SOLO.IO

White Paper

Transitioning From App Mesh to Istio for AWS EKS

Executive Summary

In the era of cloud-native architectures, organizations are increasingly adopting microservices and containers to enhance agility and scalability. AWS offers robust networking capabilities through services like AWS EKS for container orchestration and AWS App Mesh for service-to-service communication and observability. However, as the complexity of applications grows, there's a need for more sophisticated networking solutions that provide fine-grained control and advanced features.

This whitepaper explores the transition from AWS App Mesh to Istio for AWS EKS environments, focusing on advanced networking needs in cloud-native architectures. Key points include:

- Comparison of user experience, additional features, community engagement, and hybrid environment support between AWS App Mesh and Istio.
- Practical migration examples from AWS App Mesh to Istio within an Amazon EKS cluster, emphasizing ease of installation, mTLS enablement, and advanced traffic management.
- Introduction of Gloo Mesh Core by Solo.io for monitoring and managing Istio- powered service mesh deployments, enhancing visibility and simplifying upgrades.

Overall, transitioning to Istio from AWS App Mesh offers organizations greater flexibility, scalability, and control over their containerized applications, making it a compelling choice for modern cloud-native architectures. Istio combined with Solo.io's expertise and tools further simplify the migration process and empower teams to harness the full power of Istio in their infrastructure.

Background

AWS has powerful networking capabilities for building virtual private clouds, securing them, and connecting them. As organizations modernize their application architectures to be more cloud native, built on containers and microservices, more networking control is needed closer to the applications. Fine-grained application networking allows organizations to build zero trust security postures, get more accurate observability metrics to reduce mean time to recovery (MTTR), and have better operational control over load-balancing, high availability, and failover.

AWS EKS is a popular choice for running containers in AWS, however, Istio has emerged as the de facto industry standard for service mesh and the solution for fine-grained service connectivity problems. AWS App Mesh has been available within the AWS ecosystem for some time, but more and more organizations are looking at open source alternatives such as CNCF Istio to migrate from or augment their App Mesh.

Understanding AWS App Mesh

AWS App Mesh was announced back in November 2018 to solve the challenge around service-to-service communication, fine-grained networking control, and observability. AWS networking tools such as VPCs, VPC peering, and load balancers such as NLBs or ALBs are becoming more coarse grained and require more robust capabilities to solve these challenges. App Mesh is best suited for ECS, EKS, and EC2 customers who run workloads across different orchestrators and need client-side service mesh functionality, such as traffic resiliency controls (retries, timeouts, connection pooling) and mTLS.

Similar to other service mesh technologies, App Mesh leverages Envoy Proxy for its sidecar data plane and offers a managed control plane that users interact with to configure service-to-service routing rules. App Mesh is appealing because it is part of the AWS portfolio, integrates with existing AWS primitives, and the control plane components are managed by AWS.

As environments become more complex, users require additional capabilities (discussed below) beyond what AWS App Mesh offers, so customers are looking at alternatives such as Istio to help unlock the ability to adopt EKS at scale.

Comparing App Mesh and Istio on AWS EKS

Users frequently come to service meshes with specific expectations about their initial experience. These expectations can arise from comparisons with other mesh solutions or from encountering straightforward hello-world demos that create an impression of simplicity. As they progress further along the path of service mesh adoption, they will inevitably encounter situations where their applications require specific capabilities from the mesh such as:

Improved user experience	Additional features
Engaged community	On-premises/hybrid solution

These capabilities look different with App Mesh versus Istio. In the sections below, we'll explore how

Improved User Experience

The top user experiences customers expect from a service mesh are ease of installation and enabling mTLS. Let's compare:

Installation:

AWS App Mesh might not meet the user expectation of simplified installation as

the installation process might require careful handling requiring a few steps to complete a basic install. From copy/pasting credentials to using a third-party CLI (if interested in a CLI), along with creating a combination of IAM rules mapping them to Kubernetes, etc., a new user might find the process involved. After installation, managing Day 2 operations, such as upgrading the mesh data plane, could present additional considerations.

Comparable installation of Istio can be achieved with the single command:
`istioctl install`

Enabling mTLS:

Users have encountered several experience issues with AWS App Mesh, notably during the setup of mTLS, a pivotal feature within a service mesh providing authentication and encryption between services. While mTLS became available in App Mesh in February of 2021, the configuration involves cumbersome steps such as installing SPIRE and registering workloads and nodes.

Compare this to the seamless enablement of mTLS in Istio, which can easily be done by simply adding your applications to the mesh.

Additional Features:

As service mesh implementations mature, what were initially “nice-to-have” features have quickly become “must-haves.” For example, a service mesh aims to be transparent to the workload applications, but since the data plane usually runs as a sidecar, some care must be given to lifecycle management in App Mesh of the containers in a Kubernetes Pod. If a sidecar starts up after the workload, for example, the workload would not have connectivity to the network and potentially throw errors. Solving this problem is not a “big ticket” service mesh feature early on, but must be solved for use in an organization as environments continue to grow.

With Istio, not only can users expect transparency to their workload applications, but they are finding additional unique features a requirement from their mesh: For example, features such as altering headers in service-to-service requests, integrating with app-level health checks, leveraging Envoy ext-auth for advanced auth, using JWT tokens, calling https services, etc.

Another high demand feature for traffic control that Istio brings to AWS EKS is zone-aware load balancing. With this feature, organizations can benefit from better control of their ingress/egress costs across availability zones and regions.

Engaged Community:

As Istio is an open source software, users benefit from an engaged community. These users are excited to participate and provide feedback on improving the service mesh offering, and the community aligns with many open source objectives that customers are leading with today.

On-Premises/Hybrid Solution:

While AWS App Mesh is designed specifically for AWS environments, as modernization and digital transformation initiatives continue to evolve, architectures to support this are becoming more complex. These architectures require a common way to manage multi-services across hybrid and multi-cloud environments for inter-application traffic, security, and observability, which is why many organizations have turned to Istio.

Adopting Istio Into Your Infrastructure

Istio has become an App Mesh alternative due to citing maturity, project stability, and engaged community. Thousands of users and developers have helped push the Istio community to harden the project, smooth out rough areas (like UX), and fill gaps for running in production at scale.

Istio is a CNCF graduated project backed by many vendors (Red Hat, Solo.io, Google, IBM, VMWare etc). It was built with a Kubernetes-first mindset, which has significantly helped users scale their AWS EKS environments. Many of the largest AWS EKS environments globally have adopted Istio to manage their environments across highly regulated industries, financial services, high-tech, retail, healthcare, and more.

Embarking on the journey of integrating Istio into your infrastructure requires a noteworthy commitment from your team. It is essential to recognize the value of expert guidance in this process. Assistance with application migrations and the expansion of your application footprint across multiple Kubernetes clusters, whether hosted on the cloud or on-premises, can significantly smooth the initial phases of implementation.

At Solo.io, we take pride in our experience supporting some of the largest service mesh users globally. Our expertise spans design consultations, architecture reviews, implementation strategies, and day-2 operational advice, positioning us as a reliable partner in your service mesh journey.

Solo.io Gloo Mesh Core provides an advanced platform for monitoring both single and multi-cluster service mesh deployments. It offers user-friendly tools to analyze service communication metrics, identify potential misconfigurations or areas requiring attention, and recommend adjustments. Furthermore, the Istio Lifecycle Manager facilitates user-transparent, controlled, and phased service mesh upgrades, ensuring a seamless experience.

Migrating Your Application from AWS App Mesh to Istio: A Practical Example

In this section, we'll explore the process of migrating an application from AWS App Mesh to Istio within an Amazon EKS cluster. Our journey begins with a GitHub example from AWS App Mesh, which we'll deploy in EKS. Following this, we'll introduce Istio into the mix, configure it to work seamlessly alongside our existing setup, and then transition our application's namespace from App Mesh to Istio's service mesh. This strategy is designed to ensure minimal downtime and resource usage spikes, ultimately allowing the application to leverage Istio's rich service mesh capabilities without altering the user experience.

Preparing the EKS Cluster

Our starting point is the deployment of a demo application on EKS, guided by the official AWS App Mesh example documentation. To streamline this process, ensure you include the `--appmesh-access` flag to avoid manual IAM configuration – a step best described as unenjoyable. Additionally, using the `--version=1.27` flag aligns your environment with our demo, setting the stage for a smooth migration.

Observing Initial Traffic Flow

With our setup ready, let's observe the traffic flow to the user console. Begin by confirming the operation of the App Mesh-managed Envoy sidecar, indicated by a "2/2" READY status, signifying that both the application pod and the Envoy proxy are up and running:

```
$ kubectl get pods -n howto-k8s-http2 -l app=client

NAME                      READY       STATUS    RESTARTS     AGE
client-b74d67958-f549q    2/2         Running   0            64s
```

To streamline our commands, we first capture the name of the client pod into a variable. This approach simplifies future commands by allowing us to reference this variable instead of manually typing or copying the pod name each time.

```
$ CLIENT_POD=$(kubectl get pods -n howto-k8s-http2 -l app=client -o
custom-columns=NAME:.metadata.name --no-headers)
```

Next, we use kubectl port-forward to forward traffic from our local machine to the cluster, enabling direct communication with the application running in EKS. By appending & at the end of the command, we execute the port forwarding in the background, allowing us to continue using the terminal without interruption.

```
$ kubectl port-forward -n howto-k8s-http2 $CLIENT_POD 8080 >/dev/
null & [1] 352546
```

With port forwarding in place, we can now test the server response by sending requests to the `/color` endpoint. This series of requests demonstrates how the application returns random color values, effectively confirming that the App Mesh is correctly handling traffic.

```
$ for i in {1..10}; do curl localhost:8080/color;  echo; done
green
blue
blue
red
red
blue
blue
blue
green
green
```

The following command sets the weight for routing in the already deployed “color” application, transitioning from a round-robin to a controlled, weighted traffic distribution. This adjustment is essential for directing traffic flow more strategically across the application’s services, enhancing the capability to conduct targeted tests and gradual deployments without disrupting the user experience.

```
$ kubectl patch virtualrouters.appmesh.k8s.aws color \
     -n howto-k8s-http2 --type='json' -p='[
     {“op”: “replace”, “path”:
“/spec/routes/0/http2Route/action/weightedTargets”, “value”: [
           {“virtualNodeRef”: {“name”: “green”}, “weight”: 50},
           {“virtualNodeRef”: {“name”: “blue”}, “weight”: 40},
           {“virtualNodeRef”: {“name”: “red”}, “weight”: 10}
    ]}
]'
```

Let’s proceed to test the updated service setup by examining how the distribution of calls is influenced by the newly assigned weights. Execute the command designed to simulate traffic to our “color” application and observe the distribution pattern. The output should be something similar to the following, reflecting our strategic adjustments to the traffic flow among the application’s services:

```
$ for i in {1..100}; do curl -s localhost:8080/color; echo “”; done
| sort | uniq -c
     41 blue
     48 green
     11 red
```

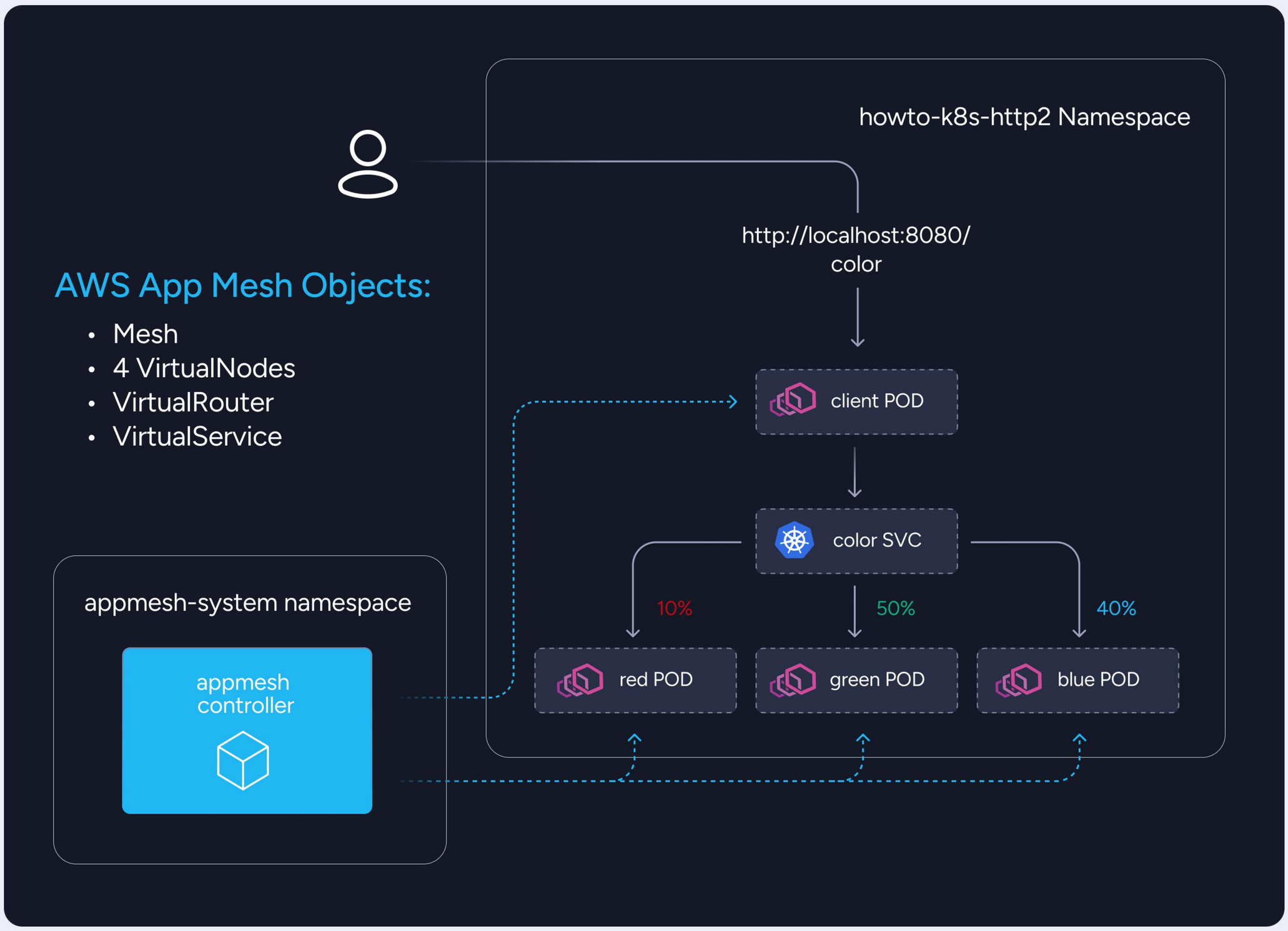
Finally, to clean up and stop the background port forwarding process, we locate theprocessID(PID)of `kubectl port-forward` using `ps` and `grep`, then terminate it using `kill`. This step is crucial for freeing up the port and system resources.

```
$ ps -ax | grep ‘[k]ubectl port-forward -n howto-k8s-http2’ | awk
‘{print $1}’ | xargs kill
```

AWS App Mesh Configuration

Let’s take a closer look at the existing deployment within our Kubernetes environment. By adhering to the App Mesh documentation, we’ve orchestrated seamless service-to-service communication using AWS App Mesh. To further enhance our control over traffic flow, we’ve leveraged the weightsTarget attribute within the App Mesh VirtualRouter, allowing us to meticulously manage how traffic is distributed among services.

Below, you’ll find a detailed exploration of the components that are pivotal to our setup, each playing a critical role in our configuration. This is illustrated in the diagram that follow:



AWS App Mesh Objects:

- Mesh
- 4 VirtualNodes
- VirtualRouter
- VirtualService

- **Mesh:** The foundation of App Mesh within the AWS ecosystem, the Mesh is defined in the AWS account, acting as a container for all the service mesh components.

- **VirtualRouter:** Functionally analogous to Istio's DestinationRule, the VirtualRouter is crucial for managing traffic distribution among connected services. Initially, our configuration relied on a manifest to establish a round- robin distribution. This foundational step ensured a fair and equal distribution of traffic across all endpoints. Moving forward to our advanced setup, we build upon this by incorporating controlled, weighted traffic management.

- **VirtualService:** This component acts as a bridge between the App Mesh defined in AWS and the Kubernetes cluster. It connects the VirtualRouter, which manages the traffic distribution logic, to the actual services running within the cluster.

- **VirtualNodes:** For each service endpoint in App Mesh, a corresponding VirtualNode is required. In our setup, three VirtualNodes represent the 'color' services (red, green, blue), and one represents the client pod that originates the call, totaling four VirtualNodes.

This Kubernetes architecture leverages AWS App Mesh to ensure efficient and reliable service-to-service communication. At the core of this setup is the Mesh, which encapsulates VirtualNodes, VirtualRouter, and VirtualService – each playing a pivotal role in traffic management. VirtualNodes correspond to each service instance, ensuring proper routing and service discovery. The VirtualRouter, devoid of default balancing behavior, is meticulously configured to distribute traffic evenly across services, a task that in Istio's environment, might not necessitate additional configuration due to its default round-robin routing. The VirtualService ties these App Mesh configurations to the Kubernetes cluster, streamlining the communication between cloud-defined settings and in-cluster service operations.

Transitioning to Istio for Simplified Service Management

In this example, our goal is to replace the existing AWS App Mesh with Istio, streamlining our service mesh infrastructure while ensuring minimal disruption. This changeover can be accomplished within our current environment – no need for a new cluster.

First, we need to pinpoint our cluster's name and region. These foundational details are essential as we integrate with the Istio ecosystem. You can retrieve these values effortlessly using the following commands:

```
$ CLUSTER_NAME=$(kubectl config view --minify -o json | jq -r
'.contexts[].context.cluster' | awk -F '.' '{print $1}' )
$ AWS_REGION=$(kubectl config view --minify -o json | jq -r
'.contexts[].context.cluster' | awk -F '.' '{print $2}' )
```

Verify that the variables are set accurately:

```
$ echo Cluster Name: $CLUSTER_NAME AWS Region: $AWS_REGION
```

This should output something akin to "Cluster Name: app-mesh-2 AWS Region: us-west-2".

While we will be using the Solo.io Istio EKS Addon, available at no extra cost on AWS (subscription through the AWS Web UI is required), it's worth noting that any version of Istio or installation method should yield comparable results. The choice of Istio distribution and deployment approach in EKS is at the user's discretion, catering to specific requirements or preferences.

To install the Solo.io Istio EKS Addon, use the following single command, providing the cluster name and region:

```
$ aws eks create-addon --addon-name solo-io_istio-distro --cluster-name $CLUSTER_NAME --region $AWS_REGION
```

You will receive a confirmation that Istio addon creation is underway.

After waiting for about a minute, confirm the successful deployment of Istio by checking for an “ACTIVE” status with this command:

```
$ aws eks create-addon --addon-name solo-io_istio-distro --cluster-name $CLUSTER_NAME --region $AWS_REGION | jq .addon.status
```

Now, with Istio in place, we patch `color` service to comply with Istio’s labeling conventions:

```
kubectl patch service color -n howto-k8s-http2 -- type='json' -p='[{“op”: “add”, “path”: “/spec/selector”, “value”: {“app”:”color”}}]'
```

Optimizing deployments in Kubernetes often involves a choice between reducing resource consumption and ensuring zero downtime. This guide outlines an in-place upgrade method that conserves resources but requires restarting services, potentially leading to temporary downtime.

To prepare a namespace for an in-place upgrade from AWS App Mesh to Istio, remove App Mesh-specific labels and enable Istio’s sidecar injection as follows:

```
$ kubectl label namespace howto-k8s-http2 appmesh.k8s.aws/sidecarInjectorWebhook- mesh- --overwrite
$ kubectl label namespace howto-k8s-http2 istio-injection=enabled
```

These commands reconfigure the howto-k8s-http2 namespace for Istio, necessitating a service restart.

For scenarios prioritizing zero downtime, an alternative involves using a new namespace. Instead of modifying existing resources, apply Istio configurations to a newly created namespace. This approach ensures uninterrupted service while transitioning to Istio, ideal for environments where continuity is critical. Steps would include creating a new namespace, applying configurations there, and gradually shifting traffic to maintain service availability during the transition.

After configuring the Istio service mesh, demonstrating the traffic flow becomes crucial. Enabling Envoy's access logging feature is a straightforward way to achieve this. Use the snippet below to activate detailed access logging:

```
kubectl apply -f - <<EOF
apiVersion: telemetry.istio.io/v1alpha1
kind: Telemetry
metadata:
     name: mesh-default
     namespace: istio-system
spec:
     accessLogging:
          - providers:
               - name: envoy
EOF
```

If the zero downtime upgrade approach is adopted, the need to restart deployments within the namespace to pick up the new configuration can be avoided:

```
kubectl rollout restart deployment --namespace howto-k8s-http2
```

Test to confirm that Istio is now balancing traffic between endpoints just as App Mesh did:

```
CLIENT_POD=$(kubectl get pods -n howto-k8s-http2 -l app=client -o custom-
columns=NAME:.metadata.name --no-headers)
```

Start port-forwarding to access the service locally:

```
kubectl port-forward -n howto-k8s-http2 $CLIENT_POD 8080 >/dev/null &
```

Run the test loop:

```
for i in {1..100}; do curl -s localhost:8080/color; echo ""; done |
sort | uniq -c
```

The output should summarize the number of responses from every `color`:

```
34 blue
36 green
30 red
```

To gain insights into the request flow, we can examine the istio-proxy logs at both the initiating and receiving ends of the connection. This approach allows us to understand the interactions between different components within our service mesh.

For the pod initiating the connection (the client pod):
Use the following command to view the last two requests made by the client pod:

```
kubectl -n howto-k8s-http2 logs $CLIENT_POD -c istio-proxy | grep -E
'GET /|POST /' | tail -n5
```

Example output:

```
[2024-02-23T23:01:33.360Z] "GET / HTTP/2" 200 - via_upstream - "-" 0311
"-" "Go-http-client/2.0" "18b928c4-21a2-4125-bd7b-05473d0bb350"
"color.howto-k8s-http2.svc.cluster.local:8080" "192.168.55.44:8080"
inbound|8080|| 127.0.0.6:37687 192.168.55.44:8080 192.168.59.9:49412
outbound_.8080_._.color.howto-k8s-http2.svc.cluster.local default
....
[2024-02-23T23:01:33.602Z] "GET / HTTP/2" 200 - via_upstream - "-" 0411
"-" "Go-http-client/2.0" "469059e8-0db4-4b2f-81da-b52062e3c91e"
"color.howto-k8s-http2.svc.cluster.local:8080" "192.168.55.44:8080"
outbound|8080||color.howto-k8s-http2.svc.cluster.local 192.168.59.9:3
406210.100.17.82:8080 192.168.59.9:34404 - default
```

For the receiving end (any of the pods that return 'color'):

To observe the requests received, execute:

```
kubectl -n howto-k8s-http2 logs -l app=color -c istio-proxy | grep -E
'GET /|POST /' | tail -n5
```

Example output:

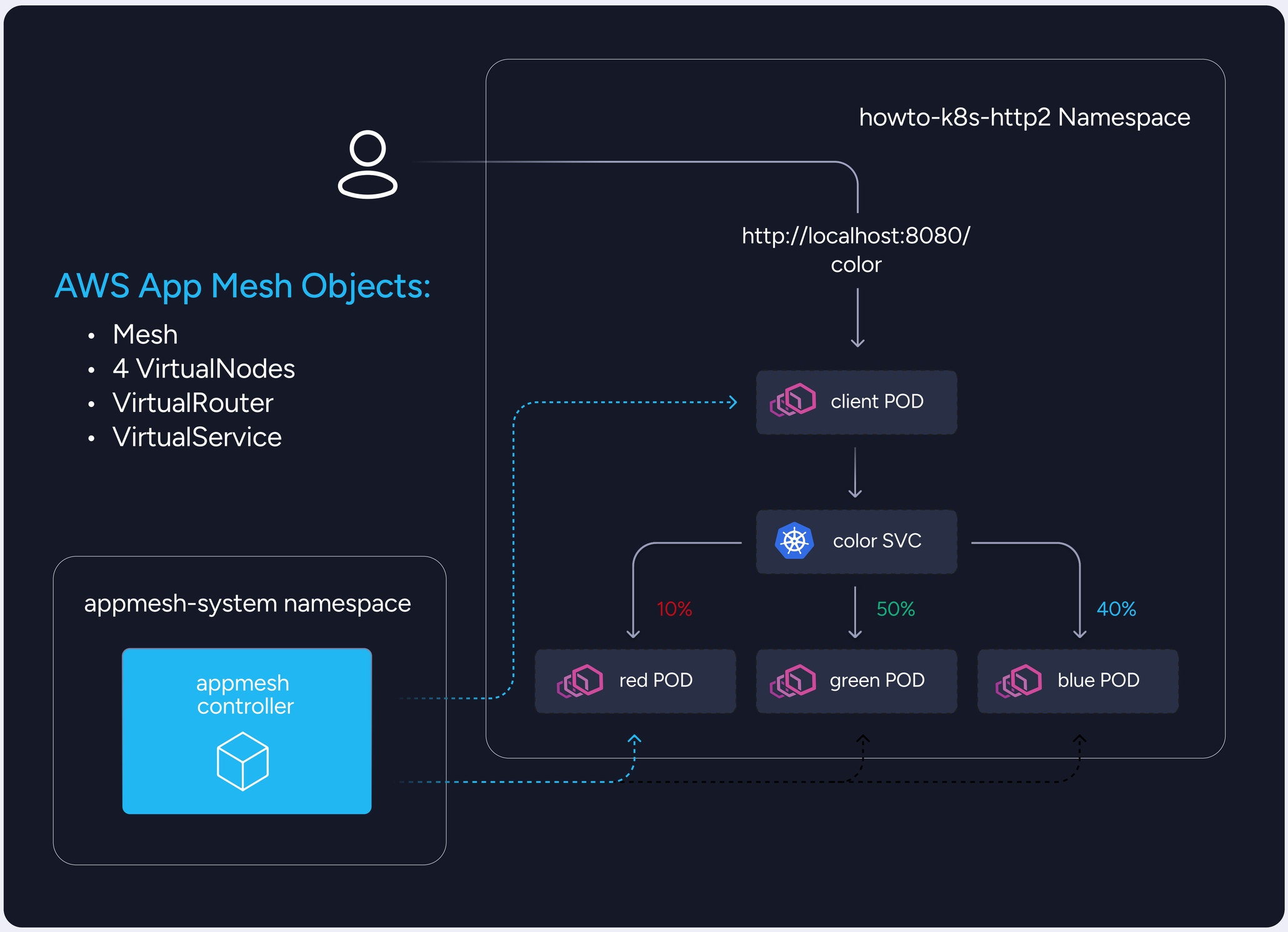
```
[2024-02-23T23:01:33.360Z] "GET / HTTP/2" 200 - via_upstream - "-" 0 3 0 0
"-" "Go-http-client/2.0" "18b928c4-21a2-4125-bd7b-05473d0bb350"
 "color.howto-k8s-http2.svc.cluster.local:8080" "192.168.55.44:8080"
inbound|8080|| 127.0.0.6:37687 192.168.55.44:8080 192.168.59.9:49412
outbound_.8080_._.color.howto-k8s-http2.svc.cluster.local default
...
[2024-02-23T23:01:33.602Z] "GET / HTTP/2" 200 - via_upstream - "-" 0 3 0 0
"-" "Go-http-client/2.0" "469059e8-0db4-4b2f-81da-b52062e3c91e"
"color.howto-k8s-http2.svc.cluster.local:8080" "192.168.55.44:8080"
inbound|8080|| 127.0.0.6:37687 192.168.55.44:8080 192.168.59.9:49412
outbound_.8080_._.color.howto-k8s-http2.svc.cluster.local default
```

With Istio successfully in place, we can remove the no longer needed App Mesh configurations:

```
kubectl delete virtualnodes, virtualservices -n howto-k8s-http2 --all
```

Through these steps, we've transitioned from AWS App Mesh to Istio, a move that brings simplicity and continuity of service operations. The result is a service mesh that integrates more fluidly with our Kubernetes environment, offering a balance of manageability and flexibility.

The diagram visualizes the transformed architecture:



Envoy Sidecars: While both App Mesh and Istio leverage Envoy proxies, Istio provides a layer of management that extends Envoy's capabilities within the Kubernetes environment. Each pod in the `howto-k8s-http2` namespace now includes an Envoy sidecar proxy managed by Istio, which facilitates advanced traffic management, security, and observability features.

Control Plane: The `istio-system` namespace contains Istio's control plane components, including `istiod`. This component configures the sidecar proxies, manages policies, and aggregates telemetry data, thereby serving as the administrative hub for the service mesh.

Service Routing: The `color` service within the namespace continues to serve as the entry point for incoming requests. Istio's control plane intelligently directs traffic to the appropriate 'color' pod, based on the configured routing rules.

User Interface: On the user's end, the interface remains consistent. The application is still accessed through the same URL (`http://localhost:8080/color`), with no perceptible change in interaction or experience, despite the shift in the underlying service mesh technology.

This evolution to Istio has refined our service mesh without interrupting service delivery, illustrating the seamless nature of the transition. The diagram above reflects the new state where Istio's nuanced management of Envoy sidecars enriches our Kubernetes service mesh, aligning with modern requirements for flexibility and scalability.

Leveraging Istio for Advanced Traffic Management Enhancements

With our migration to Istio, we've accessed an extensive suite of advanced service mesh features, bypassing the necessity for complex, Istio-specific configurations from the outset. By merely applying the Istio injection label to our Kubernetes services, we've integrated the expansive benefits of Istio's service mesh, which include secure service-to-service communication, comprehensive telemetry, as well as robust authentication and authorization mechanisms.

While our primary strategy involves an in-place update within the existing Kubernetes environment. It's crucial to acknowledge that running parallel namespaces is a good option and some have also opted to just run parallel clusters. In such cases, running two parallel namespaces — one with the current App Mesh setup and the other configured for Istio — becomes an invaluable strategy. This parallel setup allows for a phased traffic shift from App Mesh to Istio, ensuring that services remain fully operational and accessible to users throughout the transition. By leveraging this method, organizations can prioritize continuous service delivery, seamlessly migrating traffic to Istio's advanced service mesh capabilities, including robust authentication, authorization, and precise traffic management, without compromising on availability. This dual-namespace approach offers a strategic pathway for those who place a higher emphasis on maintaining zero downtime during the migration process.

Expanding further, our aim is to leverage Istio's VirtualServices and DestinationRules to achieve precise traffic management. These key Istio components enable us to implement controlled traffic management strategies that align with the advanced setup previously realized with App Mesh. Through this focused application of Istio's capabilities, we mirror the detailed, weighted traffic distribution configured in our App Mesh environment, achieving a cohesive and seamless integration of sophisticated traffic management practices within our service mesh architecture.

The Power of Istio's Custom Resources:

- **VirtualServices** allow us to define how traffic is routed to different versions of a service within the mesh. With this, we can implement advanced patterns like canary deployments, where we introduce a new service version and gradually shift traffic to it.
- **DestinationRules** are used in tandem with VirtualServices. They provide the rules that dictate traffic policies and network paths, enabling scenarios such as load balancing and circuit breaking for enhanced service resilience.

In our example, by applying a DestinationRule, we designate service subsets based on specific labels. Then, using a VirtualService, we can distribute traffic across these subsets with precise weightings. This approach is integral for:

- **Dynamic Traffic Control:** Adjust traffic flows on the fly, facilitating real-time responses to operational requirements or user demand without redeploying pods.
- **Enhanced Observability:** Leverage Istio's telemetry to gain insights into the mesh's health and traffic patterns, aiding in proactive decision-making and issue resolution.
- **Improved Resilience:** Implement robust routing strategies to ensure the mesh can handle failures or traffic spikes without degrading user experience.

Let's apply a DestinationRule for our 'color' service:

```
kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
 name: color-destination-rule
 namespace: howto-k8s-http2
spec:
 host: color
 subsets:
  - name: blue
   labels:
    version: blue
  - name: green
   labels:
    version: green
  - name: red
   labels:
    version: red
EOF
```

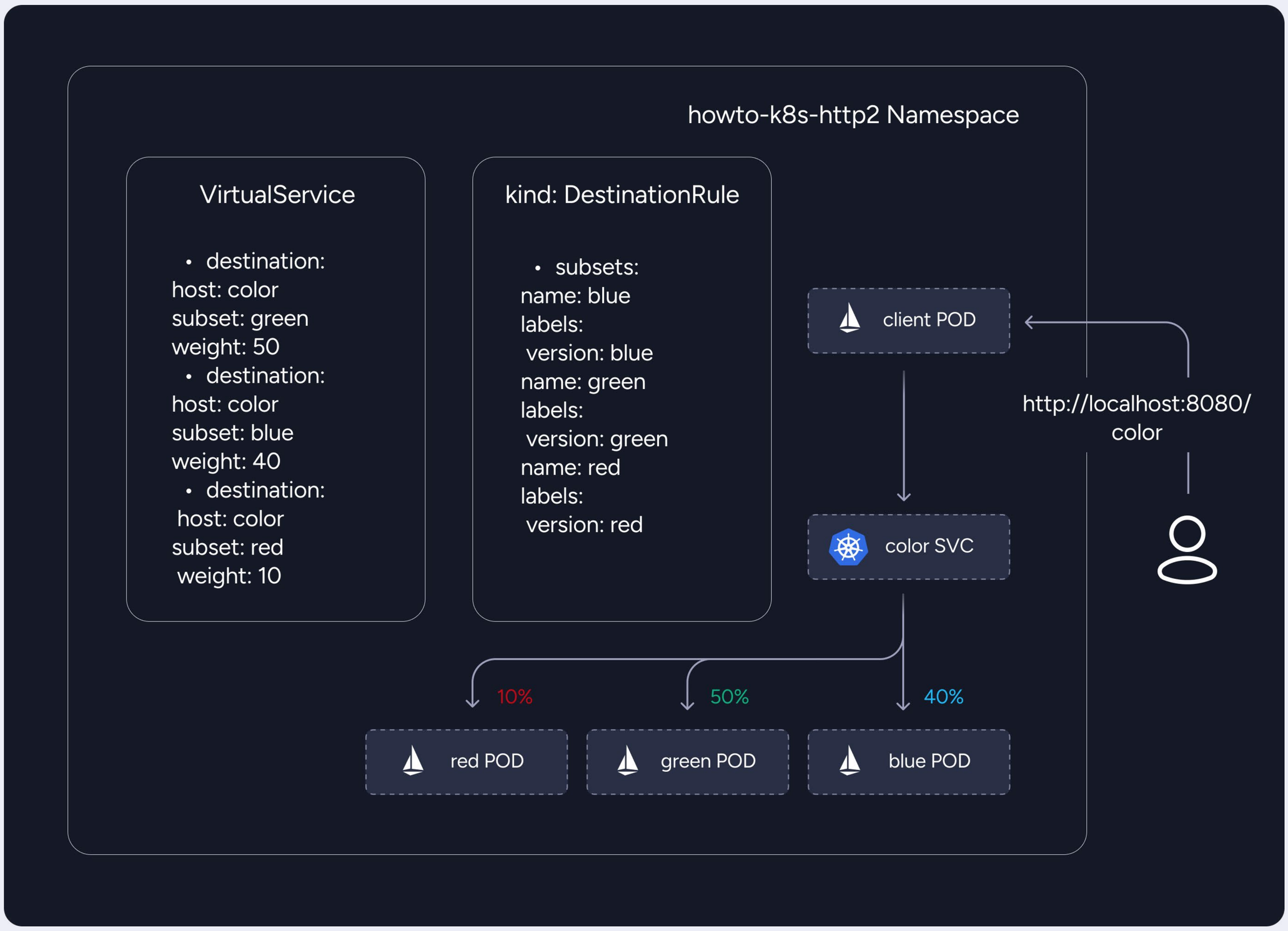
Next, we establish a VirtualService to manage the traffic distribution:

```
kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
 name: color-virtual-service
 namespace: howto-k8s-http2
spec:
 hosts:
  - color
 http:
  - route:
    - destination:
       host: color
       subset: green
     weight: 50
    - destination:
host: color
       subset: blue
     weight: 40
    - destination:
       host: color
       subset: red
     weight: 10
EOF
```

Upon executing our test command, the efficacy of Istio's traffic management policies is confirmed through the output, which aligns with the weights specified in our VirtualService configuration:

```
for i in {1..100}; do curl -s localhost:8080/color; echo “”; done |
sort | uniq -c
  38 blue
  54 green
   8 red
```

The distribution of requests clearly adheres to the rules defined in our Istio VirtualService. The service routed approximately half of the traffic to green, a significant proportion to blue, and a smaller fraction to red, just as we intended.



This real-world result exemplifies Istio's adeptness at managing traffic with precision. By integrating VirtualServices and DestinationRules, we've established a service mesh that not only ensures the continued delivery of services but also enhances the overall functionality of our network. This advanced routing capability facilitates a robust, observable, and highly manageable service architecture, setting a solid foundation for resilient operations and offering the flexibility to adapt to changing requirements.

With these configurations, Istio's ability to govern traffic flow is more than theoretical – it's a practical reality. Our setup exemplifies the service mesh's potential, enabling us to confidently manage traffic distribution, monitor service health, and improve our system's resilience. It's a testament to Istio's promise of a sophisticated, scalable, and controllable network infrastructure within Kubernetes.

To learn more, try Gloo Mesh Core or visit the open-source distribution of Istio from Solo.io on the Amazon Marketplace today!

SOLO.IO

contact@solo.io

www.solo.io

About Solo.io

Solo.io, the leading application networking company, delivers a service mesh and API platform for Kubernetes, zero trust, and microservices. The three components of the Gloo Platform – Gloo Gateway, Gloo Mesh and Gloo Network – enable enterprise companies to rapidly adopt microservice applications as part of their cloud journey and digital transformation. Solo delivers open source solutions, and is a community leader in building the technologies of the future.