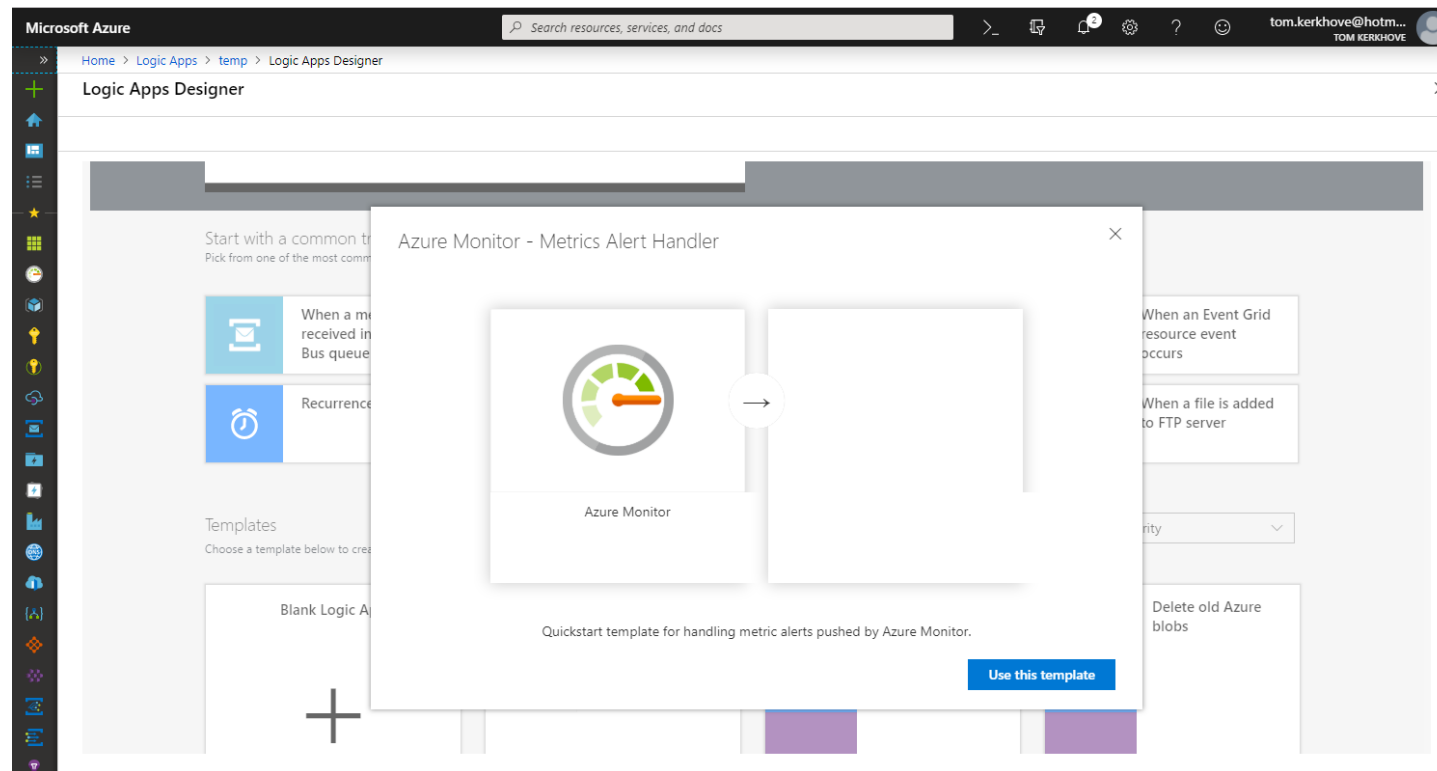
Time to move it! Azure Classic Alerts will be deprecated by end of August 2019.

<https://docs.microsoft.com/en-gb/azure/azure-monitor/platform/monitoring-classic-retirement>

Handling alerts

Monitoring

| Use the Logic App template for Azure Monitor!



Write Root Cause Analysis (RCA)

Train your team for PROD outages, write RCAs in all environments

Did our alerts detect it? Did we have enough telemetry?

Provides a structured way of analysing your platform

Use as a knowledge transfer to customers & team

Define action points and follow-up on them

Use them to detect recurring issues

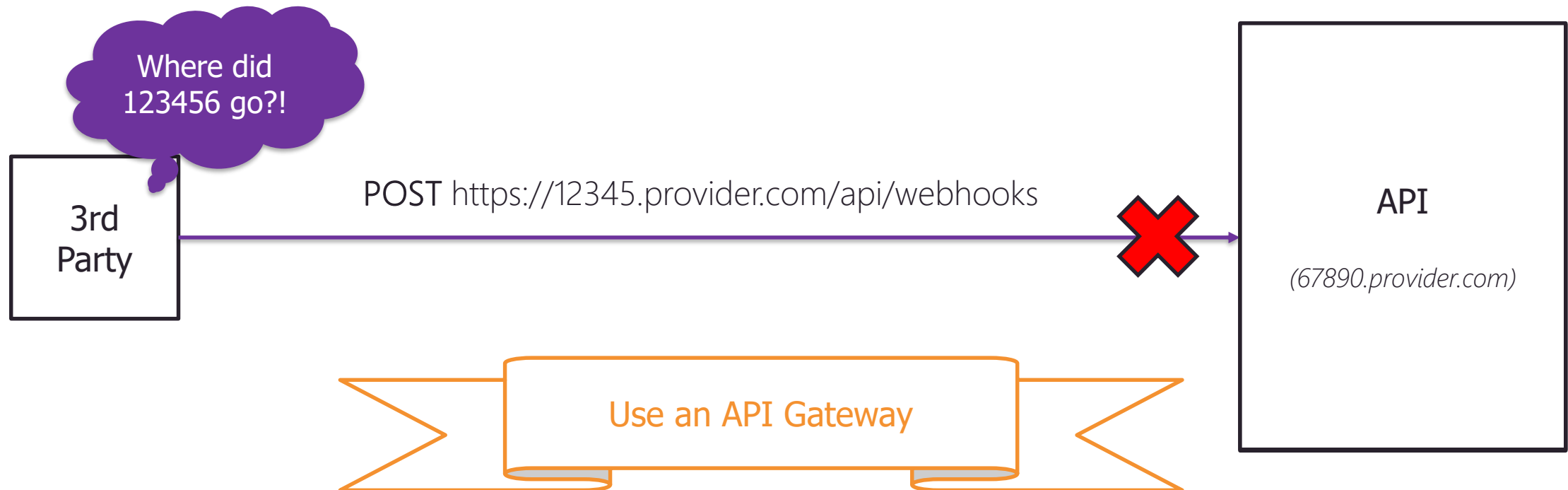
There is no such thing as failure, only opportunities to learn

Webhooks

Consuming webhooks

Webhooks

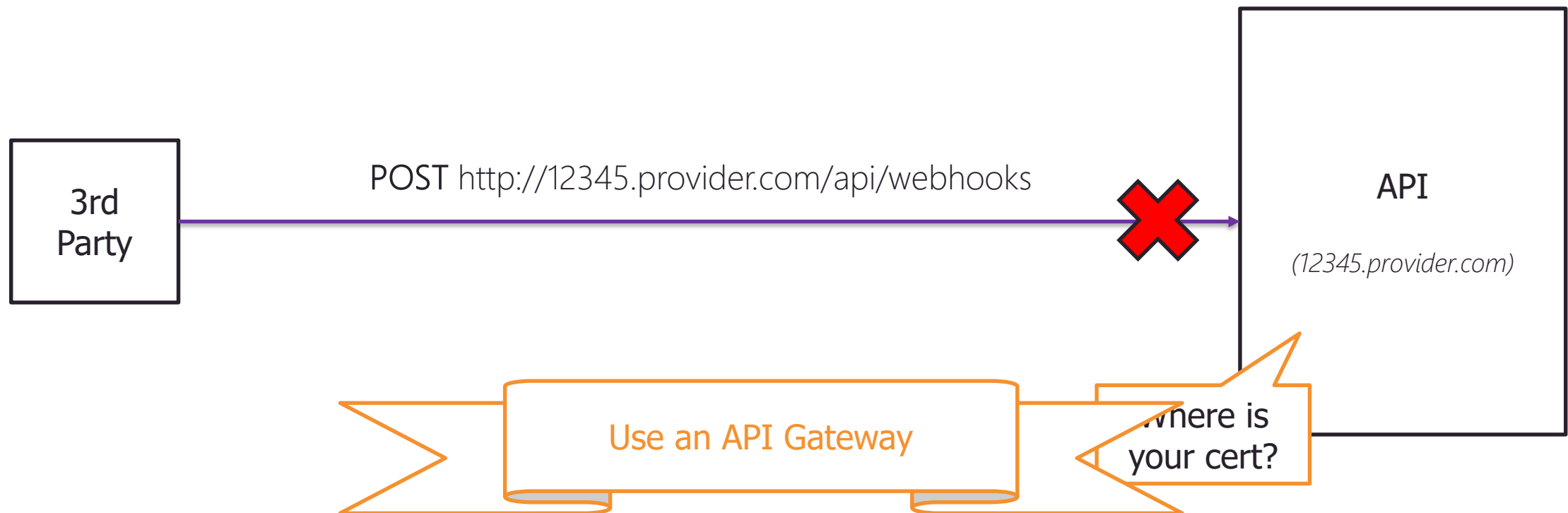
- | Generated URLs are evil, provide good DNS names of your services
 - | And this goes for everything, not only webhooks



Consuming webhooks

Webhooks

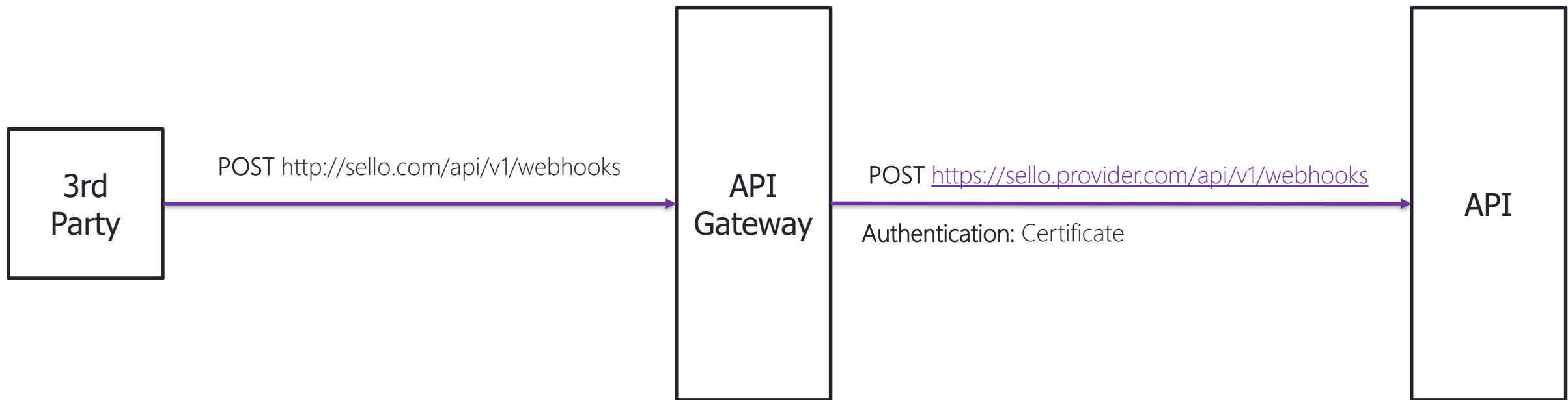
| Do not reduce your API security because of your 3rd Party



Consuming webhooks

Webhooks

| Always route webhooks through an API gateway

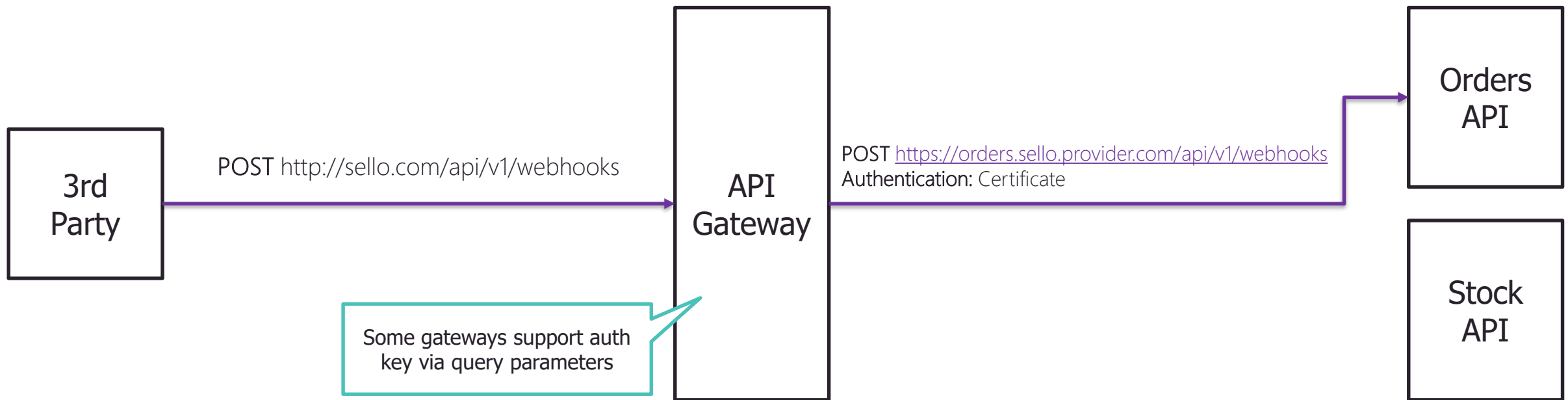


| This decouples the webhook from your internal architecture

Consuming webhooks

Webhooks

| Always route webhooks through an API gateway



| This decouples the webhook from your internal architecture

Provide user-friendly webhooks

Provide a way for consumers to provide context during registration

Provide a self-service CRUD API to register new subscriptions

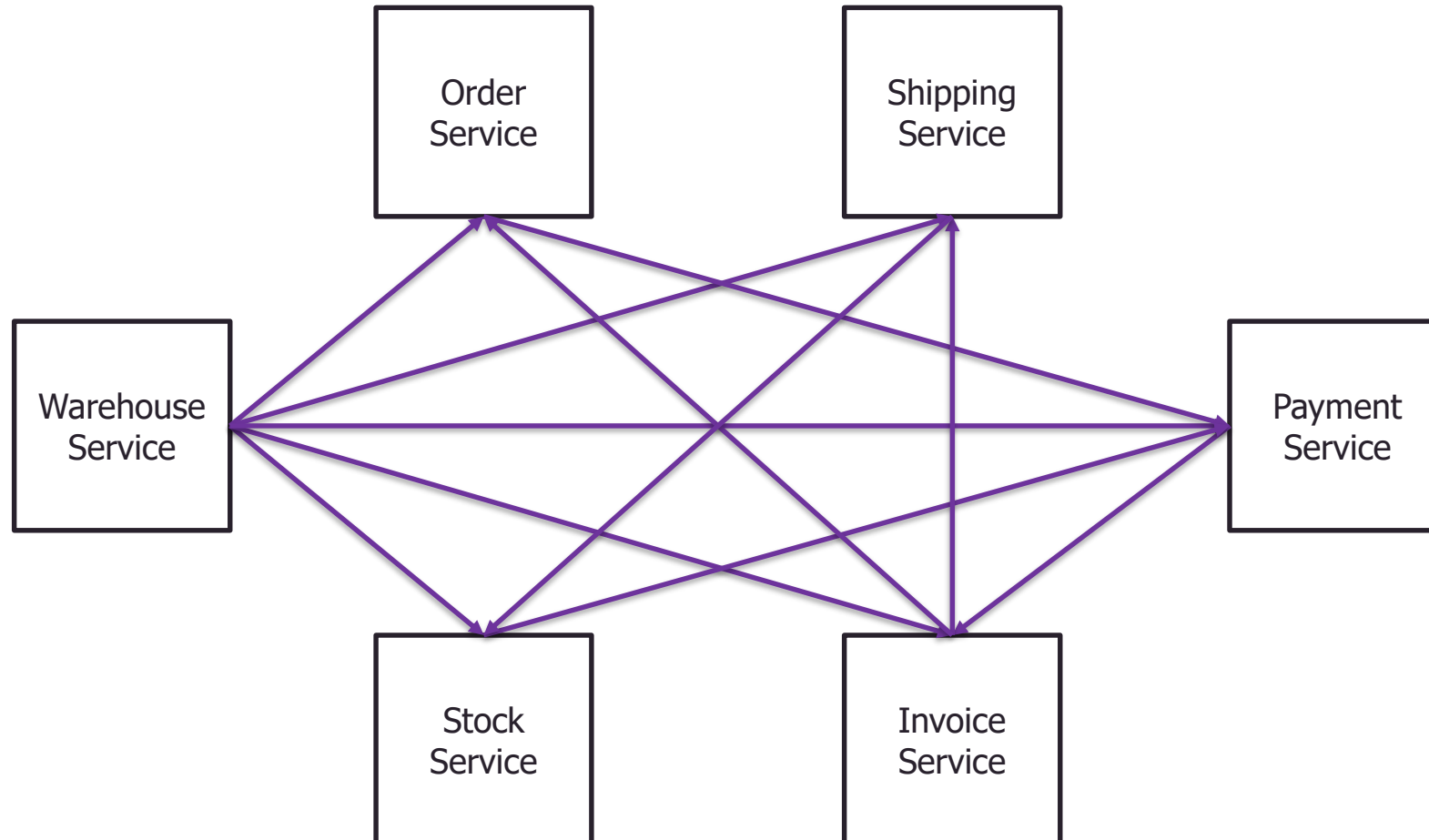
Pass your correlation id via Request-Id header

Provide an invocation history

Think as a webhook consumer, not publisher.

Spaghetti infrastructure 2.0?

Webhooks



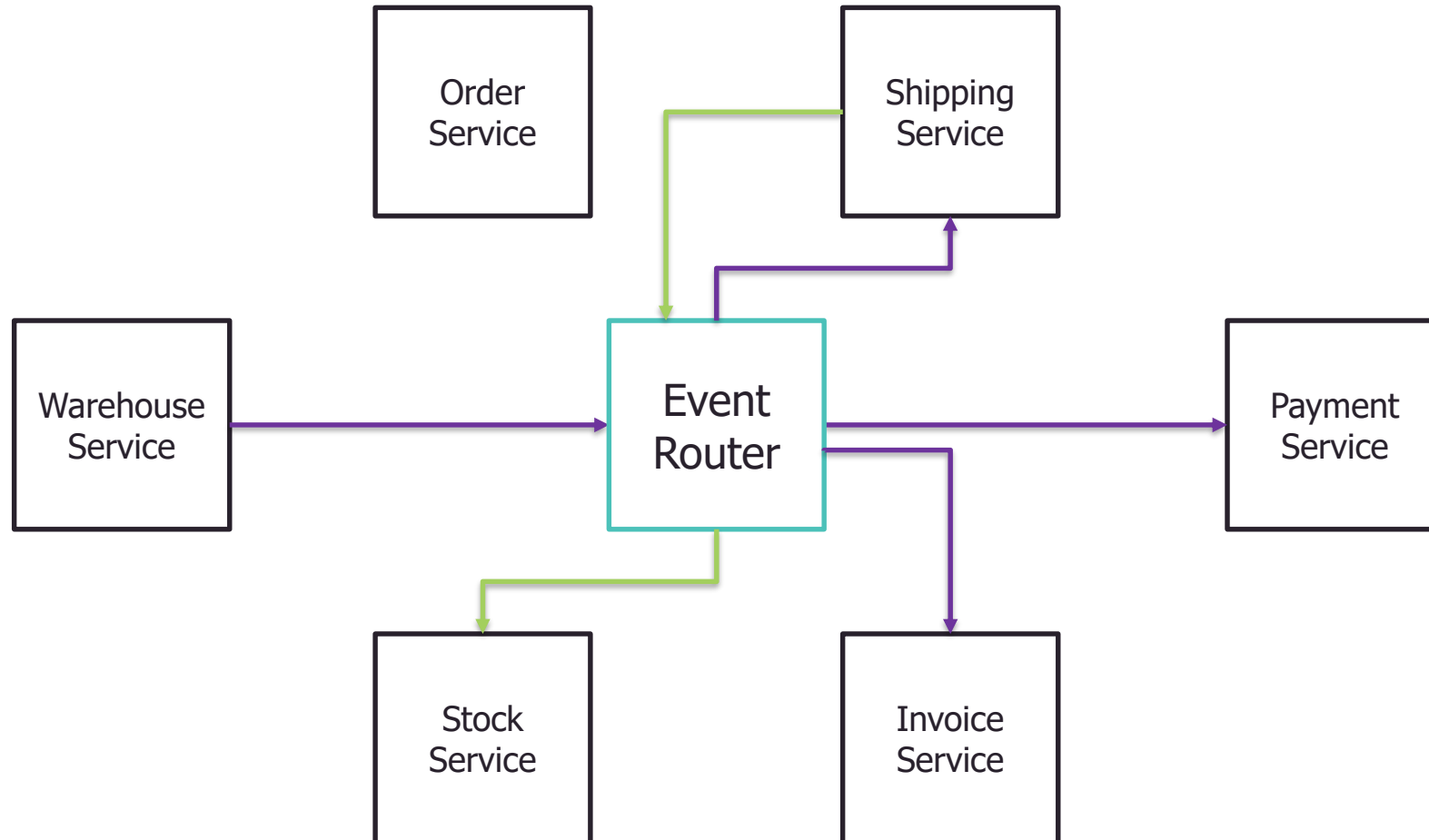
Spaghetti infrastructure 2.0?

Webhooks

- | Long-term this can start to become a burden
 - | A lot of bookkeeping to know who to update, how we should authenticate, etc
 - | No central place to route all webhooks through
- | Your platform needs to be robust
 - | What if subscriber II is not responding? Let's build a retry mechanism!
 - | Who says subscriber II owns foo.bar.com?
- | Webhooks should use a "I don't care, here's an update" approach

Event Routers

Webhooks



Event Routers

Webhooks

- | Event Routers do all the heavy lifting for you
 - | Provide a centralized hub for all things events
 - | Bookkeeping of whom subscribes to what webhooks and events
 - | They will retry sending events when they did not reply
 - | They will perform webhook validations
- | Publishers can publish events to the event router and takes it from there
- | Great for for internal usage, but harder to use with 3rd parties
 - | Webhook validation is not always easy to setup

Tips

Webhooks

- | Webhooks are not durable, if you are not around you will miss it.
 - | If you need to ensure at-least-once delivery, consider using a broker instead
- | Store audit entry of webhook that are being pushed
 - | Can be important in case of a dispute
 - | *Optionally even include the response of the consumer*
- | Do not only allow global registrations, consider serving more granular updates
 - | For example, I want updates of one flight instead of all flights
- | Provide rate limiting on your webhook endpoints
 - | Don't let your platform go down by your 3rd party provider
- | Webhooks are contracts as well
 - | Provide good documentation and version them

Use Webhooks & Events internally

Build fully automated reactive applications / data ingestion pipelines

Decouple teams from each other

Provide capability to extend

Incident report - NuGet.org downtime on March 22, 2018

March 22, 2018 by Svetlana Kofman

We did this blog post to [report about the incident that happened on March 22, 2018](#). In the last couple of days we dug deeper into the incident. Here is the summary of our findings and proposed next steps.

Customer Impact

NuGet.org website and V2 APIs were unavailable for 2 hours on March 22, 2018 between 8:45AM - 11:30AM UTC. More than 1.5 million requests failed.

What Happened?

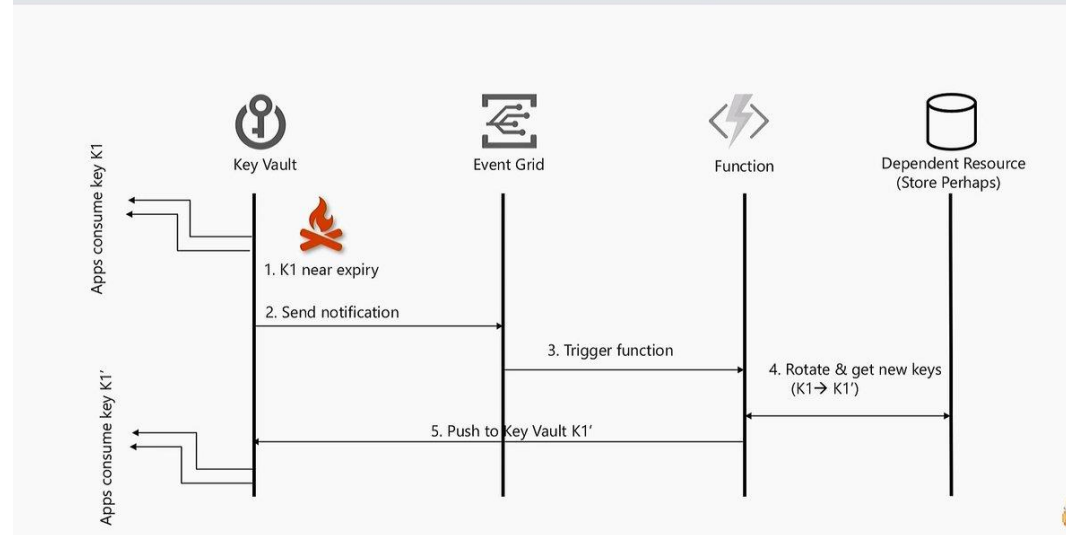
On March 22nd, a certificate used internally for authentication with Key Vault expired. It was rotated on all components except for a single Search service. This triggered a chain reaction that made the Gallery service unavailable, resulting in continuous failover attempts between the primary and secondary instances. The incident was mitigated by redeployment of the search service with a renewed certificate.

Azure Event Grid, the heart of Azure

Example

- | Azure Key Vault is working on native Event Grid Events
 - | This provides the capability to fully automate certificate management

Notifications—A touchless way to stay secure



- | The power of these events can leverage closer integration by other services such as Azure App Services, API Management who can consume latest version of cert

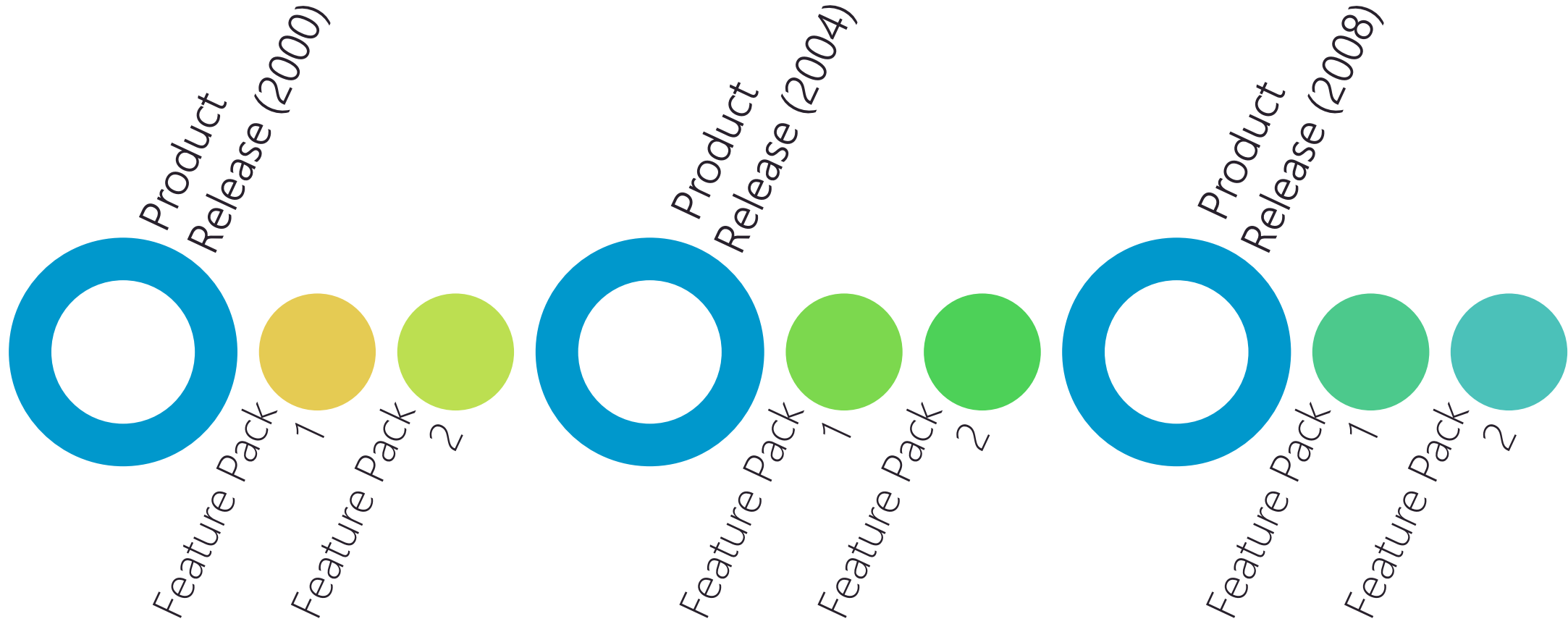
<https://www.codit.eu/blog/azure-event-grid-the-heart-of-azure/>

WEBHOOK



No.

Embrace Change



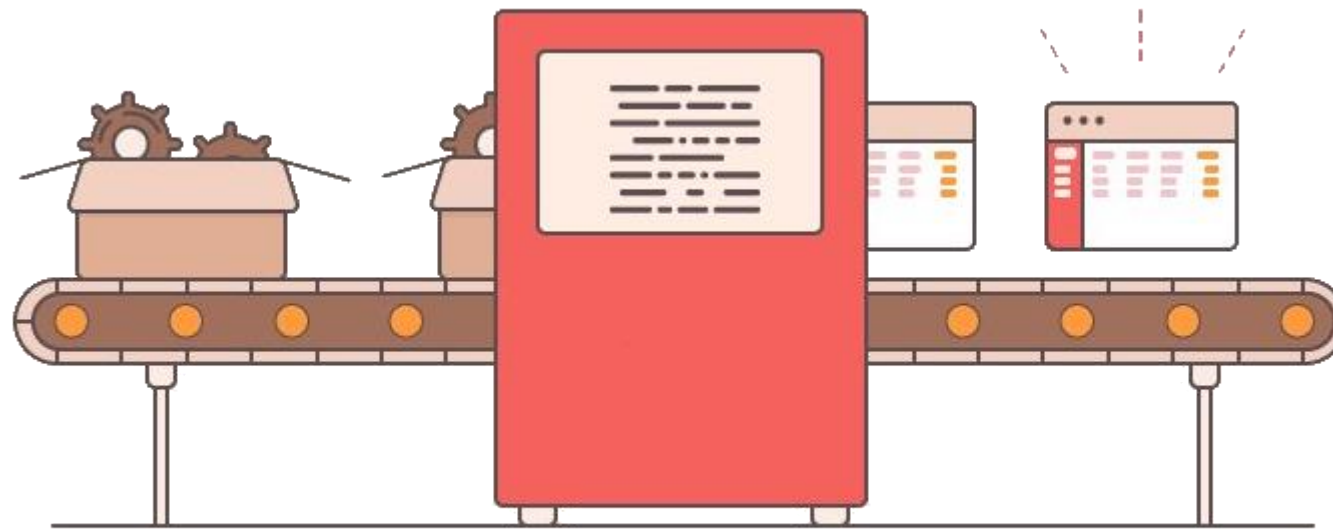
How we used to ship

Stable releases every few years

Hard to shift product focus

And then came agile...

Releasing Software to Production multiple times a day



DevOps



DevOops

You are not Netflix

DevOps is a culture and requires a mind shift

Manual interventions are evil, automate (as much as possible)

Create automated pipelines for shipping software
(deployment rings are awesome)

Use infrastructure/build as code

We live in a world of constant change

Our underlying infrastructure is constantly moving & changing

Cloud vendors are competing to offer unique services

Staying up to date is a lot harder

Service Bus for Windows Server

Azure BizTalk Services

Azure Data Factory v1

Azure Hybrid Connection

Who knows these services?

Azure RemoteApp

Azure Container Services

Azure Access Control Service

Azure Alerts

The lifecycle of a service

Embrace Change

Private Preview

- Rough version of product
- Shared under NDA to limited group

General Available

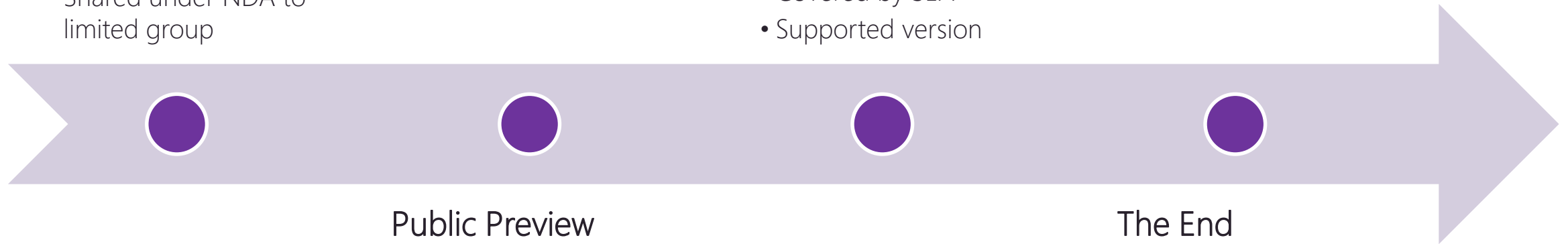
- Covered by SLA
- Supported version

Public Preview


- Available to the masses

The End

- Deprecation
- Silent Sunsetting
- Reincarnation in 2.0



The end of the road. . .



Can also be tooling cfr.
Azure DevOps Cloud-
based Loadtesting

Embrace Change

| **Official deprecation**

- | Officially announced as deprecated
- | Migration is required before service shutdown

| **Reincarnation**

- | A new version of the service arises in a new version
- | Can be part of service or service in total

| **Silent deprecation**

- | No further development in the product
- | Service is still running smoothly
- | Does not mean you should stop using it

Choosing an Alternative



Let's use the shiny one, right?!

Maybe.

Choosing an Alternative

Use the tool that fits your needs, not perse what you know

Be careful with the latest shiny technology

Decide as a team

Build or buy

There is no silver bullet

Questions you should ask

What is the learning curve? Is it worthwhile?

Does it have a vendor lock-in?

Is it operable?

Does it have a future?



You learn by doing

And sometimes, you regret your choices.

Cloud platforms are never finished

Your platform evolves, and so does its needs

Prepare for your migrations

Nothing is written in stone

Use a product mindset

Change is coming, so you'd better be prepared

Stay up to date with Azure Deprecation Notices

Dashboard with deprecation notices concerning Azure services, regions, features, APIs and SDKs

Search for services which you depend on

Get automated reminders (WIP)

@AzureEndOfLife on Twitter

Conclusion

Conclusion

Technologies have scalability capabilities & trade-offs

Provide user-friendly webhooks & route them via API gateways

Define & roll out a good monitoring strategy

Automate everything, it will save you one day

You build it, you run it

We live in a world of constant change, so be prepared

BIZTALK360

SERVERLESS360

ATOMIC SCOPE

TECHNOLOGY PARTNER



INTEGRATE 2019

PLATINUM SPONSORS



GOLD SPONSORS



Questions?



Twitter: @TomKerkhove

GitHub: @TomKerkhove

blog.tomkerkhove.be
codit.eu