ADAPTIVE
TO THE CORE

Microservices Transformation

A cautionary tale?

About Axe Group

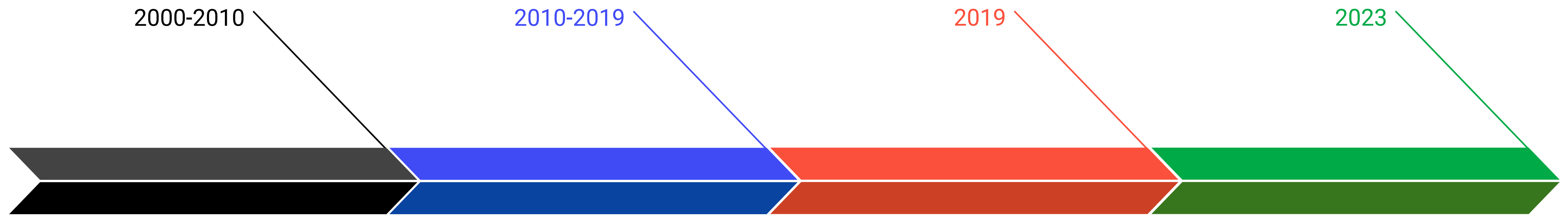
Axe Group is an InsurTech company, specialising in end-to-end Insurance systems, with a heavy focus on Life Insurance, but also supporting General Insurance.

We have a product called **Axelerator**

We've been around almost 25 years.

First and foremost a java house.

Product Evolution



Java, EJBs & Struts

Rules engine with customised front end, domain, and workflow unique to each client

Monolithic JEE Application

Standardised front end and foundation & insurance domains built upon rules engine and widget pages.

Microservices & PaaS

Standardised “Model” configuration of specific business lines (Retail & Group Life, and General Reinsurance), run as Platform-as-a-Service (PaaS).

Microservices & SaaS

Configuration now locked as a true off-the-shelf / out-of-the-box product with minimal configuration, run as Software-as-a-Service (SaaS).

Pivot

Agenda

This is case study of Axe Group’s Microservices Transformation of its flagship product Axelerator. We’ll go through the following:

- Why we embarked on a Microservices Transformation
- How did we plan out our journey
- Our tech stack
- What we didn’t plan for
- Learnings from the journey

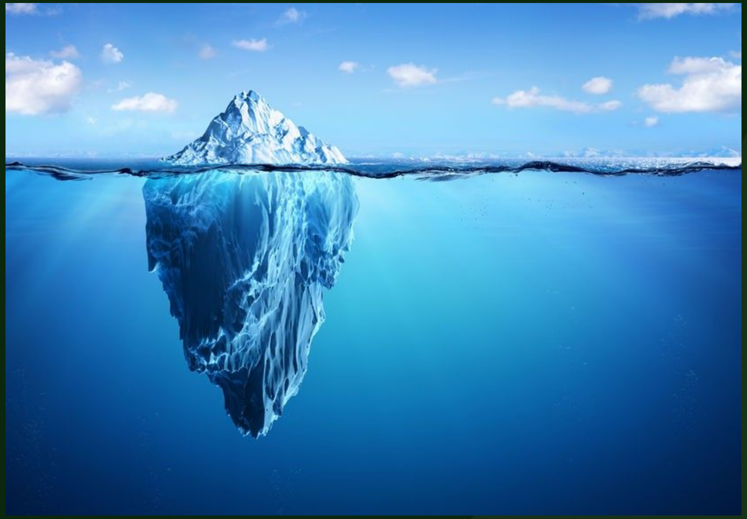
Safe Harbour

All the views expressed in this presentation are those of myself and my experience on our microservices transformation journey.

This is to offer insights from an organisation having done this and sharing our learnings so as to aid in making your own journey successful.

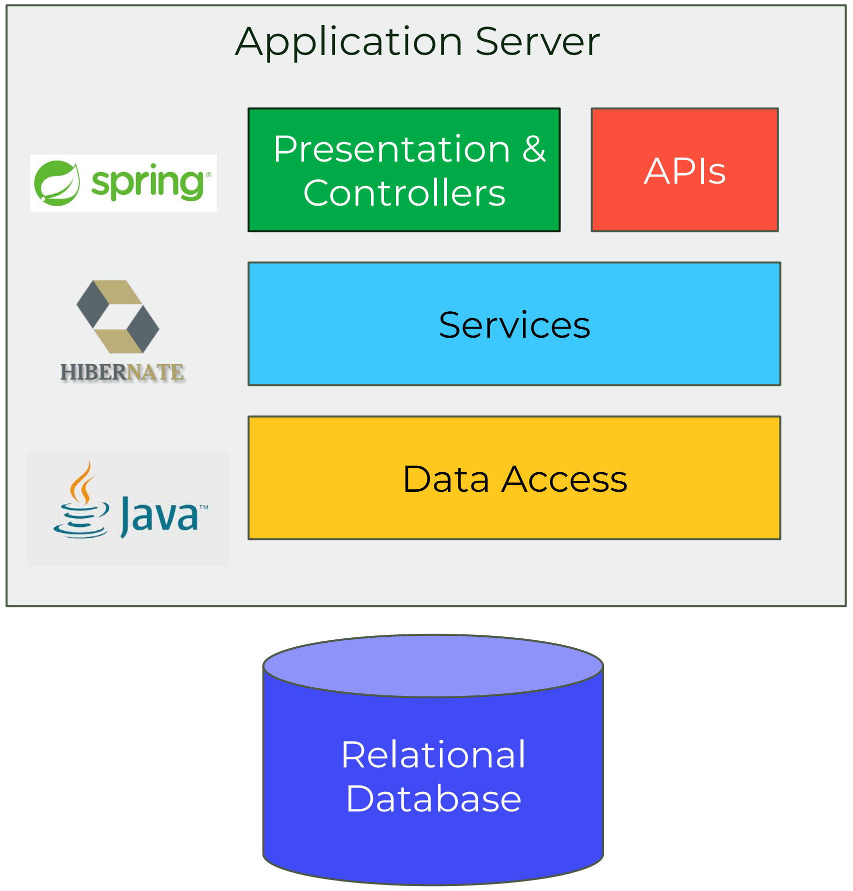
Never fear trying new things, but always have in the back of your mind that technology is like an iceberg:

the vast majority is below the surface.



Why we embarked on Microservices Transformation

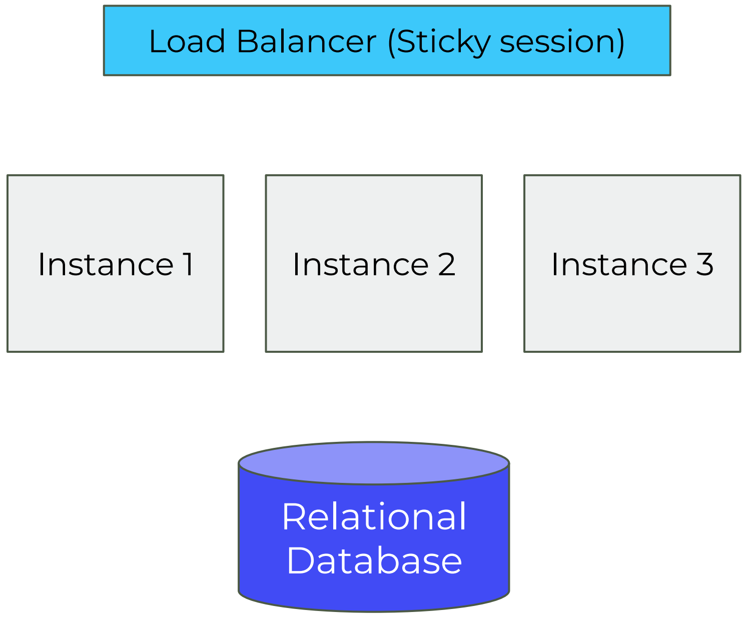
Monolithic Architecture



Axelerator's architecture was pretty standard of the JEE5 era:

- Hibernate as the JPA and DAO
- Spring for the service layer, IOC & Dependency Injection
- Servlet / Spring MVC / JSPs for Presentation Layer
- Spring REST / SOAP APIs
- Supported various relational databases (Oracle, MS SQLServer, IBM DB2, MySQL, PostgreSQL)
- Supported various application servers (IBM WebSphere, BEA Weblogic, RedHat JBoss / Wildfly)

Monolith Served us Well



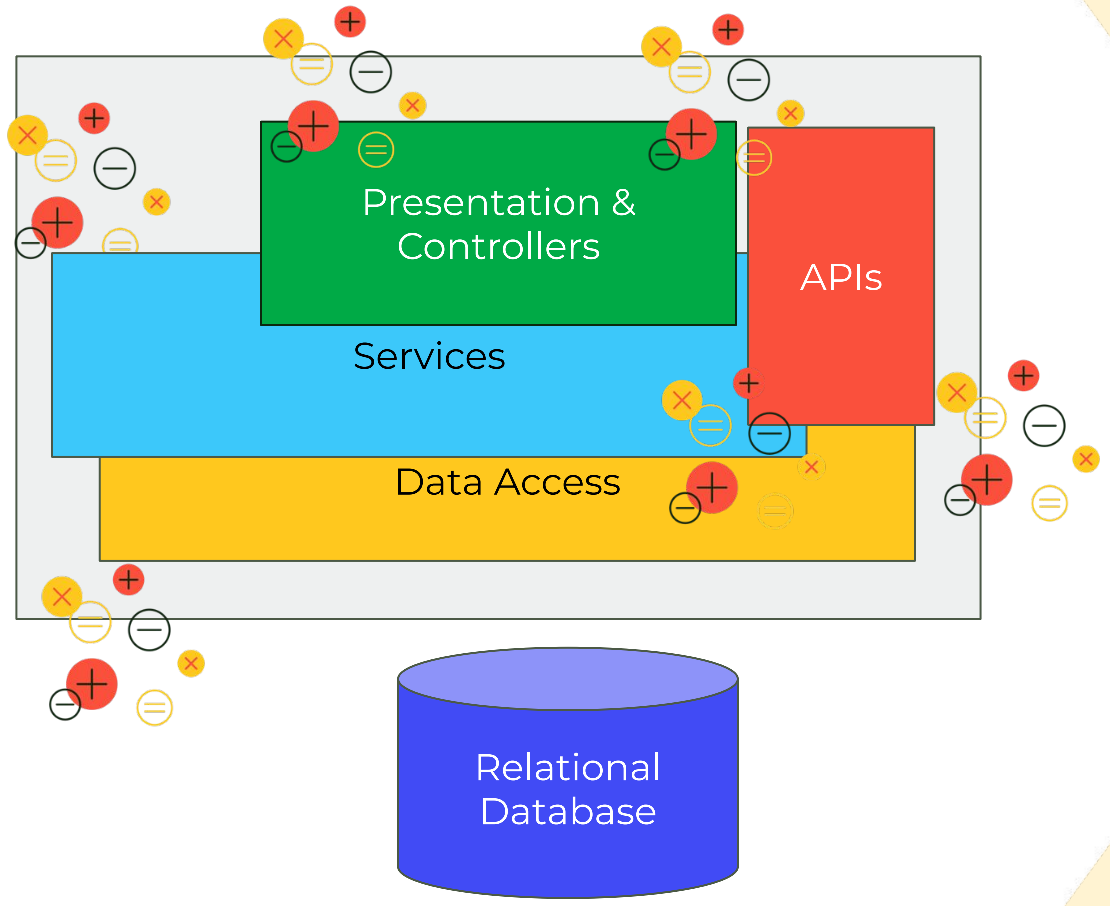
For years, the architecture held up.

- Remained stable
- Existing frameworks supported most use cases
- Scaled horizontally with minimal changes
- Easy to explain to new developers (who could visualise the changes they made)
- Simple stack with few moving parts
- Rapid development for experienced developers

With a few caveats

- Stateful back end
- Sticky sessions and unhealthy instances
- Lengthy upgrades to libraries etc

Monolith became too big



Over the course of time however, the architecture started to become unsustainable.

- Frameworks became more tightly coupled
- The encapsulation started to bleed
- Application startup time started to hit 5+ minutes
- Learning curve of the system started to balloon out (4-6 month lead time to get a sound understanding of all the systems)
- Earlier iterations of frameworks started to grow into significant technical debt

What broke us however, was the database…

Database connections and us storing Large Objects (LOBs) in there provided a bottleneck we couldn’t overcome with our current architecture.

The start of our transformation

Being a product based company, a significant change to architecture doesn’t happen without approvals.

Compounding this, our product was hosted (on premise) by our clients.

Switching to a microservices architecture would also mean making our product no longer compatible with our clients on premise infrastructure.

Fork in the road

Two possible options:

1. Axelerator 2.0 (cloud native)
   a. Full re-write of the platform
   b. Clients would need to move onto it
   c. No data migration pathway
   d. Min 3 years to achieve feature parity

2. Axelerator as SaaS / Managed Service
   a. Uplift clients onto cloud managed service (i.e. EOL on-prem installations)
   b. Data migration pathway
   c. Client configuration maintained
   d. Hollow out monolith with tactical microservices

Path Chosen

Pathway 2 was chosen.

Mainly because:

- We didn't have 3 years to build feature parity
- Our clients had significant investment in their current systems which they wanted to retain (if we wanted to retain them as clients)
- Iterative Improvement vs Hard Reset

axe

Masters of our own Destiny

Having made the decision to be a managed service, we were now masters of our own destiny and can use any technology we saw fit.

With that, we had a stocktake:

- Only support one DB - PostgreSQL - removed MS SQL Server, Oracle, DB2, MySQL
- Only support one application server - JBoss/Wildfly - removed WebSphere & WebLogic
- Progressive adoption of established tools

Since we weren't going Cloud Native, we also needed to build to run anywhere (i.e. AWS, Azure, GCP, local machines etc.)

This was especially true due to Black Swan redundancies (i.e. only 1 zone in AU for AWS meant we needed another cloud provider if that one was compromised)

Microservices Readiness: Containerisation / Docker

Our monolithic deployment architecture was quite simple:

- Web / HTTP Server (e.g. Apache HTTPD, IHS, IIS etc)
- Application Server
- Database

All tended to run as services on the operating system, with conf files.

With microservices, deployments need to be more nimble, and support infrastructure as code.

We chose Docker as our containerisation provider.

The Evolution of our Stack

JVM	Java 17 → Java 21	
Caching	redis	
Message Broker	RabbitMQ	
Language	Java	
Microservice Framework	spring boot + Spring Cloud	
Container	docker	
Container Manager (local devs)	docker desktop → minikube	*POC stage
Container Manager (cloud)	AWS ECS → kubernetes	

The Evolution of our Stack cont.

Container Visualisation & Monitoring	argo
APM, RUM & Observability	elastic → DATADOG
Container Base Image	alpine Linux → ORACLE LINUX → Amazon Linux
Security Endpoint Protection	ARMOR → CROWDSTRIKE
WAF	CLOUDFLARE

Small Beginnings

We targeted our first release to introduce one new microservice and containerised deployments via Docker.

Goal

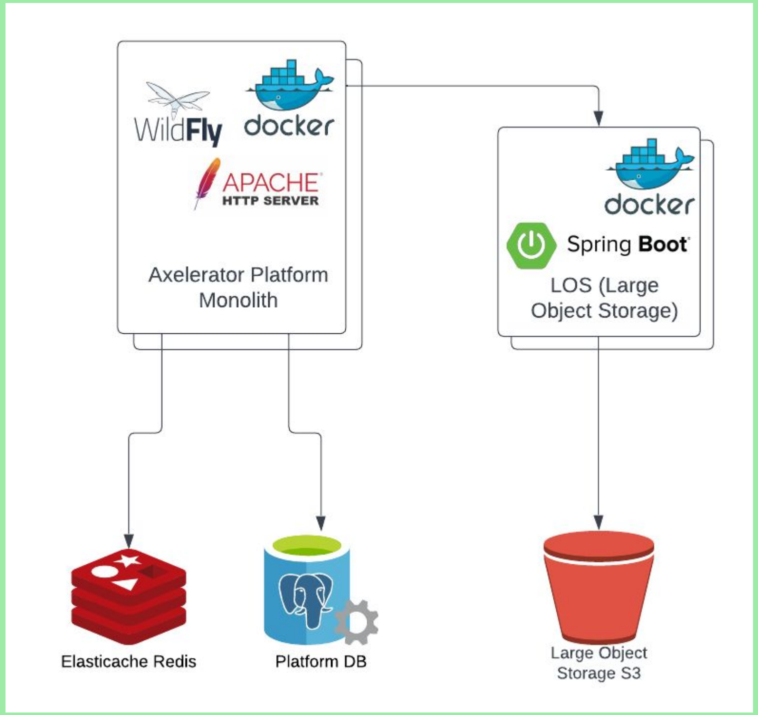
- To move large objects currently stored in DB to remove current bottleneck on performance.
- Introduce containerisation for deployments & developer familiarity

Artefacts

- New Large Object Storage microservice
  - Supports various providers like AWS S3, GCP Google Storage, plus filesystem
- New caching provider Redis replacing some http session state currently stored within the database

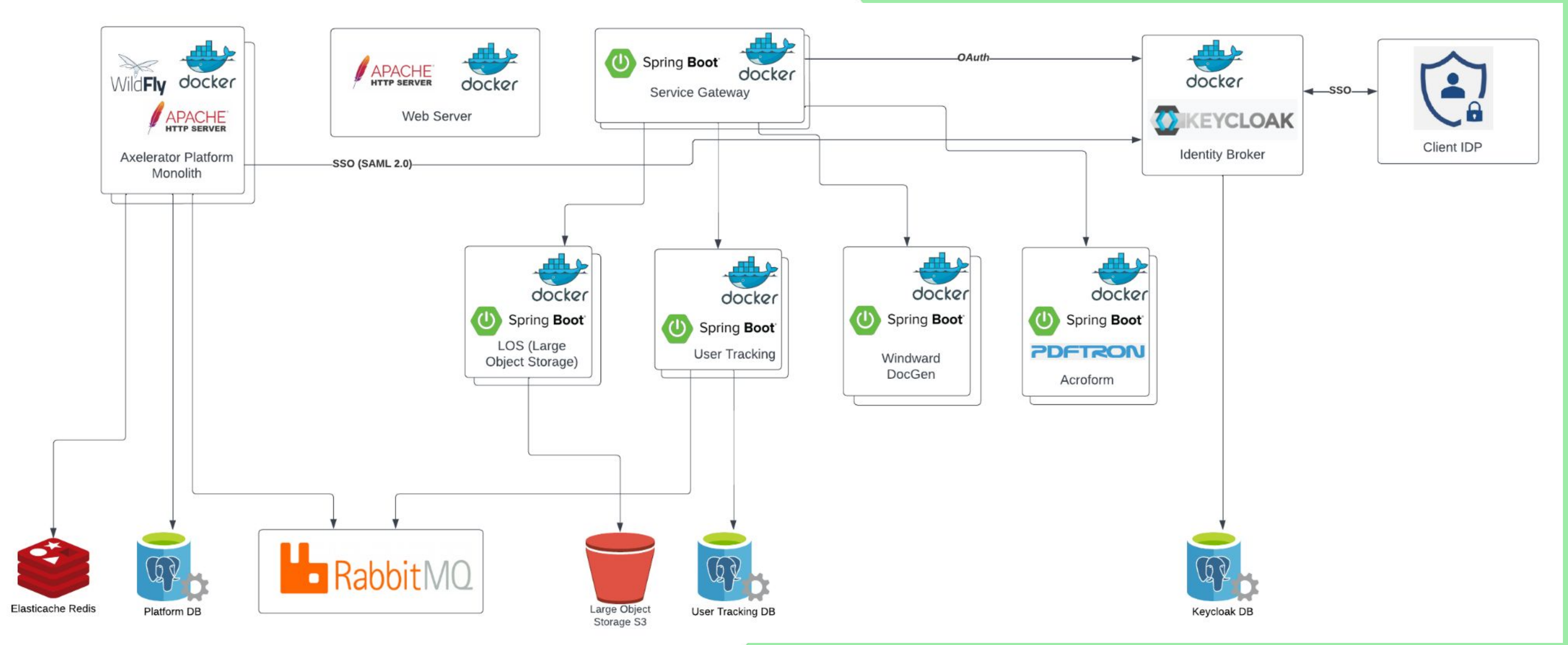
Outcome

- 30% increased performance (endurance test)
- 250% increased capacity with same database provisioning (stress test)
- Developers familiar with running Docker Desktop & Docker Compose



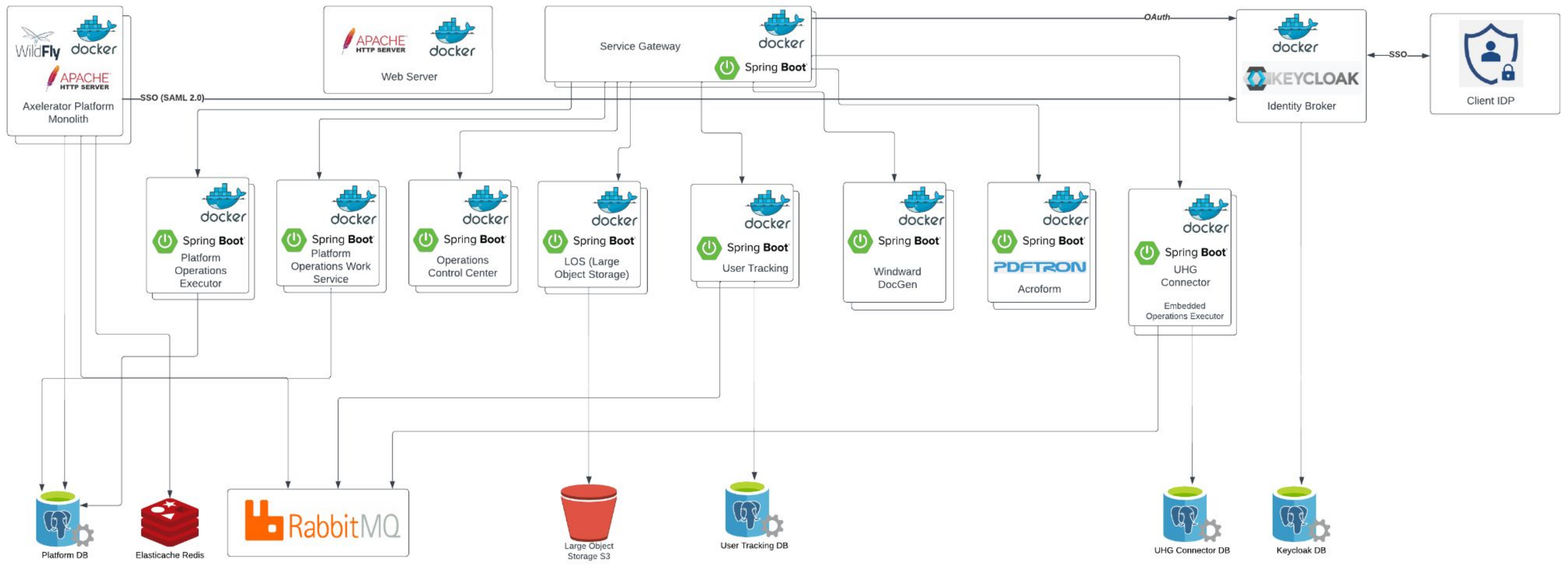
Floodgates Open

With our success of first steps into microservices, we started executing more event-driven architecture, eventual consistency and distributed computing.

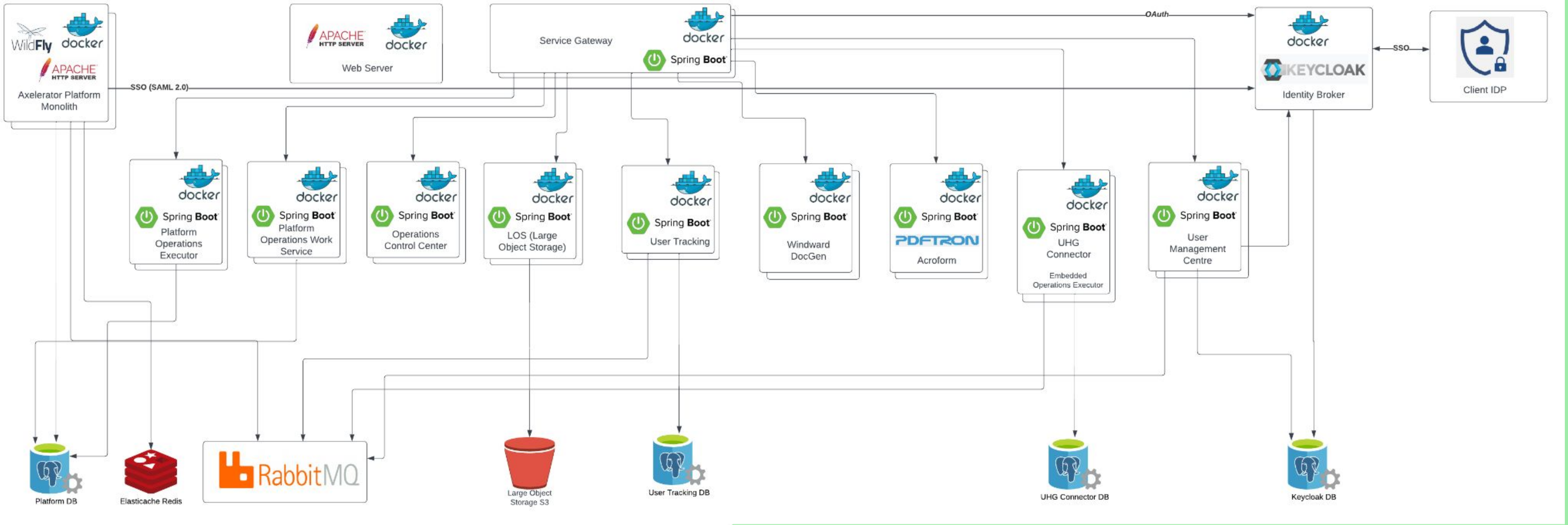


More containers

And continued with anything new we were adding would be built using microservices.



And more...



And there are even more and more now, with more on the way.

Retrospectively...

We built a Distributed Monolith more than a purist Microservices Architecture.

Could of we avoided this? Probably not, since the monolith contained so much, and we needed to solve for a particular problem - our performance and scalability.

Monolith vs Distributed Monolith vs Microservices.

There is no definitive right or wrong way to build systems. All have pros and cons.

The monolith is being hollowed out slowly and eventually the end result will be more aligned to microservices.

We're using Strangler Fig pattern to slowly eliminate the exposure of the monolith.

What we knew going in

We researched what was involved in building (and servicing) microservices so as to not go in blind and at least know what we were committing ourselves to.

What we knew we'd have to do

- Update our deployment architecture to be containerised
- Enable distributed logging & tracing
- Critical service / startup sequencing
- Configuration Service & management
- Hook up to an APM for each service
- Train developers to use Docker Desktop

The Reality

As developers, we tend to estimate on the side of everything-going-according-to-plan.

Obstacles we had to overcome

- Configuration debt
- Port conflicts
- Hostnames & Routes
- Container Reservations
- Health Checks (and their scope)
- Growing system weight
- Persistence Management & RPO
- Keeping on top of upgrades…
- Roles and responsibilities between Devs and DevOps
- Feedback loop between Devs and DevOps
- Cloud maturity
- Container level security - tools
- Cost control

Configuration Debt

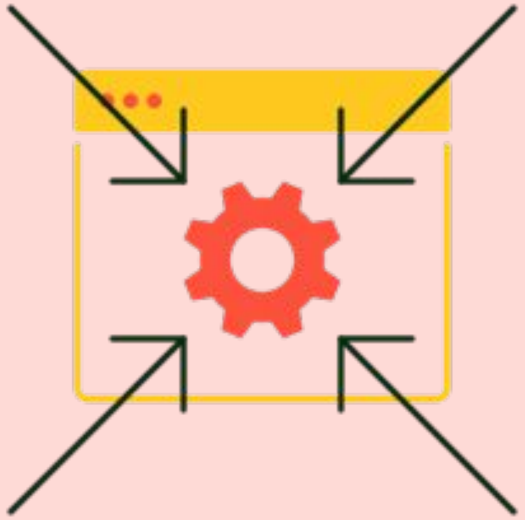
One of the greatest draw cards to microservices is the ability to have small, focused teams, owning the microservice.

I.e. they are self contained.

Developers are not the best at:

- Naming configuration variables
- Documenting / maintaining variables (especially when they are redundant)
- Preventing duplication
- Optional vs mandatory variables
- Default if not supplied variables

All of these factors lead to configuration debt whereby the cost of DevOps mis-configuration and debugging



Port Conflicts, Hostnames, and Routes

As you deploy more and more microservices, they all need to listen on a particular port, and sometimes talk to each other.

As you add to your suite of microservices, the ports they bind to, as well as hostnames, clusters.

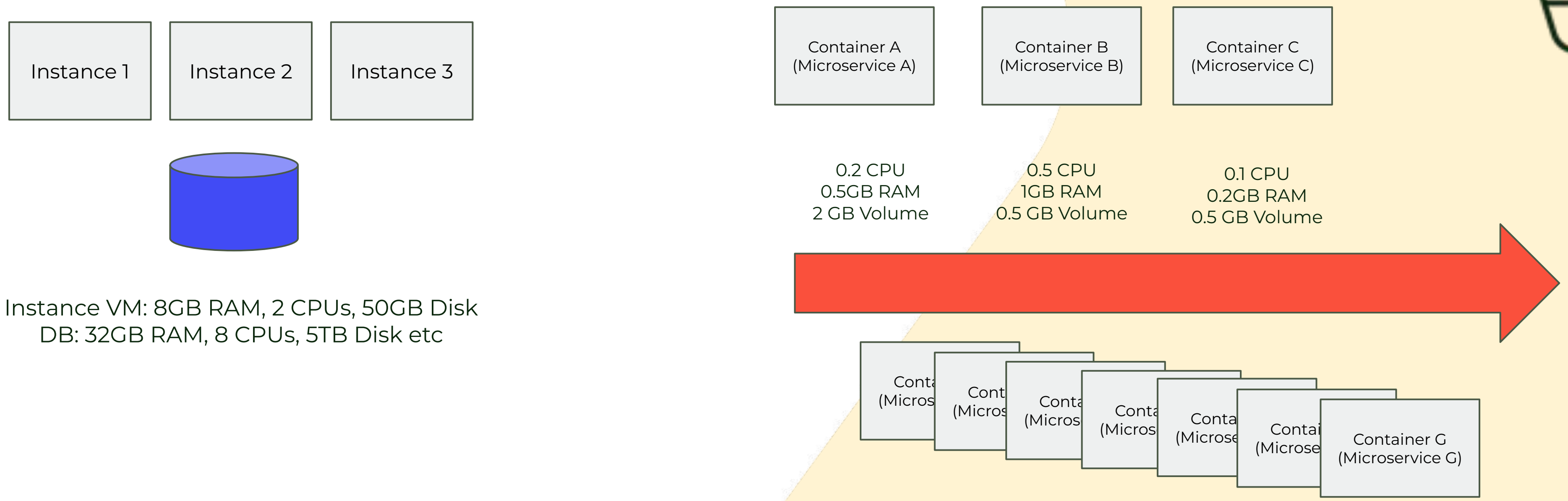
When dealing with various container deployment technologies (Docker Desktop, ECS, Kubernetes etc.), visibility and scope can come into play. I.e. localhost may work for development environments, but within Kube it requires container name / alias to resolve.

Container Reservations & Growing System Weight

CPU limits (min & max), Memory limits (min & max), Disk Space & Volumes etc.

Monolithic

Microservices



Instance VM: 8GB RAM, 2 CPUs, 50GB Disk
DB: 32GB RAM, 8 CPUs, 5TB Disk etc

Each container has its own operating system & JVM, resulting in greater size.

Costs on cloud increase.

Health Checks (and their scope)

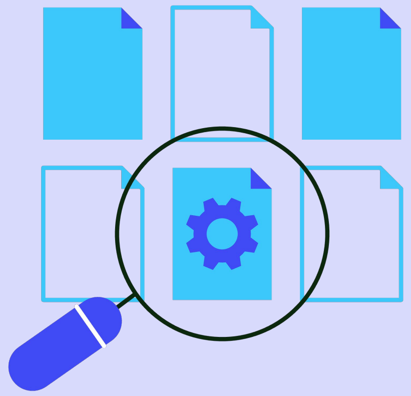
Health checks, liveness probes, readiness check, all factor into containers being marked ready to receive traffic, and also when to destroy.

Our teams wrestled with what constituted a healthy or unhealthy instance.

Is a service which is running, however cannot successfully process a request due to dependencies / resources considered a healthy instance? If yes, what is the line?

Misconfiguration of environment variables often led to developers often defensively coding health checks to check critical service access, but by solved this problem, introduced other problems with instances infinitely being destroyed and recreated.

Our eventual line we settled on was we'd check we can connect to critical infrastructure we controlled, but external resources we allowed to fail at runtime.



Persistence Management & RPO

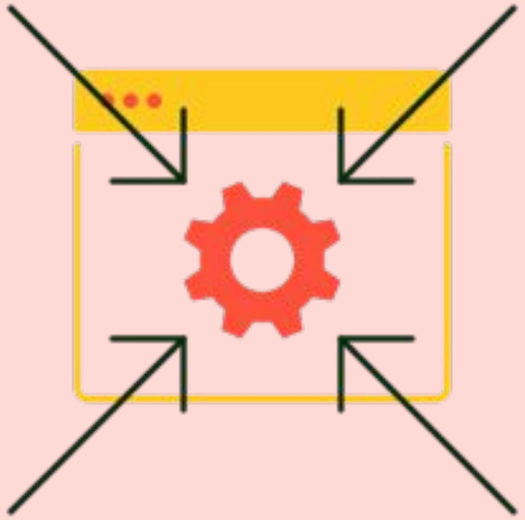
With microservices and distributed computing, comes eventual consistency.

I.e. multiple sources of persistence, owned individually by the microservice.

This then introduces the technical challenge surrounding a particular persistence being corrupted, and how to implement a RPO (Recovery Point Objective).

This, in combination with any regulatory requirements for data retention requires a fine balance between bullet proof data resiliency (high cost) vs minor data inconsistency & accepted tolerance (lower cost).

Use cases definitely factor into such things including financial transactions, point-of-no-return status changes etc.



Keeping on top of upgrades...

Microservices do tend to lead to more technology exploration. Developers as we know are rather curious to try new technologies.

That brings with it the maintenance overheads of keeping these all up to date.

Technologies are moving much faster than they were 10 years ago, and open source projects tend to be EOL'd within 12 months, with new versions replacing them.

Add to this, DevOps is also moving fast (if not more so) with their own EOL of versions.

Reserving team capacity to always be keeping on top of upgrades to spread the pain, even if the microservice isn't being functionally improved, minimises the build up where a vulnerability in your third party libraries leads you to having to drop everything (for months) and scramble to upgrade.

Devs and DevOps

It’s expected with a team newly adopting microservices, that interaction with your infrastructure / devops team will greatly increase.

What really can help is to establish clear roles and responsibilities, and a clear lines of communication and feedback loop for continual improvements.

Roles and Responsibilities

The biggest divide between devs and devops is the **POLP** security policy and **segregation of duties**, but for good reason.

Devs should own things like:

- What their service requires to run under load / capacity
- What configuration the service requires
- What secrets must be managed

Devops should own things like:

- How trace / debugging information is shared securely
- Reasonable documentation and deployment instructions

Feedback Loop

Alongside the roles and responsibilities, with the ever changing landscape of cloud and microservices, the Devs and DevOps need to work closely together to ensure deployments are as smooth as possible, and assist each other in understanding each others pain points.

Cloud Maturity & Security

Most likely, alongside moving to microservices, you're also deploying to a cloud provider such as AWS, Azure, GCP.

With that, developers contribute more than ever to comply to certifications such as SOC 2 & ISO 27001.

As well as this, all services should support security agents monitoring to detect if they had been compromised.

This ended up being how primary reason for using EC2 instances instead of Fargate to run our containers, as the security agent is able to shutdown compromised containers if they are running on the servers, and wouldnt be able to do this when running purely as sidecars.

Security takes highest priority, always!

Cost Control

Running things on the cloud isn't really cheap compared to bare metal servers in a data centre, however the industry has seen the value of rapid turnaround time to provision additional capacity, and cloud-native technologies which let us focus on developing things fast without having to worry about managing hardware.

Cloud costs however will creep up on you as your footprint grows, especially if you are not keeping a close watch on it.

Elements such as data transfer, IO, on managed services are tricky to predict.

DevOps Investment

DevOps Uplift

In order to successfully manage a system built with microservices, you must invest in DevOps.

Although not specific to microservices, hosting in the cloud you need to allocate significant resources into securing and configuring it.

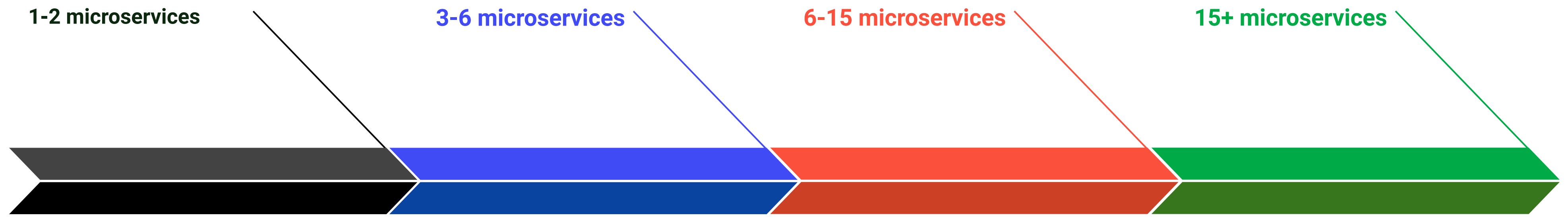
Big ticket items include:

- Ensuring AWS config get a pass in security scans based on the NIST framework
- Having a compliance framework to ensure we are aware of any changes, misconfigurations and rotation of passwords/keys.

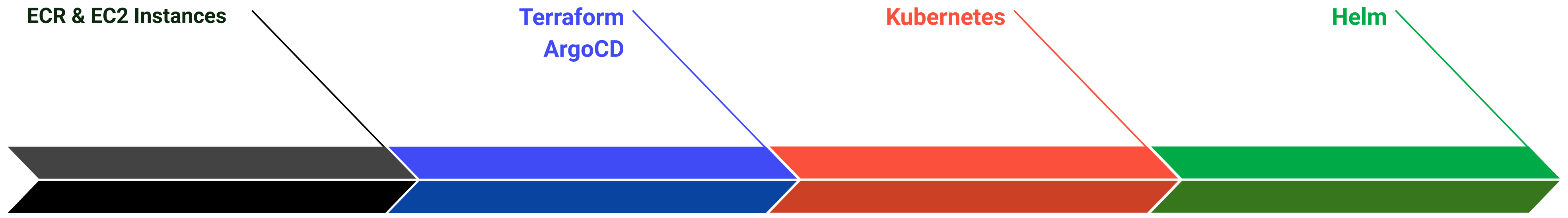
We partner with an authorised cloud services company RackSpace to assist in securing our cloud infrastructure.

Parallel Streams - Microservice & DevOps

Dev Teams



DevOps Team



Thank you for your time