Published in Google Cloud - Community

Stefan Prodan Follow

Feb 21, 2019 · 10 min read · Listen

Automated canary deployments with Flagger and Istio

Continuous delivery is accepted as an enterprise software practice, and is a natural evolution of well-established continuous integration principles. However continuous deployment continues to be notably rare, perhaps due to the complexity of management and the fear of failed deployments impacting system availability.

Flagger is an open source Kubernetes operator that aims to untangle this complexity. It automates the promotion of canary deployments utilising Istio's traffic shifting and Prometheus metrics to analyse an application's behaviour during a controlled rollout.

Flagger can run automated application analysis, promotion and rollback for the following deployment strategies:

- Canary (progressive traffic shifting)
- A/B Testing (HTTP headers and cookies traffic routing)
- Blue/Green (traffic switch or mirroring)

For Canary deployments and A/B testing you'll need a Layer 7 traffic management solution like a service mesh (Istio, Linkerd, App Mesh) or an ingress controller (Contour, NGINX, Gloo). For Blue/Green deployments no service mesh or ingress

controller is required.

What follows is a step-by-step guide to setting up and using Flagger on Google Kubernetes Engine (GKE).

Kubernetes cluster setup

You'll begin by creating a GKE cluster with the Istio add-on (if you don't have a GCP account you can sign up here for free credits).

Login into Google Cloud, create a project and enable billing for it. Install the gcloud command line utility and configure your project with `gcloud init`.

Set the default project, compute region and zone (replace `PROJECT_ID` with your own project):

```
gcloud config set project PROJECT_ID
gcloud config set compute/region us-central1
gcloud config set compute/zone us-central1-a
```

Enable the GKE service and create a cluster with the HPA and Istio add-ons:

```
gcloud services enable container.googleapis.com

K8S_VERSION=$(gcloud beta container get-server-config
--format=json | jq -r '.validMasterVersions[0]')

gcloud beta container clusters create istio \
--cluster-version=${K8S_VERSION} \
--zone=us-central1-a \
--num-nodes=2 \
--machine-type=n1-standard-2 \
--disk-size=30 \
--enable-autorepair \
--no-enable-cloud-logging \
--no-enable-cloud-monitoring \
--addons=HorizontalPodAutoscaling,Istio \
--istio-config=auth=MTLS_PERMISSIVE
```

The above command will create a default node pool consisting of two `n1-standard-2` (vCPU: 2, RAM 7.5GB, DISK: 30GB) VMs. Ideally you would want to isolate the Istio components from your workloads but there is no easy way of

running the Istio pods on a dedicated node pool. The Istio manifests are considered read-only and GKE will undo any modification like node affinity or pod anti-affinity.

Set up credentials for `kubectl`:

```
gcloud container clusters get-credentials istio
```

Create a cluster admin role binding:

```
kubectl create clusterrolebinding "cluster-admin-$(whoami)" \
--clusterrole=cluster-admin \
--user="$(gcloud config get-value core/account)"
```

Install the Helm command-line tool:

```
brew install kubernetes-helm
```

Homebrew 2.0 is now also available for Linux.

Create a service account and a cluster role binding for Tiller:

```
kubectl -n kube-system create sa tiller && \
kubectl create clusterrolebinding tiller-cluster-rule \
--clusterrole=cluster-admin \
--serviceaccount=kube-system:tiller
```

Deploy Tiller in the `kube-system` namespace:

```
helm init --service-account tiller
```

You should consider using SSL between Helm and Tiller, for more information on securing your Helm installation see docs.helm.sh.

Validate your setup with:

```
kubectl -n istio-system get svc
```

In a couple of seconds GCP should allocate an external IP to the `istio-ingressgateway` service.

Istio ingress gateway setup

Create a static IP address named `istio-gateway` using the Istio ingress IP:

```
export GATEWAY_IP=$(kubectl -n istio-system get svc/istio-
ingressgateway -ojson | jq -r .status.loadBalancer.ingress[0].ip)

gcloud compute addresses create istio-gateway --addresses
${GATEWAY_IP} --region us-central1
```

Next you'll need an internet domain and access to your DNS registrar. Add two A records (replace `example.com` with your domain):

```
istio.example.com   A ${GATEWAY_IP}
*.istio.example.com A ${GATEWAY_IP}
```

Verify that the wildcard DNS is working:

```
watch host test.istio.example.com
```

Create a generic Istio gateway to expose services outside the mesh on HTTP:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: public-gateway
  namespace: istio-system
spec:
  selector:
```

```
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - "*"
```

Save the above resource as public-gateway.yaml and then apply it:

```
kubectl apply -f ./public-gateway.yaml
```

No production system should expose services on the internet without SSL. To secure the Istio ingress gateway with cert-manager, CloudDNS and Let's Encrypt please read Flagger GKE documentation.

Install Flagger

The GKE Istio add-on does not include a Prometheus instance that scrapes the Istio telemetry service. Because Flagger uses the Istio HTTP metrics to run the canary analysis you have to deploy the following Prometheus configuration that's similar to the one that comes with the official Istio Helm chart.

```
kubectl -n istio-system apply -f \
https://storage.googleapis.com/gke-release/istio/release/1.0.6-
gke.3/patches/install-prometheus.yaml
```

Add Flagger Helm repository:

```
helm repo add flagger https://flagger.app
```

Deploy Flagger in the `istio-system` namespace with Slack notifications enabled:

```
helm upgrade -i flagger flagger/flagger \
--namespace=istio-system \
```

```
--set metricsServer=http://prometheus.istio-system:9090 \
--set slack.url=https://hooks.slack.com/services/YOUR-WEBHOOK-ID \
--set slack.channel=general \
--set slack.user=flagger
```

You can install Flagger in any namespace as long as it can talk to the Istio Prometheus service on port 9090.

Flagger comes with a Grafana dashboard made for canary analysis. Install Grafana in the `istio-system` namespace:

```
helm upgrade -i flagger-grafana flagger/grafana \
--namespace=istio-system \
--set url=http://prometheus.istio-system:9090 \
--set user=admin \
--set password=change-me
```

Expose Grafana through the public gateway by creating a virtual service (replace `example.com` with your domain):

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: grafana
  namespace: istio-system
spec:
  hosts:
    - "grafana.istio.example.com"
  gateways:
    - public-gateway.istio-system.svc.cluster.local
  http:
    - route:
        - destination:
            host: flagger-grafana
```

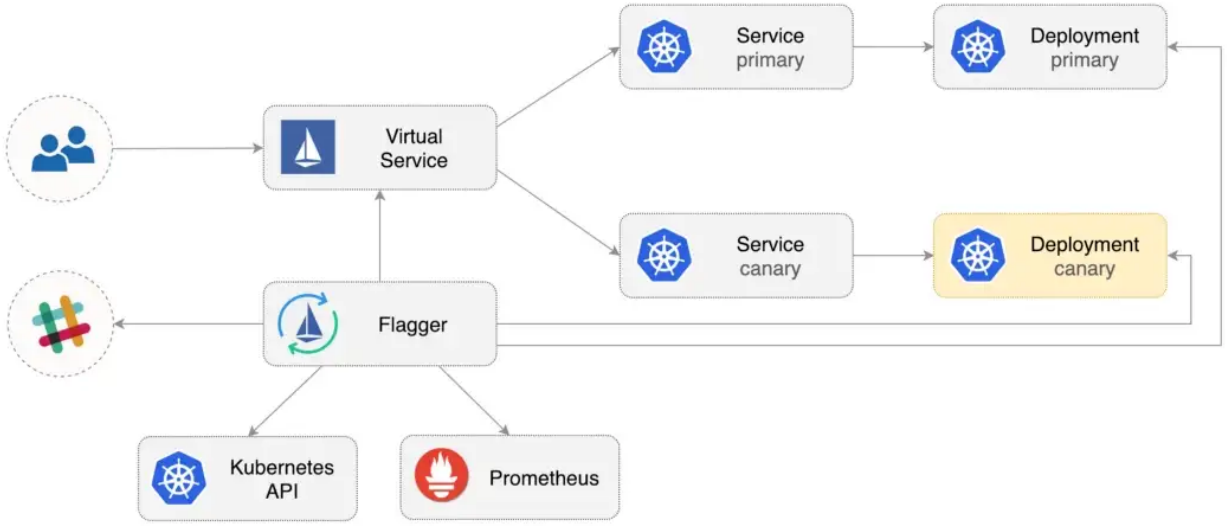
Save the above resource as grafana-virtual-service.yaml and then apply it:

```
kubectl apply -f ./grafana-virtual-service.yaml
```

Navigate to `http://grafana.istio.example.com` in your browser and you should be redirected to Grafana's login page.

Deploy web applications with Flagger

Flagger takes a Kubernetes deployment and optionally a horizontal pod autoscaler (HPA), then creates a series of objects (Kubernetes deployments, ClusterIP services and Istio virtual services). These objects expose the application on the mesh and drive the canary analysis and promotion.



Create a test namespace with Istio sidecar injection enabled:

```
kubectl create ns test
kubectl label namespace test istio-injection=enabled
```

Create a deployment and a horizontal pod autoscaler:

```
kubectl apply -k github.com/weaveworks/flagger//kustomize/podinfo
```

Deploy the load testing service to generate traffic during the canary analysis:

```
helm upgrade -i flagger-loadtester flagger/loadtester
--namepace=test
```

Create a canary custom resource (replace `example.com` with your own domain):

```
apiVersion: flagger.app/v1beta1
kind: Canary
metadata:
  name: podinfo
  namespace: test
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  progressDeadlineSeconds: 60
  autoscalerRef:
    apiVersion: autoscaling/v2beta1
    kind: HorizontalPodAutoscaler
    name: podinfo
  service:
    port: 9898
    gateways:
    - public-gateway.istio-system.svc.cluster.local
    hosts:
    - app.istio.example.com
    trafficPolicy:
      tls:
        # use ISTIO_MUTUAL when mTLS is enabled
        mode: DISABLE
  analysis:
    interval: 30s
    threshold: 10
    maxWeight: 50
    stepWeight: 5
    metrics:
    - name: request-success-rate
      threshold: 99
      interval: 30s
    - name: request-duration
      threshold: 500
      interval: 30s
    webhooks:
      - name: load-test
        url: http://flagger-loadtester.test/
        timeout: 5s
        metadata:
          cmd: "hey -z 1m -q 10 -c 2 http://podinfo-
canary.test:9898/"
```

Save the above resource as podinfo-canary.yaml and then apply it:

```
kubectl apply -f ./podinfo-canary.yaml
```

The above analysis, if it succeeds, will run for five minutes while validating the HTTP metrics every half minute. You can determine the minimum time that it takes to validate and promote a canary deployment using the formula: `interval * (maxWeight / stepWeight)`. The Canary CRD fields are documented here.

After a couple of seconds Flagger will create the canary objects:

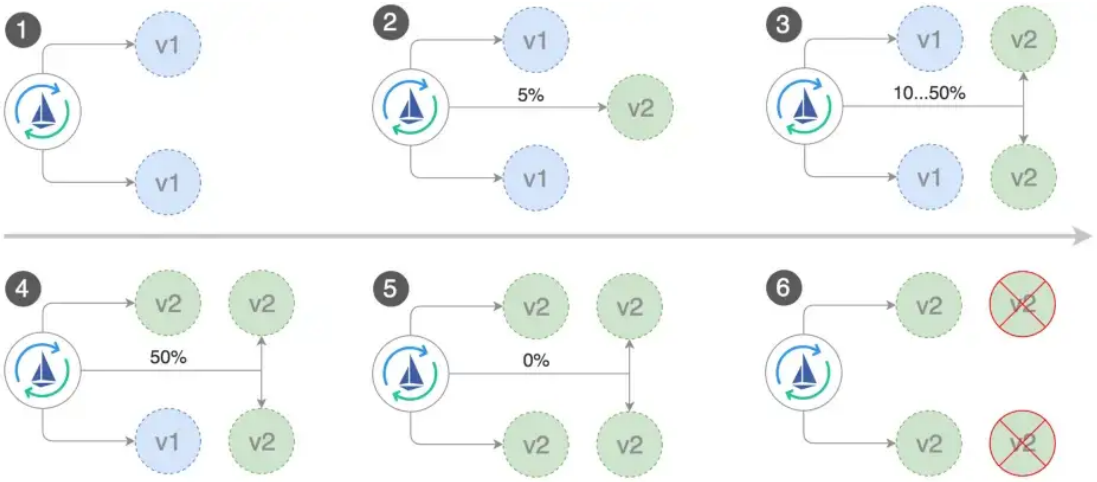
```
# applied
deployment.apps/podinfo
horizontalpodautoscaler.autoscaling/podinfo
canary.flagger.app/podinfo

# generated
deployment.apps/podinfo-primary
horizontalpodautoscaler.autoscaling/podinfo-primary
service/podinfo
service/podinfo-canary
service/podinfo-primary
destinationrule.networking.istio.io/podinfo-canary
destinationrule.networking.istio.io/podinfo-primary
virtualservice.networking.istio.io/podinfo
```

Open your browser and navigate to `app.istio.example.com`, you should see the version number of the demo app.

Automated canary analysis and promotion

Flagger implements a control loop that gradually shifts traffic to the canary while measuring key performance indicators like HTTP requests success rate, requests average duration and pod health. Based on analysis of the KPIs a canary is promoted or aborted, and the analysis result is published to Slack.



A canary deployment is triggered by changes in any of the following objects:

- Deployment PodSpec (container image, command, ports, env, etc)
- ConfigMaps mounted as volumes or mapped to environment variables
- Secrets mounted as volumes or mapped to environment variables

Trigger a canary deployment by updating the container image:

```
kubectl -n test set image deployment/podinfo \
podinfod=quay.io/stefanprodan/podinfo:3.1.1
```

Flagger detects that the deployment revision changed and starts to analyse it:

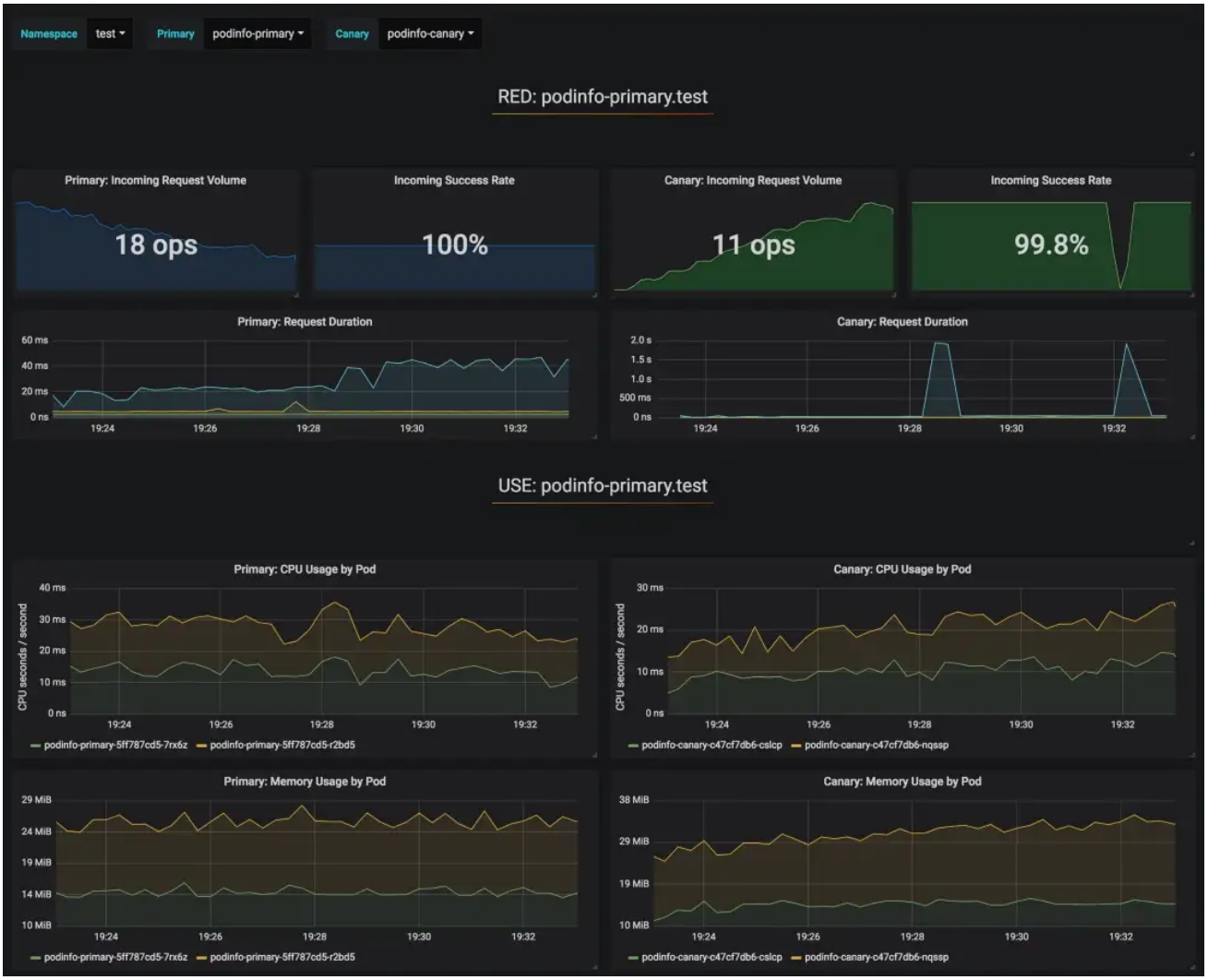
```
kubectl -n test describe canary/podinfo

Events:

New revision detected podinfo.test
Scaling up podinfo.test
Waiting for podinfo.test rollout to finish: 0 of 1 updated
replicas are available
Advance podinfo.test canary weight 5
Advance podinfo.test canary weight 10
Advance podinfo.test canary weight 15
Advance podinfo.test canary weight 20
Advance podinfo.test canary weight 25
Advance podinfo.test canary weight 30
Advance podinfo.test canary weight 35
Advance podinfo.test canary weight 40
Advance podinfo.test canary weight 45
```

```
Advance podinfo.test canary weight 50
Copying podinfo.test template spec to podinfo-primary.test
Waiting for podinfo-primary.test rollout to finish: 1 of 2 updated
replicas are available
Promotion completed! Scaling down podinfo.test
```

During the analysis the canary's progress can be monitored with Grafana:



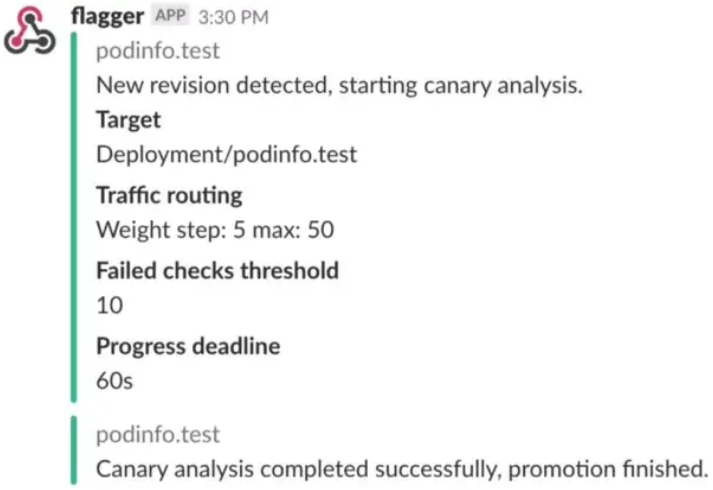
Note that if new changes are applied to the deployment during the canary analysis, Flagger will restart the analysis phase.

List all canaries in your cluster with:

```
watch kubectl get canaries --all-namespaces

NAMESPACE   NAME      STATUS        WEIGHT   LASTTRANSITIONTIME
test        podinfo   Progressing   15       2019-01-16T14:05:07Z
prod        frontend  Succeeded     0        2019-01-15T16:15:07Z
prod        backend   Failed        0        2019-01-14T17:05:07Z
```

If you've enabled the Slack notifications, you should receive the following messages:



Automated rollback

During the canary analysis it is possible to generate synthetic HTTP 500 errors and high response latency to test if Flagger pauses the rollout.

Create a tester pod and exec into it:

```
kubectl -n test run tester \
--image=quay.io/stefanprodan/podinfo:1.2.1 \
-- ./podinfo --port=9898

kubectl -n test exec -it tester-xx-xx sh
```

Generate HTTP 500 errors:

```
watch curl http://podinfo-canary:9898/status/500
```

Generate latency:

```
watch curl http://podinfo-canary:9898/delay/1
```

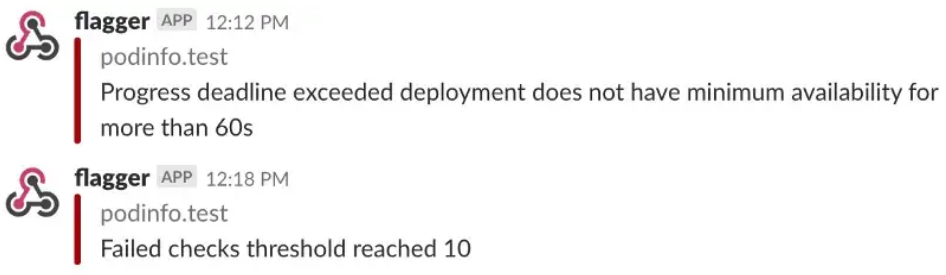
When the number of failed checks reaches the canary analysis threshold, the traffic is routed back to the primary, the canary is scaled to zero and the rollout is marked as failed.

The canary errors and latency spikes have been recorded as Kubernetes events and logged by Flagger in JSON format:

```
kubectl -n istio-system logs deployment/flagger -f | jq .msg

Starting canary deployment for podinfo.test
Advance podinfo.test canary weight 5
Advance podinfo.test canary weight 10
Advance podinfo.test canary weight 15
Halt podinfo.test advancement success rate 69.17% < 99%
Halt podinfo.test advancement success rate 61.39% < 99%
Halt podinfo.test advancement success rate 55.06% < 99%
Halt podinfo.test advancement success rate 47.00% < 99%
Halt podinfo.test advancement success rate 37.00% < 99%
Halt podinfo.test advancement request duration 1.515s > 500ms
Halt podinfo.test advancement request duration 1.600s > 500ms
Halt podinfo.test advancement request duration 1.915s > 500ms
Halt podinfo.test advancement request duration 2.050s > 500ms
Halt podinfo.test advancement request duration 2.515s > 500ms
Rolling back podinfo.test failed checks threshold reached 10
Canary failed! Scaling down podinfo.test
```

If you've enabled the Slack notifications, you'll receive a message if the progress deadline is exceeded, or if the analysis reached the maximum number of failed checks:



Traffic mirroring

For applications that perform read operations, Flagger can be configured to drive canary releases with traffic mirroring. Istio traffic mirroring will copy each incoming request, sending one request to the primary and one to the canary service. The response from the primary is sent back to the user and the response from the canary is discarded. Metrics are collected on both requests so that the deployment will only proceed if the canary metrics are within the threshold values.

Note that mirroring should be used for requests that are **idempotent** or capable of being processed twice (once by the primary and once by the canary).

You can enable mirroring by replacing `stepWeight/maxWeight` with `iterations` and by setting `canaryAnalysis.mirror` to `true`:

```
apiVersion: flagger.app/v1beta1
kind: Canary
spec:
  analysis:
    interval: 1m
    threshold: 5
    iterations: 10
    mirror: true
    metrics:
    - name: request-success-rate
      threshold: 99
      interval: 1m
    - name: request-duration
      threshold: 500
      interval: 1m
    webhooks:
      - name: acceptance-test
        type: pre-rollout
        url: http://flagger-loadtester.test/
        timeout: 30s
        metadata:
          type: bash
          cmd: "curl -sd 't' podinfo-canary:9898/token | grep
token"
      - name: load-test
        url: http://flagger-loadtester.test/
        timeout: 5s
        metadata:
          cmd: "hey -z 1m -q 10 -c 2 http://podinfo.test:9898/"
```

With the above configuration, Flagger will run a canary release with the following steps:

- detect new revision (deployment spec, secrets or configmaps changes)
- scale from zero the canary deployment
- wait for the HPA to set the canary minimum replicas
- check canary pods health
- run the acceptance tests
- abort the canary release if tests fail
- start the load tests
- mirror traffic from primary to canary
- check request success rate and request duration every minute
- abort the canary release if the metrics check failure threshold is reached
- stop traffic mirroring after the number of iterations is reached
- route live traffic to the canary pods
- promote the canary (update the primary secrets, configmaps and deployment spec)
- wait for the primary deployment rollout to finish
- wait for the HPA to set the primary minimum replicas
- check primary pods health
- switch live traffic back to primary
- scale to zero the canary
- send notification with the canary analysis result

The above procedure can be extended with custom metrics checks, webhooks, manual promotion approval and Slack or MS Teams notifications.

Wrapping up

Running a service mesh like Istio on top of Kubernetes gives you automatic metrics, logs, and traces but deployment of workloads still relies on external tooling. Flagger aims to change that by extending Istio with progressive delivery capabilities.

Flagger is compatible with any CI/CD solutions made for Kubernetes, and the canary analysis can be easily extended with webhooks for running system integration/acceptance tests, load tests, or any other custom validation. Since Flagger is declarative and reacts to Kubernetes events, it can be used in GitOps pipelines together with FluxCD or JenkinsX. If you're using JenkinsX you can install Flagger using the jx addons.

Kubernetes Istio Continuous Integration Continuous Delivery Microservices

Flagger is sponsored by Weaveworks and powers the canary deployments in Weave Cloud. The project is being tested on GKE, EKS and bare metal clusters provisioned with kubeadm.

1K 4

If you have any suggestion on improving Flagger please submit an issue or PR on GitHub at weaveworks/flagger. Contributions are more than welcome!