Using Canary Strategy with Flager, Istio, and Prometheus

This lesson demonstrates how we can use the canary strategy with Flagger, Istio and Prometheus.

We'll cover the following

Before we start exploring `Canary` deployments, let's take a quick look at what we have so far to confirm that the first release using the new definition worked.

Confirming the first release

*Is our application accessible through the **Istio** gateway?* Let's send a request to check.

```
curl $STAGING_ADDR
```

The output should say `Hello from: Jenkins X golang http rolling update`.

Now that we confirmed that the application released with the new definition is accessible through the **Istio** gateway, we can take a quick look at the Pods running in the staging Namespace.

```
kubectl --namespace jx-staging \
    get pods
```

The output is as follows.

```
NAME                          READY STATUS  RESTARTS AGE
jx-jx-progressive-primary-... 2/2   Running 0        42s
jx-jx-progressive-primary-... 2/2   Running 0        42s
jx-jx-progressive-primary-... 2/2   Running 0        42s
```

There is a change at least in the naming of the Pods belonging to *jx-progressive*. Now they contain the word `primary`. Given that we deployed only one release

using `Canary`, those Pods represent the main release accessible to all the users. As you can imagine, the Pods were created by the corresponding `jx-jx-progressive-primary` ReplicaSet which, in turn, was created by the `jx-jx-progressive-primary` Deployment. As you can probably guess, there is also the `jx-jx-progressive-primary` Service that allows communication to those Pods, even though sidecar containers injected by **Istio** further complicate that. Later on, we'll see why all those are important.

What might matter more is the `canary` resource, so let's take a look at it.

```
kubectl --namespace jx-staging \
    get canary
```

The output is as follows.

```
NAME              STATUS      WEIGHT LASTTRANSITIONTIME
jx-jx-progressive Initialized 0      2019-12-01T21:35:32Z
```

There's not much going on there since we have only the first `Canary` release running. For now, please note that `canary` can give us additional insight into the process.

You saw that we set the `Canary` gateway to `jx-gateway.istio-system.svc.cluster.local`. As a result, when we deployed the first `Canary` release, it created the gateway for us. We can see it by retrieving `virtualservice.networking.istio.io` resources.

```
kubectl --namespace jx-staging \
    get virtualservices.networking.istio.io
```

The output is as follows.

```
NAME              GATEWAYS                                  HOSTS
jx-jx-progressive [jx-gateway.istio-system.svc.cluster.local] [staging.jx-progressive.104.196.199.
```

We can see from the output that the gateway `jx-gateway.istio-system.svc.cluster.local` is handling external requests coming from `staging.jx-progressive.104.196.199.98.nip.io` as well as `jx-jx-progressive`. We'll focus on the former host and ignore the latter.

Finally, we can output **Flagger** logs if we want to see more details about the deployment process.

deployment process.

```
kubectl --namespace istio-system logs \
    --selector app=flagger
```

I'll leave it to you to interpret those logs. Don't get stressed if you don't understand what each event means. We'll explore what's going on in the background as soon as we deploy the second release.

To see `Canary` deployments in action, we'll create a trivial change in the demo application by replacing `hello, rolling update!` in `main.go` to `hello, progressive!`. Then, we will commit and merge it to master to get a new version in the staging environment.

```
cat main.go | sed -e \
    "s@rolling update@progressive@g" \
    | tee main.go

git add .

git commit \
    -m "Added progressive deployment"

git push
```

Those were such trivial and commonly used commands that there is no need to explain them.

Just as with previous deployment strategies, now we need to be fast.

```
echo $STAGING_ADDR
```

Please copy the output and go to the **second terminal**.

> 🔍 Replace `[...]` in the command that follows with the copied address of *jx-progressive* in the staging environment.

```
STAGING_ADDR=[...]

while true
do
    curl "$STAGING_ADDR"
    sleep 0.2
done
```

As with the other deployment types, we initiated a loop that continuously sends requests to the application. That will allow us to see whether there is deployment-caused downtime. It will also provide us with the first insight into how canary deployments work.

Until the pipeline starts the deployment, all we're seeing is the `hello, rolling update!` message coming from the previous release. Once the first iteration of the rollout is finished, we should see both `hello, rolling update!` and `hello, progressive!` messages alternating. Since we specified that `stepWeight` is `20`, approximately twenty percent of the requests should go the new release while the rest will continue to be forwarded to the old. Thirty seconds later (the `interval` value), the balance should change. We should have reached the second iteration, with forty percent of requests coming from the new release and the rest from the old.

Based on what we can deduce so far, `Canary` deployments are behaving in a very similar way as `RollingUpdate`. The significant difference is that our rolling update examples did not specify any delay, so the process looked almost as if it was instant. If we did specify a delay in rolling updates and if we had five replicas, the output would be nearly the same.

As you might have guessed, we would not go into the trouble of setting up `Canary` deployments if their behavior is the same as with the `RollingUpdate` strategy. There's much more going on. We'll have to go back to the first terminal to see the other effects better.

Leave the loop running and go back to the **first terminal**

Let's see which Pods do we have in the staging namespace.

```
kubectl --namespace jx-staging \
    get pods
```

The output is as follows.

```
NAME                            READY STATUS  RESTARTS AGE
jx-jx-progressive-...           2/2   Running 0        22s
jx-jx-progressive-...           2/2   Running 0        22s
jx-jx-progressive-...           2/2   Running 0        22s
jx-jx-progressive-primary-... 2/2   Running 0        9m
```

```
jx-jx-progressive-primary-... 2/2   Running 0        9m
jx-jx-progressive-primary-... 2/2   Running 0        9m
```

Assuming that the process did not yet finish, we should see that besides the `jx-jx-progressive-primary` we also got `jx-jx-progressive` (without `-primary`). If you take a closer look at the `AGE`, you should notice that all the Pods were created a while ago except `jx-progressive`. That's the new release, and we'll call it "canary Pod". **Flagger** has both releases running during the deployment process. Initially, all traffic was being sent to the `primary` Pods. But, when the deployment process was initiated, `VirtualService` started sending traffic to one or another, depending on the iteration and the `stepWeight`. To be more precise, the percentage of requests being sent to the new release is equivalent to the iteration multiplied with `stepWeight`. Behind the scenes, **Flagger** is updating **Istio** `VirtualService` with the percentage of requests that should be sent to one group of Pods or another. It is updating `VirtualService` telling it how many requests should go to the Service associated with primary and how many should go to the one associated with "canary" Pods.

Given that much of the action is performed by the `VirtualService`, we'll take a closer look at it and see whether we can gain some additional insight.

> 🔍 Your outputs will probably differ from mine depending on the deployment iteration (stage) you're in right now. Follow my explanations of the outputs even if they are not the same as what you'll see on your screen.

The output, limited to the relevant parts, is as follows.

```
...
spec:
  gateways:
  - jx-gateway.istio-system.svc.cluster.local
  hosts:
  - staging.jx-progressive.104.196.199.98.nip.io
  - jx-jx-progressive
  http:
  - route:
    - destination:
        host: jx-jx-progressive-primary
      weight: 20
    - destination:
        host: jx-jx-progressive-canary
      weight: 80
```

The interesting part is the `spec` section. In it, besides the `gateways` and the `hosts`

The interesting part is the `spec` section. In it, besides the `gateways` and the `hosts`, is `http` with two `routes`. The first one points to `jx-progressive-primary`, which is the old release. Currently, at least in my case, it has the `weight` of `40`. That means that the `primary` (the old) release is currently receiving forty percent of requests. On the other hand, the rest of sixty percent is going to the `jx-progressive-canary` (the new) release. Gradually, **Flagger** was increasing the `weight` of `canary` and decreasing the `primary`, thus gradually shifting more and more requests from the old to the new release. Still, so far all that looks just a "fancy" way to accomplish what rolling updates are already doing. If that thought is still passing through your head, you'll see very soon that there's so much more.

An easier and more concise way to see the progress is to retrieve the `canary` resource.

```
kubectl --namespace jx-staging \
    get canary
```

The output is as follows.

```
NAME              STATUS      WEIGHT LASTTRANSITIONTIME
jx-jx-progressive Progressing 60     2019-08-16T23:24:03Z
```

In my case, the process is still `progressing` and, so far, it reached `60` percent. In your case, the `weight` is likely different, or the `status` might be `succeeded`. In the latter case, the process is finished successfully. All the requests are now going to the new release. The deployment rolled out fully.

If we describe that `canary` resource, we can get more insight into the process by observing the events.

```
kubectl --namespace jx-staging \
    describe canary jx-jx-progressive
```

The output, limited to the `events` initiated by the latest deployment, is as follows.

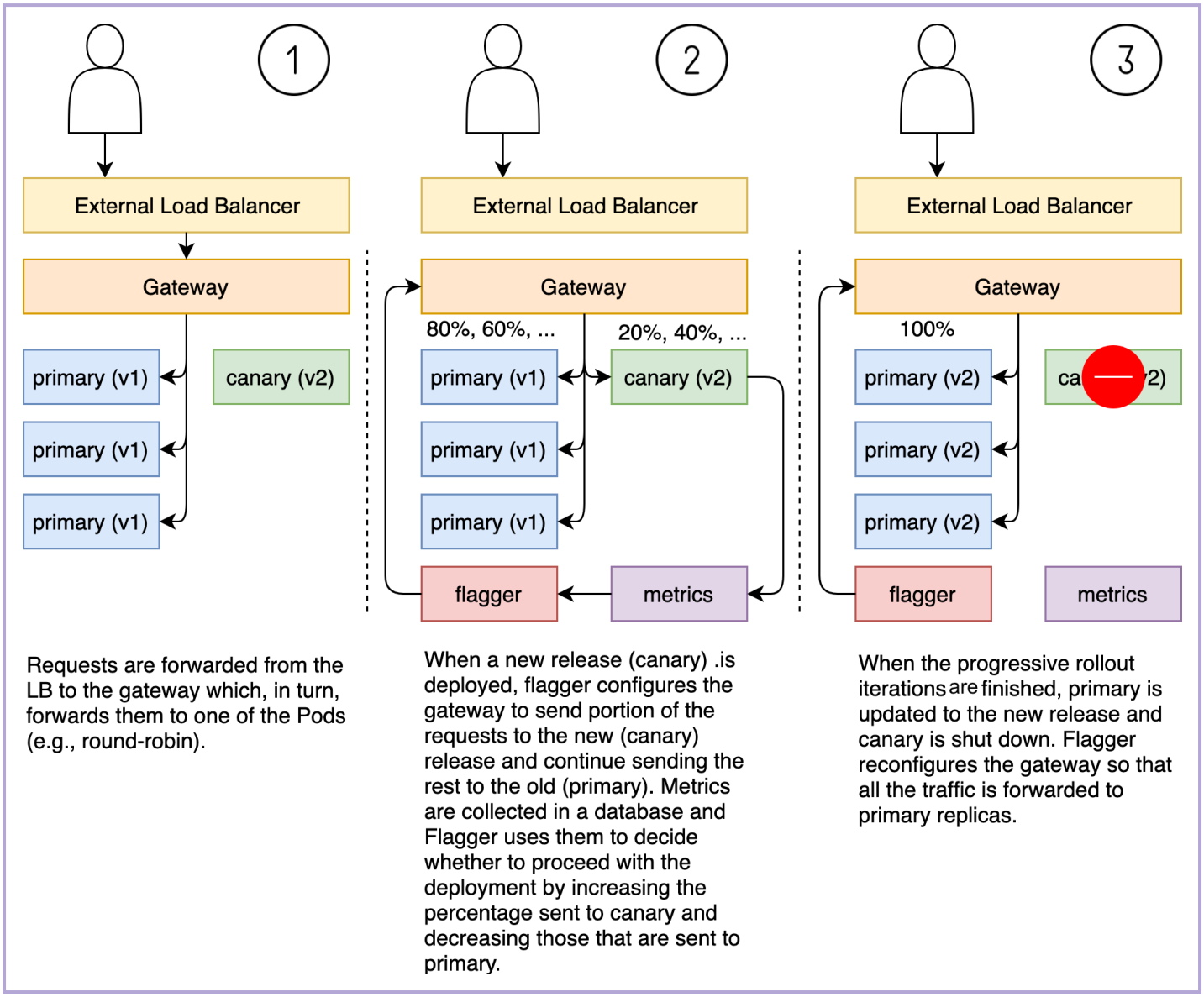
```
...
Events:
  Type    Reason Age   From    Message
  ----    ------ ----  ----    -------
...
  Normal  Synced 3m32s flagger New revision detected! Scaling up jx-jx-progressive.jx-staging
  Normal  Synced 3m2s  flagger Starting canary analysis for jx-jx-progressive.jx-staging
  Normal  Synced 3m2s  flagger Advance jx-progressive.jx-staging canary weight 20
  Normal  Synced 2m32s flagger Advance jx-progressive.jx-staging canary weight 40
```

```
  Normal  Synced 2m2s  flagger Advance jx-progressive.jx-staging canary weight 60
  Normal  Synced 92s   flagger Advance jx-progressive.jx-staging canary weight 80
  Normal  Synced 92s   flagger Copying jx-progressive.jx-staging template spec to jx-progressive-p

  Normal  Synced 62s   flagger Routing all traffic to primary
  Normal  Synced 32s   flagger Promotion completed! Scaling down jx-progressive.jx-staging
```

If one of the last two events does not state `promotion completed`, please wait for a while longer for the process to finish and re-run the `kubectl describe` command.

We can see that when the deployment was initiated, **Flagger** detected that there is a new `revision` (a new release). As a result, it started scaling the application. A while later, it initiated the analysis that consists of evaluations of the metrics (`request-success-rate` and `request-duration`) against the thresholds we defined earlier. Further on, we can see that it was increasing the weight every thirty seconds until it reached `80` percent. That number is vital given that it is the first iteration with the `weight` equal to or above the `maxWeight` which we set to `70` percent. After that, it did not wait for another thirty seconds. Instead, it replaced the definition of the `primary` template to the one used for `canary`. From that moment on, the `primary` was updated to the new release and all the traffic is being routed to it. Finally, the last event was the message that the `promotion` was `completed` and that the `canary` (`jx-progressive.jx-staging`) was scaled down to zero replicas. The last two events happened at the same time, so in your case, their order might be reverted.



Canary deployment rollout

We finally found a big difference between `Canary` and `RollingUpdate` deployments. That difference was in the evaluation of the metrics as a way to decide whether to proceed or not with the rollout. Given that everything worked correctly. We are yet to see what would happen if one of the metrics reached the threshold.

We're finished exploring the happy path, and we can just as well stop the loop. There's no need, for now, to continue sending requests. But, before we do that, I should explain that there was a reason for the loop beside the apparent need to see the progress.

If we were not sending requests to the application, there would be no metrics to collect. In such a case, **Flagger** would think that there's something fishy with the new release. Maybe it was so bad that no one could send a request to our application. Or maybe there was some other reason. In any case, lack of metrics

used to validate the release is considered a problem. For now, we wanted to see the

happy path with the new release fully rolled out. The stream of requests was there to ensure that there are sufficient metrics for **Flagger** to say: “everything’s fine; let’s move on.”

With that out of the way, please go back to the **second terminal**, stop the loop by pressing *ctrl+c*, and go back again to the **first terminal**.

Next, we’ll see how does the “unhappy” path look.