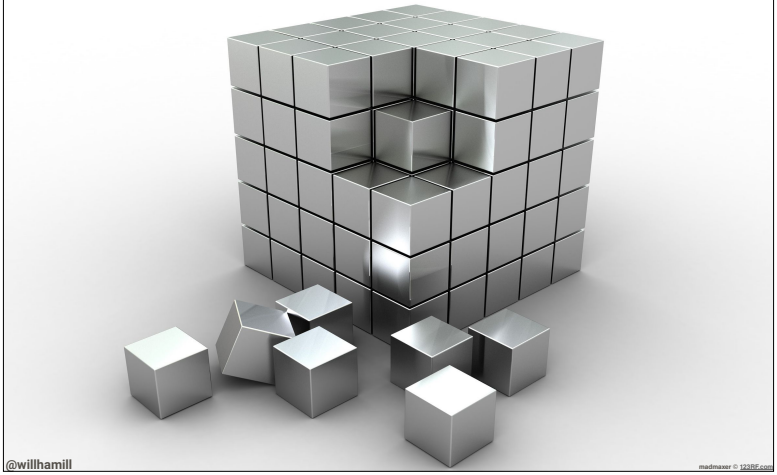
Splitting the Monolith



Who are ye?

@willhamill

Solution Architect @KainosSoftware

Worked in utilities, insurance
and public sector

Why split a monolith?

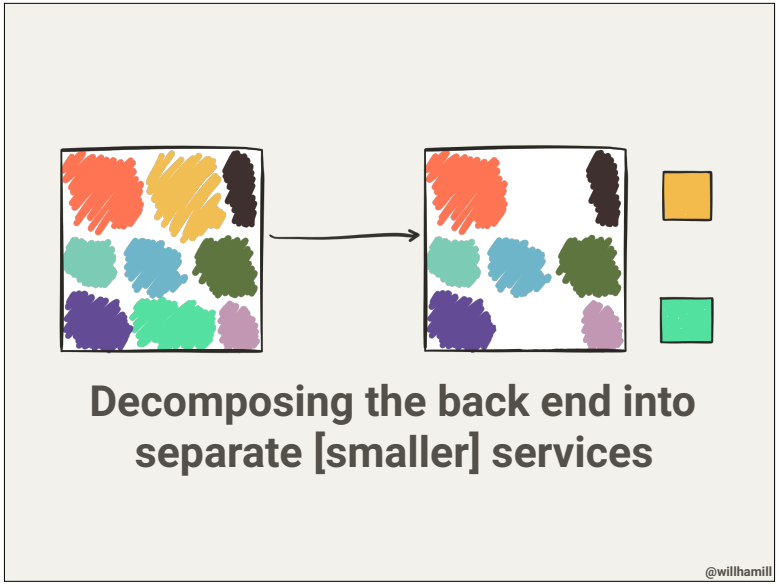
Maintenance harder than necessary, app developed to tight deadline, now paying off technical debt in most complex areas
Scaling is deploying another monolith
Performance is important and costly to increase because of scaling

The backend is known as "The API" and it does everything in one PHP project, with some areas defined for different interfaces corresponding to different behaviours

Deployment on AWS is 36 instances of one of the biggest node types that they can provide. Check out www.ec2instances.info for info on node costs. It costs a lot and has a lot of headroom in some places.

Have avoided using the term internally to sidestep hype and backlash

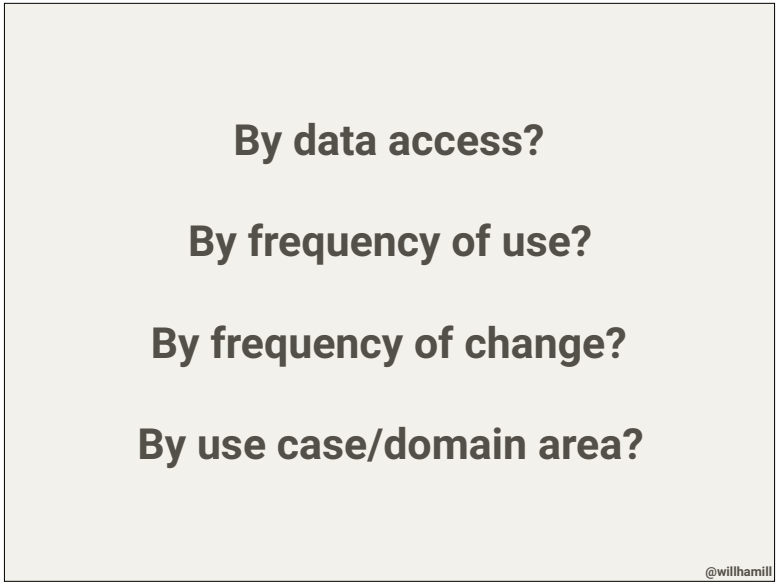
Monolith is a bit of a loaded term now. Only a few years ago it was just called "MVC application".



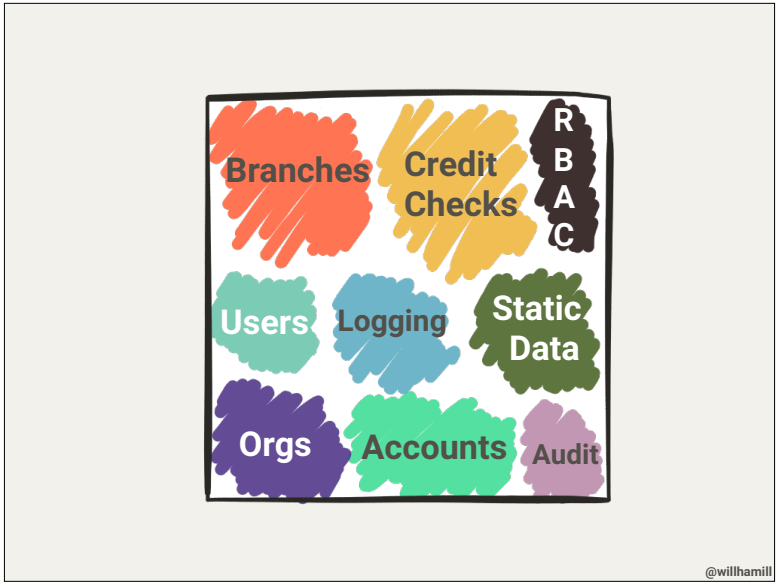
So we're trying to take this monolith and analyse its modules and decompose the most relevant parts into smaller separate services
Not near microservice hell/nirvana yet, no thousand-service distributed systems at this point

Martin Fowler has a good blog post, google it, about micro service prerequisites. He says "you must be this tall to use microservices" and lays out a requirement for rapid provisioning, good monitoring and fast automated deployment.
We've got these things, so we're tall enough to ride, but we're not rushing.

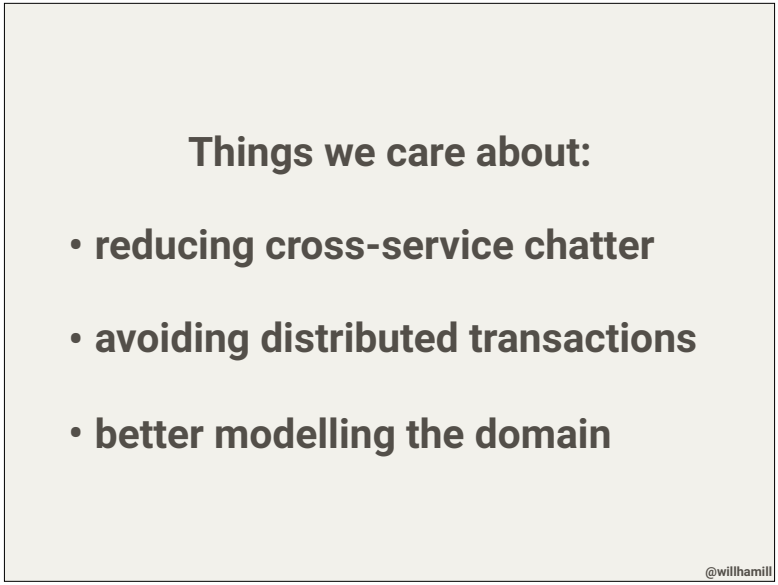
We've got a backend application that is not hugely well understood or documented, but how do we decide we want to split it up?



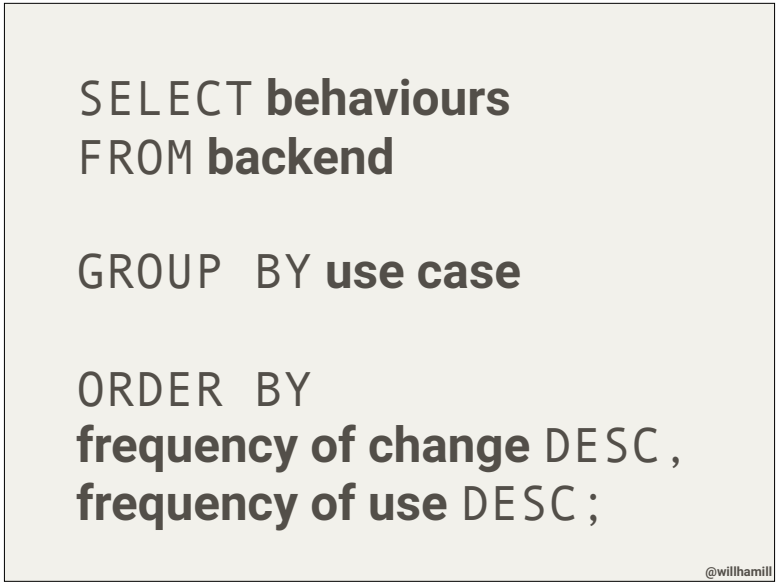
Well, firstly we need to do some analysis on it to help make the decision. Then we can think about a few different ways to carve up the turkey.



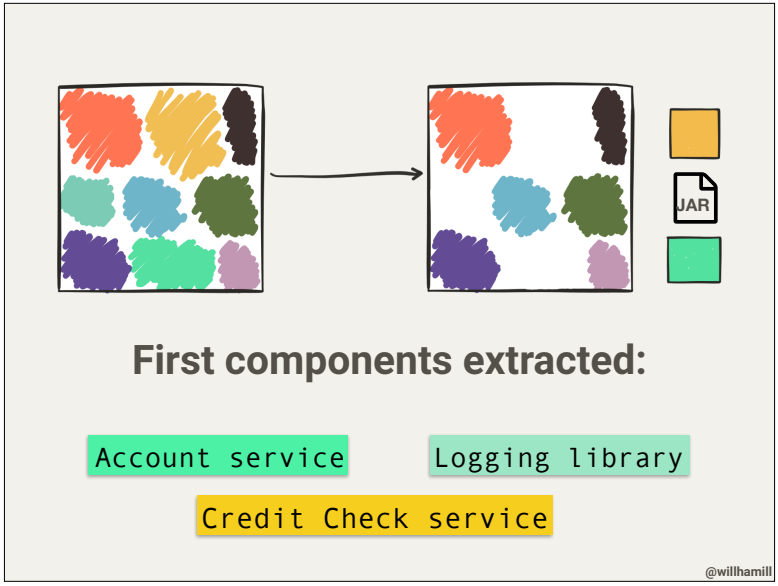
The modules inside the backend API are mostly organised around different functional areas
(this example is translated from real system)



So for us it makes most sense to try and group around use cases rather than data access, as we're not just exposing CRUD behaviours to other consumers.
We want to not just avoid more HTTP calls into the mix and we want to avoid transactions spanning service boundaries. We're not going really fine grained and having aggregator services, for these reasons.
If transactions need to span service boundaries your service boundaries are probably wrong.

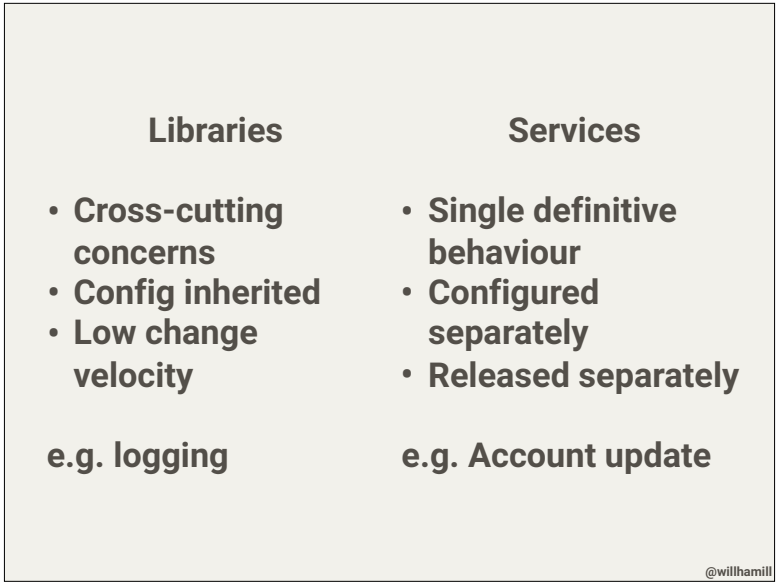


So we group by use case, or domain context, and from those areas in the application we pick out the ones which are going to change the most - as these will benefit from improved maintainability the most, and narrow by the ones that are hit the most at the backend - to get performance gain



Based on that analysis we picked out the credit checks and accounts modules into services, and the logging and authentication actions as libraries

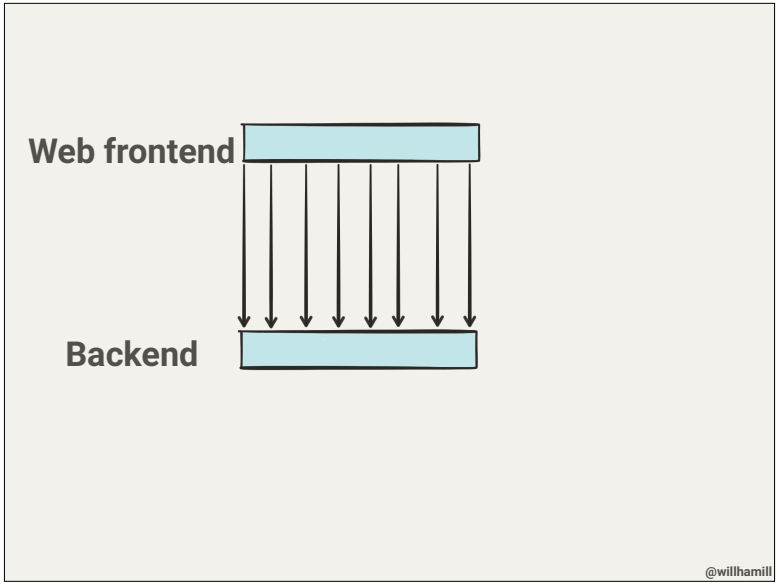
API design principles to guide service development as REST APIs & Swagger for live hosted interface docs



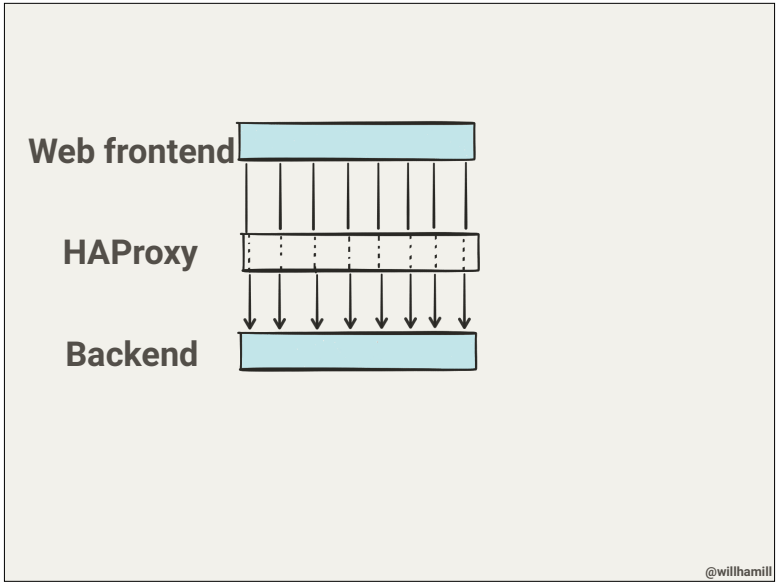
Libraries for things like log formatters or authentication filters, things that don't have internal state. Services for things like RBAC checking or account creation, things that should have a single canonical representation in the system and that can manage their own state or connection pools or the likes.

So now that we've identified which areas we want to replace how do we go about implementing it without just taking the old system offline for four months while we finish the new services?

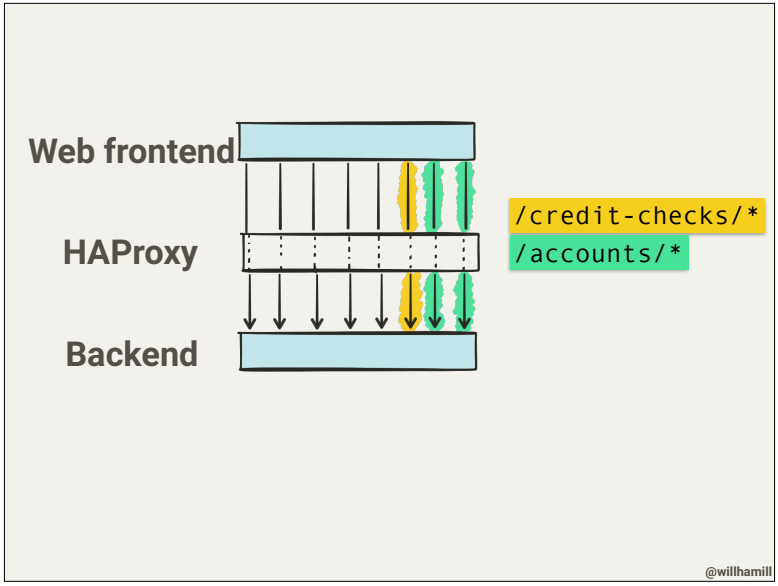
Martin Fowler has a wiki page on the strangler pattern, check it out.



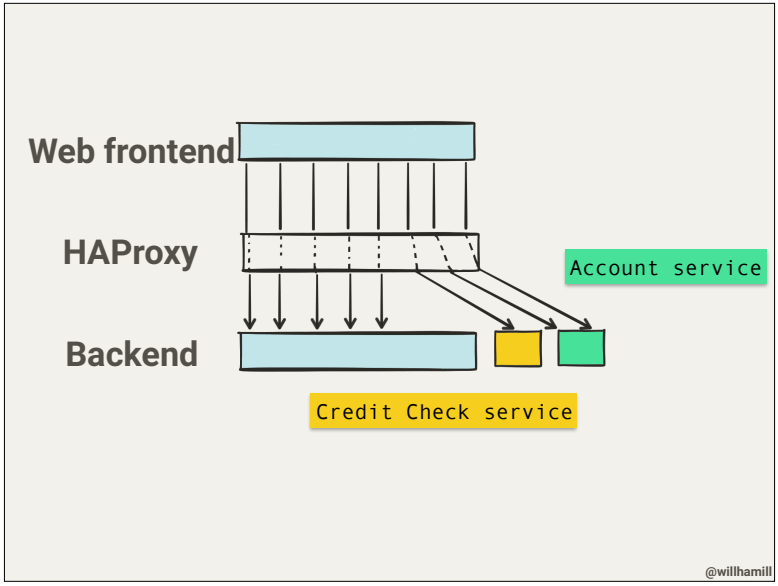
Like any self-respecting multi-tier architecture, we've got a frontend app for composing data from various actions and presenting it to users, and a backend app that provides all of the functionality the frontend needs across a few dozen HTTP interfaces.



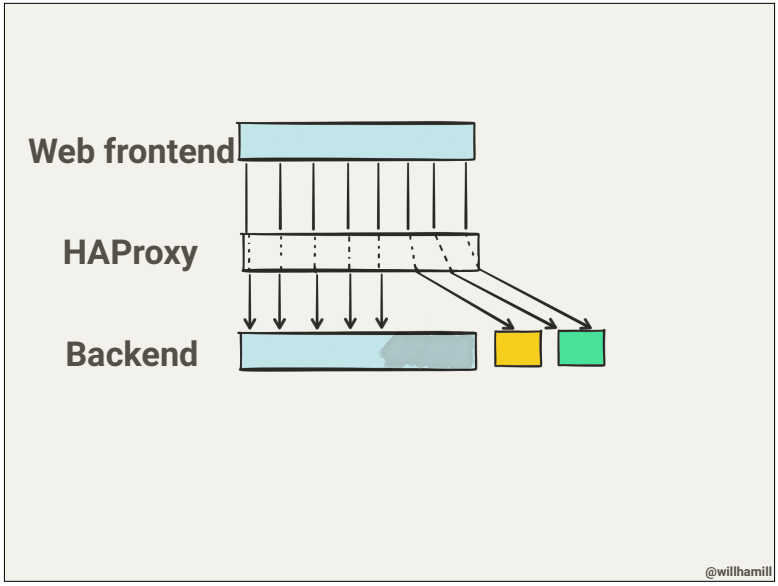
HAProxy is in the way already because of healthcheck based load balancing, so lets make it just a tiny bit smart so it can help us re-route some of the requests



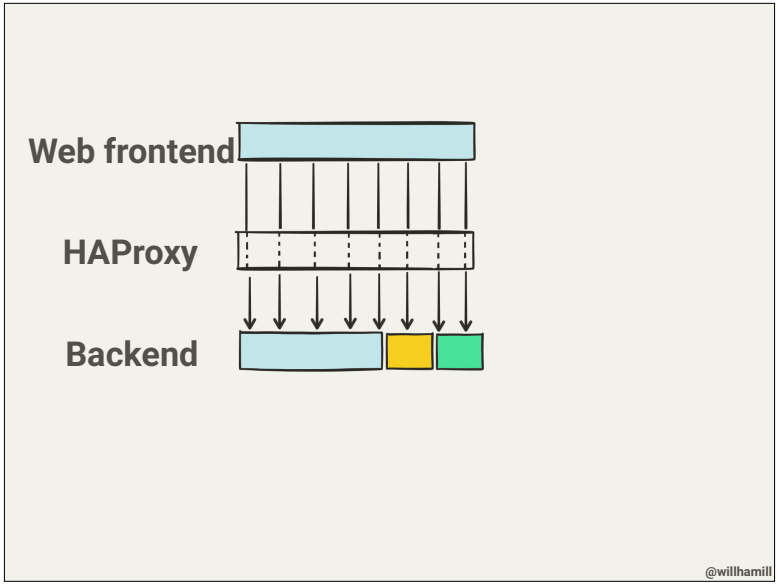
Having better defined APIs helps with this by making the route matching simpler.
All the credit-check path requests go to the credit check service, all the accounts path requests go to the accounts service.



Direct these routes to the new services and stop calling the backend for those behaviours

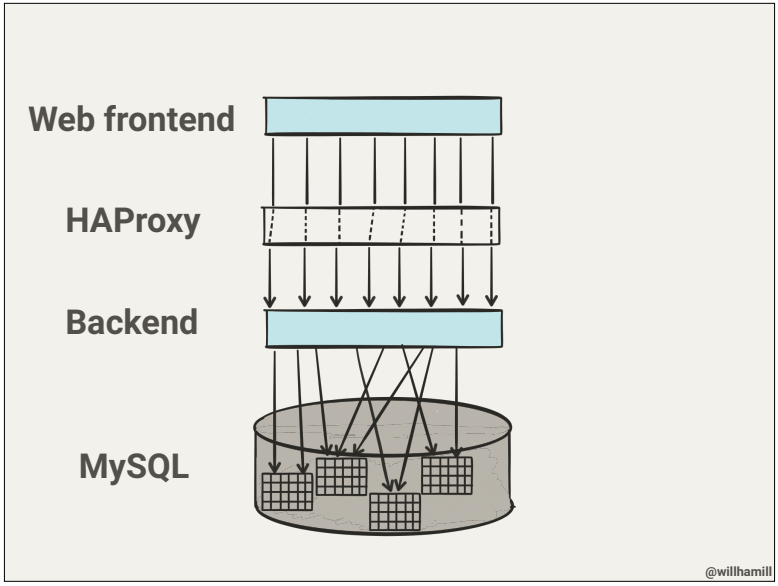


Fire up the tests, prove it all still works, then we can identify parts of the backend API that aren't being touched any more.
As the old parts of the app are no longer used, we remove the code and the tests.

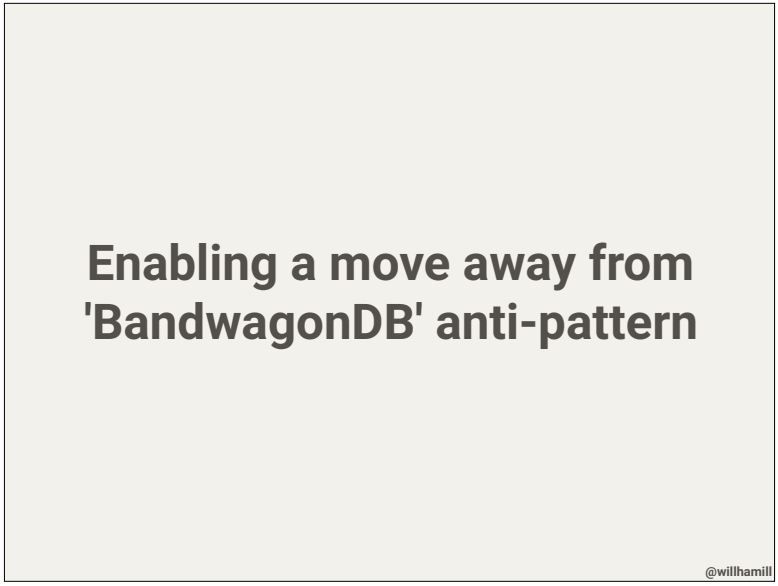


And then hopefully there's just one place to go look when someone has to go maintain the credit checking functionality or extend the account management search feature.

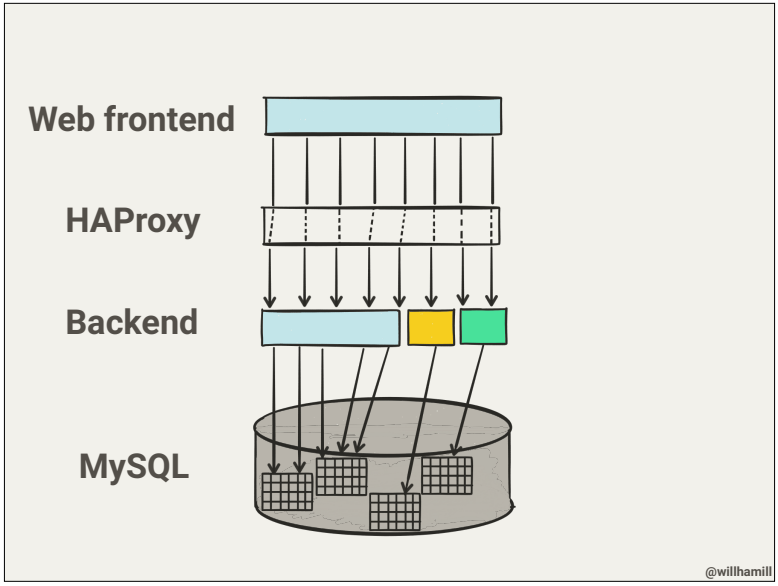
What about the database?



If only it were that simple. We also have a lot of areas in the app touching a lot of different parts of the database.



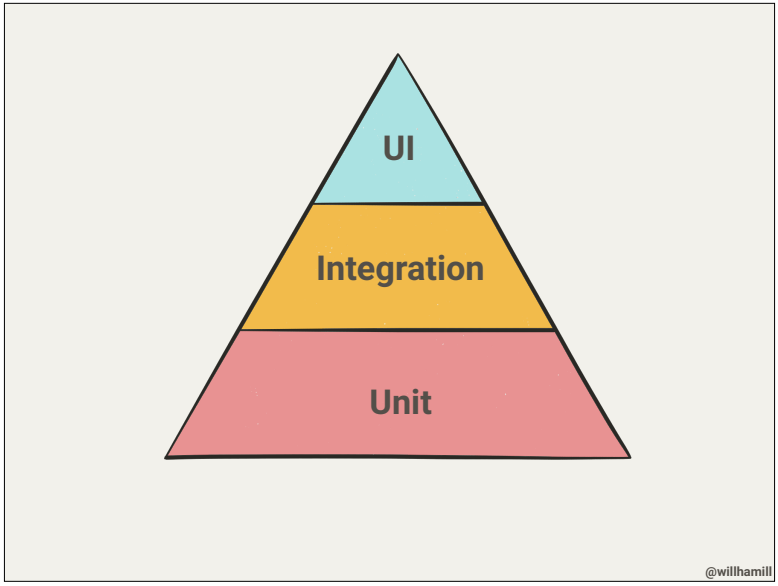
So part of the changes we need to make are to make sure that when the frontend wants data from those areas it goes to the new services to get it.



Sharing a single database is often considered an antipattern, and means your releases tend to be coupled together as the services are linked by shared data. What we're working towards here is the ability to split the database up now.

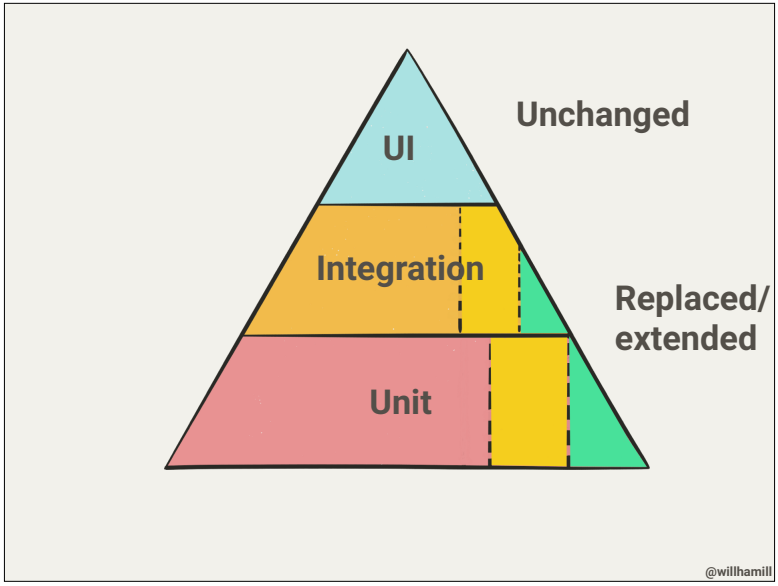
Would like to further enforce this, until DB is split, by changing application user permissions so backend app doesn't have permission to read/modify credit check or account tables, to prevent sneaky DB integration.

Testing as a safety net



Reasonably good coverage in most areas of the application, and lots of UI tests.

UI test suite pre-existing in Selenium was very valuable to us as a de-facto regression test suite.



As we went along we replaced & extended the tests corresponding to the refactored areas.
Existing integration tests were migrated where scenarios considered valuable, and extended to cover new scenarios.
Unit test coverage definitely better now, and we also added further performance tests into our JMeter suite to cover the new API interfaces that we had developed.

Creating RESTful services in Java

@willhamill

Our platform of choice is the JVM so we picked these tools to help us develop these services.
We liked most of the principles from 12 factor app approach and that helped guide our implementation of stateless web services.
Dropwizard has a funny name but is a seriously good set of libraries, and we added Metrics in to our apps to help measure everything.

Deploying onto AWS

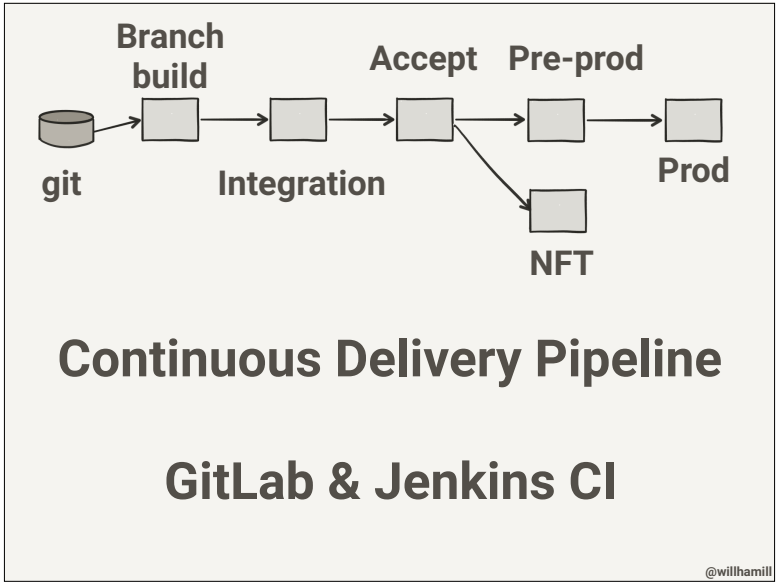
DevOps in practice:

Ops people embedded in dev team

Automation of everything after code

Measuring anything that runs

Sharing responsibilities



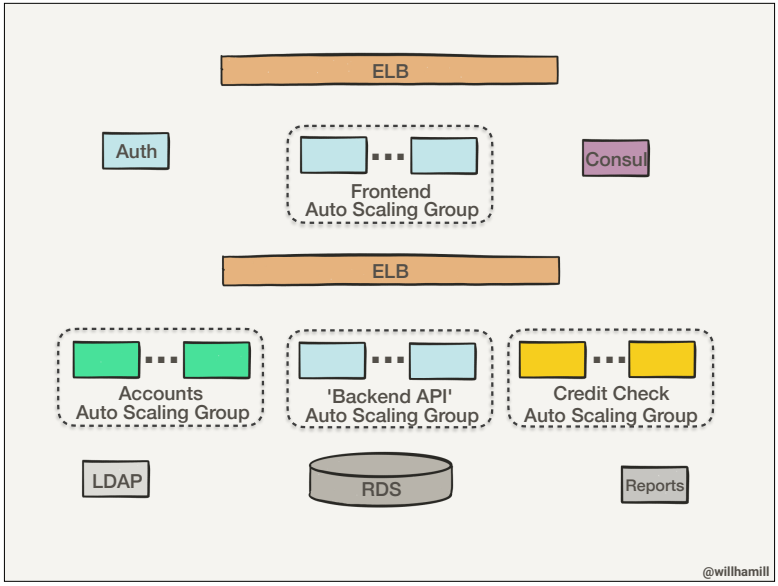
Fully automated build, test & deploy
On-demand deployment to Production
We deploy to Production after every two-week sprint

Extensive investment in automation

No unicorns

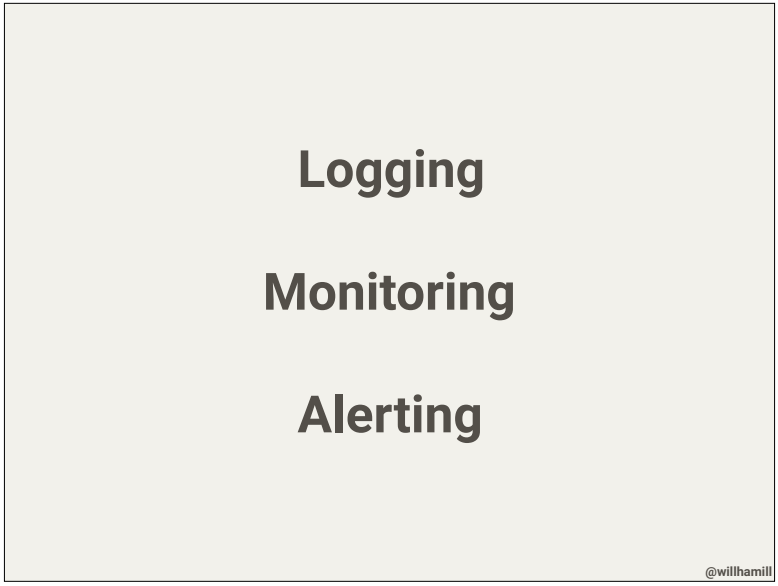
We use Terraform, Puppet & Hiera to create and configure environments

Spin up a new environment in <1hr

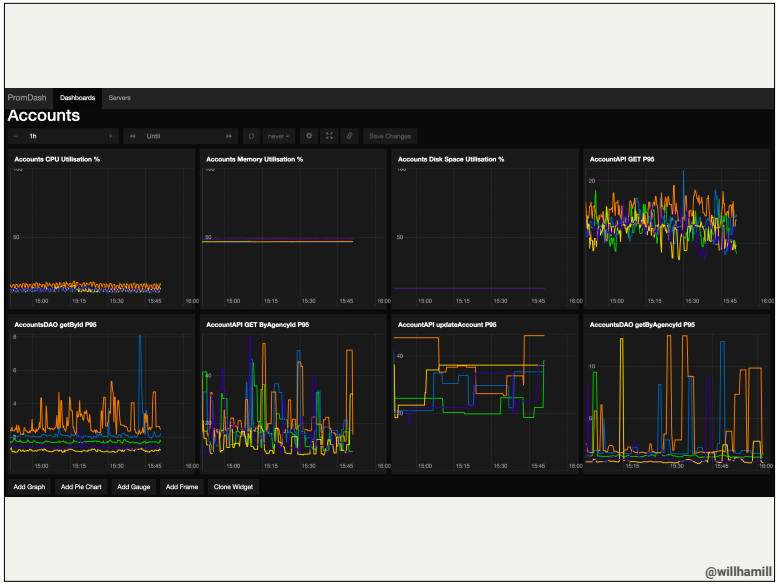


We use quite a few AWS services, only a couple of which I've shown here: All the nodes run on EC2 images and we're using things like ELB, Elasticache, RDS, Route53, S3, CloudWatch, VPC virtual network zones Autoscaling groups handle changes in load with changes to infrastructure, and can self-heal when an individual node goes down, building in antifragility.

Making sure it works



Existing platform infrastructure provided a lot of this for us, but we also wanted to be sure we're improving these areas as we go. Elasticsearch, Rsyslog and Kibana for log aggregation and graphing certain types of events or things in the logs like errors and response times. Prometheus for active monitoring, healthcheck and timing dashboards. Alerts based on certain conditions in infra monitoring like low disk space on authentication server.

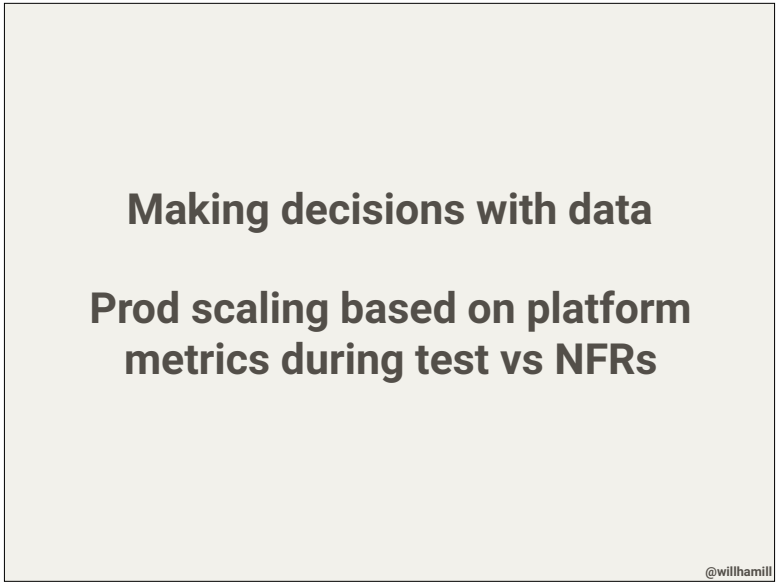


The metrics we added to the new services with the Yammer Metrics library can be hosted as an app endpoint or exposed via JMX. We really found these useful for identifying timings in different aspects of the app which help us make decisions about how the services are performing. We created Prometheus dashboards for the infra monitoring aspects and the timings.

`production_monitoring.gif`

@willhamill

Beware the signal to noise ratio.
Only alert things about which you expect people to actually stop and do something.
When people are setting up email inbox rules, we should probably change the alerting conditions.
Low-priority notices should go into logs where we can find them later or graph the rate of incidence.

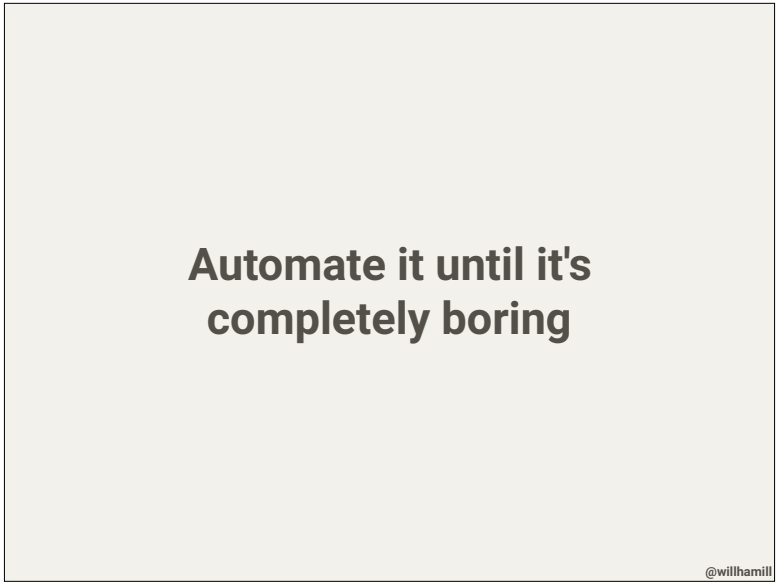


Production scaling not based on finger in the air guess from an architect or "add another one and see if it stops falling over"
Production scaling based on what we need to get main-journey throughput to target levels from NFRs and with some overhead
We use sets of JMeter tests for exercising main journeys, can watch API graphs during tests and change scale on next run

Lessons Learned

API design is important, even if your backend currently talks to your frontend
A RESTful API with endpoints clearly grouped around resources exposing domain behaviours helps organise internal structure
Interface describes intent of behaviours, should be obvious for people when extending it

Functional design documentation that was sent to team and never updated after implementation? No thanks.
Interface documentation? Yes please.
As-implemented documentation for complex areas? Yes please.
Need this to understand which parts to pull apart into which new services.
Hard to track down without this.

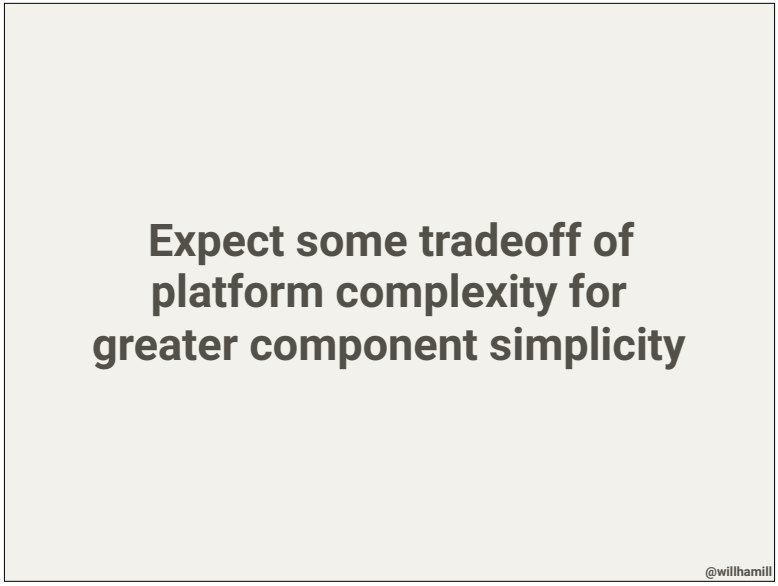


Automate anything you don't want to be doing at 11pm on a Friday night - because customer might want to do their deployments at 11pm on a Friday night when all the users are away

Automation enables fast, safe repetition.

The tools are out there, but it requires an investment.

This is a long-term win - play the long game.



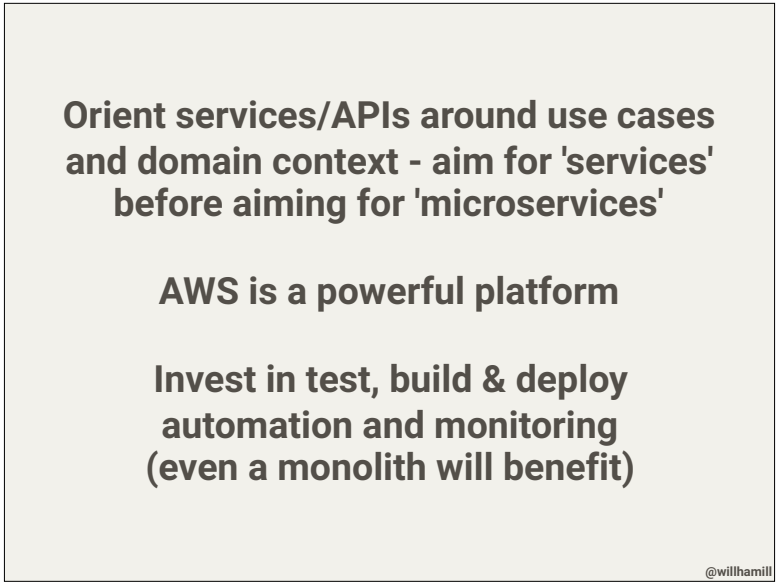
When splitting a single monolithic backend into multiple apps, you should expect that part of the tradeoff is that running the whole thing is typically more involved.

You're turning your single app into a distributed system, that's a jump in complexity.

Don't forget about dev environment setup - it's okay for AWS to run 16 different node types for all your components, but good luck throwing around 16 different VMs on a dev machine.

Think about how to automate dev environments too (we use vagrant and aggregate components onto a smaller number of VMs)

Takeaways



Strangler approach applies for general legacy system replacement

We're hiring @KainosSoftware ;)