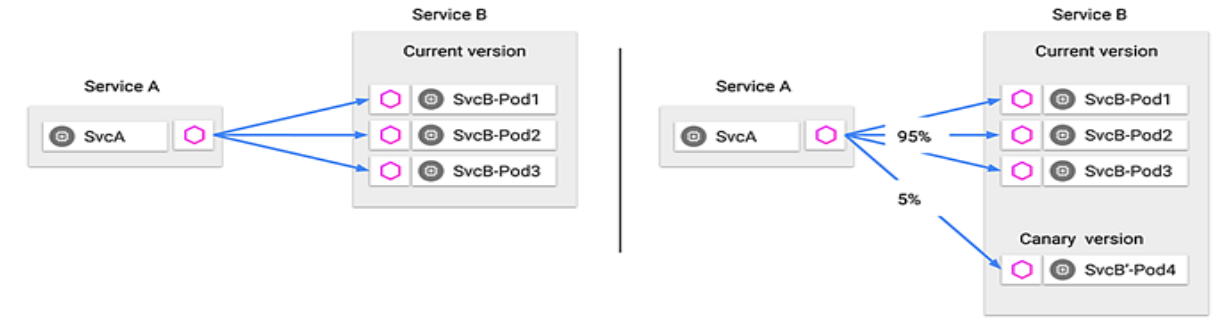
Request Routing and Canary Testing:

Istio's ability to enforce policy at any point in the network enables a number of very useful traffic control and observability features, including rate limiting, circuit breaking and programmable rollouts such as canary deployments.

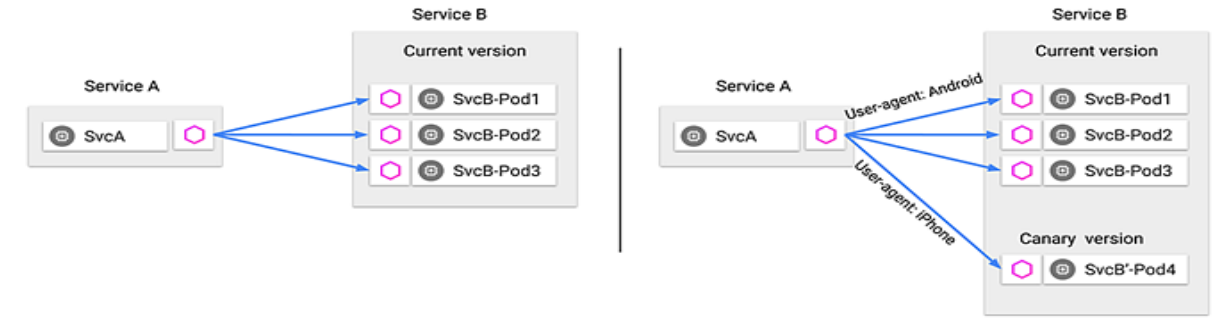
A/B Testing:

Istio's traffic routing can be used for A/B testing, the testing of new features by sending a subset of customer traffic to instances with the new feature and observing telemetry and user feedback. Although commonly used for user interfaces, A/B testing can also be employed for microservices. Istio can be configured to direct traffic based on a percentage weight, cookie value, query parameter and HTTP headers, to name a few.

Use-cases for microservice A/B testing might include trying out new features for a subset of users or geographical regions, or testing an update on a reduced scale before complete roll-out - if a significant amount of errors are detected on the new version then the rollout can be reverted and all traffic sent to the incumbent service. Testing new features via A/B testing is potentially impactful; it would be safer to perform canary releases.



Traffic splitting decoupled from infrastructure scaling - proportion of traffic routed to a version is independent of number of instances supporting the version



Content-based traffic steering - The content of a request can be used to determine the destination of a request

Canary Releases:

Canary releases could be considered a special case of A/B testing, in which the rollout happens much more gradually. The analogy being alluded to by the name is the canary in the coal mine. A canary release begins with a "dark" deployment of the new service version, which receives no traffic. If the service is observed to start healthily it is directed a small percentage of traffic (e.g. 1%). Errors are continually monitored as continued health is rewarded with increased traffic, until the new service is receiving 100% of the traffic and the old instances can be shut down. Obviously, the ability to safely perform canary releases rests upon reliable and accurate application health checks.

Istio traffic routing configuration can be used to perform canary releases by programmatically adjusting the relative weighting of traffic between service versions. Writing a control loop to observe service health and adjust weighting as part of a canary deployment is left to the user, although ***Flagger*** and ***Theseus*** are designed for this purpose.

1 Configure the default route for all services to V1:

As part of the bookinfo sample app, there are multiple versions of reviews service. When we load the `/productpage` in the browser multiple times we have seen the reviews service round robin between v1, v2 or v3. As the first exercise, let us first restrict traffic to just V1 of all the services.

Set the default version for all requests to v1 of all service using:

```
kubectl apply -f virtual-service-all-v1.yaml
```

This creates a bunch of ***virtualservice*** and ***destinationrule*** entries which route calls to v1 of the services.

To view the applied rule:

```
kubectl get virtualservice reviews -o yaml
```

Please note: In the place of the above command, we can either use kubectl or istioctl.

Output:

```
apiVersion: networking.istio.io/v1alpha3
```

```
kind: VirtualService
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"networking.istio.io/v1alpha3","kind":"VirtualService","metadata":{"ann
otations":{},"name":"reviews","namespace":"default"},"spec":{"hosts":["reviews"],"htt
p":[{"route":[{"destination":{"host":"reviews","subset":"v1"}}]}]}}
  creationTimestamp: null
  name: reviews
  namespace: default
  resourceVersion: "11595"
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
---
```

Now when we reload the /***productpage*** several times, we will ONLY be viewing the data from v1 of all the services, which means we will not see any ratings (any stars).

2. Route based on user identity:

Let's replace our first rules with a new set. Enable the `ratings` service for your user by routing `productpage` traffic to reviews v2.

Now, find the user name of the user you are logged into the Bookinfo app from the browser.

Once you have found out the user name, we can update the `virtual-service-reviews-test-v2.yaml` file with the right user name in the placeholder USER_NAME we have in place:

Edit the file `virtual-service-reviews-test-v2.yaml` and update the user name.

After you have updated the user name in the right place, let us save the changes and then apply it on the cluster:

```
kubectl apply -f virtual-service-reviews-test-v2.yaml
```

Confirm the rule is created:

```
kubectl get virtualservice reviews -o yaml
```

Output will be similar to the one below:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"networking.istio.io/v1alpha3","kind":"VirtualService","metadata":{"ann
otations":{},"name":"reviews","namespace":"default"},"spec":{"hosts":["reviews"],"htt
p":[{"match":[{"headers":{"end-
user":{"exact":"USER_NAME"}}}],"route":[{"destination":{"host":"reviews","subset":"v2
"}}]},{"route":[{"destination":{"host":"reviews","subset":"v1"}}]}]}}
  creationTimestamp: null
  name: reviews
  namespace: default
  resourceVersion: "10366"
spec:
  hosts:
  - reviews
  http:
  - match:
    - headers:
        end-user:
          exact: USER_NAME
    route:
    - destination:
        host: reviews
        subset: v2
  - route:
    - destination:
        host: reviews
        subset: v1
---
```

Now if we login as your user, you will be able to see data from `reviews v2`. While if you NOT logged in or logged in as a different user, you will see data from `reviews v1`.