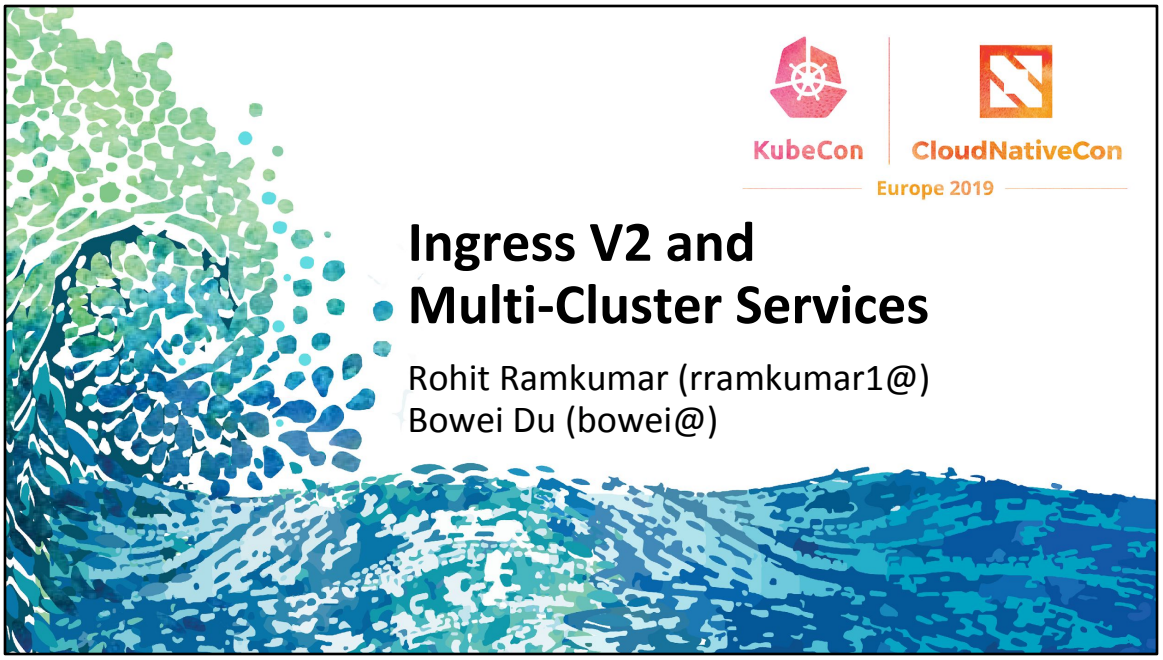
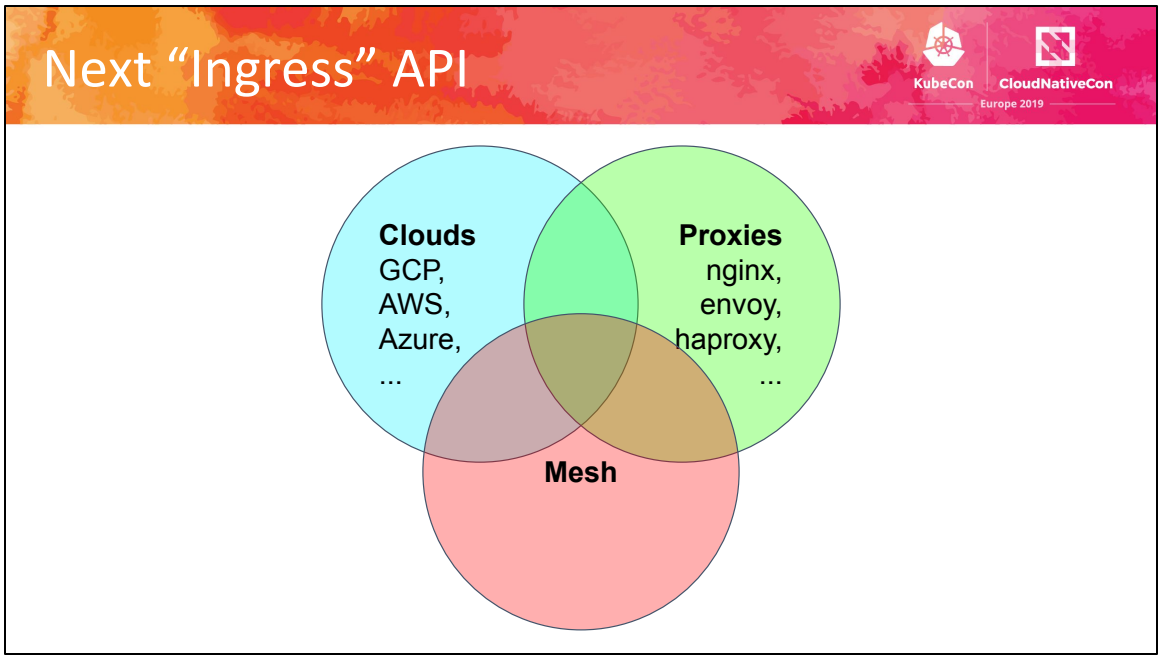

KubeCon
CloudNativeCon
Europe 2019
Ingress V2 and
Multi-Cluster Services
Rohit Ramkumar (rramkumar1@)
Bowei Du (bowei@)



After the survey from 2018, we look at the landscape of what Ingress was trying to fulfill:

- Describe L7 proxy-based load balancing (Note: could be extended to L4)

Does this make sense in the core Kubernetes?

- Would there be a project that we can point to and say “use that”

Could we have a strong statement on portability and featureset?

- Option: It will not be portable.
- Option: We standardize on a given implementation and wait for convergence…

The gaps in featureset/portability between cloud infrastructure and proxies will persist for quite some time:

- It is likely infeasible for clouds to allow for the kind of customization and expressiveness of a self-managed proxy
- However, we do see convergence, but it is slow.

We also had a newer technology, that of service meshes, that have different deployment model, featuresets, but intersect the space.

Next “Ingress” API

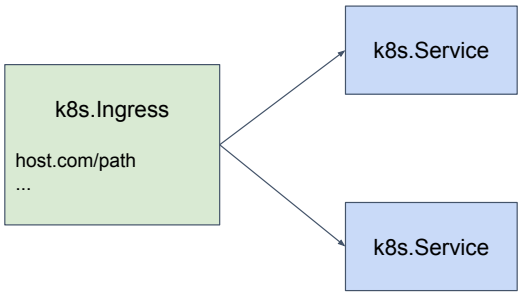
Challenges: portability, expressiveness, features

Lots of APIs now in this space.

What can we can learn from them?

Note: reference systems below were chosen for illustrative purposes, not meant to be exhaustive.

API models: Ingress (Beta)



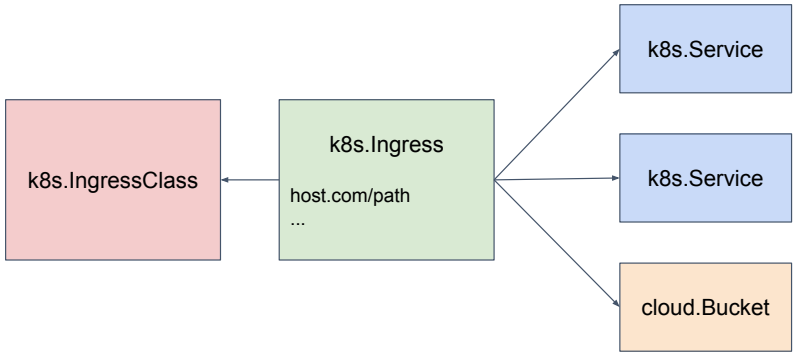
Key attributes: self-service, single-role, simple

Decompose each of these model into key attributes.

I know there are a lot of nuances, such as support features. But for the purposes of this presentation, we are focusing on the big picture.

self-service: a single developer can define and configure all aspects of the frontend (TLS termination, protocols, etc) and the backends.
single-role: single dev
simple: just one resource to understand

API models: Ingress GA (proposed)

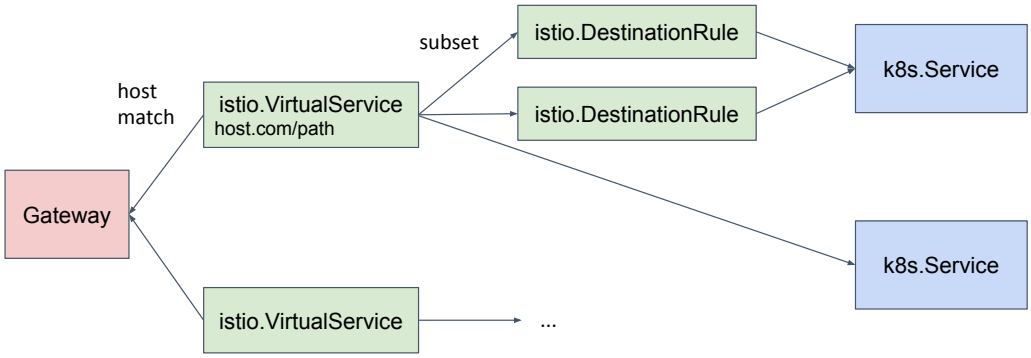


Key attributes: “flavor” of Ingress via class, heterogenous backends.

GA codifies some common practice, fixes bugs and adds more flexibility to the backends.

Ingress class: which controller manages the ingress, akin to storage class
Currently in discussion: more flexibility in backends that can be used.

API models: Istio



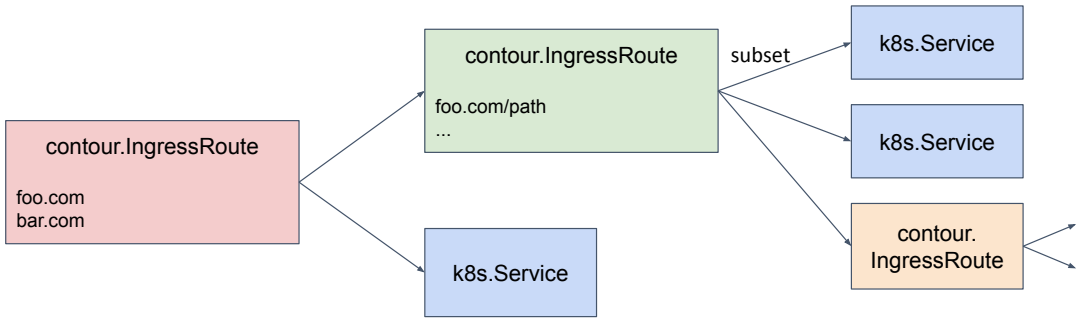
Key attributes: Separate roles for proxy infrastructure, application definition; rich feature set based on Envoy

Popular model istio:

Istio is a service mesh, but the same resources are intended to be used with “traffic ingress” purposes

Separate “proxy infra” from app description
Traffic splitting
Rich feature set

API models: IngressRoute

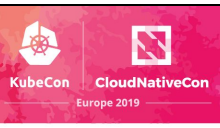


Key attributes: Separate roles, recursive delegation + composability, additional features such as traffic splitting.

Another interesting API model:

IngressRoute -- recursive model

API models: Others



Lots of annotations for features (Everybody)

Decorators on k8s.Service instead of k8s.Ingress (Ambassador)

Use Custom Resources (Gloo)

Some other interesting API models

What have we learned?

KubeCon CloudNativeCon Europe 2019

Multi-role environments

- Infrastructure vs App dev

How much portability?

- Should be a user choice.

Support future API growth

- Claim: providers/features will converge over time (but quite slowly).

Ingress was first designed, focus was on simplicity and single dev.

A modest proposal

Warning: this is very early…

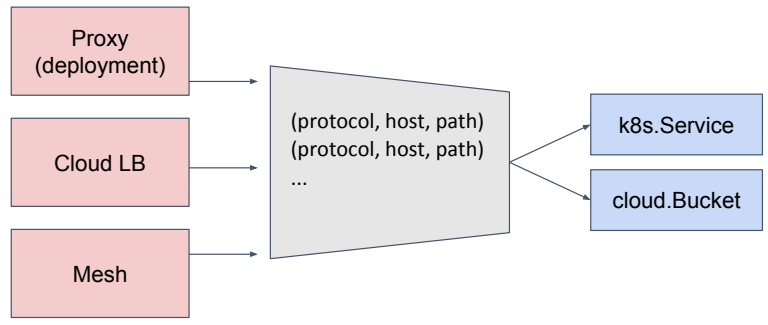
Takes from heavily from existing work...

Shape of the resources (model)

Portability and extensibility

Future directions

A modest proposal: the model

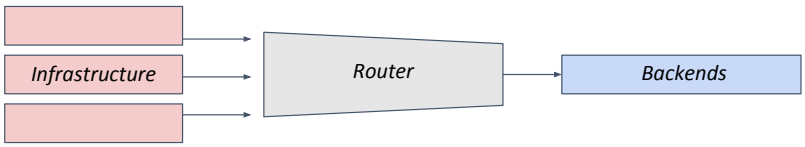


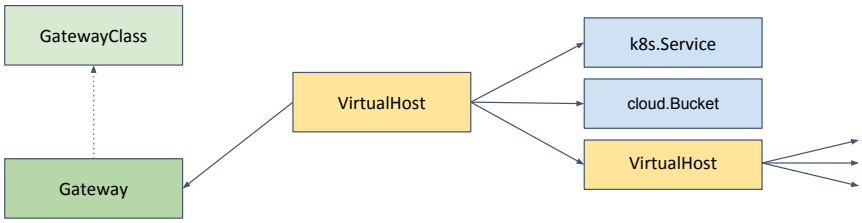
Infrastructure
Load-balancer implementation

Router
Application description

Backends
Application implementation

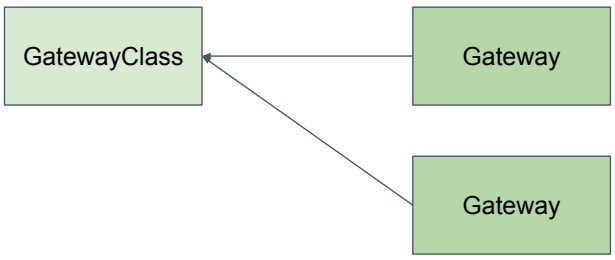
A modest proposal: the model





*Names are temporary

Gateway



Deployment specific options - e.g. mergeable

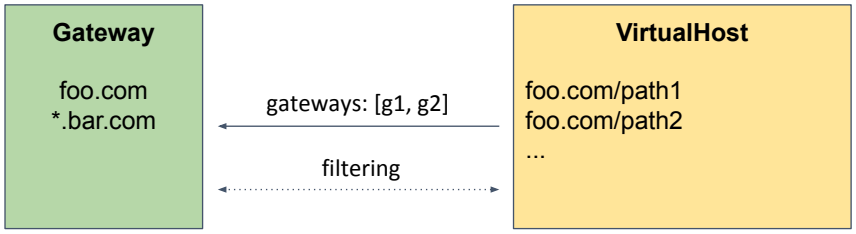
Abstracts available implementations

Represents actual instance of LB/proxy infrastructure, capacity

Protocol termination (<IP:port>, TLS)

Gateway - represents the actual instance of an LB

Gateway ↔ VirtualHost



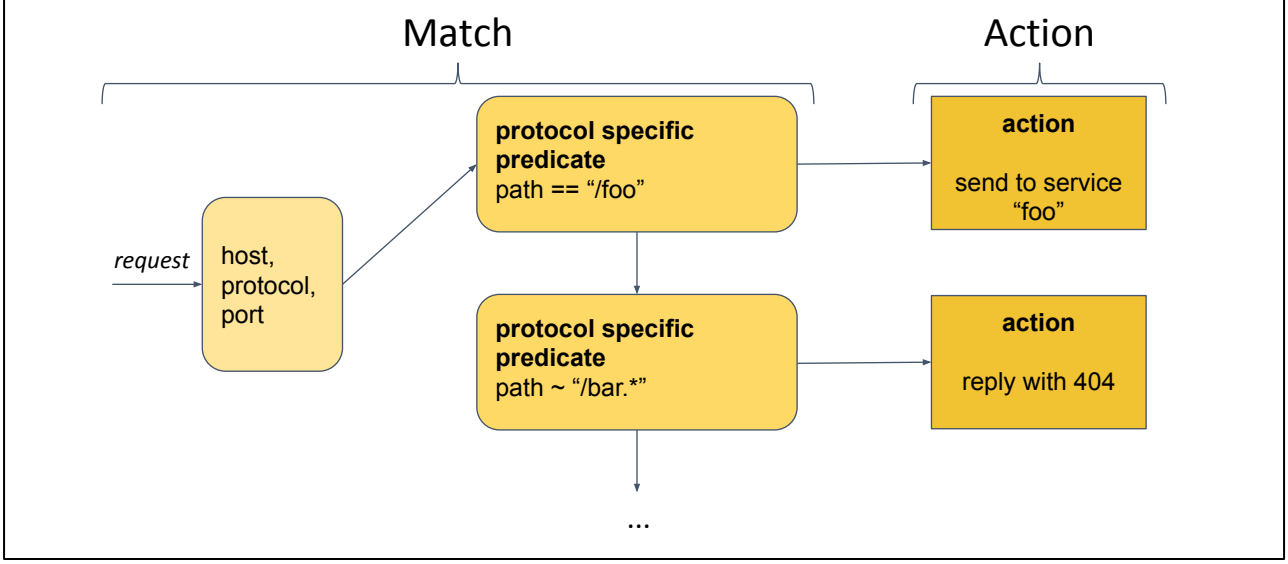
How to match Gateway and VirtualHost?

- VirtualHost attaches to a Gateway
- Gateway filters VirtualHosts -- wildcard allows for self-service.

How to match Gateways to VirtualHost?

If we make a direct reference from gateway to virtualhost, this limits the ability to self-service

VirtualHost: Match → Action

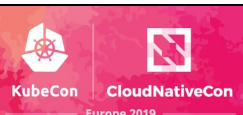


This is a quite generic model that lets us add matchers and predicates later

we can start with the existing very simple matchers (e.g. path-based)
hang other predicates off of this layer

Actions gives us a way to expand the range of possible processing effects later

VirtualHost




```
type VirtualHost struct {
     Routes []Route
}
type Route struct {
     HTTP *HTTPRoute
     TCP  *TCPRoute
}
type HTTPRoute str
     Host string
     Rules []HTTPR
}
type HTTPRouteRule
     // Match
     Path     string
     ...
     // Action
     Backend    *HTTPRouteActionBackend
     StatusCode *HTTPRouteActionStatusCode
     ...
}
```

<blink> just a sketch </blink>

```
# VirtualHost
spec:
  routes:
  - http:
      host: "foo.com"
      rules:
                                  foo-app
                                  ort: www
      - backend:
          service: tcp-app
          servicePort: my-protocol
```

VirtualHost extensibility

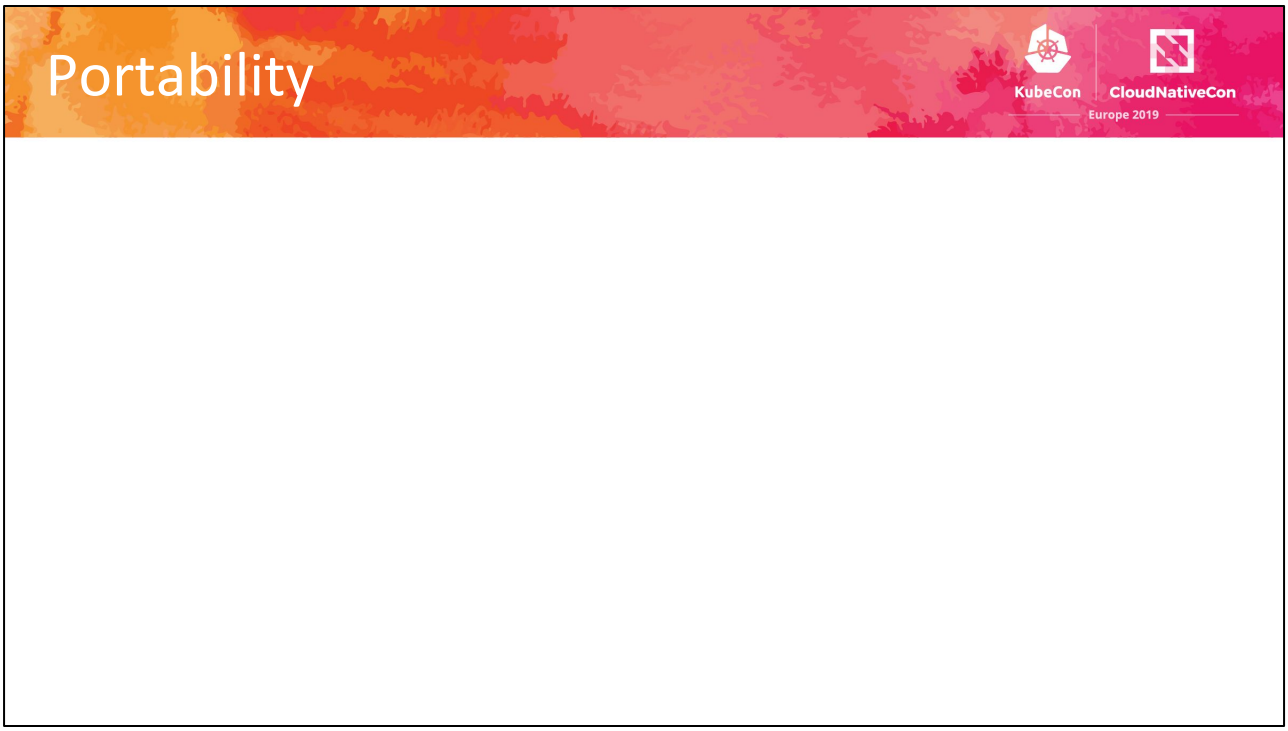
KubeCon CloudNativeCon Europe 2019

Complex syntax tree -- needs a decorator pattern

- Better API machinery?
- map[string]string vs Raw Objects (inline CRDs) vs CRD link

Work through examples and UX for users

Portability



Now, let's talk about portability -- where we propose a new strategy should be pursued.

In terms of portability, we are proposing a slightly different model than what k8s has done before:

Why?

Lots of k8s implementations have grown up along with k8s: e.g. container runtimes.
Thus, they are easier to bring into line with portable standard.
Storage classes -- have very orthogonal features

proxy implement predate k8s, feature sets are pretty heterogeneous.

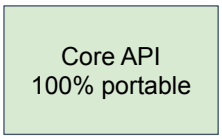
What we want is a force that will drive features inwards to the core API.
Annotations have not been very easy (impossible?) to corral among the various implementations.

Portability

KubeCon CloudNativeCon Europe 2019

Core

MUST be supported.



Proposal: expanding rings of support.

Core API

-- guaranteed portable across providers
-- really the minimal set necessary to be “an ingress”

Portability

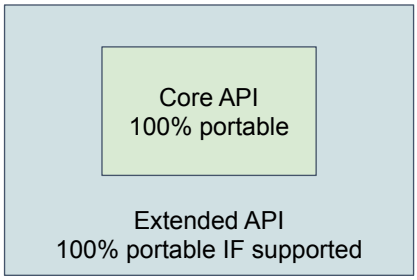
KubeCon CloudNativeCon Europe 2019

Core

MUST be supported.

Extended

Feature by feature.
MAYBE supported, but
MUST be portable.
Part of k8s API schema.



Extended features

-- portable IF supported

Portability

KubeCon CloudNativeCon Europe 2019

Core

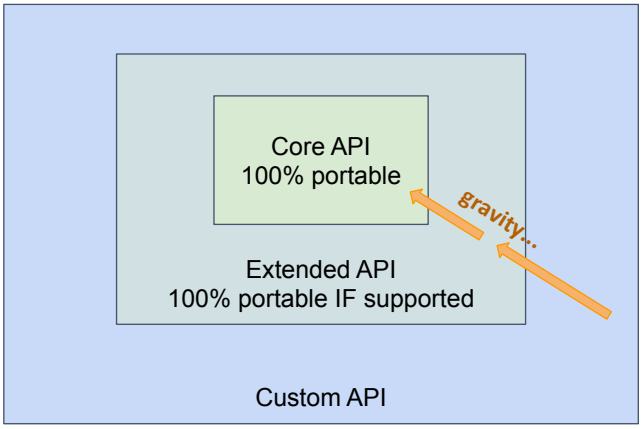
MUST be supported.

Extended

Feature by feature.
MAYBE supported, but
MUST be portable.
Part of k8s API schema.

Custom

No guarantee for portability,
No k8s API schema.



Create “gravity” to pull features into the core

One issue we have seen with the previous ingress -- there is simply no force that makes annotations uniform, and it’s actually quite hard to chase down all of the implementations to make sure that the extended features are compatible.

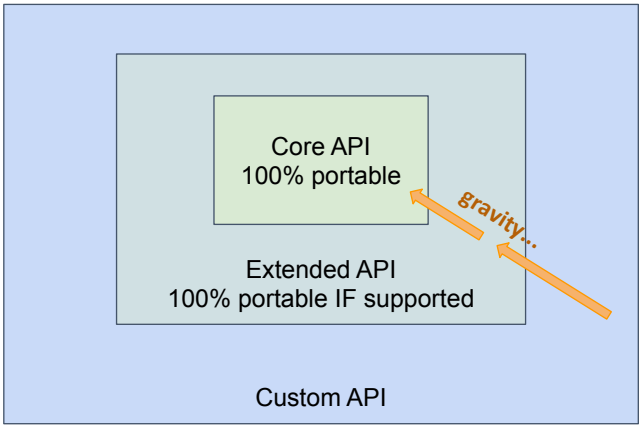
Portability

KubeCon CloudNativeCon Europe 2019

Enforcement by conformance tests.

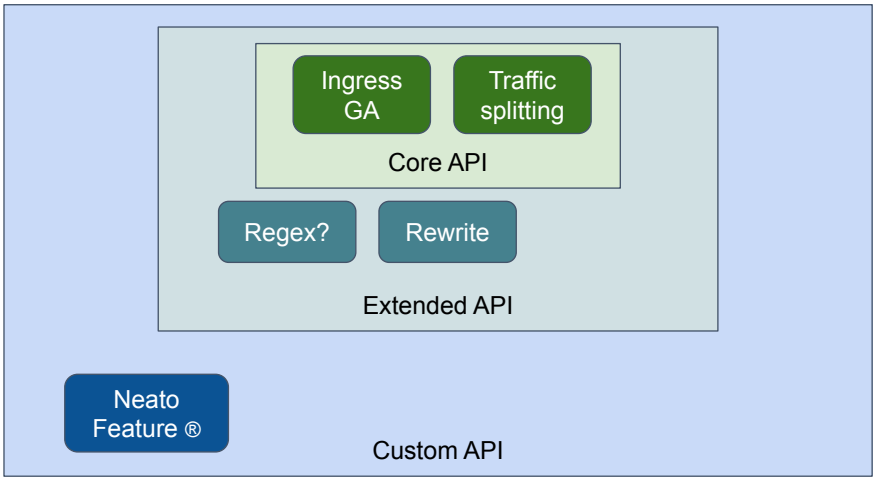
Extended feature definition **requires** self-contained conformance.

Require all extended features be checked statically?



How to make sure these things are actually portable?

Portability



Future directions

Interesting alternative backends:

- Storage bucket
- Multi-cluster Services

Multicluster Service

Multi-Cluster Services: Use Cases

- Canary - Deploy new version of application in one Kubernetes cluster in one locality before rolling it out worldwide.
- Low latency - Users get routed to the application backends that are closest to them geographically.
- High availability - Users requests are served even if one cluster holding application backends is completely down.
- Hybrid - Application spans multiple cloud providers or on-prem

Note: All these use cases are in the context of a global application.

Canary - When shipping a new version, you want to make sure you canary the version to a locality in order to isolate the changes and verify they work. Once you have done X amount of verification you need to do, you can slowly roll out the new version to the rest of your localities.

Low Latency - In most cases, you want to ensure your users experience the lowest latency when talking to your application backends. This means that a user is Australia is routed to the backends that are closest to Australia.

High Availability - You want to ensure that users do not experience downtime when your application infrastructure is broken. If your backends in Australia are down, you want to ensure that they are still getting served traffic but perhaps from a location that is a bit further away. In general for an HA application, you are most likely willing to sacrifice a increase a little bit in increased latency for no downtime for your customers.

Hybrid - You could have some of your application backends still in on-prem for business or compliance reasons

Multi-Cluster Services: User Journey

Persona: Admin

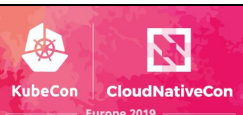
- I want **L7 load balancing across** a global service that is deployed on Kubernetes clusters.
- I want the ability to add / remove clusters from this load balancing based on business requirements.
- I want teams in my organization (service owners) to manage their own ingress traffic but maintain a clear boundary between the “admin” and the “Kubernetes end user”.
- I want my teams to use a native Kubernetes API to leverage default features (e.g namespacing, labels) & to reduce the learning curve.

We just looked at some use cases for why you would want multi-cluster services, let’s look at an example user journey to demonstrate at a high-level how it can be done.

For this user journey we want to highlight the following;

1. These API’s are solving north-south, not east-west. By north-south we mean traffic that is ingressing / egressing outside of your “service mesh”. East-west is traffic within your “mesh”.

Multi-Cluster Services: APIs

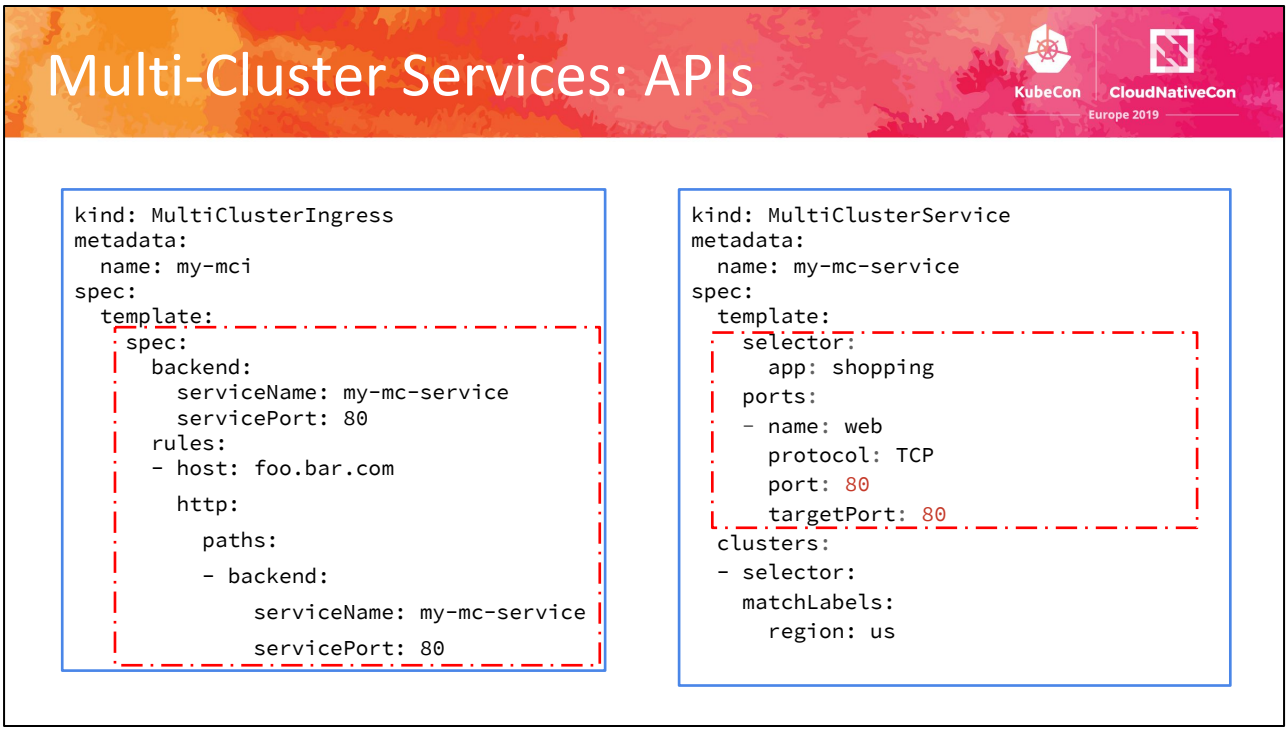



```
kind: MultiClusterIngress
metadata:
  name: my-mci
spec:
  template:
    spec:
      backend:
        serviceName: my-mc-service
        servicePort: 80
      rules:
      - host: foo.bar.com
        http:
          paths:
          - backend:
              serviceName: my-mc-service
              servicePort: 80
```

```
kind: MultiClusterService
metadata:
  name: my-mc-service
spec:
  template:
    selector:
      app: shopping
    ports:
    - name: web
      protocol: TCP
      port: 80
      targetPort: 80
  clusters:
  - selector:
    matchLabels:
      region: us
```

From here on, we may reference abbreviate MultiClusterIngress and MultiClusterService to MCI & MCS, respectively

Multi-Cluster Services: APIs

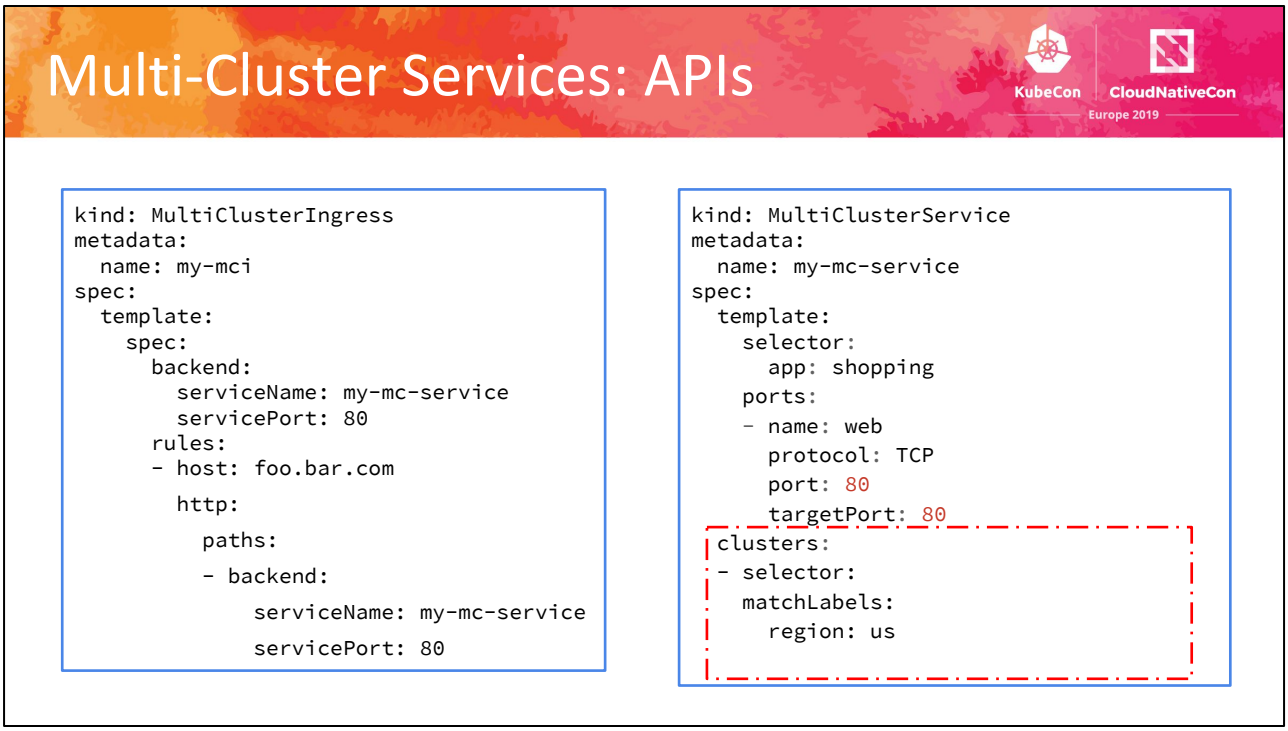


Similar to Ingress and Service. Core spec for each is the exact same. Namely, we see the same vanilla virtual hosting specification for MCI and service selector + port specification for MCS.

In fact, this proposed API simply embeds the standard “v1” Ingress & Service inside of the “template” field. Benefit of this is its very pluggable. As Ingress and Services evolve, their older counterparts can simply replaced in a “drag and drop” fashion.

Conceivably, as the API evolves, we could embed the “v2” Ingress & Service.

Multi-Cluster Services: APIs

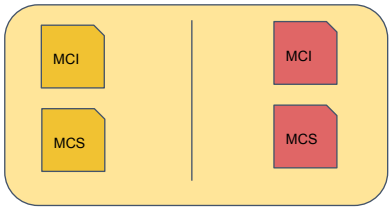


Cluster selection is the piece of MCS that is new. We leverage the power of kubernetes labels to identify which clusters are targeted by a MultiClusterService.

You might be wondering, how are you mapping a set of selected labels to actual clusters. Well, this is where a “list of clusters” implementation comes in.

Right now, there is an implementation that uses a Kubernetes API server and a Custom Resource Definition to represent a cluster called “Cluster Registry”. Ideally how you implement a “list of clusters” is up to you.

Multi-Cluster Services: Workflow

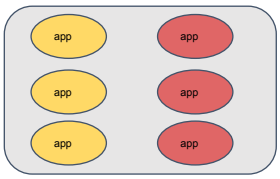


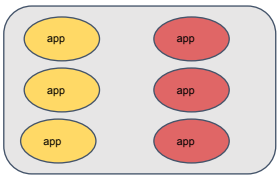
"Config Source"

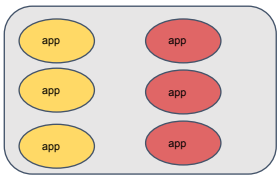
Admin

Cluster turnup / turndown

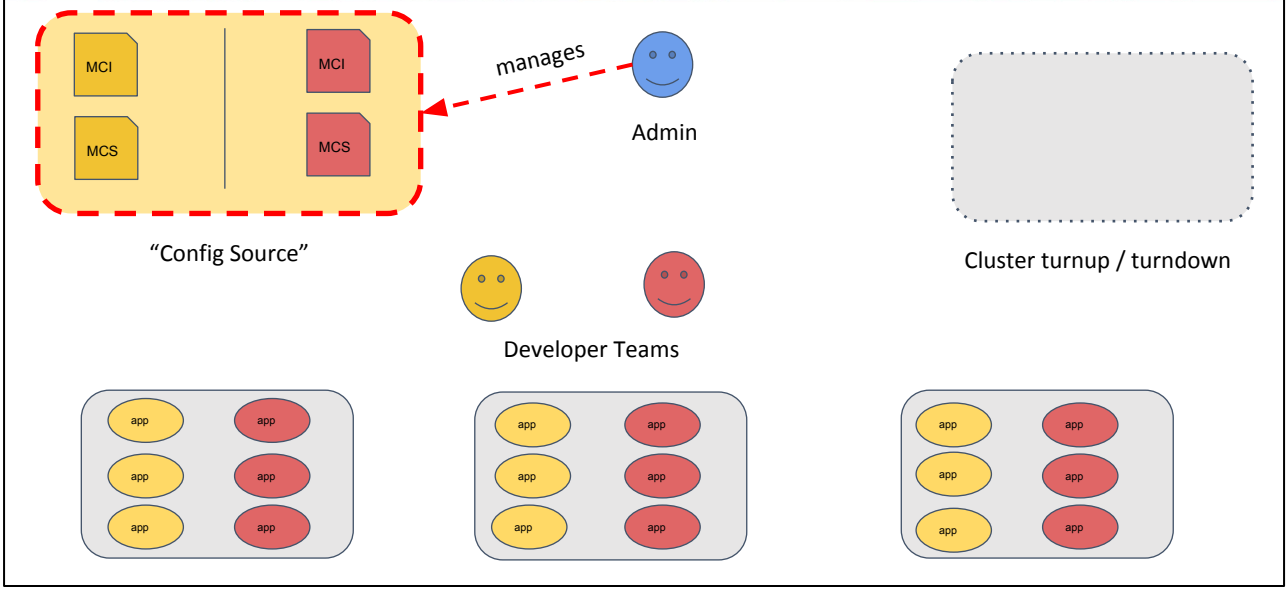
Developer Teams





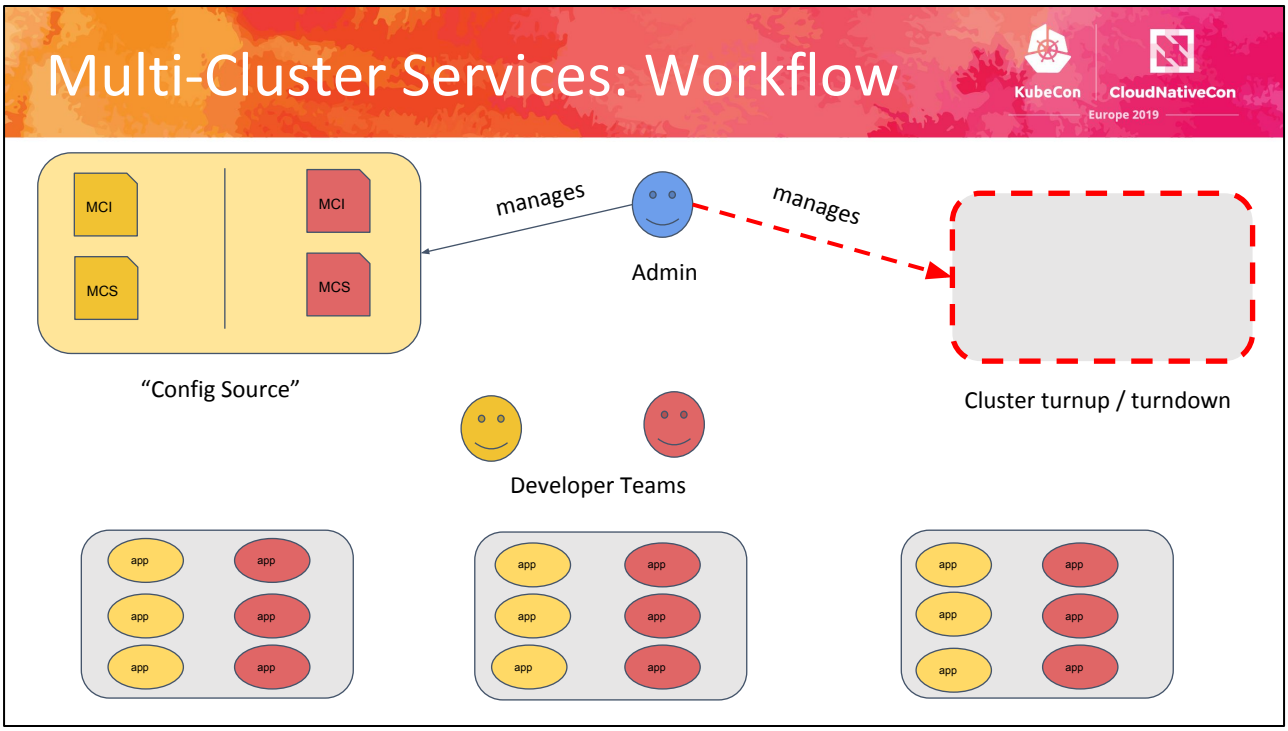


Multi-Cluster Services: Workflow



Need some place to store MCI & MCS. This is where the "Config Source" comes in. This is your standard kubernetes API server.

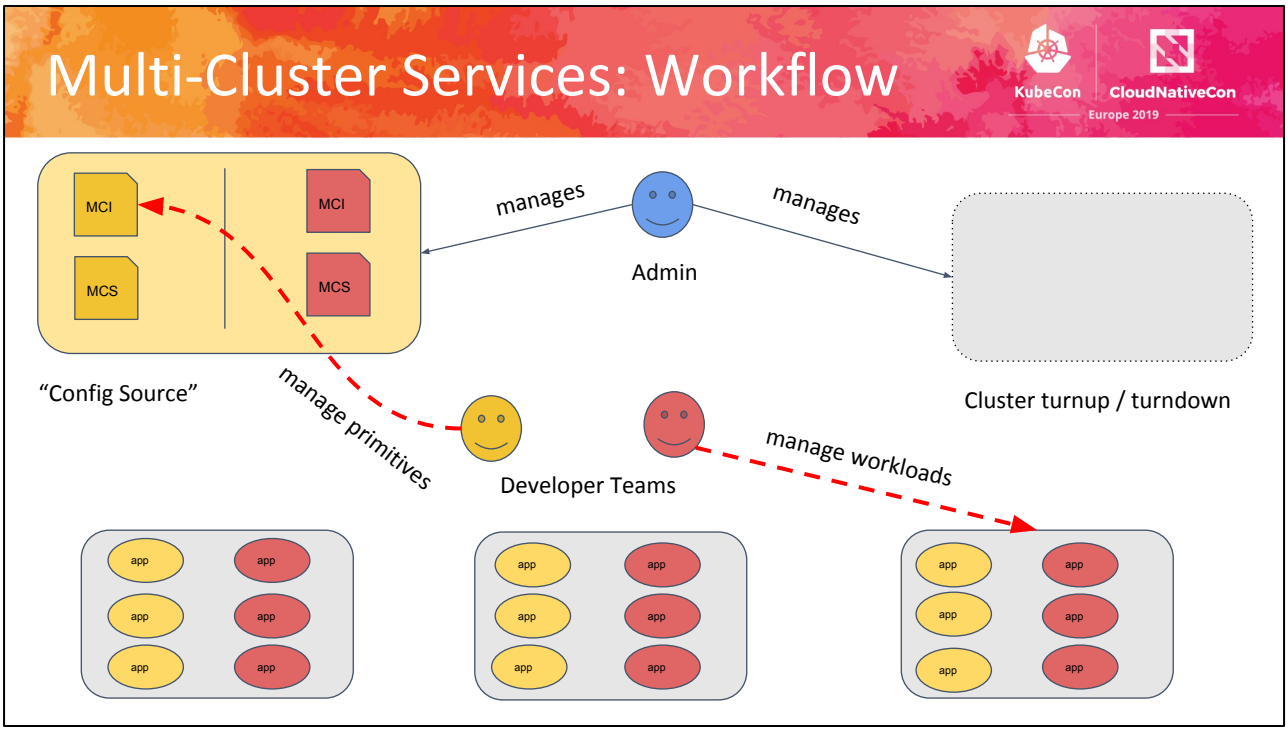
Ideally, an administrative figure maintains the API server for the "Config Source" since it is infrastructure.



You need k8s clusters to run your workloads. Ideally, the administrative figure also managed turnup / turndown of individual clusters since clusters are also infrastructure.

Note that nothing is precluding from having the config source be on the same cluster as the application backends.

Thing that is omitted here is a "list of clusters" (e.g cluster registry)



Ideally individual developers / teams manage their own workloads and their own ingress

Multi-Cluster Services: FAQ

- Why not have a MultiClusterDeployment? That alleviates the need for having to do the manual work of replicating their workload across clusters.
  - Do not want to be opinionated on how users deploy their workloads. It's not really the spirit of what the API is trying to provide.
  - Best practices in this space are divided, no one-size fits all approach.

Multi-Cluster Services: FAQ

What about Federation v2?

- Multi-cluster models and assumptions are slightly different.
- Federation v2 is opinionated on cross-cluster DNS & SD, deployment of workloads, etc.
- Federation v2 is trying to target a much wider set of use cases, multi-cluster services are an “à la carte” offering

Where does Istio’s Multicluster story fit?

- Potential for existing API’s to support use case.

Notes from federation docs:

Federation V2 is designed to allow users to deploy services and workloads to multiple clusters from a single API.

Use cases for fed v2:

- distribution of applications, services and policy to multiple clusters
- migration of applications and services and their storage between clusters
- disaster recovery for those applications and services

As Federation matures, we expect to add features dealing with storage, workload placement, etc.

There is no reference architecture (yet) for how to accomplish north-south load balancing across services using Istio API’s. However, there is potential for the existing API’s to support the use case. It’s a matter of waiting and seeing.

Conclusion

Conclusion

- Initial KEP proposal to come shortly, although we have to get V1 GA wrapped up first.
- **Feedback from community is key**
- Reference implementation

Thanks!

Rohit
rramkumar@google.com
github:rramkumar1

Bowei
bowei@google.com
github:bowei