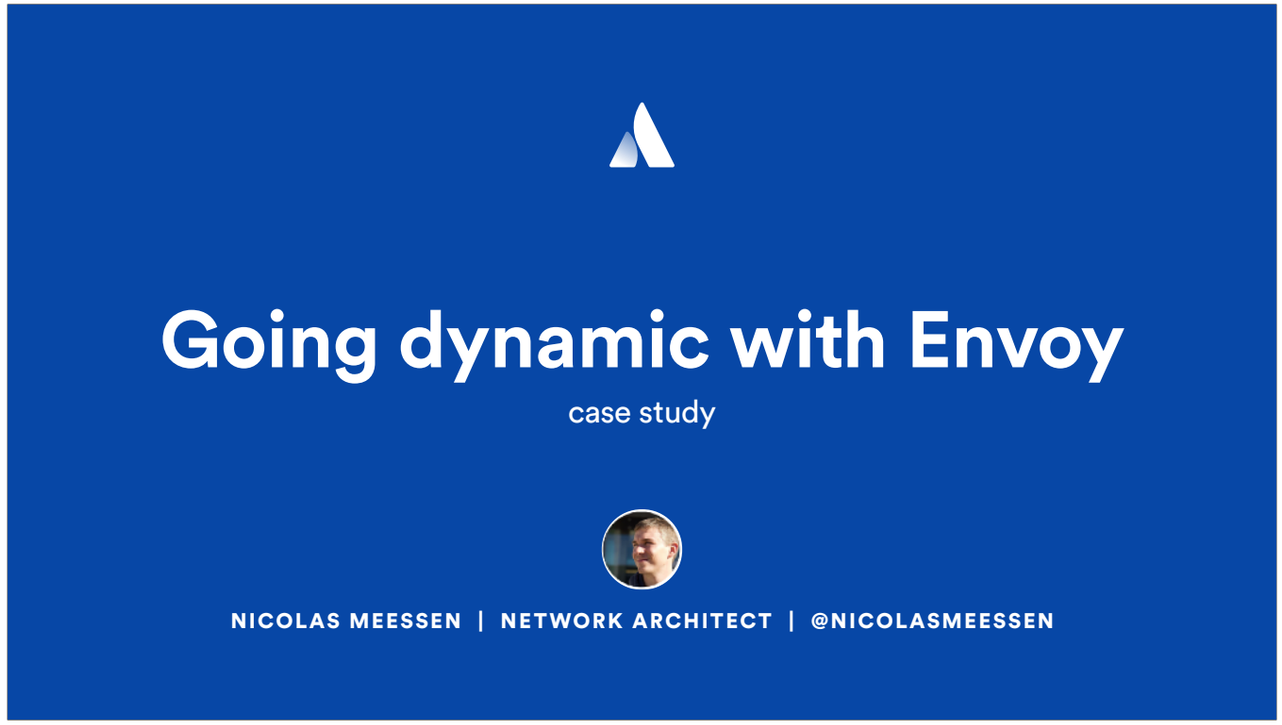
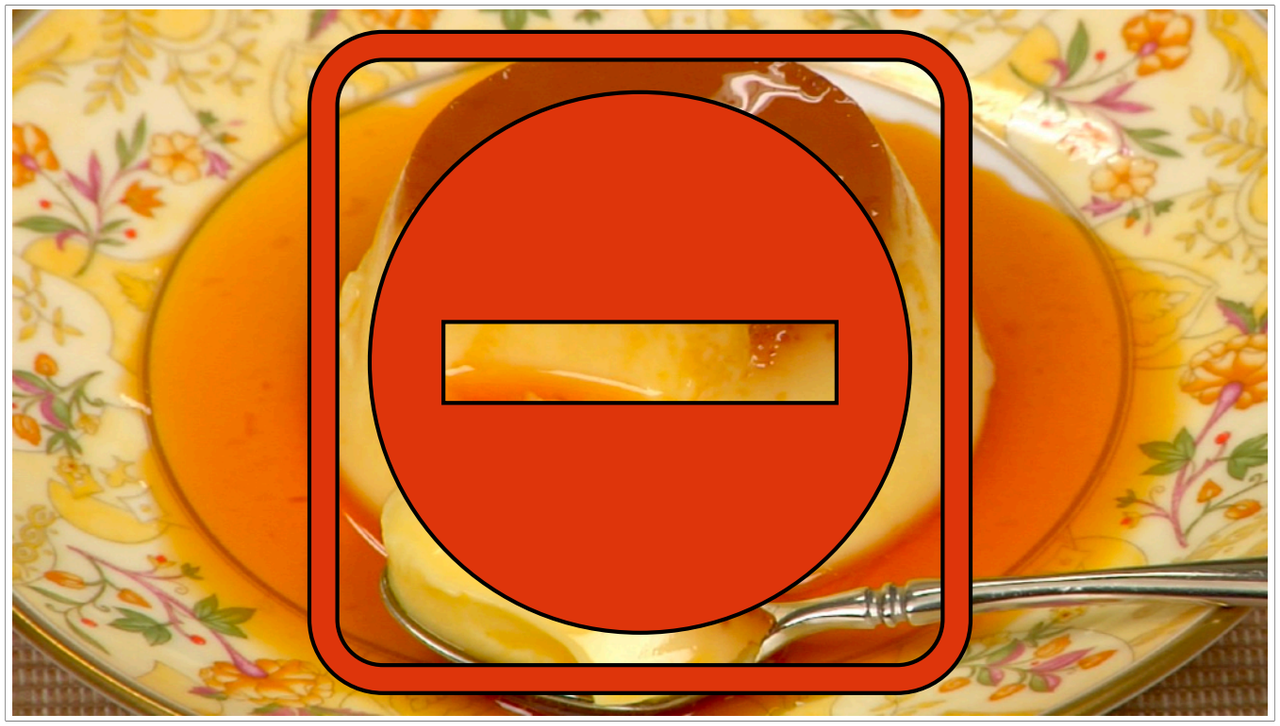

ATLASSIAN



My name is Nicolas, I’m a network architect at Atlassian and I work with our Cloud Infrastructure teams to deliver all the things that our internal developers need to have their services communicating with each other and the world securely, reliably and effectively.



There is an expressions saying that “the proof is in the pudding”… Sorry to disappoint you but there will be no pudding! Atlassian does not run Istio in production.
That expression makes no sense so as a professional I looked it up to find out (Thanks Merriam Webster) that the expression originally was “the proof of the pudding is in the eating”.

Agenda

Tasting: user stories

Ingredients: dataplane

Recipe: control-plane

So what we will do today is talk through the tasting we did to figure out what our developers internally want to get out of this pudding.

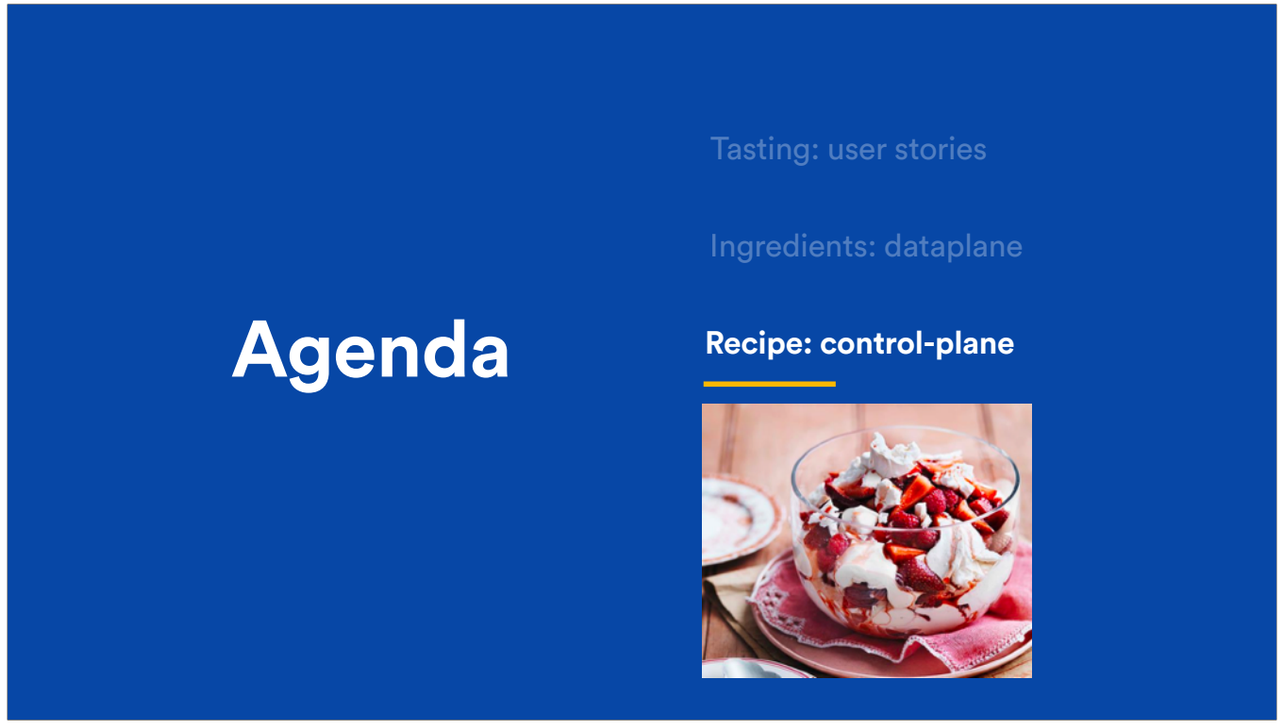
Agenda

Tasting: user stories

Ingredients: dataplane

Recipe: control-plane

Then we will cover the journey we went through to decide on our ingredients: what dataplane proxies worked best for us



And finally we will talk about how we decided on which recipe we were going to use to turn our ingredients (the proxies) into a hopefully delicious pudding by adding a control-plane. The foodies amongst you might have recognised the dessert pictured as the famous Eton Service Mess.

My hope is that by sharing what led us to decide to use Istio, it will make it easier for those of you that do not use Istio in prod today to decide if and when Istio might be right for you.

Tasting

What's the cake, the icing and the "cherry on top" for our developers?

We talked to our internal developers and all stakeholders around the company to ask them what they expected the platform to solve when it came to service-to-service communication. The main personas were the service developer (as a consumer and a provider), the service operator (we operate in a you-build-it-you-run-it model so it's generally the same people but with different needs) and the platform operator (us)
We collated all those, grouped them by teams and also categorised them in various themes: devx, cost, security, trust, ops, perf, reliability

The cake (1/2)

HTTP client/ server tuning

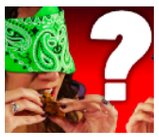
keepalives, timeouts, …

API facade

a gateway but not a gateway

Service discovery

DNS is a distributed database I don't want to know about



Authentication

who are you anyway?

Let's start with the cake: the things that developers want the most.

HTTP Client/Server tuning: reliability, performance
API facade: cost, devx
Service Discovery + dependency mapping: devx, ops, trust
Auth: security, trust

The cake (2/2)

gRPC

sometimes you want to let REST in peace

Rate-limiting

Not so fast, buddy!

Metrics

if I can't measure it, it doesn't exist

Automated failure handling

gRPC: performance, devx
Rate-limiting: reliability
Metrics: reliability, ops
Failure handling and auto-recovery: reliability (retries, health-checks, circuit-breaking)

The icing

Authz

Tenanted sharding

Cost control

cross-AZ, ALB/ELB

Fault injection

Then we have a few other things that rank as medium with our tasters:

Authz: security (it’s important but generally done in the service code, moving it out is a nice-to-have for devs, more sought after by security)
Routing and observability for tenanted shards: reliability, trust. (in order to run 100s of 1000s of instances of our products, we need to distribute our customers over separate shards or cells or partitions each of them connected to the datastore that has their stuff. Internal services don’t want to have to know which shard to talk to.)
Cost control: cost
Fault injection: reliability

The cherry on top

Tracing

Service Injection

Request Mirroring

Request tapping

And finally, the things that developers see as nice to have are
tracing: ops (this is tracing as part of the cross-service calls, all requests have a trace ID assigned at the edge and service owners can emit trace events for those)
service injection: devx - insert my local dev env into the infra
mirroring: reliability (send a copy of traffic to a new service version)
tapping: reliability, ops (allow me to sniff targeted exchanges in clear-text for troubleshooting)

Tasting

What's the cake, the icing and the "cherry on top" for our developers?

The use-cases in your org may vary a bit but keeping what you're actually trying to help solve as top of mind is crucial. Our developers do not want a service mesh, they just want us to solve some problems for them and help them ship their product faster and better. We'll come back to those use-cases later.

Ingredients

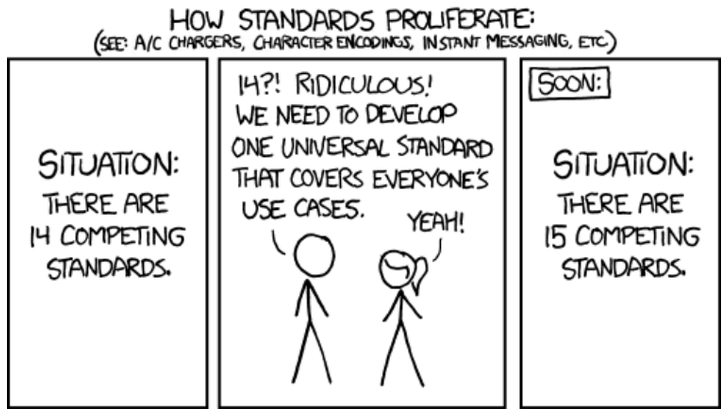
What we learned with various dataplane technologies over the years

To make pudding, you need milk! To make services communicate, you need proxies

Chances are that your organisation has a history of using various proxies that is as long as the organisation itself. Atlassian is no exception.

Our journey with ~~milk~~ proxies

6 years ago we had our own datacenter where different parts of the business ran nginx & haproxy which then got replaced by a vendor product called vTM (virtual traffic manager)
4 years ago we moved into AWS, this introduced ALB/ELB into the picture, we also built a proxy dedicated to our new sharded architecture based on openRESTY (nginx+lua), an API gateway based on Zuul and then through product acquisitions we added varnish and we reintroduced haproxy (and of course nginx and vTMs are still around)

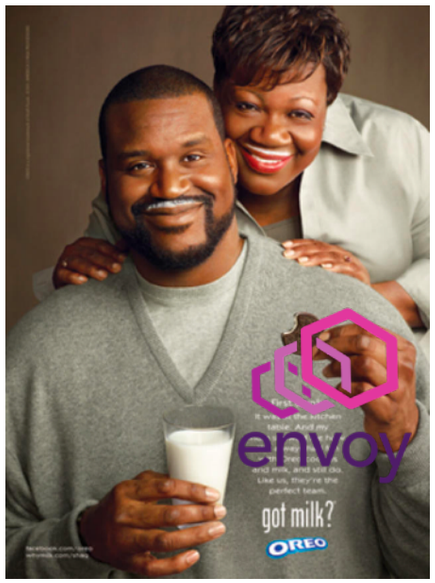


source: https://xkcd.com/927/

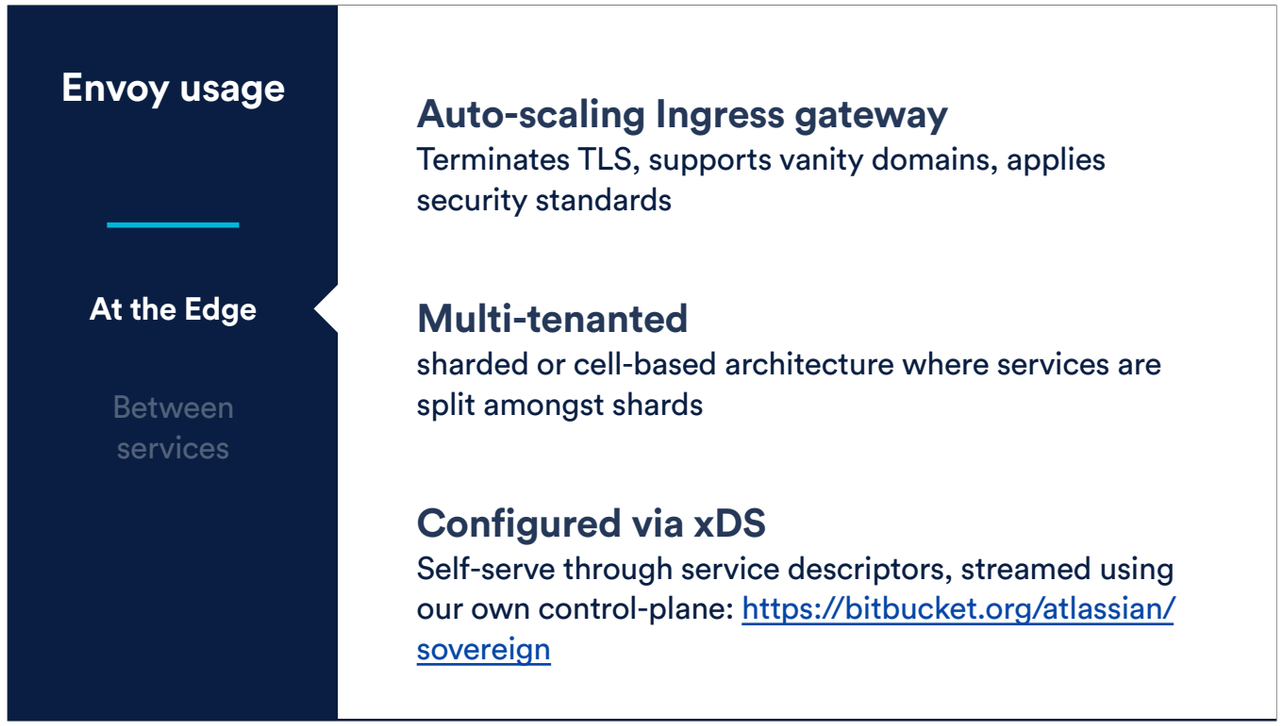
We had to standardize of course, so you all know the stories, 2 years ago we introduced a new proxy solution: Envoy

ENVOY PROXY

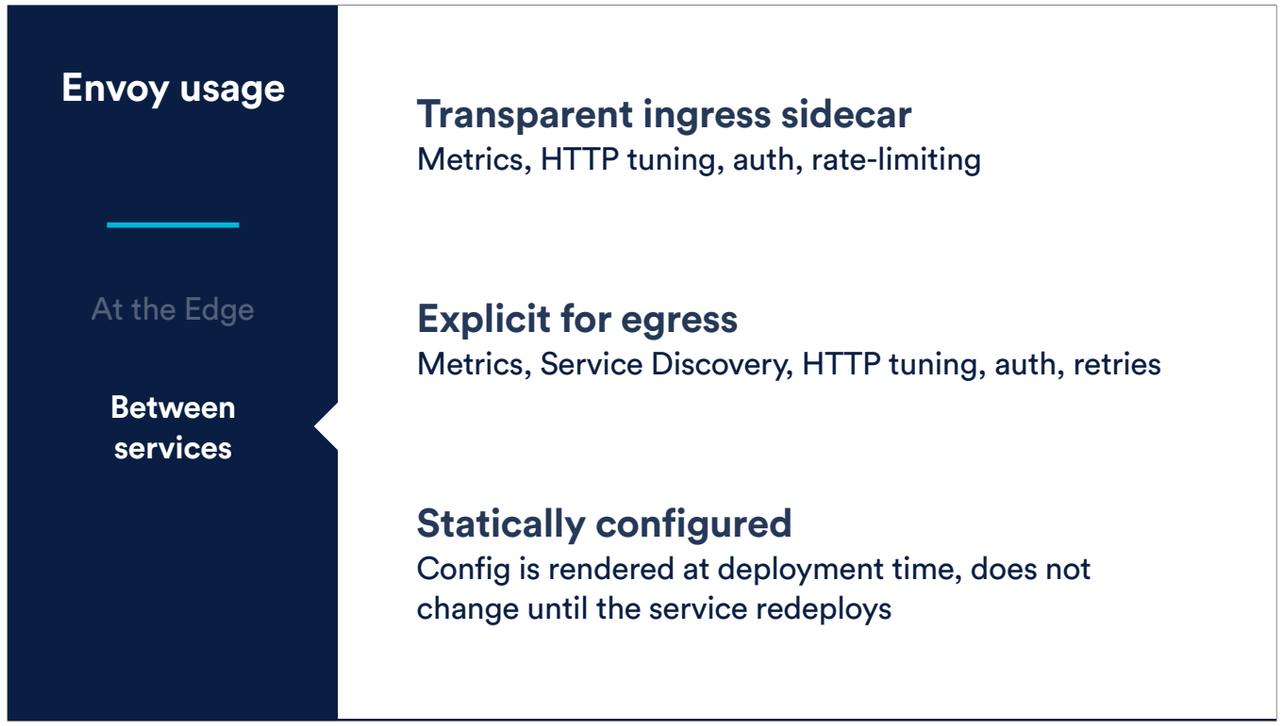
Observability
Performance
Streaming config
Extensibility
Ecosystem



Those were the main reasons why several teams converged on Envoy and we've been happily using it in prod for over 2 years!
We're using Envoy in 2 distinct ways…



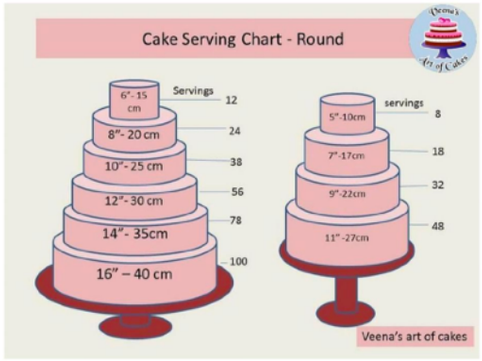
When we started out with Envoy at the Edge, pretty much all solutions out there were extremely focused on Kubernetes and barely any worried about the edge. This is why we decided to build our own control-plane simply to be able to distribute config dynamically to sets of auto-scaling instances behind NLBs. We wanted the platform to be self-serve so developers could describe the kind of public edge they wanted with a few lines in their service descriptor and have that safely streamed into a config change applied to the edge proxies and the xDS API was a great fit for that.
This has allowed us to build a lot of expertise running Envoy at scale and using the xDS API.



Our internal PaaS runs largely on the technologies that were the best building blocks in AWS when we started it 5-6 years ago: Auto-scaling group of EC2 instances fronted by a elastic load-balancer, managed as CloudFormation stacks with extra resources associated for datastore and so on. Kubernetes wasn't really a viable solution for us at the time.

We started introducing Envoy as a sidecar on every EC2 instance as a transparent ingress proxy. Our first feature was to standardise the ingress HTTP server and generate better metrics than what we could get from the ALB. We've been adding more and more features as well as supporting egress, though egress is not transparent, developers need to change their code to stop using the DNS name of the service they want to talk to and pull an environment variable we inject for the service instead.

In order to minimise risks, services are mostly immutable and new versions are rolled out as new deployments where extra guardrails can be put in place. So far we've applied the same principle on the Envoy proxy config that's colocated on those EC2 instances. The config is rendered once when the new deployment takes place and populated as a file on instances that are part of the auto-scaling group for the service. We don't use xDS here



Wedding cake servings size of 1" x 2" portion.

So how much cake does this give me?

What use-cases do we satisfy with our current approach?

Let’s go back to our use-cases to see how far our current setup can get us

The cake (1/2)

HTTP client/ server tuning

keepalives, timeouts, …

API facade

a gateway but not a gateway

Service discovery

DNS is a distributed database I don't want to know about

Authentication

who are you anyway?

API facading requires applying routing rules on the egress client-side but in our static config model, we can't change the client-side config since clients aren't redeployed when a service that they consume is redeployed.

The cake (2/2)

gRPC

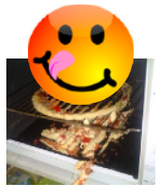
sometimes you want to let REST in peace

Rate-limiting

Not so fast, buddy!

Metrics

if I can't measure it, it doesn't exist



Automated failure handling

gRPC used to require direct node-to-node communication because ALBs did not support it but this changed a few months ago.

The icing

Authz

Tenanted sharding

Cost control

cross-AZ, ALB/ELB

Fault injection

Because shards may come and go, we need a way to update the list of upstream clusters dynamically on the egress proxies of our clients.
To start doing az-affinity and avoiding load-balancer costs, we need to go node-to-node and given the auto-scaling nature of the infra, this requires a control-plane to dynamically update upstream clusters with the live nodes.
Chaos experiments need to have a much faster lifecycle than deployments in order to give a decent developer experience

The cherry on top

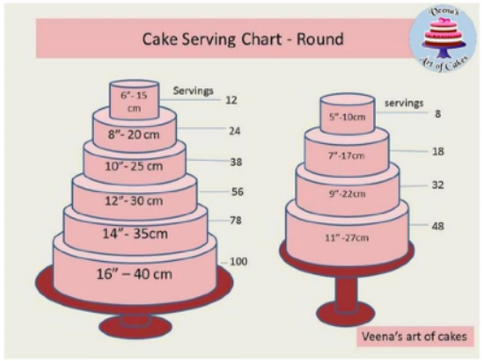
Tracing

Service Injection

Request Mirroring

Request tapping

and some of our low-priority feature also require the ability to make changes on-the-fly to be usable.



Wedding cake servings size of 1" x 2" portion.

So how much cake does this give me?

What use-cases do we satisfy with our current approach?

So really, the main drivers for us to need a dynamic control-plane are:

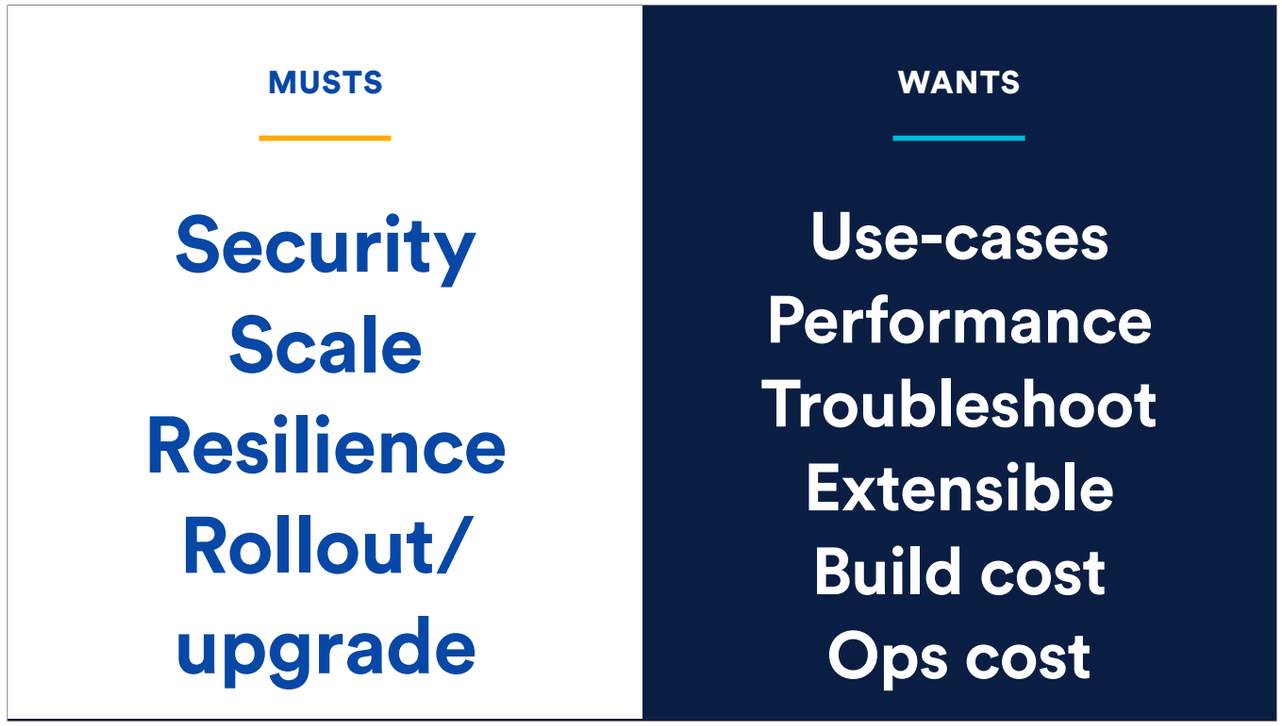
- the ability to apply client-side features and allow our developers to have the benefits of internal API gateways without actual gateways
- the ability to natively support routing to sharded services
- the ability to cut ELB/cross-AZ costs when going node-to-node
- the ability to run chaos experiments at the HTTP level and other smaller improvement to developer experience.

Atlassian has gotten to a point where the things above are sufficiently highly sought after that we've decided to introduce a dynamic control-plane for the service-to-service Envoy sidecars.

Recipe

Which control-plane will turn our milk into pudding?

It’s now time to figure out which control-plane will be the best to deliver on what we’re after.
When selecting a technology, we looked at our use-cases but we also looked at all the objectives we wanted our chosen solution to fulfil based on our experience running the platform. We followed the Kepner Tregoe “Decision analysis” methodology and we organised all our objectives, sorted them between the MUST and the WANTs and gave a relative weight to each of the WANTs



This is an abridged list of the things we considered in our decision:
We required the solution to have a proven track record for handling of security issues and scale similar to ours.
It had to be resilient to all expected infrastructure failures (instance, az, …)
And we needed to be able to gradually rollout changes and do upgrades without interruption

On the WANT side, we need to check how well each solution fulfils our use-cases but we also consider things like performance, cost, extensibility and ability to troubleshoot

ALTERNATIVES

We considered: DIY, Istio, App Mesh, Consul Connect, Kuma, Open Service Mesh

It was time to look at what's out there. We know that the list is not exhaustive and we know that some of the decisions are subjective, this is acknowledged and because the whole decision matrix is documented, we can always revisit it if we find that our judgement was incorrect.

We considered: DIY, Istio, App Mesh, Consul Connect, Kuma, Open Service Mesh

ALTERNATIVES

We considered: DIY, Istio, App Mesh, Consul Connect, ~~Kuma~~, ~~Open Service Mesh~~

While screening those through our must, Kuma and Open Service Mesh did not meet our requirements in terms of having a proven track record on security and scale

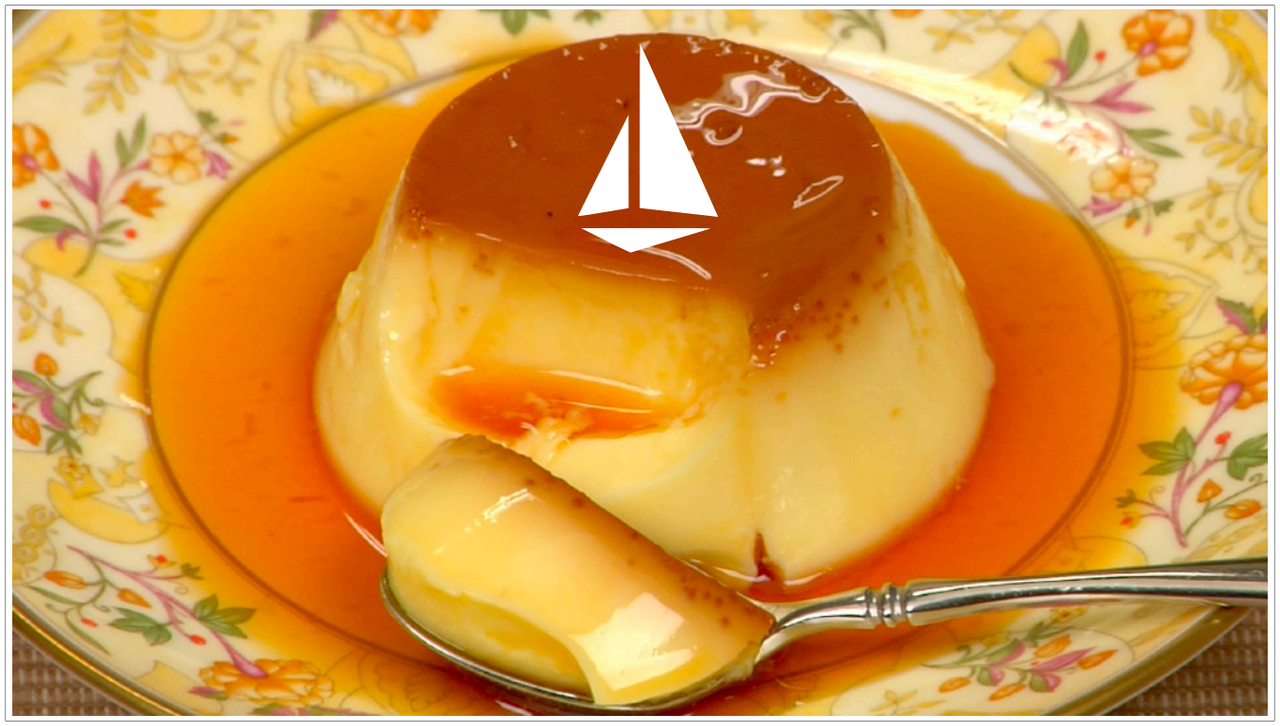
ALTERNATIVES

We considered: ~~DIY~~, Istio, App Mesh, ~~Consul Connect~~, ~~Kuma~~, ~~Open Service Mesh~~

While doing relative scoring, the cost of building a control-plane ourselves ended up causing that option to score poorly.
Consul Connect also suffered from relatively high cost since we don’t run Consul at all (but we do run Kubernetes API and etcd and have plenty of expertise there) and a lot of the features we wanted are not available out-of-the-box and would have needed us to extend (escape hatch)

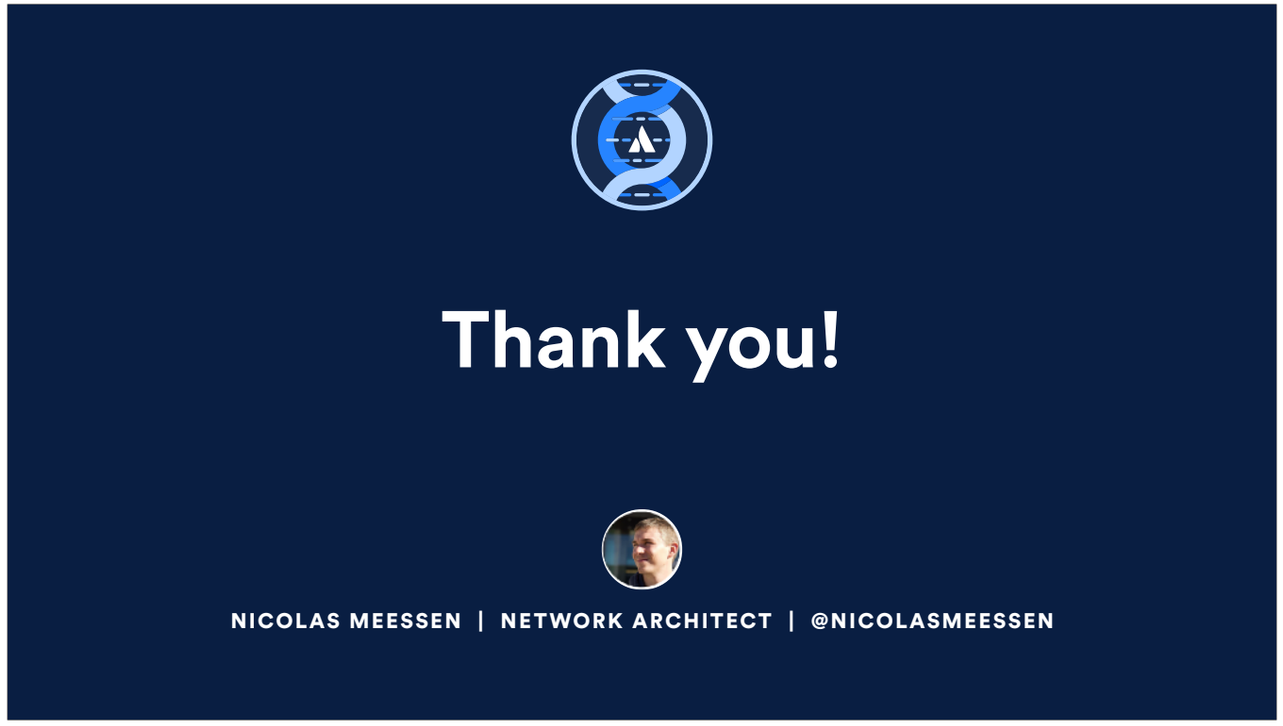
In the end, timing was everything:

- in the last year or so, Istio has tremendously improved its support for non-K8s compute (VM or mesh expansion), it has simplified its operating model, guarantees us the possibility to extend and benefits from the support of a large part of the k8s ecosystem.
- app mesh on the other hand would be massively simpler to build and operate but critical features are still missing (rate-limit, external authz, …) the story around extensibility is also lagging behind. The public roadmap shows huge progress is being made though and we suspect that if we have to make the same decision in a year's time, the playing field might be more levelled then.



So here we are, we’ve chosen Istio to make our pudding, we’ve done a few test cooks but

we’re yet to really eat it in production



Thank you for listening and I hope that our story will help you decide if and when Istio can help you delight your customers

Q and A


ATLASSIAN