

How Heptio Built Contour

*Any What You Can Learn From
Our Experiences*



heptio
Contour

contour

What does an Ingress Controller do?





Ingress

To step in



Ingress controllers have 99
problems, but service discovery,
cross cluster replication, outgoing
NAT, are not some of them



Why can't I just use a Service
type: LoadBalancer?



(Dave's) Philosophy of Ingress



An ingress controller should
take care of the 90% case



Getting to the 90% case

Traffic consolidation

TLS management

Abstract configuration

Reverse proxy table stakes

Path based routing

HTTP → HTTPS 3xx redirects

(limited) Request rewriting



What is Contour?



Why did Contour choose Envoy as its foundation?



Contour is the *control plane*
Envoy is the *data plane*



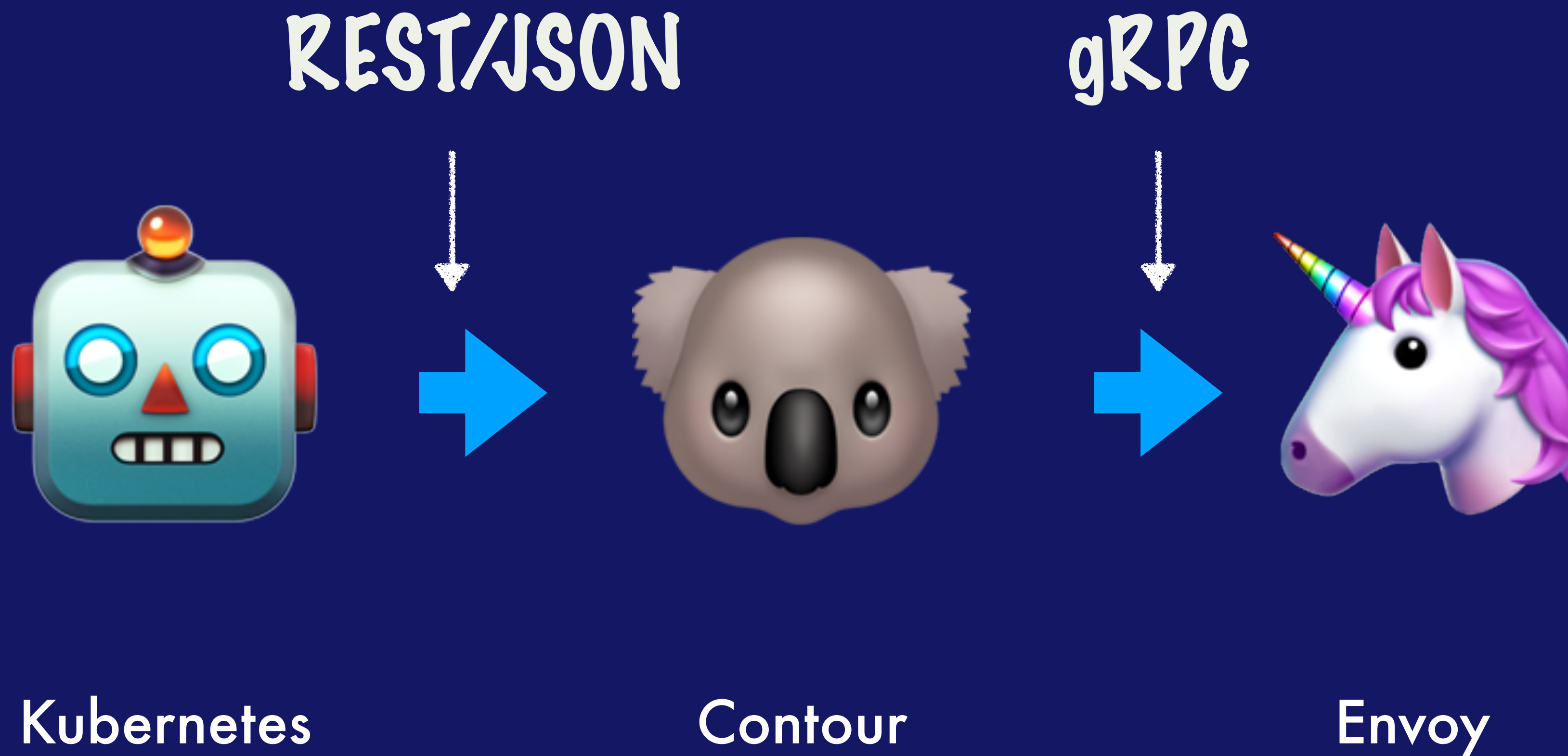
Envoy is a proxy *designed* for
dynamic configuration



Contour is the API *server*
Envoy is the API *client*



Contour Architecture Diagram



Envoy handles configuration
changes *without* reloading



Kubernetes and Envoy interoperability

	Ingress	Service	Endpoints	Secret
LDS	😊			😊
RDS	😊			
CDS		😊		
EDS			😊	

Contour, the project



As of April 30, Contour is
around 9900 LOC
2900 source, 7000 tests



Do as little as possible in
`main.main`



Ruthlessly refactor your main package to move as much code as possible to its own package



contour

└─ cmd

| └─ contour

└─ internal

| └─ contour

| └─ e2e

| └─ envoy

| └─ grpc

| └─ k8s

| └─ workgroup

└─ vendor

The actual contour command

The translator; turns k8s
objects into Envoy

Integration tests

Envoy helpers; bootstrap config

gRPC server; implements the
xDS protocol

Kubernetes helpers

Goroutine helpers



Consider `internal/` for
packages that you don't want
other projects to depend on



Goroutine management

github.com/heptio/contour/internal/workgroup



Contour needs to watch for
changes to
Ingress, Services, Endpoints,
and Secrets




Contour also needs to run a
gRPC server for Envoy, and a
HTTP server for the
/debug/pprof endpoint



```
// Group manages a set of goroutines with related lifetimes.  
type Group struct {  
    fn []func(<-chan struct{})  
}
```

**Run each function in its own
goroutine; when one exits
shut down the rest**



```
// Add adds a function to the Group.  
// The function will be executed in its own  
// goroutine when Run is called.  
func (g *Group) Add(fn func(<-chan struct{})) {  
    g.fn = append(g.fn, fn)  
}
```

**Register functions to be run
as goroutines in the group**



```
// Run executes each function registered with Add in  
// its own goroutine.  
// Run blocks until each function has returned.  
// The first function to return will trigger the closure of  
the channel
```

```
var g workgroup.Group
```

Make a new Group

```
client := newClient(*kubeconfig, *inCluster)
```

Register the gRPC server

```
k8s.WatchServices(&g, client)
```

```
k8s.WatchEndpoints(&g, client)
```

```
k8s.WatchIngress(&g, client)
```

```
k8s.WatchSecrets(&g, client)
```

**Create individual watchers
and register them with the
group**

```
g.Add(debug.Start)
```

```
g.Add(func(stop <-chan struct{} {
```

Register the /debug/pprof server

```
    addr := net.JoinHostPort(*xdsAddr, strconv.Itoa(*xdsPort))
```

```
    l, err := net.Listen("tcp", addr)
```

```
    if err != nil {
```

Start all the workers,

wait until one exits

```
        log.Errorf("could not listen on %s: %v", addr, err)
```

```
    }
```

Prefer crash only software



Handling concurrency

`github.com/heptio/contour/internal/k8s.Buffer`

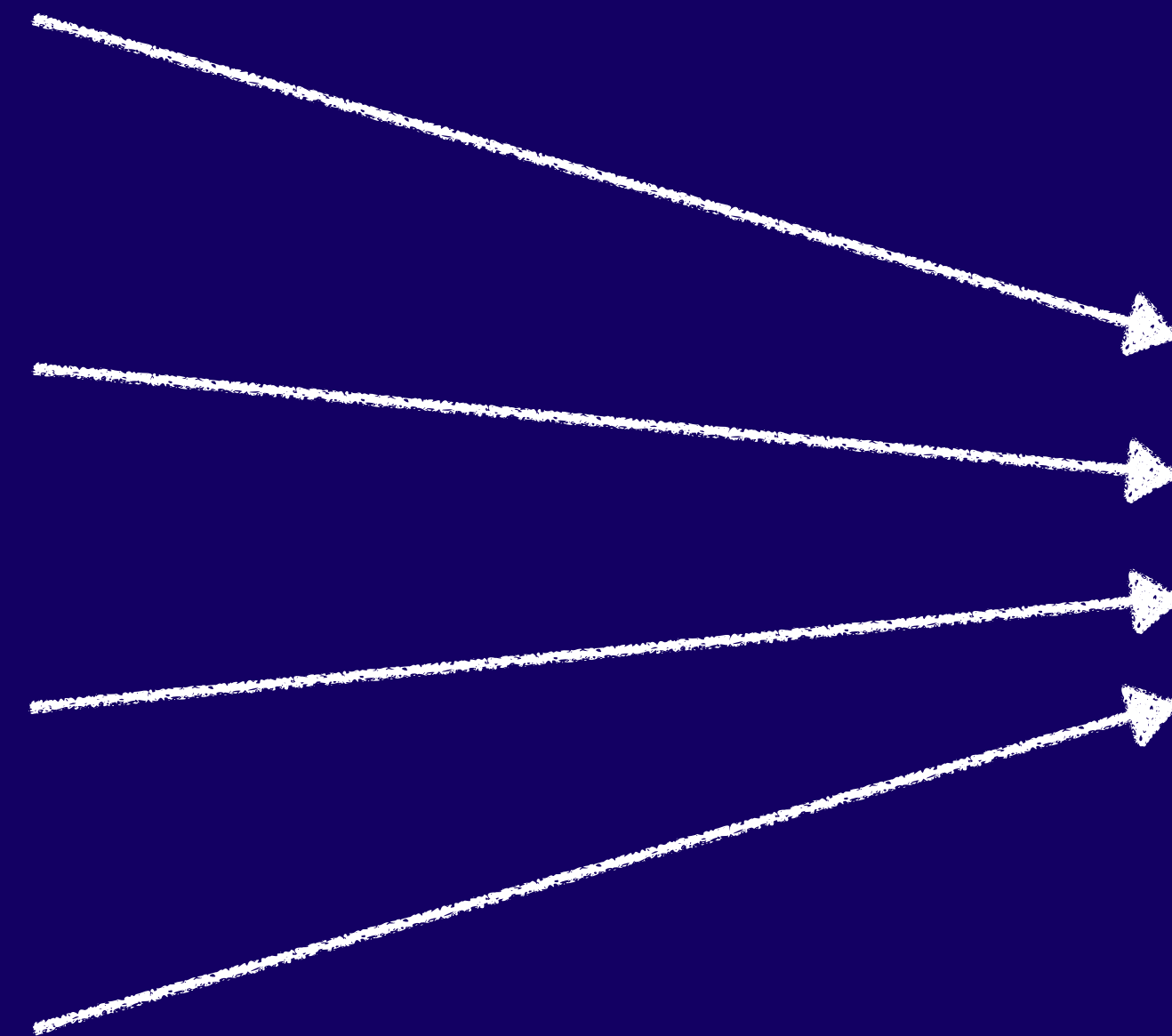


Watchers call back to Contour concurrently



Kubernetes

Ingress
Services
Endpoints
Secrets



Contour

Kubernetes and Envoy interoperability

	Ingress	Service	Endpoints	Secret
LDS	😊			😊
RDS	😊			
CDS		😊		
EDS			😊	

`sync.Mutex`
to the rescue ...



Channels to the rescue



```

func NewBuffer(g *workgroup.Group, rh cache.ResourceEventHandler, size int)
cache.ResourceEventHandler {
    buf := &buffer{
        ev:
        rh:
    }
    g.Add(buf.loop)
    return buf
}

```

Loop takes events off the channel,
calls the backing handler, until
Create group defer, register
with group

The diagram shows two arrows originating from the right side of the code. One arrow points to the 'make(chan interface{}, size)' line, and the other points to the 'g.Add(buf.loop)' line. These arrows indicate the flow of data from the function parameters to the buffer struct and then to the group.

```

func (b *buffer) OnAdd(obj interface{}) {
    b.send(&addEvent{obj})
}

```

```

func (b *buffer) OnUpdate(oldObj, newObj interface{}) {
    b.send(&updateEvent{oldObj, newObj})
}

```

Buffer fulfills the
ResourceEventHandler
to a channel interface

The diagram shows two arrows originating from the right side of the code. One arrow points to the 'b.send(&addEvent{obj})' line, and the other points to the 'b.send(&updateEvent{oldObj, newObj})' line. These arrows indicate the flow of data from the methods to the buffer struct and then to the channel interface.

```

func (b *buffer) OnDelete(obj interface{}) {
    b.send(&deleteEvent{obj})
}

```

Dependency management with dep



Gopkg.toml

```
[[constraint]]  
  name="k8s.io/client-go"  
  version="v7.0.0"  
  
[[constraint]]  
  name="k8s.io/api"  
  branch="release-1.10"  
  
[[constraint]]  
  name="k8s.io/apimachinery"  
  branch="release-1.10"
```



We don't commit vendor / to
our repository



```
% go get -d github.com/heptio/contour  
% cd $GOPATH/src/github.com/heptio/contour  
% dep ensure -vendor-only
```



What about vgo?

*TL;DR vgo can't handle
client-go, yet*



Living with Docker



`.dockerignore`



When you run `docker build` it
copies *everything* in your
working directory to the docker
daemon 🤪



```
% cat .dockerignore  
/.git  
/vendor
```




```
% cat Dockerfile
FROM golang:1.10
WORKDIR /go/src/github.com/heptio/contour
```

```
RUN go get github.com/golang/dep/cmd/dep
COPY Gopkg.toml Gopkg.lock ./
RUN dep ensure -v -vendor-only
```

**only runs if Gopkg.toml or
Gopkg.lock have changed**

```
COPY cmd cmd
COPY internal internal
RUN CGO_ENABLED=0 GOOS=linux go install -ldflags="-w -s" -v
github.com/heptio/contour/cmd/contour
```

```
FROM alpine:latest
RUN apk --no-cache add ca-certificates
COPY --from=0 /go/bin/contour /bin/contour
```



contour — dfc@Daves-MacBook-Pro: ~/src/github.com/heptio/contour — -bash — 79x24

(reverse-i-search)`':

Try to avoid the
docker build && docker push
workflow in your inner loop



contour — dfc@Daves-MacBook-Pro: ~/src/github.com/heptio/contour — -bash — 79x24

Daves-MacBook-Pro(~/src/github.com/heptio/contour) %

Local development against a live cluster



%

%

I

Functional Testing



~~Functional~~ End to End tests are terrible

- Slow ...
- Which leads to effort expended to run them in parallel ...
- Which tends to make them flakey ...
- IMO end to end tests become a boat anchor on velocity



So, I put them off as long as I
could

But, there are scenarios that
unit tests cannot cover ...

... because there is a moderate
impedance mismatch between
Kubernetes and Envoy

We need to model the
sequence of interactions
between Kubernetes and Envoy



What are Contour's e2e tests *not* testing?

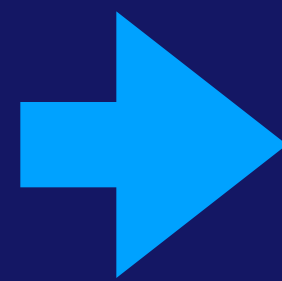
- We are *not* testing Kubernetes (we assume it works)
- We are *not* testing Envoy (we hope someone else did that)



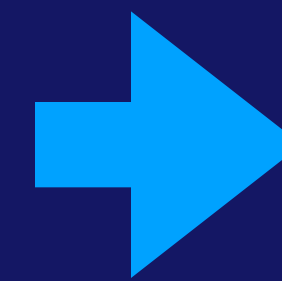
Contour Architecture Diagram



Kubernetes



Contour



Envoy


```
func setup(t *testing.T) (cache.ResourceEventHandler, *grpc.ClientConn, func()) {
    log := logrus.New()
    log.Out = &testWriter{t}

    tr := &contour.Translator{
        FieldLogger: log,
    }

    l, err := net.Listen("tcp", "127.0.0.1:0")
    check(t, err)
    var wg sync.WaitGroup
    wg.Add(1)
    srv := cgrpc.NewAPI(log, tr)
    go func() {
        defer wg.Done()
        srv.Serve(l)
    }()
    cc, err := grpc.Dial(l.Addr().String(), grpc.WithInsecure())
    check(t, err)
    return tr, cc, func() {
        // close client connection
    }
}
```

Create a new gRPC client and dial our server

Create a new gRPC server and bind it to a loopback address, return a resource handler, client, and shutdown function

Resource handler, the input

// pathological hard case, one service is removed, the other
// is moved to a different port, and its name removed.

```
func TestClusterRenameUpdateDelete(t *testing.T) {  
    rh, cc, done := setup(t)  
    defer done()
```

gRPC client, the output

```
    s1 := service("default", "kuard",  
        v1.ServicePort{  
            Name:      "http",  
            Protocol:  "TCP",  
            Port:      80,  
            TargetPort: intstr.FromInt(8080),  
        },  
        v1.ServicePort{  
            Name:      "https",  
            Protocol:  "TCP"
```

Insert s1 into
API server

Query Contour
for the results

Low lights 🙄

- Verbose, even with lots of helpers ...
- ... but at least it's explicit; after this event from the API, I expect this state.



High Lights 🤗

- High success rate in reproducing bugs reported in the field.
- Easy way for contributors to add tests.
- Easy to model failing scenarios which enables Test Driven Development 🎉
- Avoid docker push && k delete po -l app=contour style debugging



Thank you for listening!

Questions?

@davecheney – dfc@heptio.com

