mercari

Istio is a long wild river: how to navigate it safely

About me

Raphael Fraysse

@la1nra (Twitter)

Tech Lead, Networking
Mercari, Inc.

Github / **@lainra**

Today’s agenda

- Istio at Mercari
- Stabilizing Istio
- Adopting Istio

Istio at Mercari

mercari

What Is Mercari?

- **Service start: July 2013**
- **OS: Android, iOS**

**Can also be accessed by web browsers*

- **Usage fee: Free**

**Commission fee for sold items: 10% of the sales price*

- **Regions/languages supported: Base specs for Japan/Japanese**
- **Total number of listings to date: More than 2 billion**

**As of December 2020*

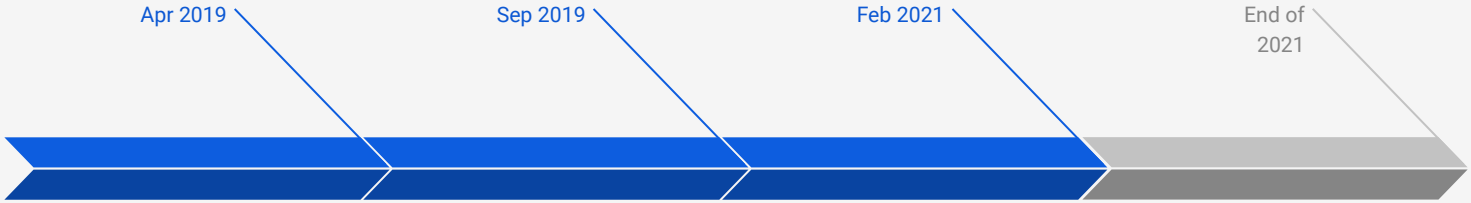
The Mercari app is a C2C marketplace where individuals can easily sell used items. We want to provide both buyers and sellers with a service where they can enjoy safe and secure transactions. Mercari offers a unique customer experience, with a transaction environment that uses the payments Mercari holds in escrow, and simple and affordable shipping options.

Many sellers enjoy having the items they no longer need purchased and used by buyers who need them, and buyers enjoy the feeling of hunting for treasure as they search through unique and diverse items for lucky finds. In addition to buying and selling, users actively communicate through the buyer/seller chat and the “Like” feature.

Istio at Mercari

- 200+ microservices (200+ namespaces)
- 100K RPS at peak on API Gateway
- 1 main production Google Kubernetes Engine (GKE) cluster
- 12k+ pods
- 750+ nodes

Istio at Mercari



Features currently used:

- HTTP/2 Load-balancing
- Traffic Shifting
- mTLS

Features under investigation:

- Retries
- Circuit breaking

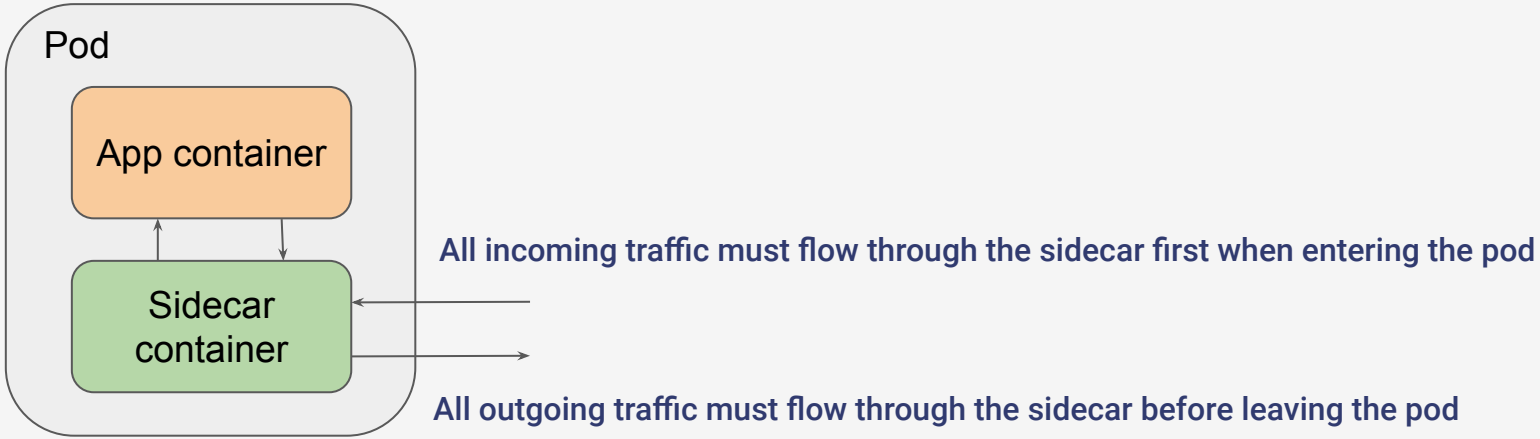
Stabilizing Istio

mercari

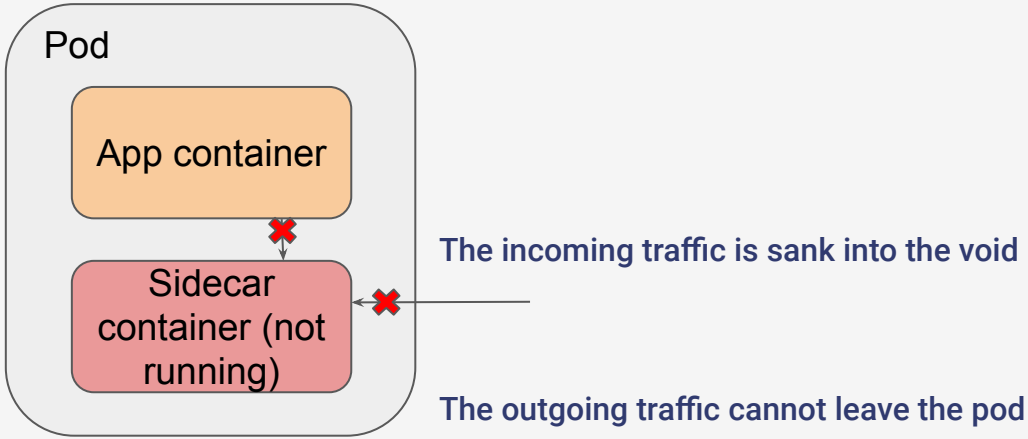
Stabilizing Istio

- Istio sidecar proxy specifications
- Kubernetes shortcomings with sidecar containers
  - Controlling containers lifecycle
  - Autoscaling pods with sidecar containers
- Are you prepared to handle Istio?
- A full mesh is utopian, know what you need only
- Guardrails for Istio

Istio sidecar proxy specifications



What happens when the sidecar container is not ready?



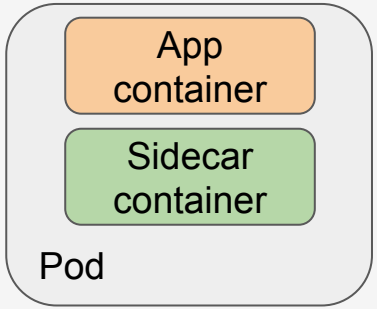
What happens when the sidecar container is not ready?

- 2 cases where it happens frequently:
  - During pod creation
  - During pod deletion

- **To prevent it, we need to make sure that:**
  1. Envoy is **started before** any other container in a pod
  2. Envoy is **stopped after** any other container in a pod

Kubernetes shortcomings with sidecar containers

A Pod is the Kubernetes atomic unit



Pods are the atomic unit, not containers.

Shortcoming 1: Controlling the running order for containers

Kubernetes lacks good control APIs to customize the containers lifecycle in a pod.

There is no official way to instruct a pod to:

1. Start the sidecar container first
2. Stop the sidecar container after the app container is stopped

However, we can wrap a pod lifecycle using container lifecycle hooks to achieve our goal.

Workaround: Use postStart and preStop lifecycle hooks

1. Ensure that Envoy is **started before** any other container in a pod
   - Use a `postStart` lifecycle hook in the istio-proxy container manifest

```
lifecycle:
  postStart:
    exec:
      command:
      - pilot-agent
      - wait
```

Fortunately, it is handled automatically since Istio 1.8 by setting the ***`holdApplicationUntilProxyStarts`*** field to ***true*** in ProxyConfig under MeshConfig options:

```
meshConfig:
  defaultConfig:
      holdApplicationUntilProxyStarts: true
```

Workaround: Use postStart and preStop lifecycle hooks

2. Ensure that Envoy is **stopped after** any other container in a pod
   - Use a `preStop` lifecycle hook in the **istio-proxy container** manifest:

```
lifecycle:
  preStop:
    exec:
      command: [“/bin/sh”, “-c”, “while [ $(netstat -plunt | grep tcp | grep -v envoy |
wc -l | xargs) -ne 0 ]; do sleep 1; done”]
```

This preStop hook will wait for application connections to be drained before stopping the container.

Workaround: Use postStart and preStop lifecycle hooks

2. Ensure that Envoy is **stopped after** any other container in a pod
    - Use a `preStop` lifecycle hook in the **application container** manifest:

```
     lifecycle:
       preStop:
         exec:
           command: ["/bin/sh", "-c", "sleep 30; wget -qO- --post-data ''
localhost:15000/healthcheck/fail; sleep 45; wget -qO- --post-data ''
localhost:15000/healthcheck/ok;"]
```

This preStop hook will sleep to let downstream gRPC connections terminate, drain the Envoy listeners and sleep to give enough time for draining remaining connections. The last command is to handle container restart cases.

Workaround: Use postStart and preStop lifecycle hooks

2. Ensure that Envoy is **stopped after** any other container in a pod
    - Adjust your pods ***terminationGracePeriodSeconds*** to be more than the sum of all sleeps in the preStop hooks.

➔ If the pod is terminated too early, connection draining may not complete, leading to 5xx errors

Example: for sleep 30 + sleep 45 in the application container, we set ***terminationGracePeriodSeconds*** to 90 seconds.

Warning: These are workarounds, not solutions!

Test before using!

These workarounds are based on the Kubernetes pod/container lifecycles and only recommended if you know what you are doing.
Once Kubernetes supports the sidecar pattern in a better way, these workarounds should be deprecated.

Shortcoming 2: Autoscaling multi-containers pods

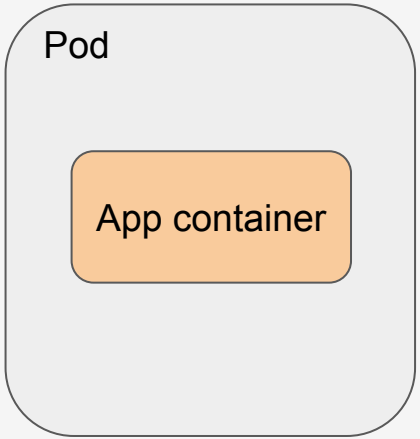
Kubernetes offers 2 ways to autoscale pods:

- HorizontalPodAutoscaler (HPA)
- VerticalPodAutoscaler (VPA)

Unfortunately, Kubernetes is (was) not very smart at scaling out pods with multiple containers with HPA.

- Fixed in Kubernetes 1.20 by specifying a container resource as an HPA target
- In the meantime, we need to add the Istio sidecar into the HPA calculation

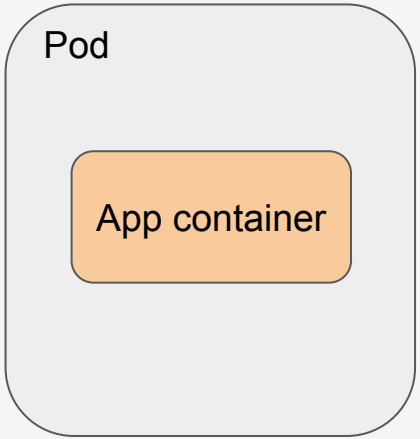
Define HPA target for multi-containers pods



Container requests

CPU: 1
Memory: 100MB

Define HPA target for multi-containers pods



Container requests

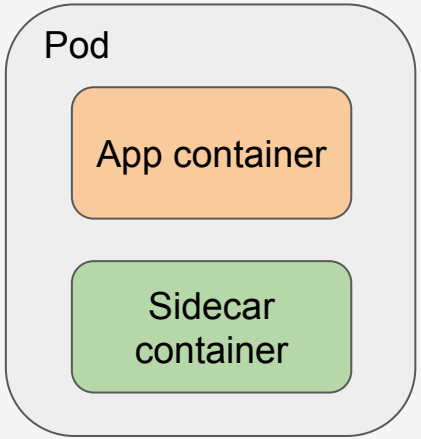
CPU: 1

HPA configuration (70% CPU)

```
metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

Will trigger when the container is using more than **700m CPU**

Define HPA target for multi-containers pods



Container requests

CPU: 1

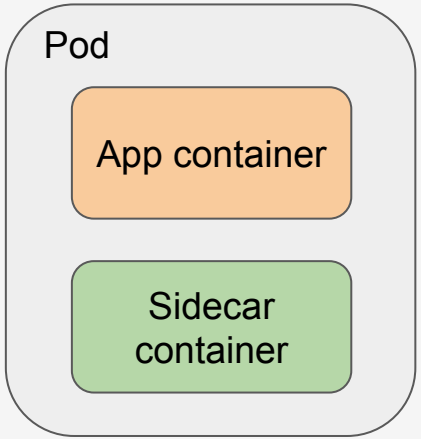
CPU: 100m

HPA configuration (70% CPU)

```
metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

The HPA takes the average of all containers CPU requests values.

Define HPA target for multi-containers pods



Container resources

CPU: 1

CPU: 100m

HPA configuration (70% CPU)

```yaml
metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

Will trigger when the container is using more than **770m CPU**

Define HPA target for multi-containers pods

Two options:

1. **Make the istio-proxy CPU very low compared to the application CPU**
   (Between x% and y% of app CPU) to minimize the variance

2. **Adjust the HPA threshold to match the original CPU absolute target (700m):**
   Target % = Original CPU absolute target /Sum of CPU resources = 63.6%.

Define HPA target for multi-containers pods

Both options have their drawbacks, since **you need to involve users in the calculation**, making it a big blocker in spreading the Istio adoption…

The other big problem is estimating what is the Istio sidecar container CPU usage, which we'll talk about in the second part of the presentation.

Are you prepared to handle Istio?

Main time consumers with Istio:

1. Troubleshooting
2. Spreading adoption
3. Supporting new features

To succeed in Istio adoption you need to have:

- Dedicated resources for it (the more the better)
- A good in-house knowledge of networking : Linux, Kubernetes and Envoy
- Be patient and resisting the temptations from users to open features too early
- Mechanisms to improve the reliability of Istio

Choose your fights, start small

Start with few simple features such as:

- Injecting sidecars, HTTP/2 LoadBalancing
- Traffic shifting for canaries

Build confidence in the system and understanding of Istio. Then you can onboard some users, get feedback, improve, rinse and repeat.

A full mesh is utopian, know what you need only

The dream:

- Service meshes usually promise full mesh observability, reachability
- Plug it in, and shall the magic unleash! they said

A full mesh is utopian, know what you need only

The reality:

- The control plane is burning down when pushing your thousand services updates to the hundreds of proxies running
- Proxies are OOM Killed every X minutes since they cannot handle the change frequency
- Proxies are heavily CPU throttling and consuming CPU without traffic
- Envoy configuration files are > 100K Lines

A full mesh is utopian, know what you need only

In fact, Istio is impossible to use at any scale other than small PoCs without restricting the exposed resources to each proxy in the mesh.

It is written in the official documentation, and actually, reference values are **only disclosed for when namespace isolation is enabled.**

The Sidecar CRD to save the mesh

The Sidecar CRD (Custom Resource Definition) allows to control the exposure of mesh configuration to a specific proxy, based on namespace or labels.

```
apiVersion: networking.istio.io/v1beta1
kind: Sidecar
metadata:
  name: default
  namespace: mercari-echo-jp-dev
spec:
  egress:
  - hosts:
    - ./*
    - istio-system/*
```

The Sidecar CRD to save the mesh

The Sidecar CRD (Custom Resource Definition) allows to control the exposure of mesh configuration to a specific proxy, based on namespace or labels.

```
apiVersion: networking.istio.io/v1beta1
kind: Sidecar
metadata:
  name: default
  namespace: mercari-echo-jp-dev
spec:
  egress:
  - hosts:
    - ./*
    - istio-system/*
```

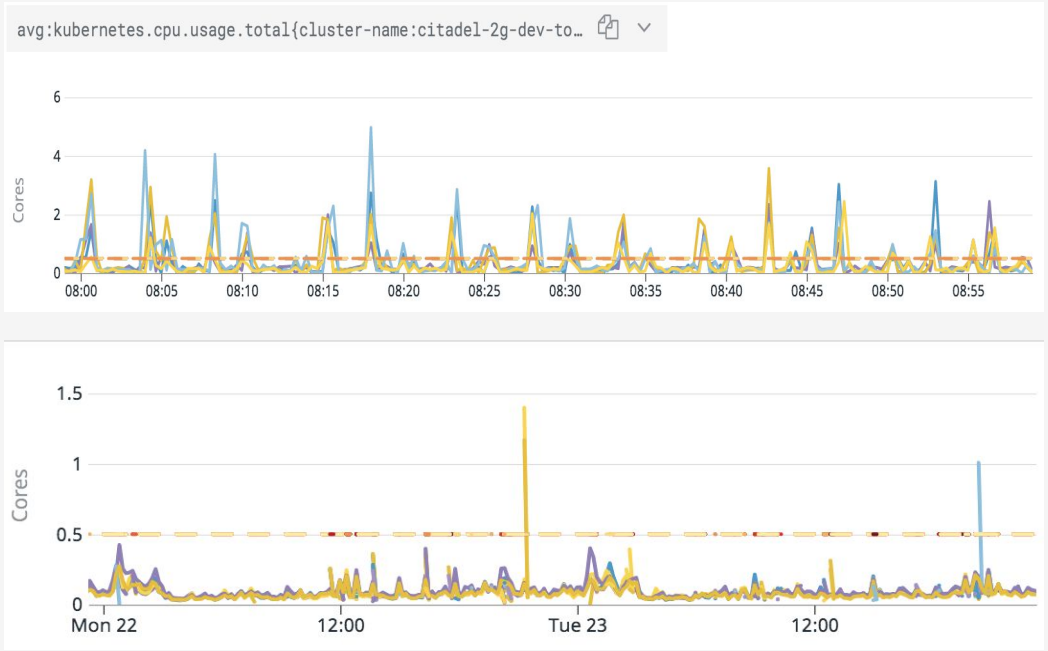
Only Istio and the local namespace configuration is pushed to namespace-local proxies:

- Listeners
- Clusters
- Endpoints

The Sidecar CRD to save the mesh

Without sidecar CRD

With sidecar CRD



Istiod average CPU usage

The Sidecar CRD to save the mesh

Main drawback

Services must know their dependencies, document and update them.

If this wasn't the case before, Istio may not feel welcoming to users.

When a dependency is not in the allowed list of a Sidecar CRD, **the service mesh features will not be available for that traffic**. (Because it goes through the *PassthroughCluster*)

Some approaches to handle Sidecar CRDs

- Do not expose Sidecar CRD to users, use a service definition to generate Sidecar
- Use protocol specific traffic sniffing (i.e. gRPC call discovery) to find out dependencies
- eBPF magic to get service calls?

We use the first approach currently as it is protocol-agnostic and works before live traffic.

Guardrails for Istio

- The service mesh is common to all users
- Any change to it spreads across the whole mesh
  - **Any misconfiguration spread too,** be it intentional or not

Humans are error-prone, both users and operators are humans so:

Errors will happen, with a large blast radius!

How can we mitigate errors and their impact?

- Leverage linters (conftest) to catch issues at CI-level, keeping a short feedback loop
- Leverage admission webhooks (OPA Gatekeeper) to
  - protect the resources
  - check what cannot be checked at linter-level (inventory)

Please check my last year presentation:

"Preparing the guardrails for Istio at scale" (Slides, Video) for more details

Takeaways

- Kubernetes doesn’t handle sidecar containers well
  - Use postStart and preStop container hooks to gracefully handle the pod lifecycle
- Kubernetes doesn’t scale Istio-enabled pods well
  - Use ContainerResource to fix HPA on the application container (From K8S 1.20)
  - Otherwise, add the Sidecar proxy CPU usage into calculation for HPA scale target.
- Exposing only a few Istio features helps with Istio adoption and stability
- Use Sidecar CRDs to keep Istio healthy and find mechanisms to handle this automatically
- Guardrails such as Gatekeeper OPA are crucial to ensure the long-term stability of Istio

Adopting Istio

mercari

Adoption challenges

- Moving HTTP/2 load-balancing from client-side to Envoy
- Label selector updates for app and version labels
- Istio default retry policy
- Istio proxy performance and load testing
- Abstracting the Istio features

Moving HTTP/2 load-balancing from client-side to Envoy

- We use gRPC heavily in our microservices
- But Kubernetes is pretty bad at load-balancing it
- So we solved it by using a client-side load-balancing library + Headless Services

Headless services are to us what ClusterIP services are to common people!

However, our KubeDNS was not happy at all with the SRV requests...

Promises of brighter days with Istio

- Then Istio came, with its awesome HTTP/2 load-balancing capabilities out-of-the-box
- We tried it as-is, with existing gRPC services
- Result: **Weird 5XXs on upstream service pod rollout**
- No matter how well our services handled graceful termination, Istio would make headless services worse.

Conclusion: We stopped using headless services and gradually migrated to ClusterIP services

The hell of migrating hundreds of services

- Services are immutable (for some good reasons) so for each service migration, we need to:
  - Write the ClusterIP service equivalent
  - Make sure Istio-enabled callers update their config with the ClusterIP service
  - Keep a double standard during migration

Compounding to hundreds of services, the cost is terrible so be strategic

Strategy to migrate services

1. Abstract
2. Explain
3. Support
4. Track

Label selector updates for *app* and *version* labels

- Is there anyone in the audience who was prescient enough to use the *app* or *version* before starting Istio?
- Chances are huge that you need to modify your Deployments to put these labels
  - **Because we all want fancy Traffic Shifting features!**
- Then you try to update, and:

`Error: .LabelSelectorRequirement(nil)}: field is immutable` (Since k8s 1.16)

Label selector updates for *app* and *version* labels

First, headless services, now labels...

Who said that migrating to Istio is only about adding sidecars??

Label selector updates for *app* and *version* labels

Fair enough, let's do it:

1. Create a new Deployment with new name (immutable field) with the *app* and *version* labels
2. Make sure the Service is serving both Deployments
3. Create HPAs to target the new Deployment
4. Delete old Deployment

Simple, isn't it? Now, repeat for hundreds of services! Good luck :D

Label selector updates for *app* and *version* labels

A more sustainable approach:

- Use your CD tooling (i.e. Spinnaker) to automate this migration
- Ask users to use the migration pipeline when onboarding with Istio

This approach is quite similar to canary release so you gain time by investing into it

Istio default retry policy

Another good surprise from Istio:

All HTTP requests are **retried twice!**

The other even better surprise is:

You cannot disable it or change it!

Istio default retry policy

So you're stuck with adding a RetryPolicy for every single Kubernetes service served by Istio...

➔ Isn't it loose coupling? This issue opened last year explains the problem and its fatality.

Thankfully, the community is working on a solution. **(Contributing is important!!!)**

But we didn't have the time to wait for it so what did we do?

We forked Istio!

Forking Istio to change the default retry policy

It’s not a big deal, actually a one-liner change in the code:

```
-       RetryOn:        "connect-failure,refused-stream,unavailable,cancelled,retriable-status-codes",
+       RetryOn:        "connect-failure”,
```

Connect-failure is retry-safe even for non-idempotent methods as it is triggers when a server is unavailable at the TCP level.

Build your Istiod image, push your tag and use it in the IstioOperator manifest.

Istio proxy performance and capacity

- Putting sidecars everywhere has a cost
  - Latency
  - Compute resources

The Istio 1.9 community reference values for sidecar performance are:

- Latency: +2.65 ms at p90 (no telemetry)
- Compute resources: 0.35 vCPU and 40 MB memory / 1000 RPS

Istio proxy performance and capacity

- What do we want when implementing Istio?
  - Added value to the business
  - Reliable performance
  - Reasonable cost

- Put in another way, know your tradeoffs:
  - How acceptable is the performance loss for the added value?
  - How much should we pay for the added value?

Istio proxy performance and capacity

- Each workload may be different, even in a same product. Some examples:
  - Latency-sensitive workloads
  - Long-lived batches (ML)
  - Web platforms
- How do you define a common answer to the previous questions?
  - It’s nearly impossible
  - At best, requires to involve each owner and brainstorm it

Istio proxy performance and capacity

Fact: If Istio is enabled in all pods in a cluster, for n pods, there are n sidecars

- Case 1: One size fits all (need to fit the biggest workload)
  - \+ Easy to set, one default value for sidecar resources
  - \- Bigger default size = bigger cost
- Case 2: Adjust based on workloads
  - \+ Resource cost is low
  - \- Tremendous cost in load-testing and adjusting values

Istio proxy performance and capacity

- One size fits all is too costly for us (and should probably be for you too)
- So how can we adjust the sidecar size?
  - VPA? Not working
  - HPA? Not applicable
  - Load testing application, load testing the sidecar -> seems the only way

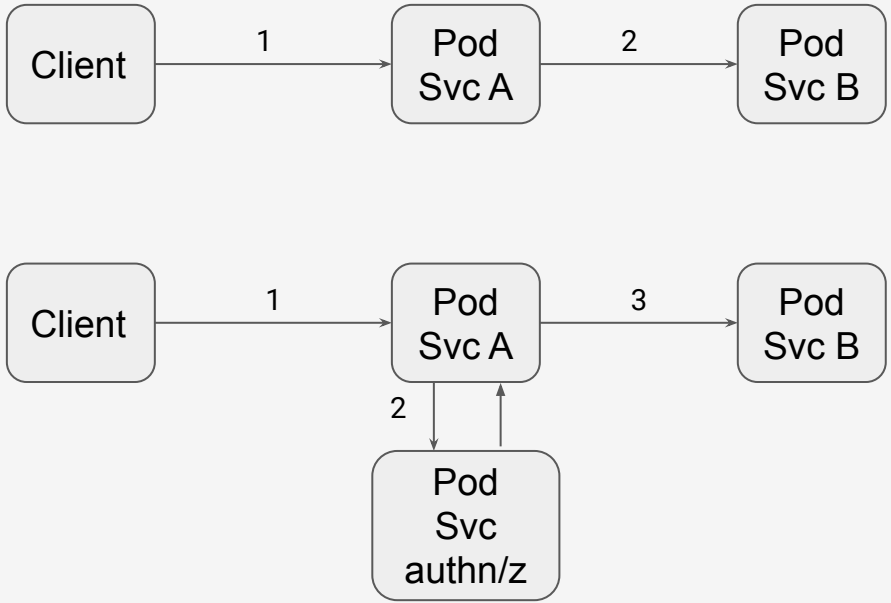
We just want a dynamic smart autoscaler for Istio sidecars!

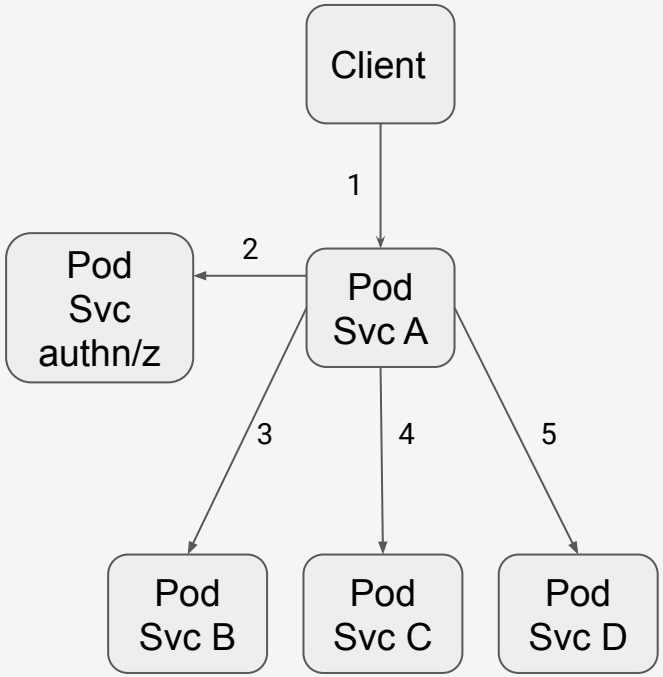
Istio proxy performance and capacity

- When load-testing a service Istio sidecar, questions to ask:
  - How many RPS without Istio?
  - How many hops per request?
    - Single request per call?
    - Multiple requests per call?
    - Calling authn/z service on each call?

Depending on the answers, the application RPS measured in library may vary between 2 and n times when using Istio.

Istio proxy performance and capacity





Istio proxy performance and capacity

Service with 2 requests:

10000 RPS at library level

Istio RPS: 20000 RPS

Service with 5 requests:

10000 RPS at library level

Istio RPS: 50000 RPS

Istio proxy performance and capacity

Service with 2 requests:	20 pods
10000 RPS at library level	RPS/pod: 500
Istio RPS: 20000 RPS	Istio RPS/pod: 1000
Service with 5 requests:	10 pods
10000 RPS at library level	RPS/pod: 1000
Istio RPS: 50000 RPS	Istio RPS/pod: 5000

Istio proxy performance and capacity

Envoy concurrency setting is also very important for performance.

- Default -> 2
- For minimal performance impact -> Workers = vCPU (1 worker/vCPU)
- Load test your workloads at different level of concurrency and resources
- Account for RPS/pod when calculating the capacity and beware of HPA
- **Capacity differs greatly depending on both CPU resources and concurrency**

Abstracting Istio

- Should you expose a whole new layer of YAMLs to people that are already overfed with?

The answer is no.

- Should you require your users to understand every single parameter in a VirtualService?

The answer is also no.

The main reason is probably that you are paid to improve your users productivity, not decreasing it.

Abstracting Istio

The same way as we build libraries and interfaces to improve productivity, we need to build proper abstractions to maximize the added value of Istio to our users:

- Automating the onboarding
- Making a feature fully automated and managed

It improves by a lot:

- The user experience for developing services
- The maintainability of Istio for operators

How we abstract Istio

- We are using Terraform to handle the Sidecar CRD Policy and GitOps CI/CD pipeline to apply them
- We are exploring Cuelang to template a simple DSL for managing various features
  - Full Istio onboarding (lifecycles, injection…)
  - True Managed Canary Release with Spinnaker
  - And more coming in the future!

Takeaways

- Headless services are erratic with Istio, use ClusterIP services instead, plan the migration wisely
- Use automation pipelines to label Deployments for traffic shifting
- Istio has a risky default retry policy for non-idempotent APIs, we forked Istio to solve it temporarily
- Having sidecars everywhere is a huge cost so make sure to mitigate it by proper sizing with testing
- Abstracting the Istio features is the only way to spread the adoption and maximize their added value

Thank you very much for joining!

We’re hiring :)