
plain concepts
Webinar
Resilient Architectures:
The power of
embracing failure
13/5/2020
16:00 (GMT+02)
Duración: 1 h

Carlos Landeras

Development Team Lead at @Plainconcepts
Madrid & Benelux

Microsoft MVP Developer Technologies

@Carlos_Lande

CarlosLanderas

Resilience

Is the ability to absorb or avoid damage without suffering complete failure

Resilience

Is the ability to recover from failures and continue to function

Resilience

Is about accepting the fact that failures...

Will occur!

Resilience

Fail fast, fail often!

Detecting failure early reduces the cost of a fix.

Resiliency experiments detect potential failures before they become a catastrophe

Resilience facts

“Failures are a given, and everything will eventually fail over time.” Werner Vogels, Amazon CTO

“Resilience is all about being able to overcome the unexpected”

“The goal of resilience is to thrive”

“Push your system almost to the breaking point”

plain concepts

Resilience

It’s all about balance…

Cost of being resilient

Loss of money due to outages

Clients happyiness

Software provider reputation

Resilience at different levels

- Infrastructure Layer
- Networking and Data
- Software and application design
- Site Reliability engineering team & Developers

Resilience Patterns

Redundancy

- Duplicate elements to avoid having a single point of failure

- Increase overall availability of the system

Component	Availability	Downtime
X	99% (2-nines)	3 days 15 hours
Two X in parallel	99.99% (4-nines)	52 minutes
Three X in parallel	99.9999% (6-nines)	31 seconds

Multi-Region redundancy

- One region goes down, traffic routes to the closest region without intervention
- Closest region routing by latency

Azure Load Balancers

Global

Front Door (layer 7)

Great for HTTP Acceleration, Affinity, SSL offload, **instant failover**, path routing, WAF, Rate limit, Caching (HTTP/S)

Traffic Manager – Dns Resolver

Balances at domain level
Great for TCP, UDP (non-HTTP/s)
Slower failover (DNS Caching, TTLs honoring)

Regional

Application Gateway (layer 7)

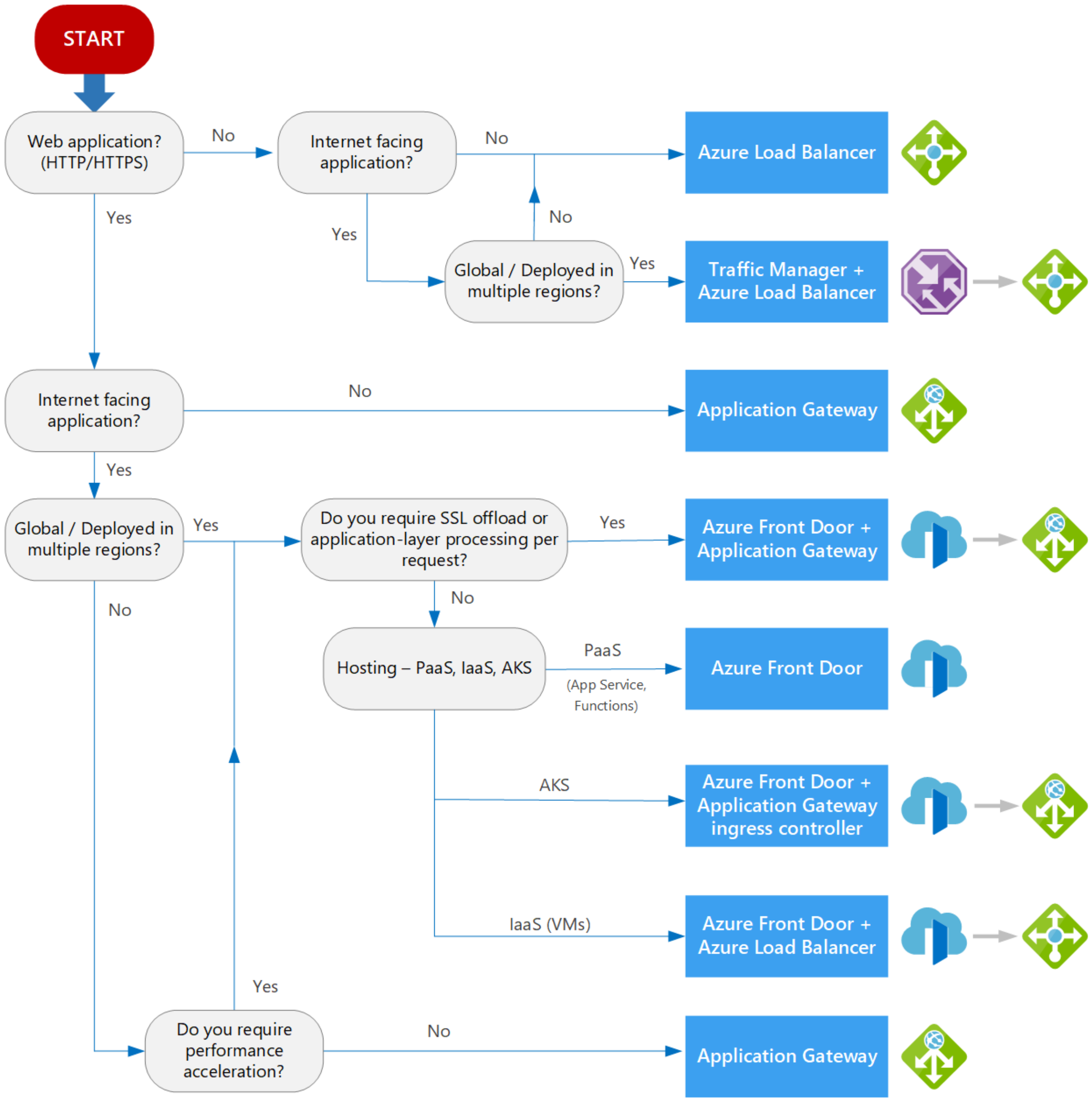
SSL offload, E2E SSL, WAF, L7 Load balancer (HTTP/S)

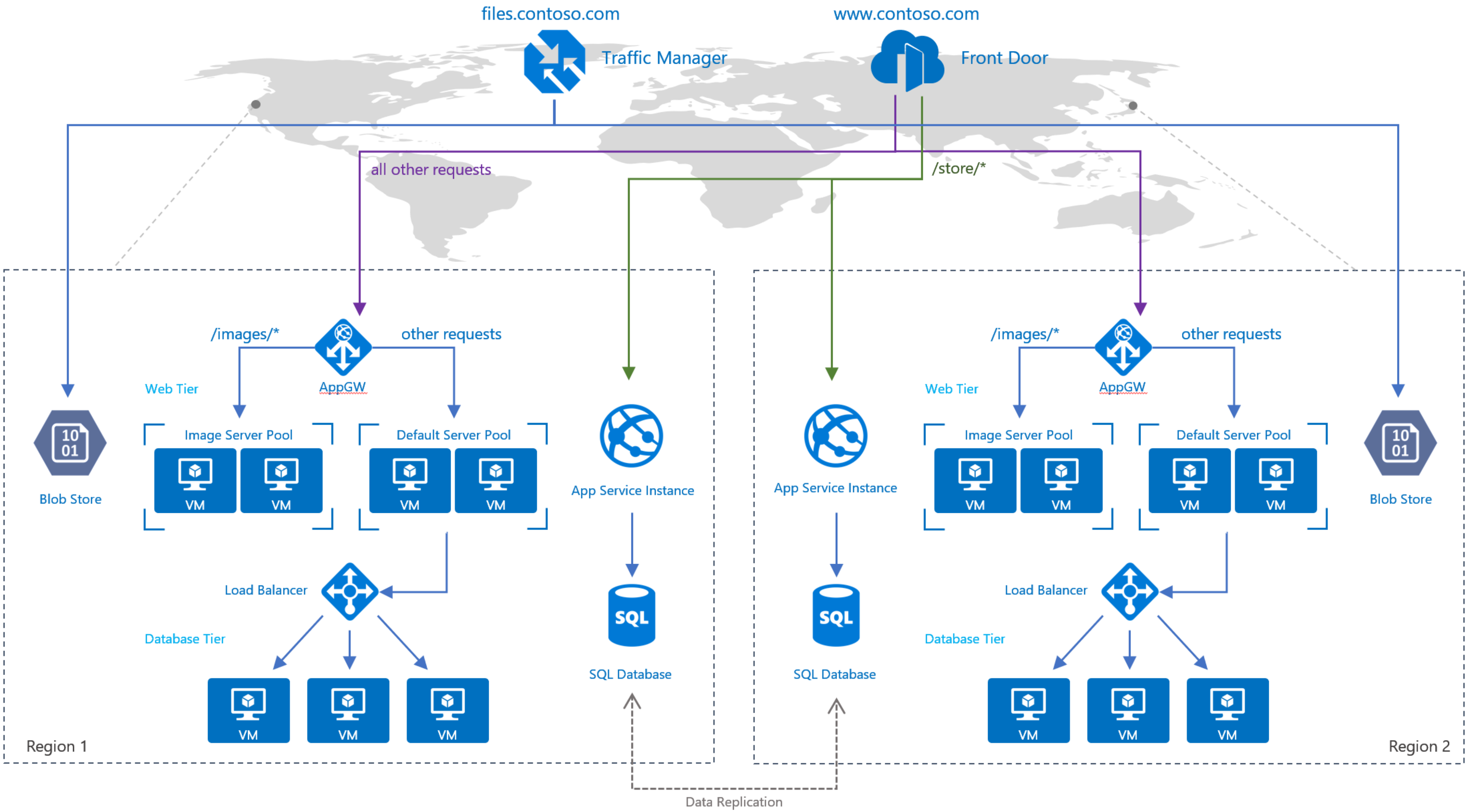
Azure Load Balancer (layer 4)

Great for TCP, UDP (non-HTTP/s)
Low latency, designed to handle millions rps

Front Door Fail Over Demo

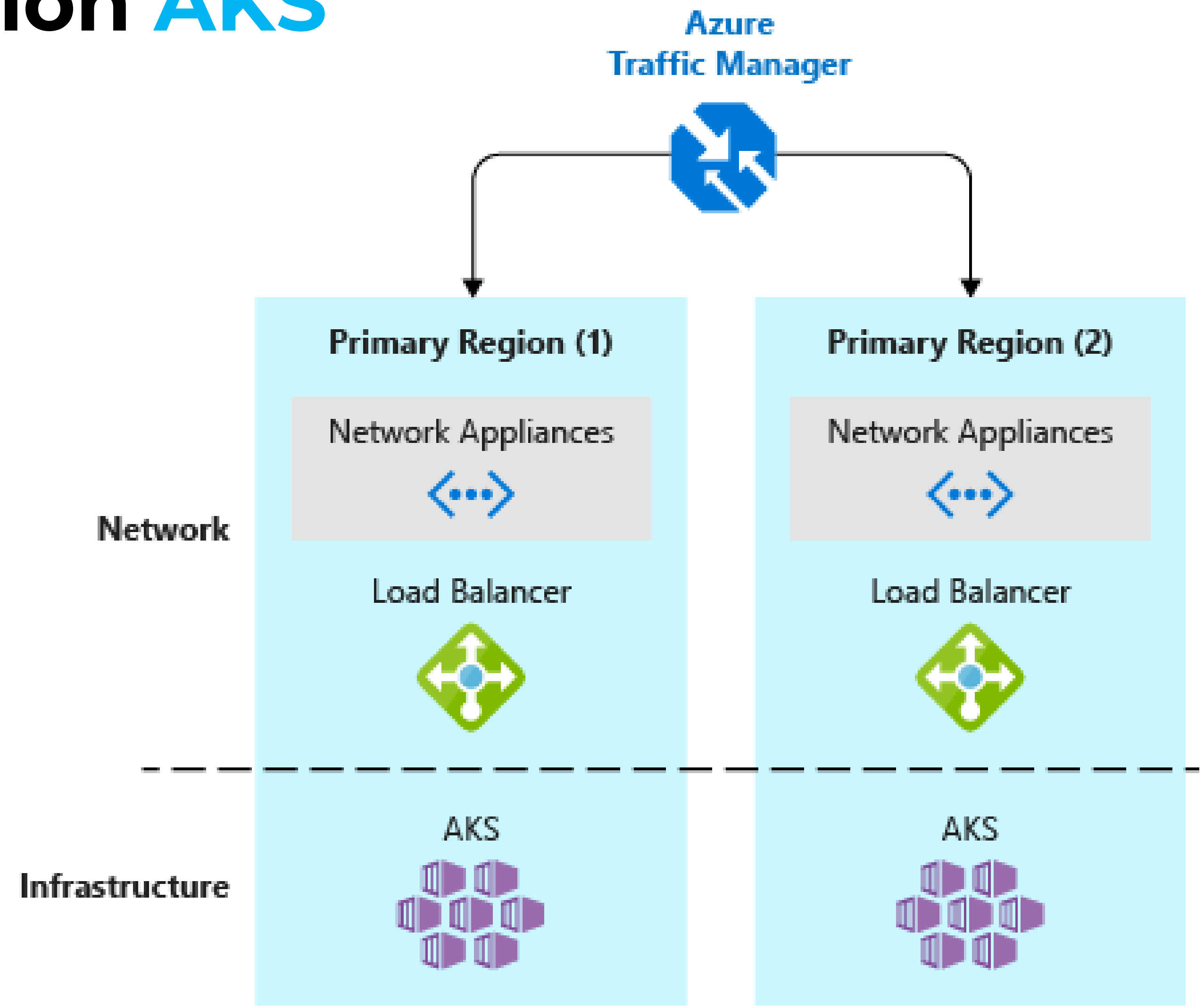
Decision tree



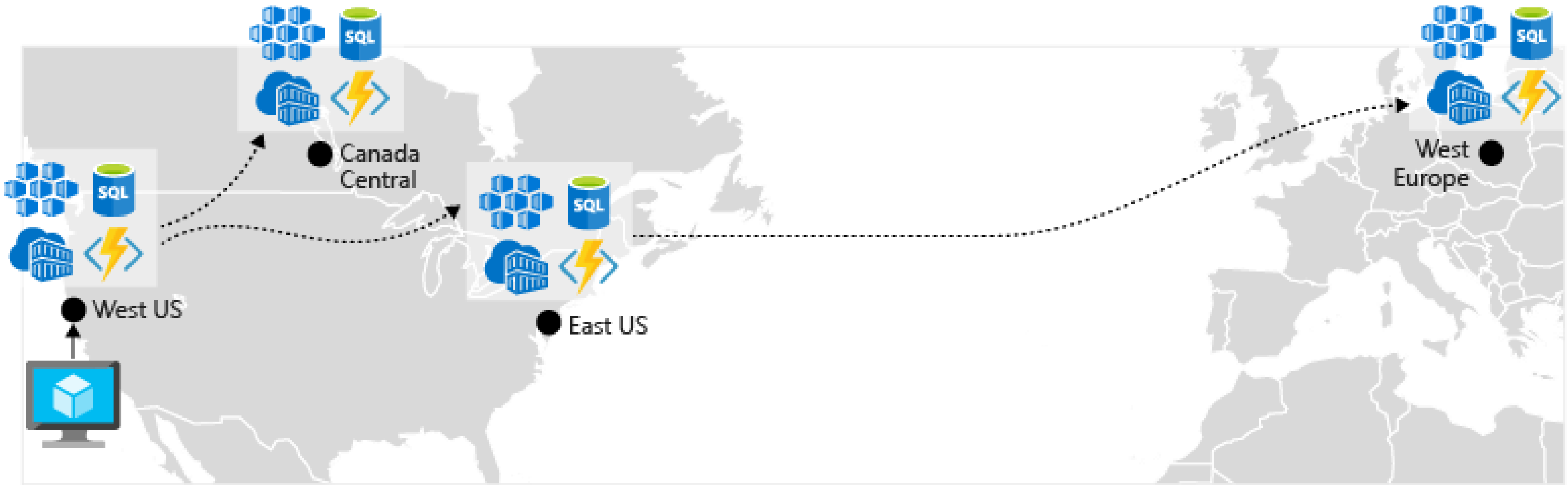


files.contoso.com
Traffic Manager
www.contoso.com
Front Door
all other requests
/store/*
/images/*
other requests
AppGW
Web Tier
Blob Store
Image Server Pool
Default Server Pool
VM
App Service Instance
Load Balancer
Database Tier
SQL
SQL Database
Region 1
Region 2
Data Replication

Multi-Region AKS



Multi-Region AKS



Geo Replication tips

- **Stateless Applications** (Avoid sessions, local state)
- State should be shared across regions (Distributed cache, replicated databases
- Only use region resources
- If you wan't clients to stick to resources, use "stickiness"
- If you do not respect this, you'll probably have balancing issues

Auto Scaling

- Know the different scaling models for your cloud resources

- Choose your tiers and configure scaling (when needed)

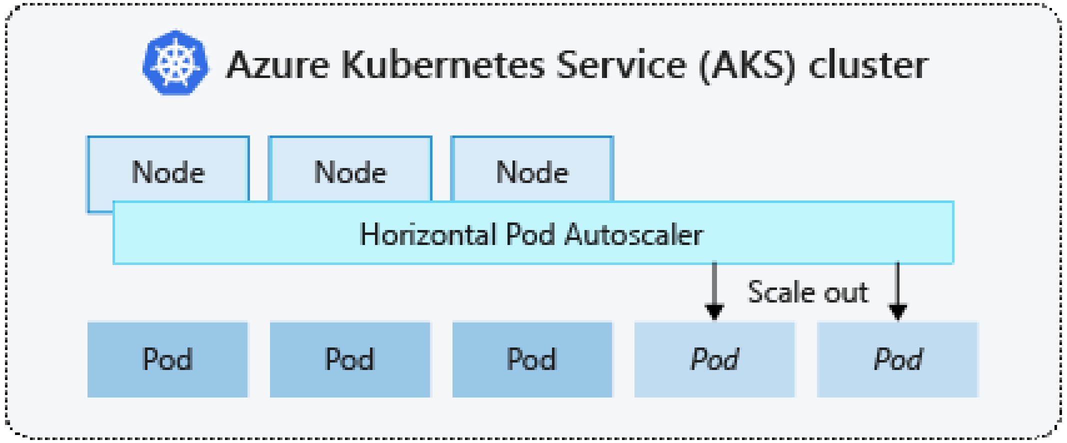
Infrastructure Scaling:

- Azure Firewall (auto scaled)
- Azure Front Door (auto scaled)
- Azure Traffic Manager (DNS based)
- Azure Application Gateway (manual / autoscaling)
- Azure Load Balancer (Basic / Standard 10x tiers)
- Azure Kubernetes Service Nodes (manual / autoscaling)
- Azure Sql Server (V/H), Redis (tiers), and Cosmos DB (Region RUs)

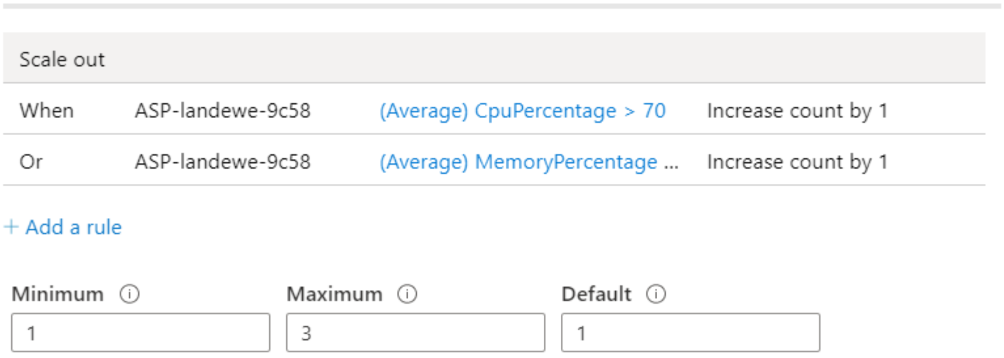
Auto **Scaling**

Application Level Auto Scaling

Kubernetes HPA



Web App metrics-based auto scaling



Infrastructure as Code

Some facts

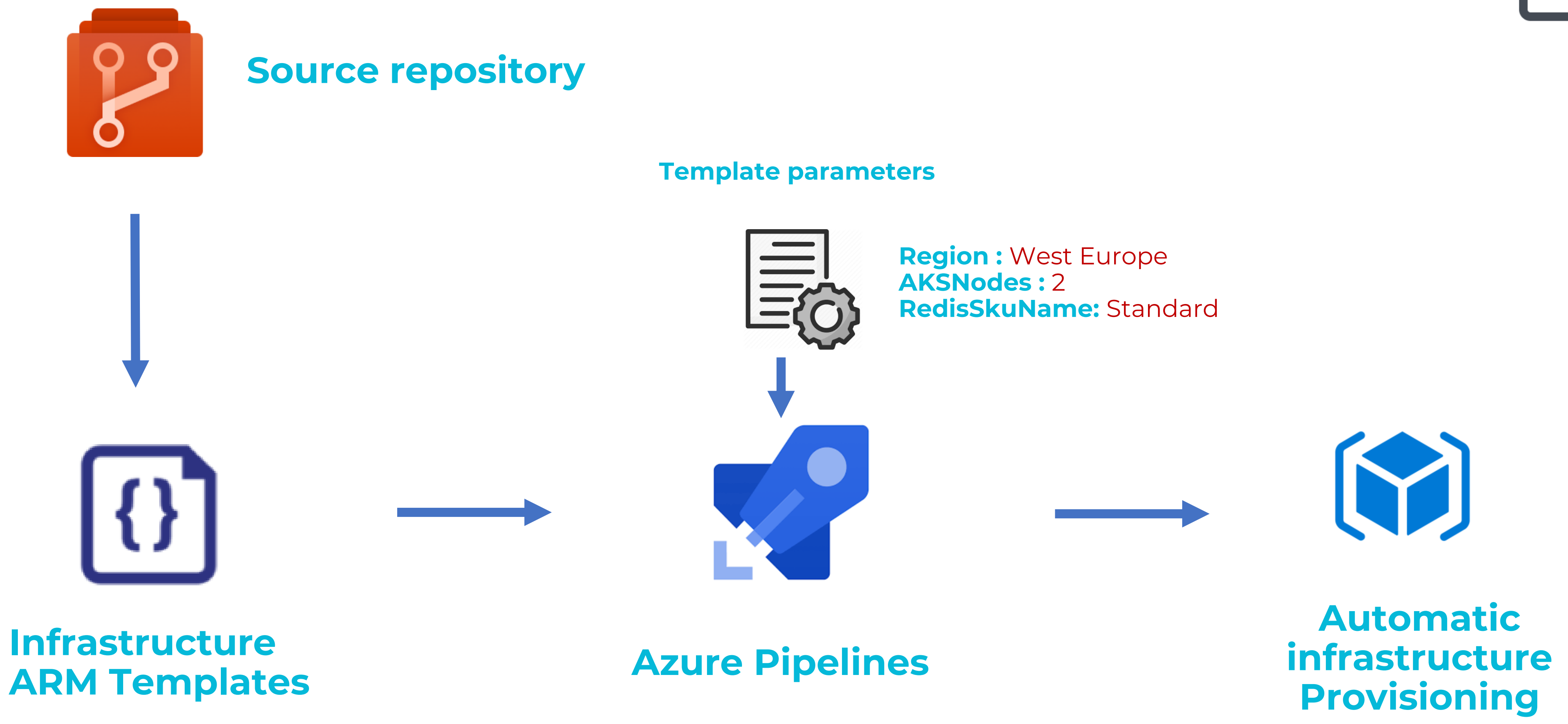
- As humans, we are not very good at repetitive tasks. We are very error prone.
- Complex systems are hard to manually reproduce right at first
- If we suffer a datacenter or infrastructure disaster, we are in a race to prevent lost of revenue.
- Human errors are a reality. Ooops, I deleted the production Resource Group. I swear I was in the Development blade!
- **All above = Unhappy clients :_(**

Infrastructure as Code

Solutions

- Infrastructure as code guarantees **repeatability** and being up and running in minutes
- Machines are very good at repeating tasks with the same exact output. We are not.
- Infrastructure as code provides system history preservation
- We can fastly reproduce our infrastructure in other region if needed

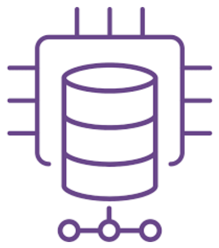
Infrastructure as code in Azure



Caching for resilience

Cache: hardware or software component that stores data so that future requests for that data can be served faster (Wikipedia)

Why caching?



- Accelerate content delivery = Better user experience
- Distributed caches provides shared state in distributed architectures (Stateless)
- Improves application scalability avoiding I/O operations, network connections and relieves database stress
- In a short timespan, we can be serving the exact same content to hundreds / thousands of users. Caching increases throughtput and RPS and helps preventing database connections exhaustion.
- Enables graceful degradation and fast fail (We will see this later!)

plain concepts

Why caching?

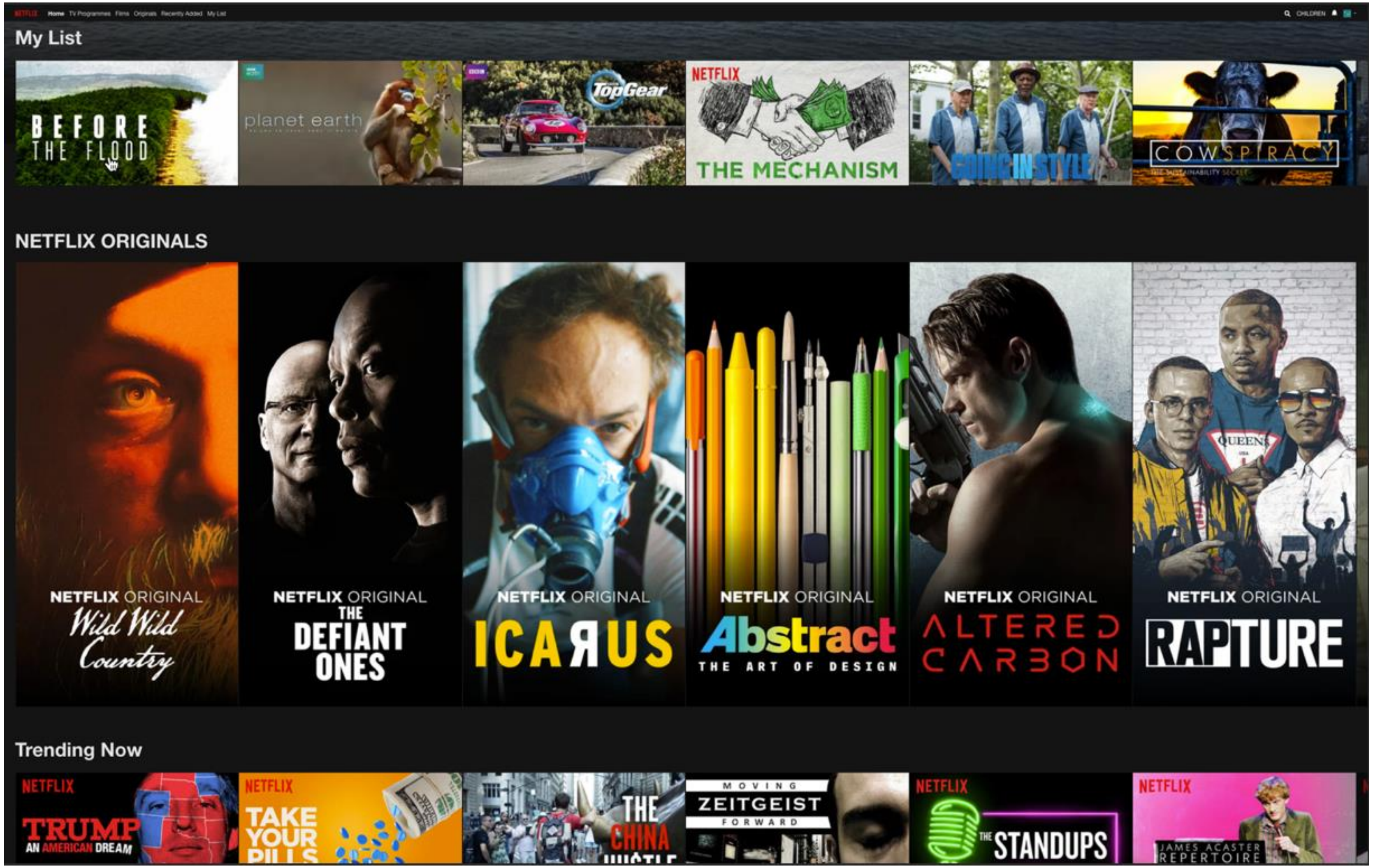


Image credits: [Adrian Hornsby] Principal Developer Advocate awscloud

Typical cache usage : Cache-Aside

```
public async Task<Data> GetSomeData(string key)
{
    var data = _cache.GetAsync(key);
    if(data == null)
    {
        data = _query.Get(key);
        await _cache.SetAsync(key, data, options);
    }

    return data;
}
```

Risks of caching

- **Data Staleness:** Risk of serving old data
- You need to balance right keys expiration times to prevent long aged data.
- If you evict keys very often, this causes performance problems.
- You need to measure application request patterns and volume in order to properly adjust expiration times
- **Eventual consistency:** Changes in the database or cache nodes are not immediately reflected.

Health Checks

Health Checks are designed to retrieve information about the health of a service / application and its dependencies.

Health Checks are present in several cloud resources. They are used to ensure availability and decide where to route requests

- Azure Front Door
- Azure Traffic Manager
- Load Balancers

Health Checks

In the times we are living in, Health Checks are a must have for applications as well.

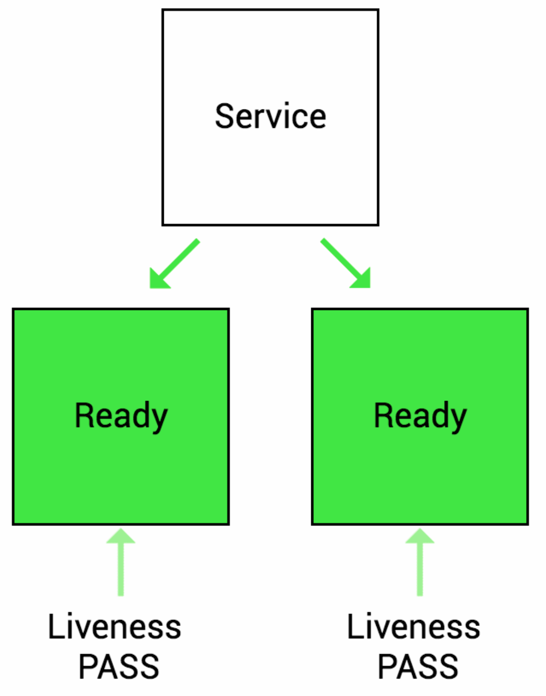
Every application should expose health checks mechanisms. Self health and dependencies status.

Implementing health checks, allow infrastructures and container orchestrators to execute healing and high-availability strategies

Liveness Probes

Kubernetes **liveness** probes allow the orchestrator to kill a pod container instance that is not working properly.

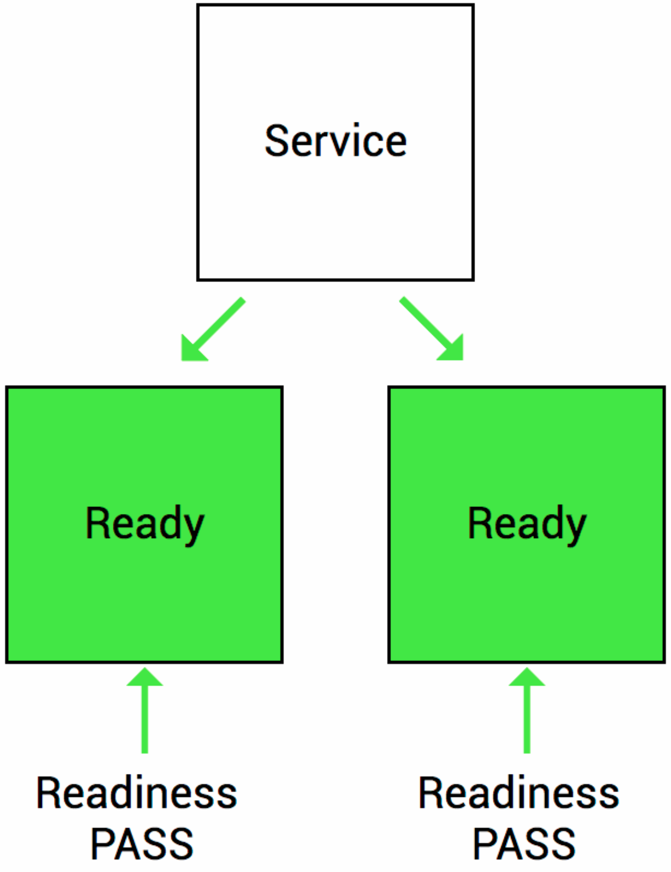
This helps to recover from eventual or transient failures.



Readiness **Probes**

Kubernetes **readiness** probes allow the orchestrator to exclude non ready containers from traffic.

While not ready, the container won't serve requests.



plain concepts

Health Checks

More on Health Checks

SDN Cast – Carlos Landeras about AspNetCore and Kubernetes HealthChecks

https://www.youtube.com/watch?v=kzRKGCmGbqo&t=198s

Kubernetes liveness and readiness probes using HealthChecks

https://github.com/Xabaril/AspNetCore.Diagnostics.HealthChecks/blob/master/doc/kubernetes-liveness.md

Kubernetes AspNetCore HealthChecks Operator

https://github.com/Xabaril/AspNetCore.Diagnostics.HealthChecks/blob/master/doc/k8s-operator.md

Kind of Failures

plain concepts

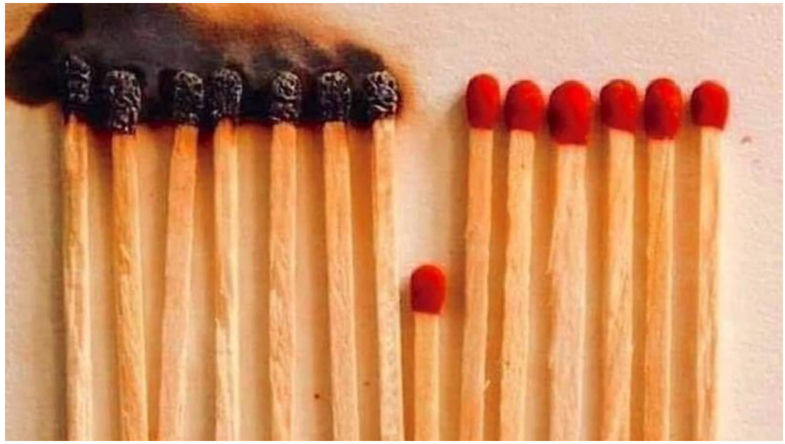
Transient Failures

All applications that communicate with remote services and resources must be sensitive to transient faults.

Transient faults include the momentary loss of **network connectivity** to components and services, the **temporary unavailability** of a service, or **timeouts** that arise when a service is busy

Cascading Failures

A **cascading failure** is a process in a system of interconnected parts in which the **failure** of one or few parts can trigger the **failure** of other parts and so on (Wikipedia)



A very common cascading failure is **overload**. The service struggles to serve requests and ends suffering resource exhaustion

Application Resilience Strategies

plain concepts

Resilience Strategies

Retry (policy family) (quickstart ; deep)	Many faults are transient and may self-correct after a short delay.	"Maybe it's just a blip"	Allows configuring automatic retries.
Circuit-breaker (policy family) (quickstart ; deep)	When a system is seriously struggling, failing fast is better than making users/callers wait. Protecting a faulting system from overload can help it recover.	"Stop doing it if it hurts" "Give that system a break"	Breaks the circuit (blocks executions) for a period, when faults exceed some pre-configured threshold.

plain concepts

Resilience **Strategies**

Timeout (quickstart ; deep)	Beyond a certain wait, a success result is unlikely.	"Don't wait forever"	Guarantees the caller won't have to wait beyond the timeout.
Bulkhead Isolation (quickstart ; deep)	When a process faults, multiple failing calls backing up can easily swamp resource (eg threads/CPU) in a host. A faulting downstream system can also cause 'backed-up' failing calls upstream. Both risk a faulting process bringing down a wider system.	"One fault shouldn't sink the whole ship"	Constrains the governed actions to a fixed-size resource pool, isolating their potential to affect others.

Resilience Strategies

Cache (quickstart ; deep)	Some proportion of requests may be similar.	"You've asked that one before"	Provides a response from cache if known. Stores responses automatically in cache, when first retrieved.
Fallback (quickstart ; deep)	Things will still fail - plan what you will do when that happens.	"Degrade gracefully"	Defines an alternative value to be returned (or action to be executed) on failure.

plain concepts

Chaos Injection

Exception: Inject random exceptions in our system randomly

Result: Substitute results to fake faults in our system

Latency: Random latency injection into system executions

Behaviour: Allows extra behaviour injection (Restart a service, kill a container, reboot a virtual machine, stress the cpu, block DNS resolution, Continuously write on a hard disk)

Chaos to the cluster Demo

GREMLIN

VS

Chaos and Resilience Demos

VS

POLLY

Credits:

Microsoft Azure documentation

Adrian Hornsby Principal Developer Advocate, Architecture @awscloud

plain concepts

Rediscover
the meaning of technology

Thank you for your time

www.plainconcepts.com

For further information

info@plainconcepts.com