

# Agenda

- Securing Kubernetes
  - Setup our cluster
  - Kubernetes architecture overview
  - Securing Kubernetes components
  - Pod security context
  - Rabc, roles and service accounts
  - mTLS with Istio



# Course assumptions

## Prior knowledge

- A basic understanding of Linux
- Be able to read bash scripts
- Understand what a container is
- A basic understanding of Kubernetes



# Set up our cluster



@scottcoulton

# Kubernetes architecture overview



@scottcoulton

# Kubernetes components

Kubernetes is broken down into two node types.

- Master node
- Worker node



# Master node

A master node is responsible for

- Running the control plane
- Scheduling workloads
- Security controls



# Worker node

A worker node is responsible for

- Running workloads



# Master node

A master nodes components (control plane)

- kube-apiserver
- etcd
- kube-scheduler
- kube-controller-manager
- cloud-controller-manager



# Worker node

A worker nodes components

- kubelet
- Kube-proxy



@scottcoulton

# Kube-apiserver

The kube-apiserver is responsible for

- The entry point into the cluster
- It exposes the Kubernetes API
- It's a REST service
- Validates and configures data for the api objects

# etcd

Consistent and highly-available key value store used as  
Kubernetes' backing store for all cluster data

# kube-scheduler

Kube-scheduler is responsible for

- watches newly created pods that have no node assigned, and selects a node for them to run on

Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines

# kube-controller-manager

Kube-controller-manager is responsible for

- Node Controller: Responsible for noticing and responding when nodes go down.
- Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
- Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods)
- Service Account & Token Controllers: Create default accounts and API access tokens for new namespaces.

# cloud-controller-manager

Cloud-controller-manager is responsible for

- For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- For setting up routes in the underlying cloud infrastructure
- For creating, updating and deleting cloud provider load balancers
- For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes

# kubelet

Kubelet is responsible for

- All containers in a pod are running

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy

# Kube-proxy

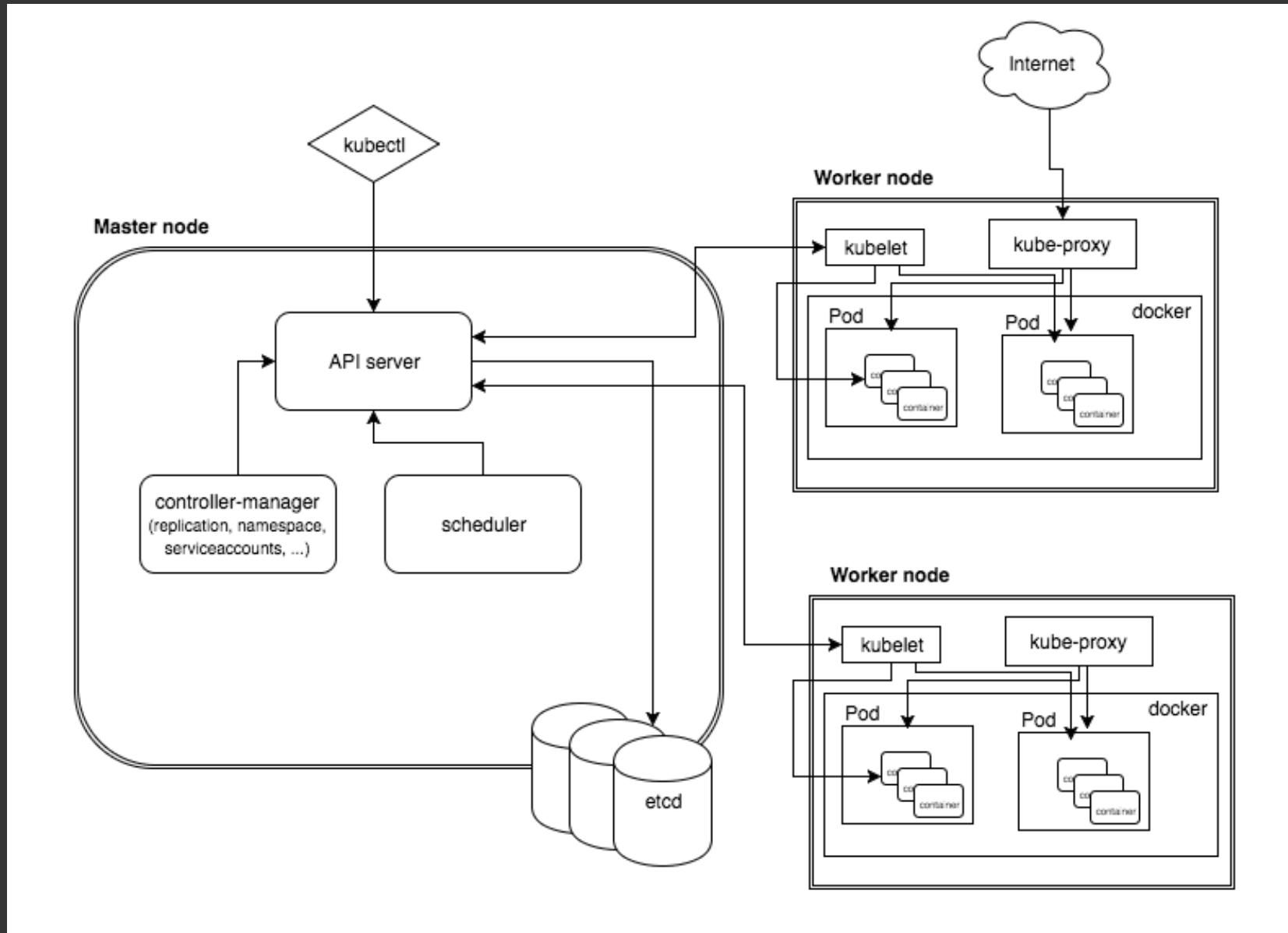
This reflects services as defined in the Kubernetes API on each node and can do simple TCP, UDP, and SCTP stream forwarding or round robin TCP, UDP, and SCTP forwarding across a set of backends

# Container runtimes

Kubernetes can use different container runtimes

- Docker
- Moby
- Containerd
- Cri-o

# Kubernetes architecture



# Securing Kubernetes components



@scottcoulton

# Networks ports need for Kubernetes

Port	Process	Description
4149/TCP	kubelet	Default cAdvisor port used to query container metrics
10250/TCP	kubelet	API which allows full node access
10255/TCP	kubelet	Unauthenticated read-only port, allowing access to node state
10256/TCP	kube-proxy	Health check server for Kube Proxy
9099/TCP	calico-felix	Health check server for Calico (if using Calico/Canal)
6443/TCP	kube-apiserver	Kubernetes API port

# Securing the API

In this section we will look at some flags that should be set on the API server and why

- Authorization mode & anonymous auth
- Insecure Port
- Using certificates
- AdmissionController

# Authorization mode and anonymous auth

By default anonymous auth is turned on

Safe configurations on the kube api flags

- --authorization-mode=RBAC

Bad configuration would be

- --anonymous-auth=true

# Insecure port

Running Kube API on a insecure port is not recommended  
Safe configurations on the kube api flags

- --insecure-port=0

Bad configuration would be

- --insecure-port=6443

# Using certificates

Running Kube API without x.509 certificates not recommended

Safe configurations on the kube api flags

- kube-apiserver \
- <... other flags> \
- --client-ca-file=/path/to/ca.crt \
- --tls-cert-file=/path/to/server.crt \
- --tls-private-key-file=/path/to/server.key

# Admission controllers

An admission controller is a piece of code that intercepts requests to the Kubernetes API server prior to persistence of the object, but after the request is authenticated and authorized.

# Admission controllers

There are a couple of recommended ways you could handle your admission controllers

Safe configurations on the kube api flags

- --admission-control=...,DenyEscalatingExec
- --admission-control=...,PodSecurityPolicy

# Making sure we have the right configurations

There are tools and documentation to make this process easier

- CIS Kubernetes bench mark

<https://www.cisecurity.org/benchmark/kubernetes/>

- Aqua securities Kube bench

<https://github.com/aquasecurity/kube-bench>

# Pod security context



@scottcoulton

# Pod security context

Just because we are using Kubernetes means we are secure by default.

There are a lot of good security features in Kubernetes that are not turned on.

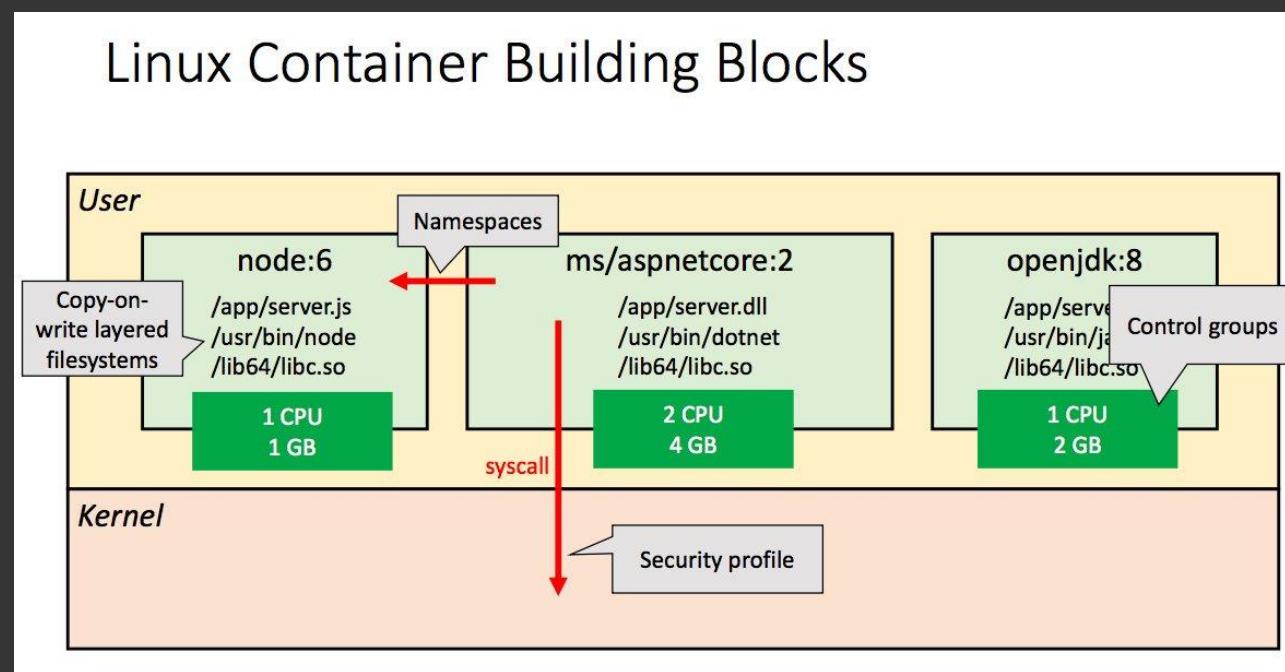
# Pod security context

Pod security is an abstraction from the Linux security subsystem.

- Apparmor
- Selinux
- Secomp

# Pod security context

A container is a process that is isolated via kernel namespaces and cgroups



@scottcoulton

# Pod security context

In Azure our pods are talking to the kernel via Moby

Today we are going to look at three of the important default policies.

In a production environment I would personally use seccomp

[https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp\\_filter.html](https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html)

With *libseccomp*

<https://github.com/seccomp/libseccomp>

# Pod security context

The three default policies are

- runAsUser
- readOnlyRootFilesystem
- allowPrivilegeEscalation

# Pod security context

We are going to run a series of deployments with our webapp changing the security context each time.

First, we must see why security context are so important.

# Deploy our webapp

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: webapp-deployment
spec:
  selector:
    matchLabels:
      app: webapp
  replicas: 1
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
        - name: webapp
          image: scottyc/webapp:latest
      ports:
        - containerPort: 3000
          hostPort: 3000
EOF
```

# Check your deployment

`kubectl get svc` (to get our public ip)

Check your browser with <http://<public-ip>:3000>

# The hack

```
kubectl get pods | grep webapp (to get your pod name)
```

We will then get a shell inside our container  
kubectl exec -it <pod\_name> sh

Change the index.html file

```
cd static && vim index.html
```

Change the url on line 16

```
to https://media.giphy.com/media/DBfYJqH5AokgM/  
giphy.gif
```

# The hack

See what user is the pod running as

**whoami** from inside the pods terminal

# Check your deployment

`kubectl get svc` (to get our public ip)

Check your browser again with <http://<public-ip>:3000>

Make sure you refresh your browser

# Defining pod security context

Pod security policy is set in your deployment yaml under:

```
spec:  
  containers:  
    securityContext:
```

# Adding runAsUser policy

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: webapp-deployment
spec:
  selector:
    matchLabels:
      app: webapp
  replicas: 1
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
        - name: webapp
          image: scottyc/webapp:latest
      ports:
        - containerPort: 3000
          hostPort: 3000
      securityContext:
        runAsUser: 1000
EOF
```

@scottcoulton

# Check the user running in the container

```
kubectl get pods | grep webapp (to get your pod name)
```

We will then get a shell inside our container

```
kubectl exec -it <pod_name> sh
```

Check the user with

```
whoami
```

Now let's test if we can change the file ? Or change to root

# Clean up

```
kubectl delete deployments.apps webapp-deployment
```

# Adding readOnlyRootFilesystem policy

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: webapp-deployment
spec:
  selector:
    matchLabels:
      app: webapp
  replicas: 1
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
        - name: webapp
          image: scottyc/webapp:latest
          ports:
            - containerPort: 3000
              hostPort: 3000
      securityContext:
        readOnlyRootFilesystem: true
EOF
```

@scottcoulton

# Check the readonly file system

```
kubectl get pods | grep webapp (to get your pod name)
```

We will then get a shell inside our container

```
kubectl exec -it <pod_name> sh
```

Now let's test if we can change the file ?

# Clean up

```
kubectl delete deployments.apps webapp-deployment
```

# Adding allowPrivilegeEscalation policy

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: webapp-deployment
spec:
  selector:
    matchLabels:
      app: webapp
  replicas: 1
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
        - name: webapp
          image: scottyc/webapp:latest
      ports:
        - containerPort: 3000
          hostPort: 3000
      securityContext:
        allowPrivilegeEscalation: false
EOF
```

@scottcoulton

# Check the readonly file system

```
kubectl get pods | grep webapp (to get your pod name)
```

We will then get a shell inside our container

```
kubectl exec -it <pod_name> sh
```

Now let's test if we can change the file ? Or change to root

# Clean up

```
kubectl delete deployments.apps webapp-deployment
```

# Adding all three policies

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: webapp-deployment
spec:
  selector:
    matchLabels:
      app: webapp
  replicas: 1
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
        - name: webapp
          image: scottyc/webapp:latest
          ports:
            - containerPort: 3000
              hostPort: 3000
      securityContext:
        runAsUser: 1000
        readOnlyRootFilesystem: true
        allowPrivilegeEscalation: false
EOF
```

# Check with all policies applied

Now let's run all our tests again on this pod

# Clean up

```
kubectl delete deployments.apps webapp-deployment
```

# Rbac, roles and service accounts



@scottcoulton

# rbac

Role based access control (rbac)

- Separation of applications
- Access control for users
- Access control for applications (service accounts)

# namespaces

Namespaces are the logical separation in kubernetes

Things that are namespaced

- dns <service-name>.<namespace-name>.svc.cluster.local
- Deployments, services and pods
- Access control for applications (service accounts)
- Resource quotas
- Secrets

# namespaces

Things that are NOT namespaced

- Nodes
- Networking
- Storage

# Service accounts vs user accounts

The differences are

- User accounts are for humans. Service accounts are for processes, which run in pods.
- User accounts are intended to be global. Names must be unique across all namespaces of a cluster, future user resource will not be namespaced. Service accounts are namespaced.

# Create a namespace

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Namespace
metadata:
  name: webapp-namespace
EOF
```

# Create a service account

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: ServiceAccount
metadata:
  name: webapp-service-account
  namespace: webapp-namespace
EOF
```

# Create a role

```
cat <<EOF | kubectl apply -f -
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: webapp-role
  namespace: webapp-namespace
rules:
  - apiGroups: [""]
    resources: ["pods", "pods/log"]
    verbs: ["get", "list", "watch"]
EOF
```

# Create a role binding

```
cat <<EOF | kubectl apply -f -
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: webapp-role-binding
  namespace: webapp-namespace
subjects:
- kind: ServiceAccount
  name: webapp-service-account
  namespace: webapp-namespace
roleRef:
  kind: Role
  name: webapp-role
  apiGroup: rbac.authorization.k8s.io
EOF
```

# Deploying an application to our namespace

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-deployment
  namespace: webapp-namespace
spec:
  selector:
    matchLabels:
      app: webapp
  replicas: 1
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
        - name: webapp
          image: scottyc/webapp:latest
          ports:
            - containerPort: 3000
              hostPort: 3000
EOF
```

@scottcoulton

# Now let's set up kubectl to use the service account



@scottcoulton

# We need the secret for the service account

```
SECRET_NAME=$(kubectl get sa webapp-service-account --namespace webapp-namespace -o json | jq -r '.secrets[].name')
```

We will get the ca

```
kubectl get secret --namespace webapp-namespace  
"${SECRET_NAME}" -o json | jq -r  
'.data["ca.crt"]' | base64 --decode > ca.crt
```

# We will get the user token

```
USER_TOKEN=$(kubectl get secret --namespace  
webapp-namespace "${SECRET_NAME}" -o json | jq  
-r '.data["token"]' | base64 --decode)
```

@scottcoulton

# Then we will create our kubeconfig file

```
context=$(kubectl config current-context)

CLUSTER_NAME=$(kubectl config get-contexts "$context" | awk '{print $3}' | tail -n 1)

ENDPOINT=$(kubectl config view -o jsonpath=".clusters[?(@.name == \"${CLUSTER_NAME}\")].cluster.server")

kubectl config set-cluster "${CLUSTER_NAME}" --kubeconfig=admin.conf --server="${ENDPOINT}" --certificate-authority=ca.crt --embed-certs=true

kubectl config set-credentials "webapp-service-account-webapp-namespace-${CLUSTER_NAME}" --kubeconfig=admin.conf --token="${USER_TOKEN}"

kubectl config set-context "webapp-service-account-webapp-namespace-${CLUSTER_NAME}" --kubeconfig=admin.conf --cluster="${CLUSTER_NAME}" --user="webapp-service-account-webapp-namespace-${CLUSTER_NAME}" --namespace webapp-namespace

kubectl config use-context "webapp-service-account-webapp-namespace-${CLUSTER_NAME}" --kubeconfig="${KUBECFG_FILE_NAME}"
```

# Export the file to use in the current terminal

```
export KUBECONFIG=admin.conf
```

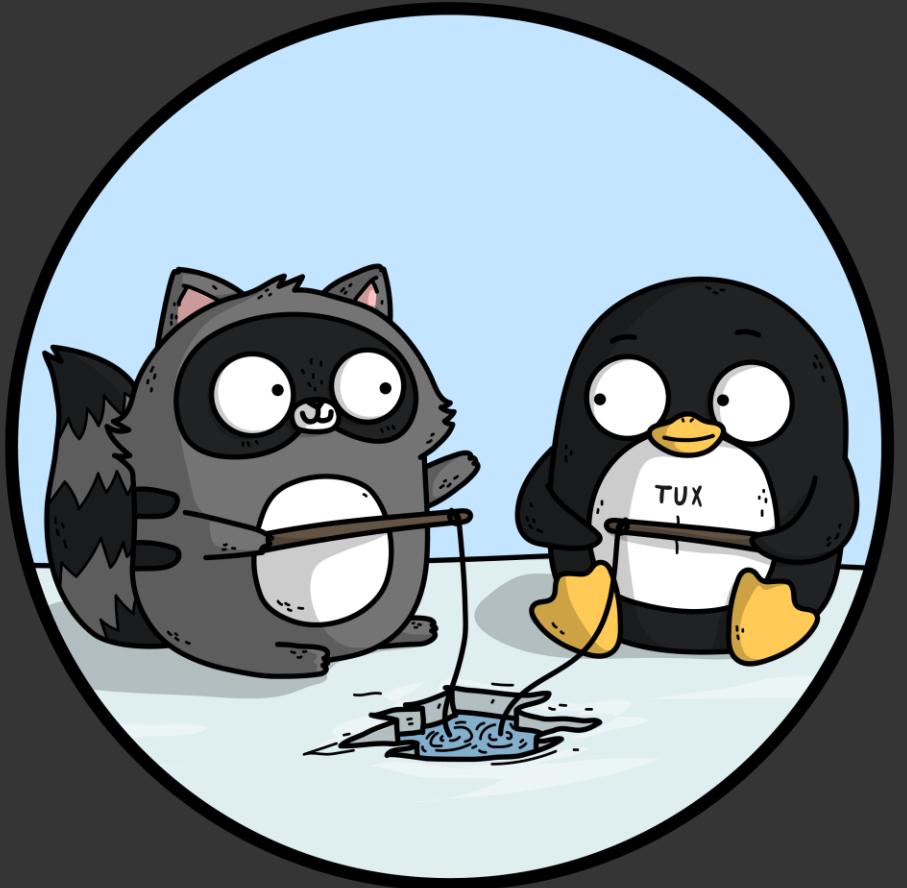
@scottcoulton

# Use your kubectl commands to see what you have access too



@scottcoulton

# Istio 101



@scottcoulton

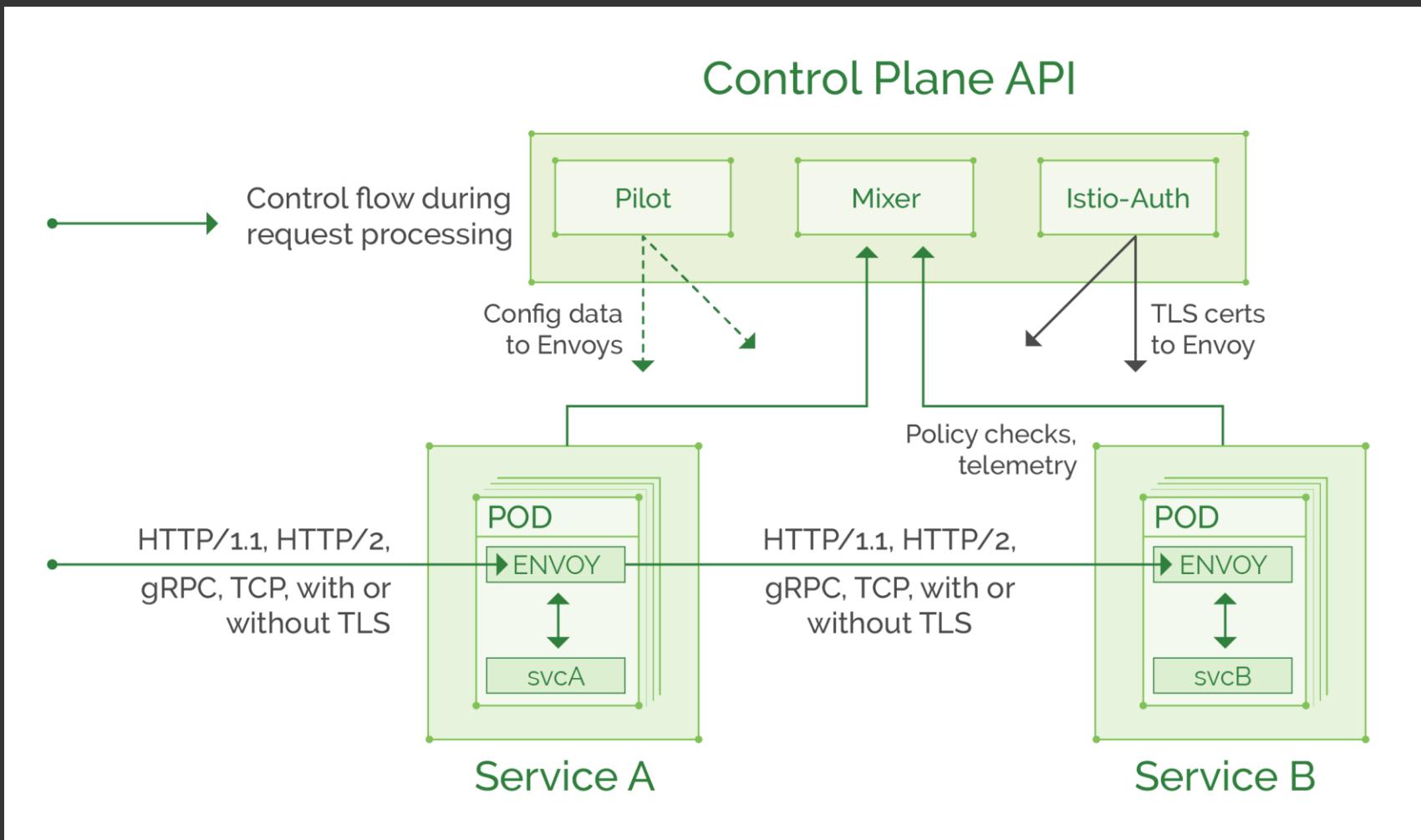
# Istio 101

What is Istio trying to solve ?

- Application routing
- Ingress/Egress
- authentication
- authorisation



# Istio architecture



# Before we go on

A screenshot of a Twitter post from user @davecheney. The post features a profile picture of a cartoon owl, the name 'Dave Cheney' in bold black text, and the handle '@davecheney' in smaller grey text. The tweet content is a question: 'Service mesh real talk; how are y'all dealing with the latency overhead of CPU switching every packet in user space?'. Below the tweet is the timestamp '9:30 AM · Feb 9, 2019 · Twitter Web Client'. A horizontal line separates the tweet from the engagement metrics. The metrics show '46 Retweets' and '186 Likes'. Below the metrics are four interaction icons: a speech bubble for replies, a retweet symbol, a heart for likes, and an upward arrow for sharing.

Dave Cheney  
@davecheney

Service mesh real talk; how are y'all dealing with the latency overhead of CPU switching every packet in user space?

9:30 AM · Feb 9, 2019 · Twitter Web Client

---

46 Retweets 186 Likes

Reply Retweet Like Share

@scottcoulton

# Before we go on

Some security gotchas

- Istio runs as user 1337
- It needs CAP\_NET\_ADMIN (This is privileged)
- You will need to adjust your pod security policies accordingly

# My two cents

Before we go to all the good things istio gives you.

You need to weigh up the following

- The extra complexity istio adds to the stack
- Is the over head on the application worth it



# Istio components



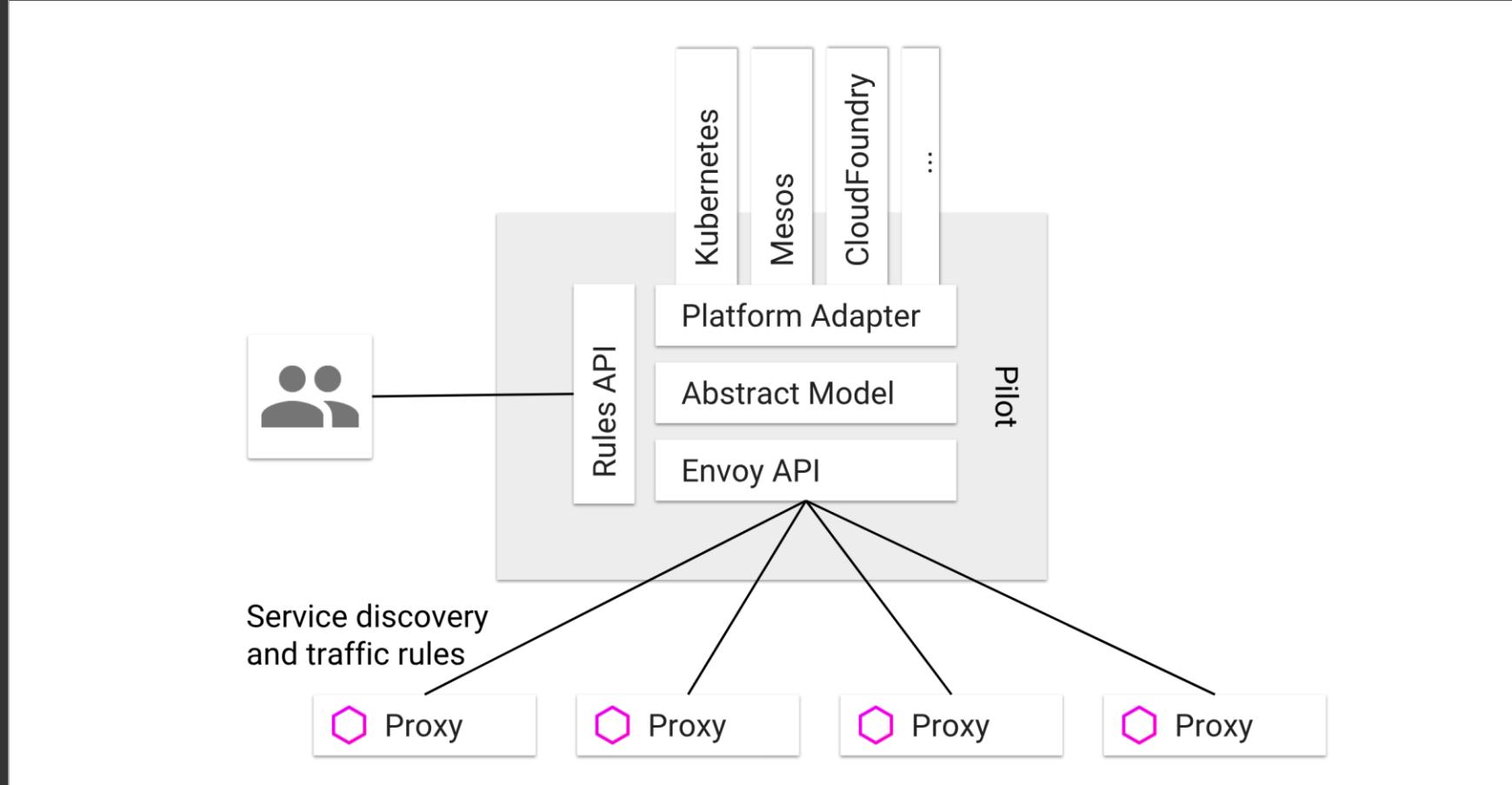
@scottcoulton

# Istio components

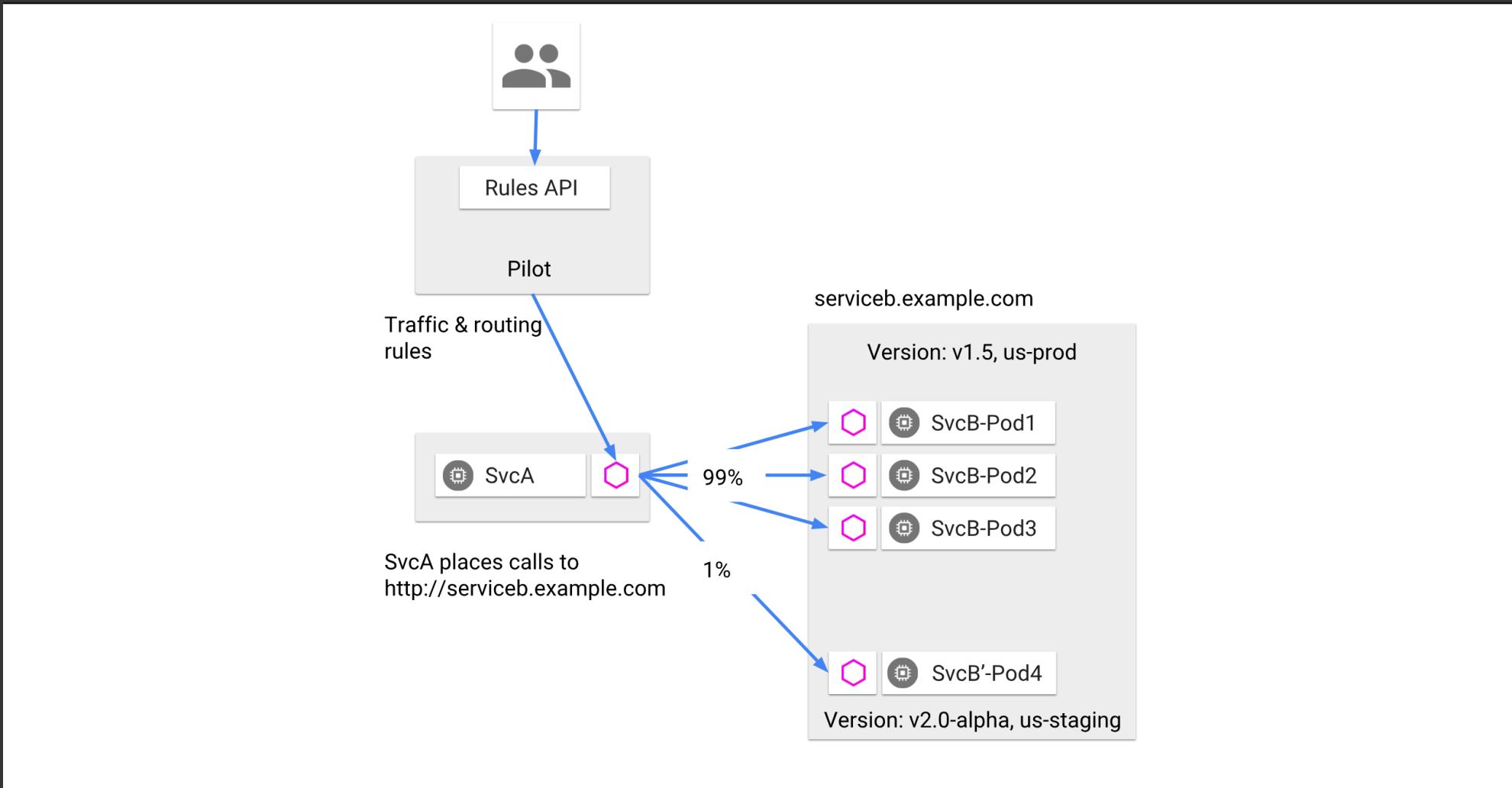
Istio is made up of the following components

- Envoy and Piolit (ingress, egress & authentication policies)
- Citadel (certificate management)
- Mixer (authorization and auditing)

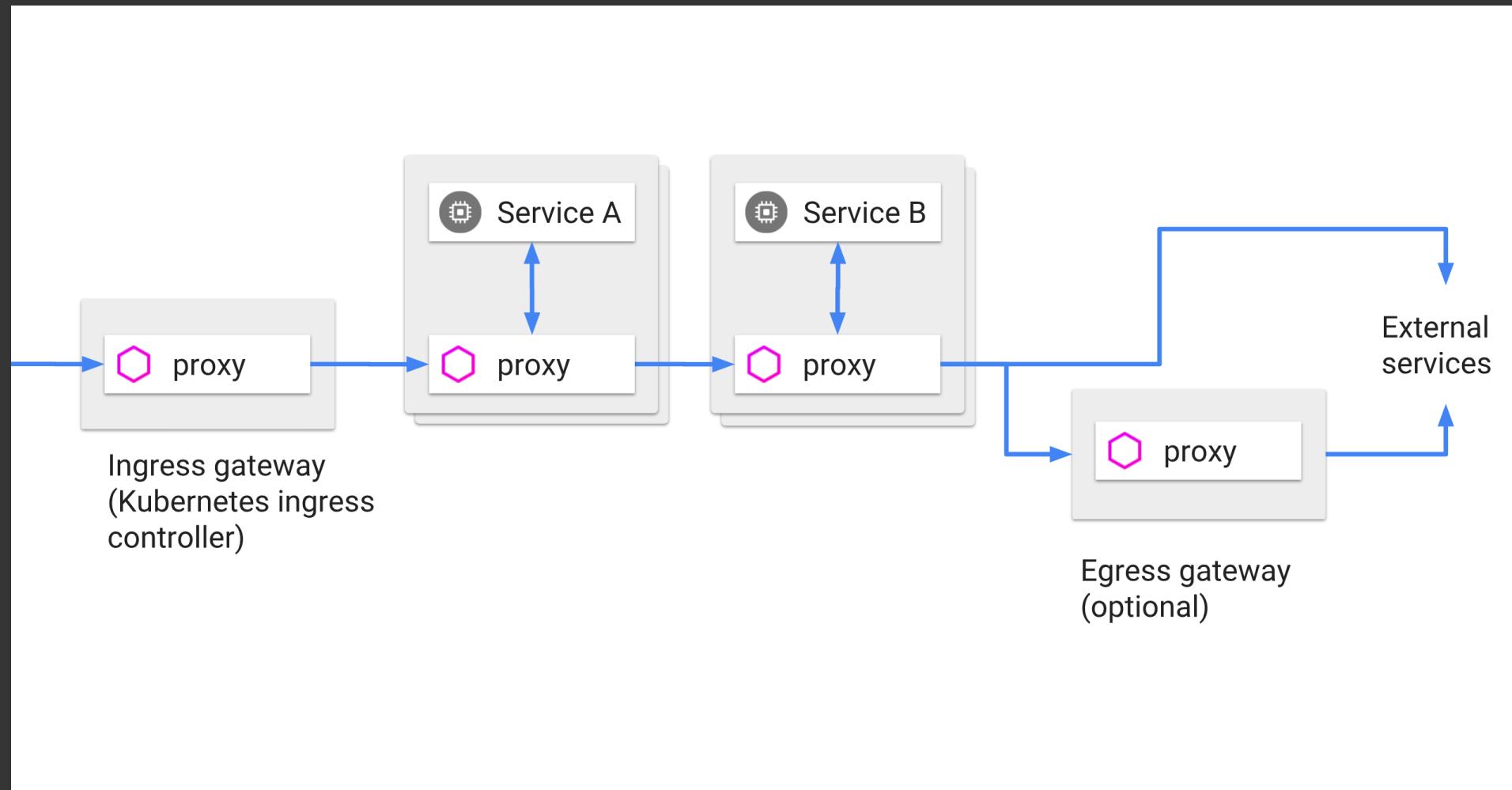
# Envoy and Piolt



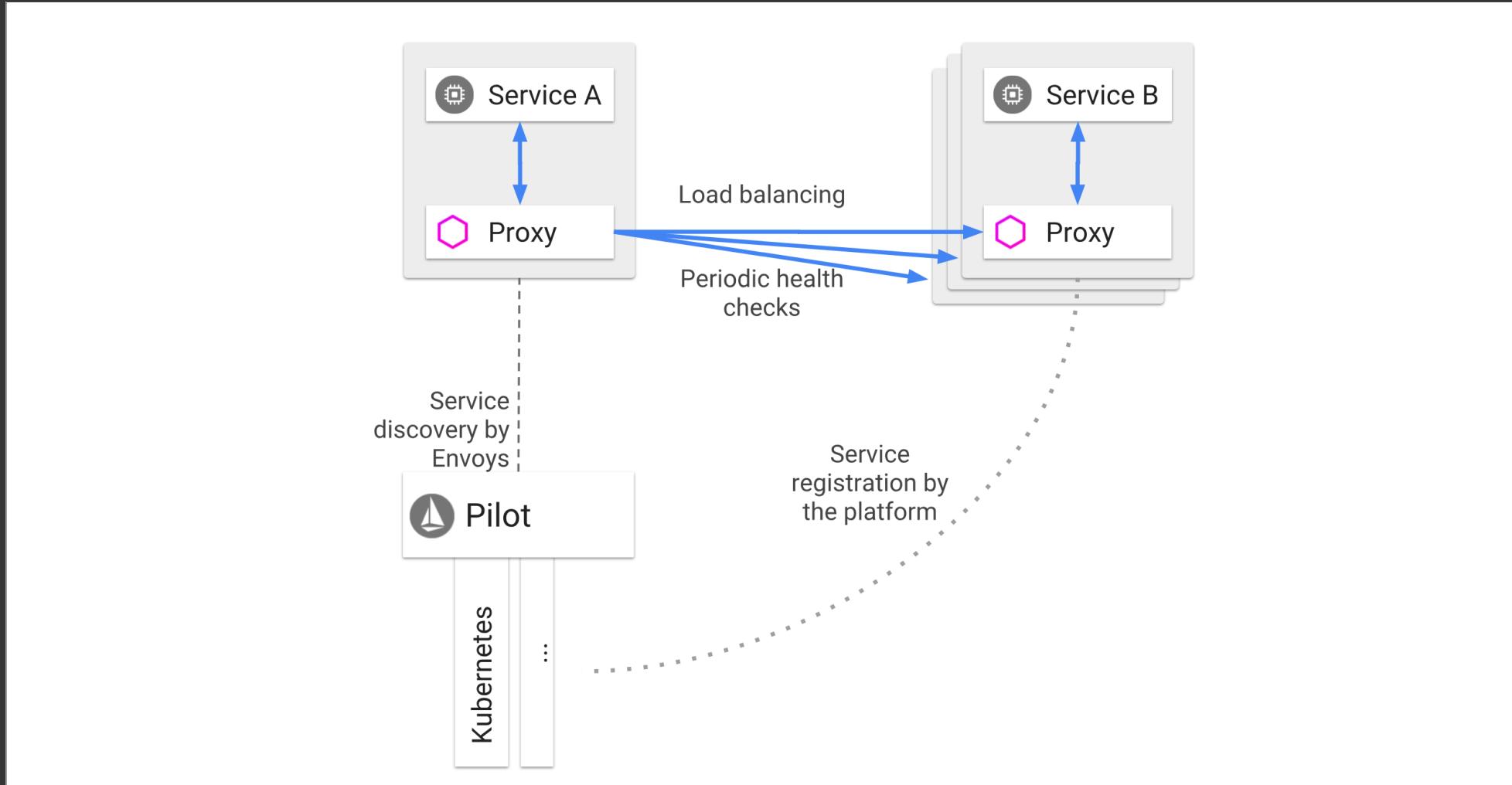
# Communication between services



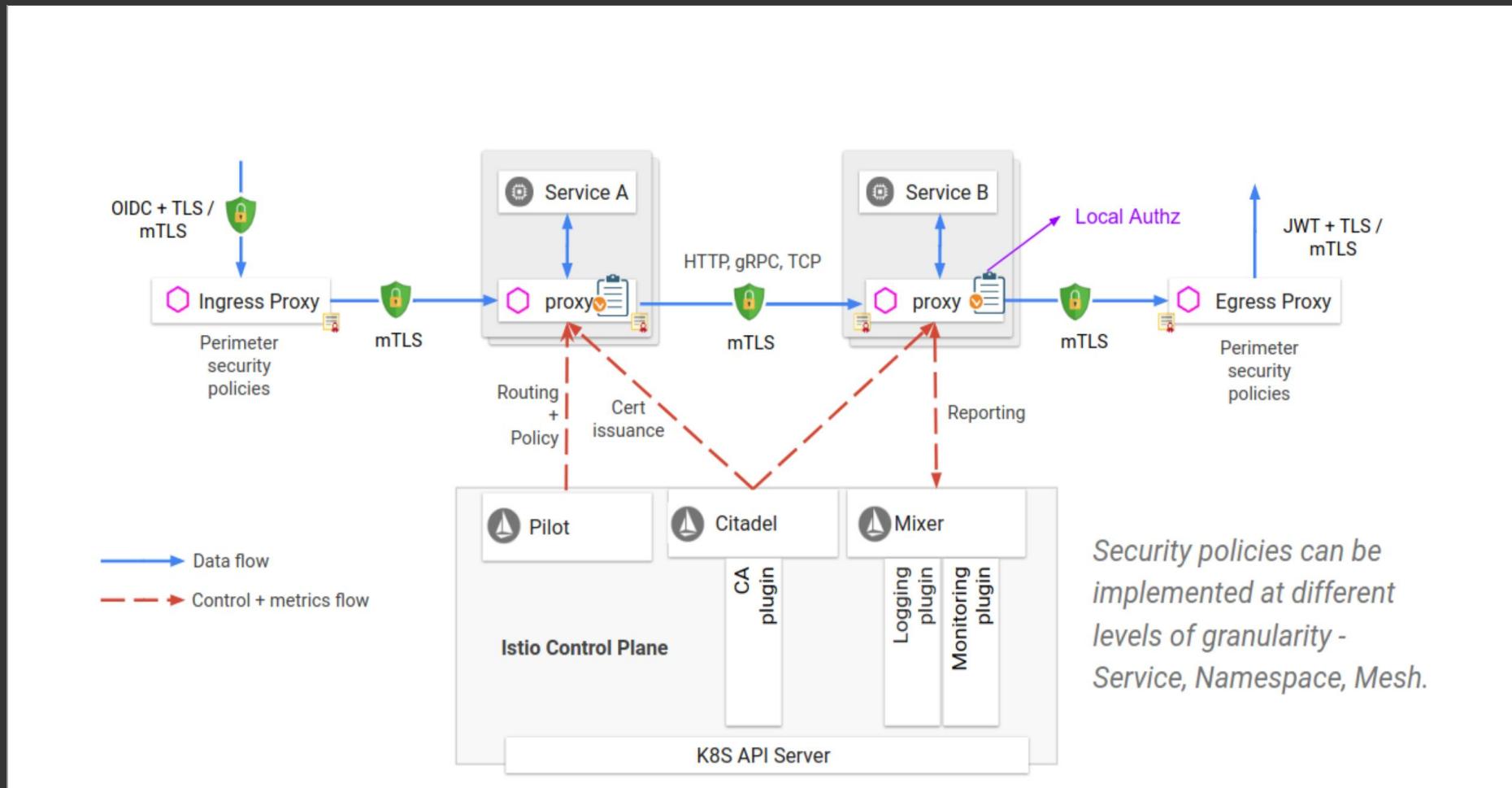
# Ingress and egress



# Discovery and load balancing



# Certificate architecture



# Installing istio



@scottcoulton

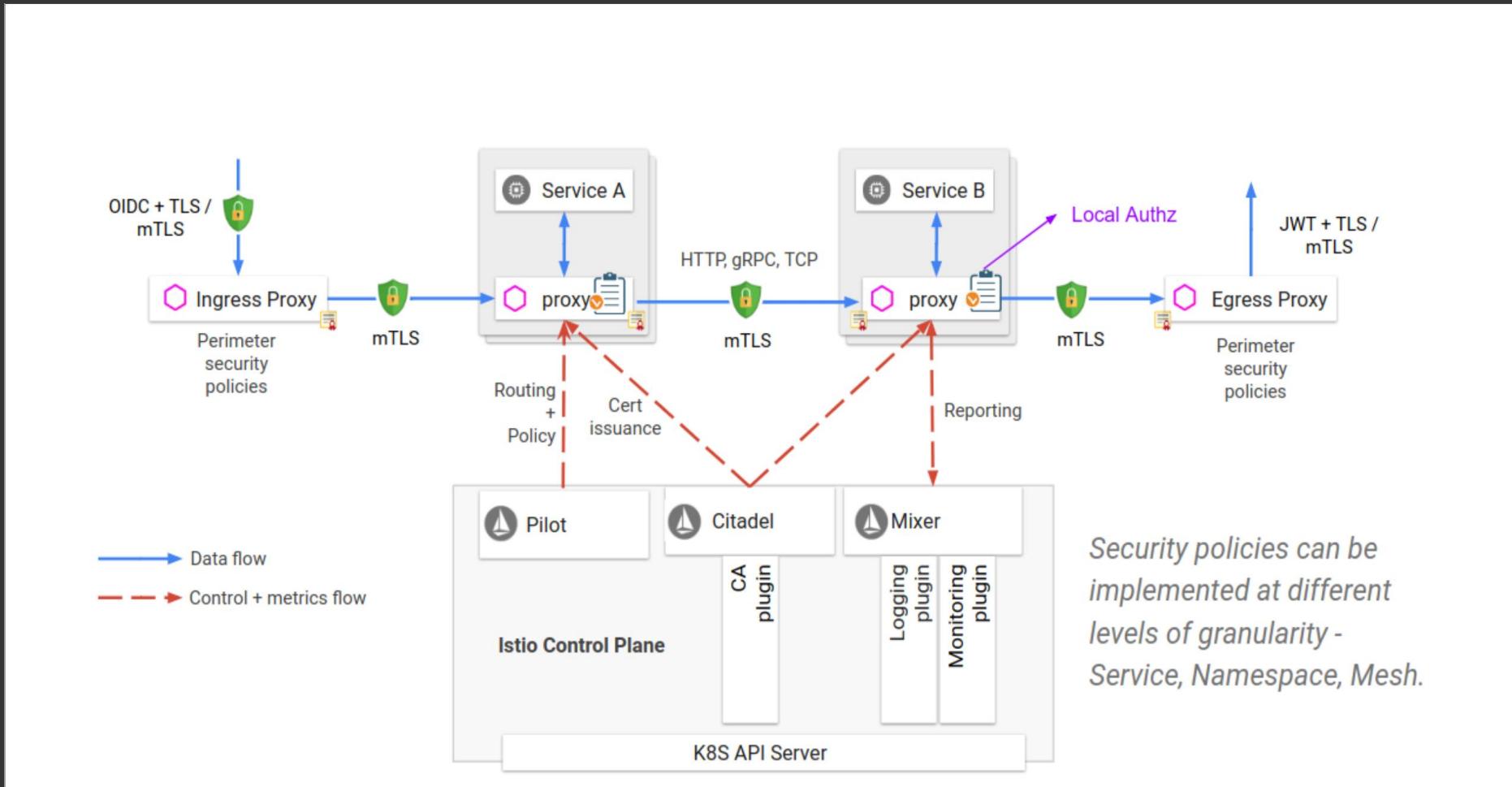
# mTLS



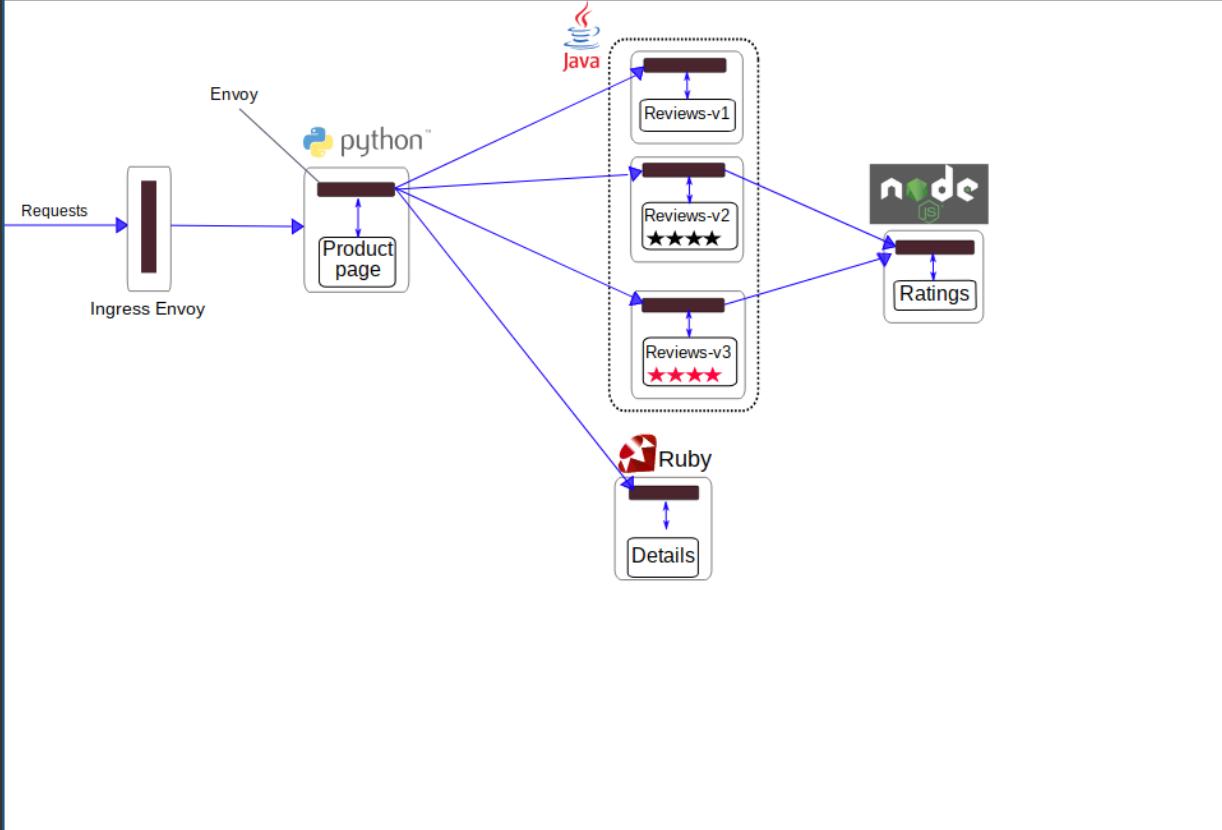
# Warning !!!!!

mTLS needs to be set up in an empty namespace.

# A refresher of the certificate architecture



# The application we are deploying



@scottcoulton

# Create our namespace

```
kubectl create namespace istio-app
```

Make sure we turn on istio injection to our new namespace

```
kubectl label namespace istio-app istio-injection=enabled
```

# Enable mTLS across our namespace

```
cat <<EOF | istioctl create -f -
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: default
  namespace: istio-app
spec:
  peers:
  - mtls:
EOF
```

# Create a destination rule

```
cat <<EOF | istioctl create -f -
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: default
  namespace: istio-app
spec:
  host: "*"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
EOF
```

# Deploy our application

```
kubectl create -n istio-app -f istio-  
1.0.4/samples/bookinfo/platform/kube/bookinfo  
.yaml
```

# Create our virtual service

```
cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
  namespace: istio-app
spec:
  gateways:
  - bookinfo-gateway
  hosts:
  - '*'
  http:
  - match:
    - uri:
        exact: /productpage
    - uri:
        exact: /login
    - uri:
        exact: /logout
    - uri:
        prefix: /api/v1/products
  route:
  - destination:
      host: productpage
      port:
        number: 9080
EOF
```

@scottcoulton

# Create our gateway

```
cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
  namespace: istio-app
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
EOF
```

# Create our gateway

```
kubectl -n istio-system get service istio-ingressgateway -o  
jsonpath='{.status.loadBalancer.ingress[0].ip}'
```

We will then use the ip address in our browser to make sure our site is working correctly.

[http://<YOUR\\_IP>/productpage](http://<YOUR_IP>/productpage)

# Check our mTLS status

```
istioctl authn tls-check | grep .istio-  
app.svc.cluster.local
```

@scottcoulton

# Check our mTLS status

Exec into our envoy proxy sidecar

```
export POD_NAME=$(kubectl get pods --  
namespace=istio-app | grep details | cut -d' '\"  
-f1)
```

```
kubectl exec -n istio-app -it $POD_NAME -c  
istio-proxy /bin/bash
```

# Hit a service internally

```
curl -k -v http://details:9080/details/0
```

@scottcoulton

# Let's capture the traffic with tcpdump

```
IP=$(ip addr show eth0 | grep "inet\b" | awk  
'{print $2}' | cut -d/ -f1)  
sudo tcpdump -vvv -A -i eth0 '((dst port 9080)  
and (net $IP))'
```

# From another terminal

```
curl -o /dev/null -s -w "%{http_code}\n"  
http://$(kubectl -n istio-system get service  
istio-ingressgateway -o  
jsonpath='{.status.loadBalancer.ingress[0].ip}'  
)/productpage
```

# Output

```
^[[22:47:36.978639 IP (tos 0x0, ttl 64, id 19003, offset 0, flags [DF], proto TCP (6), length 60)
  10.244.0.12.50662 > details-v1-6764bbc7f7-x7x99.9080: Flags [S], cksum 0x162b (incorrect -> 0xb758), seq 2995501799, win 29200, options [mss 1460,sackOK,TS val 1887650117 ecr 0,nop,wscale 7], length 0
E..<J;@.@...
...
... .#x.....r...+.....
p.AE.....
22:47:36.978681 IP (tos 0x0, ttl 64, id 19004, offset 0, flags [DF], proto TCP (6), length 52)
  10.244.0.12.50662 > details-v1-6764bbc7f7-x7x99.9080: Flags [.], cksum 0x1623 (incorrect -> 0x3e8d), seq 2995501800, ack 2809488904, win 229, options [nop,nop,TS val 1887650117 ecr 2183432464], length 0
E..4J<@.@...
...
... .#x....uf.....#....
p.AE.$..
22:47:36.978742 IP (tos 0x0, ttl 64, id 19005, offset 0, flags [DF], proto TCP (6), length 254)
  10.244.0.12.50662 > details-v1-6764bbc7f7-x7x99.9080: Flags [P.], cksum 0x16ed (incorrect -> 0xe838), seq 0:202, ack 1, win 229, options [nop,nop,TS val 1887650117 ecr 2183432464], length 202
E...J=@.@...
...
... .#x....uf.....
p.AE.$.....A.....}.Q..d.....+.../. ....../.,.0.
....5...|.....7.5..2outbound|9080||details.istio-app.svc.cluster.local.....#. .........istio.....
.....
22:47:36.979801 IP (tos 0x0, ttl 64, id 19006, offset 0, flags [DF], proto TCP (6), length 52)
  10.244.0.12.50662 > details-v1-6764bbc7f7-x7x99.9080: Flags [.], cksum 0x1623 (incorrect -> 0x38ac), seq 202, ack 1280, win 251, options [nop,nop,TS val 1887650118 ecr 2183432465], length 0
E..4J>@.@...
...
... .#x....uk.....#....
p.AF.$..
22:47:36.981099 IP (tos 0x0, ttl 64, id 19007, offset 0, flags [DF], proto TCP (6), length 1245)
  10.244.0.12.50662 > details-v1-6764bbc7f7-x7x99.9080: Flags [P.], cksum 0x1acc (incorrect -> 0xba00), seq 202:1395, ack 1280, win 251, options [nop,nop,TS val 1887650120 ecr 2183432465], length 1193
E...J?@.@...
...
... .#x....uk.....
p.AH.$.....6..3..00..,0.....->.....).....0... *.H.....0.1.0...U.
..k8s.cluster.local0...190108215913Z..190408215913Z0.1 0...U.
...
... .#x....up....7.#....
p.AM.$..
```