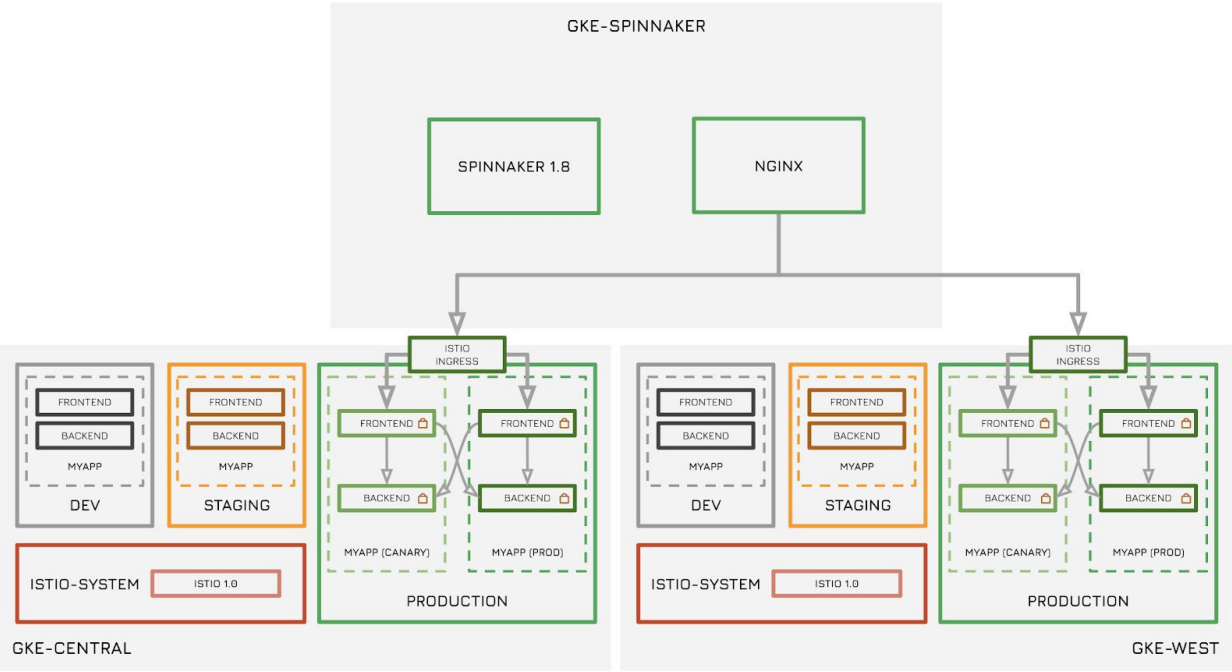
Application and Traffic Management in a Multi-cluster Kubernetes Environment

Overview



Kubernetes is quickly becoming the platform of choice for many enterprise developers and devops. As environments grow, a single Kubernetes cluster is no longer an option. Deploying and running applications in multiple Kubernetes clusters comes with added complexity. Thanks to the open source community, we have numerous tools to help us manage multiple clusters and application deployment across these clusters. This lab focuses on best practices around operating multiple clusters, managing application deployments across multiple clusters, and advanced routing scenarios in a multi-cluster Kubernetes environment.

What You'll Learn

In this workshop you will learn how:

- **Build**

    - Developers use tools such as Skaffold for local development and testing prior to deploying it in staging/production environment.
    - Once local development is completed, developers can use Cloud Build to store the image in a container repository such as Google Container Registry (GCR.io).
- **Deploy**
    - Deploy applications to multiple Kubernetes clusters using Spinnaker (an open source continuous delivery tool).
    - Trigger Continuous Delivery pipelines using Spinnaker to deploy an application across multiple clusters to staging, canary and production phases.
- **Control**
    - Use NGINX to globally load balance traffic to multiple Kubernetes clusters
    - Use Istio advanced routing to manipulate traffic to production and canary versions of the application within each cluster
    - Secure microservices traffic using mTLS
    - Control traffic using policy/quota management and enforcement with Istio.
- **Monitor**
    - Monitor traffic using open source tools such as:
        - Metric and visualization with Prometheus and Grafana
        - Distributed tracing with Jaeger
        - Microservices visualization with ServiceGraph
        - Logging and troubleshooting with Google Stackdriver (GCP Monitoring and management for services, containers, applications, and infrastructure)

For this workshop, you utilize 3 Kubernetes Engine clusters.

- `gke-central` and `gke-west` are deployed in `us-central1` and `us-west1` regions respectively, and are used for running a simple two tier web application written in Go.
- `gke-spinnaker` is used for continuous delivery/deployment of the web app using Spinnaker to `gke-central` and `gke-west`. This cluster is also used for globally load balancing application traffic to `gke-central` and `gke-west` using an NGINX load balancer.
- `gke-central` and `gke-west` also have Istio Service Mesh installed. You use Istio's advanced traffic management features to manipulate traffic to different versions of the application within each cluster. You also use Istio to secure service-to-service communication. And finally, use traffic policy and quota management to rate limit traffic to the application.

Setup

What you'll need

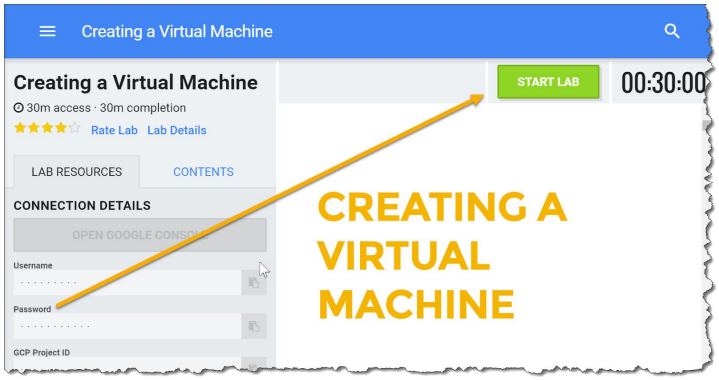
To complete this lab, you'll need:

- Access to a standard internet browser (Chrome browser recommended).
- Time. Note the lab's **Completion** time in Qwiklabs, which is an estimate of the time it should take to complete all steps. Plan your schedule so you have time to complete the lab. Once you start the lab, you will not be able to pause and return later (you begin at step 1 every time you start a lab).
- You do NOT need a Google Cloud Platform account or project. An account, project and associated resources are provided to you as part of this lab.
- If you already have your own GCP account, make sure you do not use it for this lab.
- If your lab prompts you to log into the console, **use only the student account provided to you by the lab**. This prevents you from incurring charges for lab activities in your personal GCP account.

Use a new Incognito window (Chrome) or another browser for the Qwiklabs session. Alternatively, you can log out of all other Google / Gmail accounts before beginning the labs.

Start your lab

When you are ready, click **Start Lab**. You can track your lab's progress with the status bar at the top of your screen.

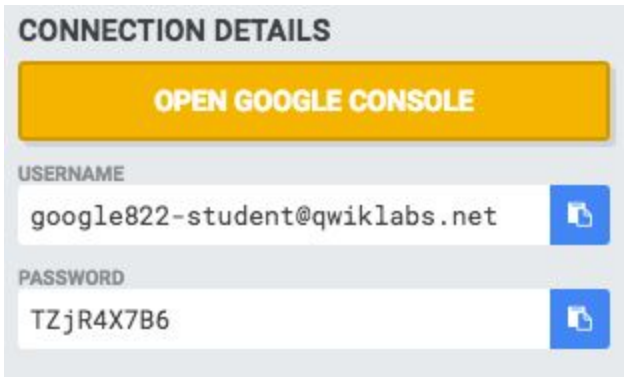


Important: What is happening during this time?

Your lab is spinning up GCP resources for you behind the scenes, including an account, a project, resources within the project, and permission for you to control the resources you will need to run the lab. This means that instead of spending time manually setting up a project and building resources from scratch as part of your lab, you can begin learning more quickly.

Find Your Lab's GCP Username and Password

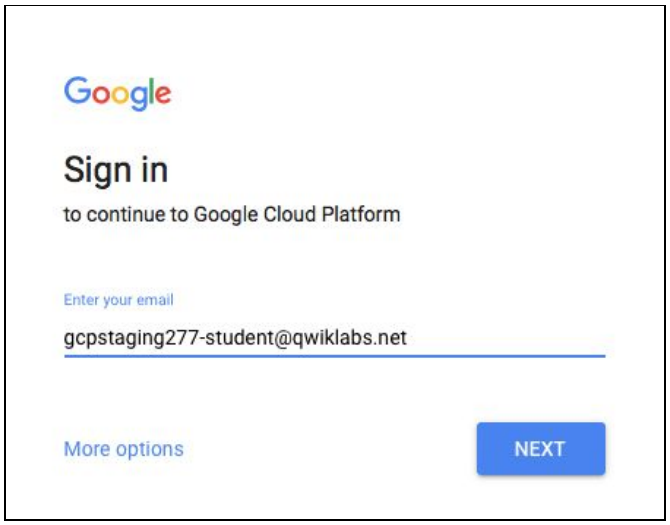
To access the resources and console for this lab, locate the Connection Details panel in Qwiklabs. Here you will find the account ID and password for the account you will use to log in to the Google Cloud Platform:



If your lab provides other resource identifiers or connection-related information, it will appear on this panel as well.

Sign in to Google Cloud Console

Using the Qwiklabs browser tab/window (preferably in Incognito mode) or the separate browser you are using for the Qwiklabs session, copy the Username from the Connection Details panel and click **Open Google Console**. Paste in the Username and then the Password as prompted:

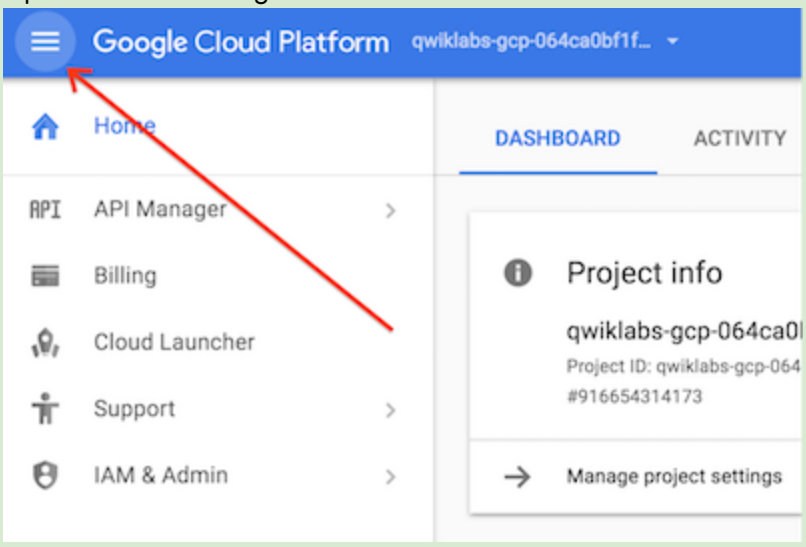


Accept the terms and conditions.

Because this is a temporary account, which you will only have access to for this one lab:

- Do not add recovery options
- Do not sign up for free trials

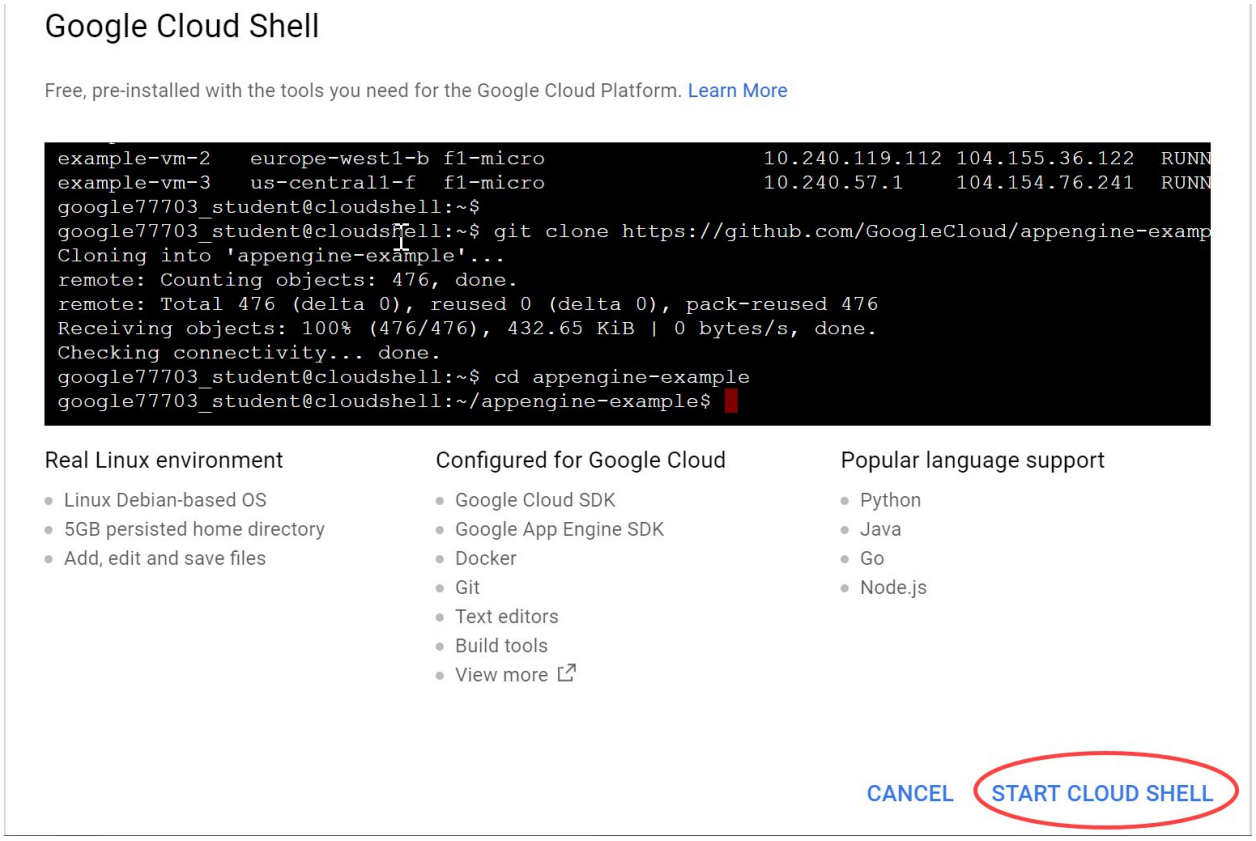
Note: You can view the menu with a list of GCP Products and Services by clicking the button at the top-left next to "Google Cloud Platform."



Activate Google Cloud Shell

From the GCP Console click the Cloud Shell icon on the top right toolbar:

Then click “Start Cloud Shell”:

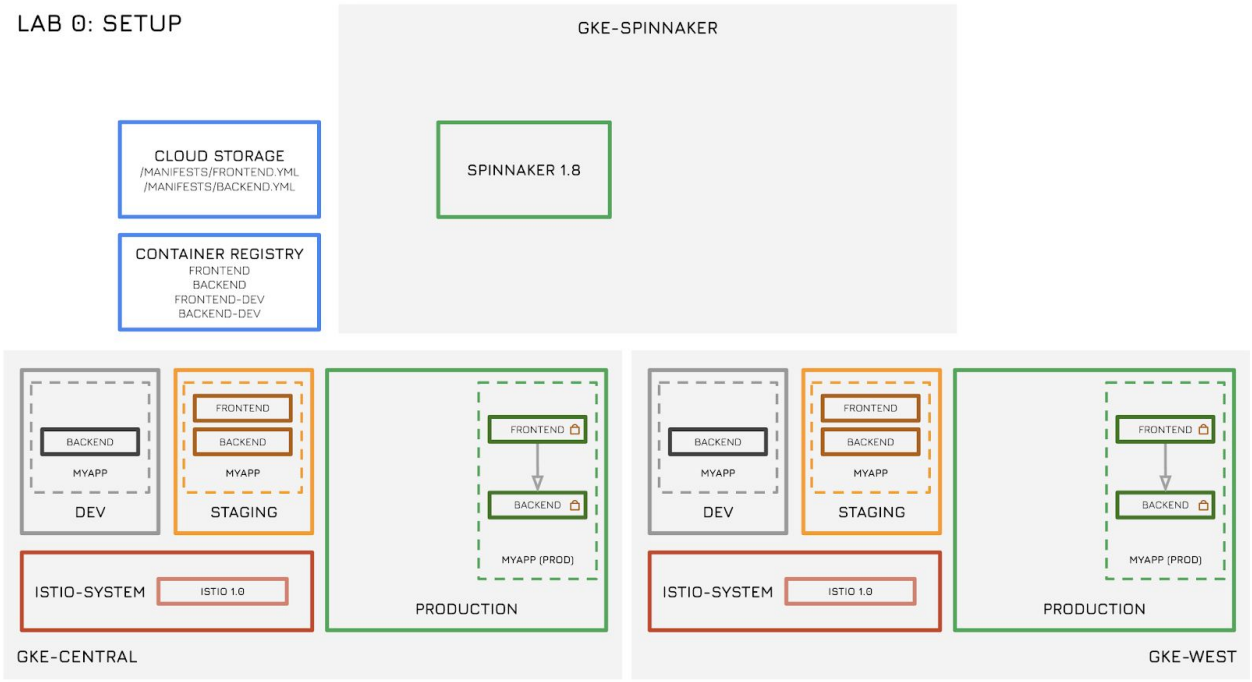


It should only take a few moments to provision and connect to the environment:

This virtual machine is loaded with all the development tools you'll need. It offers a persistent 5GB home directory, and runs on the Google Cloud, greatly enhancing network performance and authentication. Much, if not all, of your work in this lab can be done with simply a browser or your Google Chromebook.

Once connected to the cloud shell, you should see that you are already authenticated and that the project is already set to your *PROJECT_ID*.

Lab 0: Cluster Setup (30 mins)



In this lab, you:

- Download all associated files from a gitlab repo.
- Run a script that performs the following bootstrapping tasks:
    - Installs pertinent tools for the workshop in Cloud Shell. The following tools are installed:
        - kubectx/kubens
        - Helm
        - Html2text
        - kube-ps1
        - Hey

        - Terraform
        - Skaffold
    - Create service accounts for Spinnaker and Terraform to be able to access and edit GCP resources.
    - Prepare Google Container Registry (GCR) with the application. You use a simple two tier web application with a `frontend` and a `backend` microservice.
    - Prepare Google Cloud Storage (GCS) for Spinnaker configurations and kubernetes manifests.
    - Create Google PubSub topics for GCR and GCS to trigger Spinnaker pipelines.
    - Install three Kubernetes Engine Clusters. `gke-west`, `gke-central` and `gke-spinnaker`.
    - Install Istio on `gke-west` and `gke-central` clusters.
    - Install and configure Spinnaker and Halyard on `gke-spinnaker` cluster.

Cloning the repo

Get all files associated with this workshop by cloning the following repo.

```
git clone https://github.com/FairwindsOps/advanced-kubernetes-workshop
```

Running install scripts

Run the following command to kick off the script to install all resources for this workshop.

```
source ~/advanced-kubernetes-workshop/setup/setup.sh
```

Note: This step takes about 20 - 25 minutes to complete. Please ensure you do not close Cloud Shell during the setup.

Reviewing the environment

Once the script has finished, review the environment and ensure everything was installed properly.

Kubernetes Engine Clusters

Run the following command to ensure three Kubernetes Engine clusters were installed.

```
kubectx
```

Output (Do not copy)

```
gke-central
gke-west
gke-spinnaker
```

Confirm Istio is installed on `gke-west` and `gke-central` clusters. Ensure all services are up-to-date and available.

```
kubectl get deployments -n istio-system --context gke-west
kubectl get deployments -n istio-system --context gke-central
```

Output (Do not copy)

```
NAME                       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
grafana                    1         1         1            1           1m
istio-citadel              1         1         1            1           1m
istio-egressgateway        1         1         1            1           1m
istio-galley               1         1         1            1           1m
istio-ingressgateway       1         1         1            1           1m
istio-pilot                1         1         1            1           1m
istio-policy               1         1         1            1           1m
istio-sidecar-injector     1         1         1            1           1m
istio-statsd-prom-bridge   1         1         1            1           1m
istio-telemetry            1         1         1            1           1m
istio-tracing              1         1         1            1           1m
prometheus                 1         1         1            1           1m
servicegraph               1         1         1            1           1m
```

Confirm Spinnaker is installed on `gke-spinnaker` cluster. Ensure all services are up-to-date and available.

```
kubectl get deployments --context gke-spinnaker
```

Output (Do not copy)

```
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
spin-clouddriver   1         1         1            1           2m
spin-deck          1         1         1            1           2m
spin-echo          1         1         1            1           2m
spin-front50       1         1         1            1           2m
spin-gate          1         1         1            1           2m
spin-igor          1         1         1            1           2m
spin-minio         1         1         1            1           5m
spin-orca          1         1         1            1           2m
spin-rosco         1         1         1            1           1m
```

Check namespaces in `gke-west` and `gke-central` clusters. Confirm *ISTIO-INJECTION* is enabled on both `gke-central` and `gke-west` for `default`, `staging` and `production` namespaces.

```
kubectl get namespace -L istio-injection --context gke-central
kubectl get namespace -L istio-injection --context gke-west
```

Output (Do not copy)

```
NAME           STATUS    AGE       ISTIO-INJECTION
default        Active    1h        enabled
dev            Active    1h
istio-system   Active    1h
kube-public    Active    1h
kube-system    Active    1h
production     Active    1h        enabled
spinnaker      Active    1h
staging        Active    1h        enabled
```

Application

The application used in this workshop is composed of two microservices: `frontend` and `backend`. Both are web servers written in Go.

The application is installed in three namespaces on both clusters. `production` and `staging` namespaces are used for their named functions and `dev` namespace is used for local development.

Confirm the application is installed on both `gke-west` and `gke-central` clusters in all three namespaces.

```
kubectl get all -n production --context gke-central
kubectl get all -n staging --context gke-central
kubectl get all -n dev --context gke-central
```

```
kubectl get all -n production --context gke-west
kubectl get all -n staging --context gke-west
kubectl get all -n dev --context gke-west
```

Output excerpt for production namespace (Do not copy)

```
NAME                     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/backend-primary   5         5         5            5           1h
deploy/frontend-primary  5         5         5            5           1h

NAME           TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
svc/backend    ClusterIP   10.2.0.49     <none>        80/TCP    1h
svc/frontend   ClusterIP   10.2.12.227   <none>        80/TCP    1h
```

Output excerpt for staging namespace (Do not copy)

```
NAME                     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/backend-primary   5         5         5            5           1h
deploy/frontend-primary  5         5         5            5           1h
```

```
NAME           TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
svc/backend    ClusterIP   10.2.12.194   <none>        80/TCP    1h
svc/frontend   ClusterIP   10.2.14.225   <none>        80/TCP    1h
```

Output excerpt for dev namespace (Do not copy)

```
NAME                         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/backend-deploy-dev    1         1         1            1           1h


NAME                    TYPE           CLUSTER-IP   EXTERNAL-IP     PORT(S)
AGE
svc/backend-svc-dev     ClusterIP      10.2.13.66   <none>          80/TCP
1h
svc/frontend-svc-dev    LoadBalancer   10.2.4.239   35.224.34.160   80:32430/TCP
1h
```

The `production` and `staging` namespaces look exactly the same. Both `frontend` and `backend` deployments have 5 replicas and both services are exposed as *ClusterIP* (you use Istio `ingressgateway` in Lab 3 to expose these externally). The `dev` namespace only has a single replica deployment for both services and the `frontend` service is being exposed as service type `LoadBalancer` to quick be able to access it externally during local development phase.

Google Container Registry

Ensure that you have four images in GCR.

```
gcloud container images list
```

Output (Do not copy)

```
NAME
gcr.io/qwiklabs-gcp-04480262ec37ae63/backend
gcr.io/qwiklabs-gcp-04480262ec37ae63/backend-dev
gcr.io/qwiklabs-gcp-04480262ec37ae63/frontend
gcr.io/qwiklabs-gcp-04480262ec37ae63/frontend-dev
```

The `frontend` and `backend` images are used for `staging` and `production` namespace deployments. `frontend-dev` and `backend-dev` images are used for local development. This is discussed further in Lab 1.

Google Cloud Storage

Ensure you have the kubernetes manifests in Google Cloud Storage.

```
export PROJECT=$(gcloud info --format='value(config.project)')
gsutil ls gs://$PROJECT-spinnaker/manifests
```

Output (Do not copy)

```
gs://qwiklabs-gcp-04480262ec37ae63-spinnaker/manifests/backend.yml
gs://qwiklabs-gcp-04480262ec37ae63-spinnaker/manifests/frontend.yml
```

Each of the `yml` files correspond to the respective microservice of the application. This is discussed further in Lab 2.

Google Cloud PubSub

Ensure you have topics and subscriptions created in your GCP Project for **GCR** and **GCS**. This is discussed further in Lab 2.

```
gcloud pubsub topics list
```

Output (Do not copy)

```
---
name: projects/qwiklabs-gcp-04480262ec37ae63/topics/gcr
---
name: projects/qwiklabs-gcp-04480262ec37ae63/topics/spin-gcs-topic
```

And

```
gcloud pubsub subscriptions list
```

Output (Do not copy)

```
---
ackDeadlineSeconds: 10
messageRetentionDuration: 604800s
name: projects/qwiklabs-gcp-04480262ec37ae63/subscriptions/my-gcs-sub
pushConfig: {}
topic: projects/qwiklabs-gcp-04480262ec37ae63/topics/spin-gcs-topic
---
ackDeadlineSeconds: 10
messageRetentionDuration: 604800s
name: projects/qwiklabs-gcp-04480262ec37ae63/subscriptions/my-gcr-sub
pushConfig: {}
topic: projects/qwiklabs-gcp-04480262ec37ae63/topics/gcr
```

Exposing service ports

In this workshop, you access various Kubernetes add-ons using kubectl port-forward functionality. Here is a list of services and their respective ports for access for both `gke-west` and `gke-central` clusters.

Service	gke-central port	gke-west port	gke-spinnaker port
Spinnaker	-	-	8080
Prometheus	9000	9001	-
Grafana	3000	3001	-
Jaeger	16686	16687	-
Servicegraph	8088	8089	-

Run the following script to expose all of these port.

```
workshop_connect
```

Output (Do not copy)

```
Spinnaker Deck Port opened on 8080
Prometheus Port opened on 9090 for GKE Central
Prometheus Port opened on 9091 for GKE East
Grafana Port opened on 3000 for GKE Central
Grafana Port opened on 3001 for GKE East
Jaeger Port opened on 16686 for GKE Central
Jaeger Port opened on 16687 for GKE East
Servicegraph Port opened on 8088 for GKE Central
Servicegraph Port opened on 8089 for GKE East
```

Congratulations! You have successfully installed all the pertinents components required for this workshop.

Troubleshooting Lab 0

If you run into problems once the script has run, there are a few steps you can try to recover on your own.

First, try re-running the setup script. It will go much more quickly the second time:

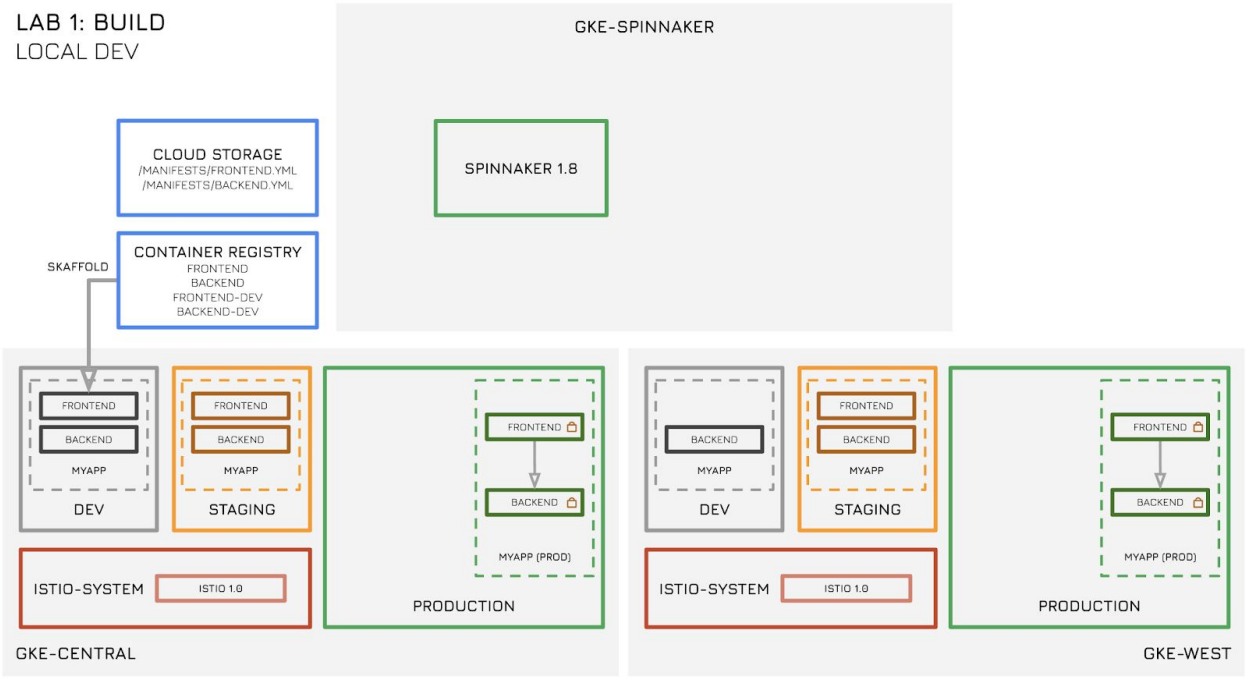
```
source ~/advanced-kubernetes-workshop/setup/setup.sh
```

Next, you can at any time re-run the terraform used to setup most of the lab:

```
workshop_terraform-apply
```

END OF LAB 0 - Please STOP and wait for instructions before proceeding!

Lab 1: Build (45 mins)



In this lab, you:

- Build and iterate on the web application.
- Use Skaffold for local development. Iterate, build ,deploy and test using Skaffold.
- Use dev namespace for local development.
- Clean up dev namespace after local development.

Developers typically build applications on their laptops. For developing on Kubernetes, sometimes minikube is used which is a single node Kubernetes cluster that runs locally on a laptop. Alternatively, you can use any Kubernetes cluster for local development. For this lab, you use `gke-central` cluster and namespace `dev` for local development.

Inspecting the current dev environment

Check your current `dev` namespace and environment.

```
kubectl get all -n dev --context gke-central
```

Output excerpt (Do not copy)

```
NAME                          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/backend-deploy-dev     1         1         1            1           1h

NAME                                      READY     STATUS    RESTARTS   AGE
po/backend-deploy-dev-745858bcdc-8hnfx    1/1       Running   0          2h

NAME                    TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)
AGE
svc/backend-svc-dev     ClusterIP      10.2.13.66    <none>          80/TCP
1h
svc/frontend-svc-dev    LoadBalancer   10.2.4.239    35.224.34.160   80:32430/TCP
1h
```

The only thing currently running is a single replica of `backend` deployment. There is no `frontend` deployment currently running. You can also see a service exposing the (future) `frontend` service via a `LoadBalancer`. The *EXTERNAL-IP* of that `LoadBalancer` will be used to access the dev environment.

You can now make changes to your `frontend`. You use Skaffold for this task.

Skaffold is a command line tool that facilitates continuous development for Kubernetes applications. You can iterate on your application source code locally then deploy to local or remote Kubernetes clusters. Skaffold handles the workflow for building, pushing and deploying your application.

Navigate to the `frontend` folder.

```
cd ~/advanced-kubernetes-workshop/services/frontend/
```

List files in folder.

```
ls -l
```

Output (Do not copy)

```
total 36
-rwxr-xr-x 1 googlece9765_student googlece9765_student  134 Sep  5 09:22 build.sh
-rwxr-xr-x 1 googlece9765_student googlece9765_student  102 Sep  5 09:22 check.sh
drwxr-xr-x 2 googlece9765_student googlece9765_student 4096 Sep  5 09:22 content
-rw-r--r-- 1 googlece9765_student googlece9765_student  150 Sep  5 09:22 Dockerfile
-rw-r--r-- 1 googlece9765_student googlece9765_student  818 Sep  5 09:23
```

```
k8s-frontend-dev.yml
-rw-r--r-- 1 googlece9765_student googlece9765_student 1206 Sep  5 09:22 main.go
-rw-r--r-- 1 googlece9765_student googlece9765_student  283 Sep  5 09:23
skaffold.yaml
-rw-r--r-- 1 googlece9765_student googlece9765_student  146 Sep  5 09:22
update-frontend.sh
```

The `frontend` service is composed of two files.

1. `main.go` file containing the source code
2. `index.html` file inside the `content` subfolder, which presents the `frontend` webpage

Review the `Dockerfile` for the `frontend` service. This is the file used for building the container. Skaffold uses this for its build phase.

```
cat Dockerfile
```

Output (Do not copy)

```
FROM golang
ADD . /go/src/spinnaker.io/demo/k8s-demo
RUN go install spinnaker.io/demo/k8s-demo
ADD ./content /content
ENTRYPOINT /go/bin/k8s-demo
```

Skaffold also uses two additional YAML files.

1. `skaffold.yaml` which consists the build, push and deploy configurations for skaffold.
2. `k8s-frontend-dev.yml` which consists of the `frontend-dev` deployment yaml.

Review the `skaffold.yaml` file. The `build`, `deploy` and `profiles` sections correspond to artifacts, kubernetes deployment and container registry details respectively.

```
cat skaffold.yaml
```

Output (Do not copy)

```
apiVersion: skaffold/v1alpha2
kind: Config
build:
  artifacts:
  - imageName: gcr.io/qwiklabs-gcp-04480262ec37ae63/frontend-dev
deploy:
  kubectl:
    manifests:
      - k8s-*
profiles:
  - name: gcb
```

```
    build:
      googleCloudBuild:
        projectId: qwiklabs-gcp-04480262ec37ae63
```

Review the `k8s-frontend-dev.yml` file. This looks like a typical single replica kubernetes deployment and configmap. The configmap sets an environment variable that points to the `backend-dev` service address.

```
cat k8s-frontend-dev.yml
```

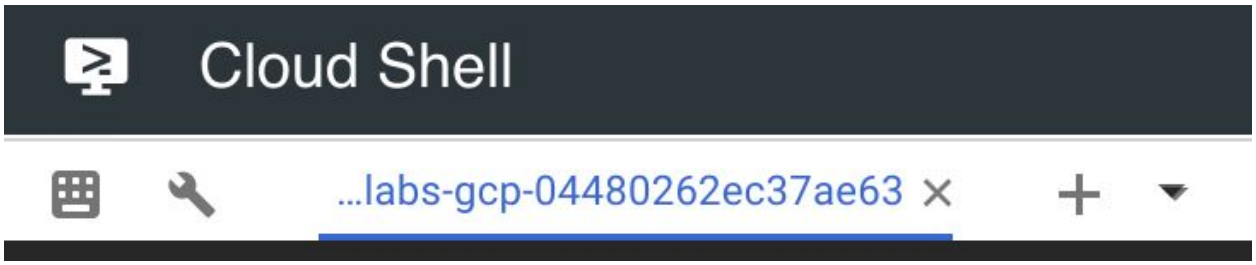
Output (Do not copy)

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: frontend-config-dev
  namespace: dev
data:
  BACKEND_ENDPOINT: 'http://backend-svc-dev.dev'
---
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: frontend-deploy-dev
  namespace: dev
  labels:
    app: frontend
    version: dev
    stack: frontend
    tier: dev
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
      version: dev
      stack: frontend
      tier: dev
  template:
    metadata:
      labels:
        app: frontend
        version: dev
        stack: frontend
        tier: dev
    spec:
      containers:
      - name: primary-dev
        image: gcr.io/qwiklabs-gcp-04480262ec37ae63/frontend-dev
        ports:
```

```
        - containerPort: 8000
        envFrom:
        - configMapRef:
            name: frontend-config-dev
```

Developing with Skaffold

For this lab, you use two Cloud Shell terminal windows. Open another Cloud Shell window by clicking the plus button next to the existing Cloud Shell window.



Ensure you are in the `~/advanced-kubernetes-workshop/services/frontend` in both Cloud Shell windows.

```
cd ~/advanced-kubernetes-workshop/services/frontend/
```

From Cloud Shell window # 1, switch to `gke-central` cluster context and run Skaffold in dev mode by running the following command.

```
kubectx gke-central
kubectl config set-context $(kubectl config current-context) --namespace=dev
skaffold dev
```

Output (Do not copy)

```
Starting build...
Building [gcr.io/qwiklabs-gcp-04480262ec37ae63/frontend-dev]...
...
...
Build complete in 44.728100636s
Starting deploy...
configmap "frontend-config-dev" created
deployment "frontend-deploy-dev" created
Deploy complete in 3.268022353s
[frontend-deploy-dev-fbc9fccc4-7qc4p primary-dev] Starting to service on port :8000
```

Skaffold builds the image and stores it in GCR in the `frontend-dev` image. After the build it deploys the `k8s-frontend-dev.yml` manifests which deploys the configmap and the deployment.

In the Cloud Shell window #2, confirm that the `frontend-dev` deployment is up and running in the `dev` namespace in the `gke-central` cluster.

```
kubectl get deploy -n dev --context gke-central
```

Output (Do not copy)

```
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
backend-deploy-dev   1         1         1            1           3h
frontend-deploy-dev  1         1         1            1           3m
```

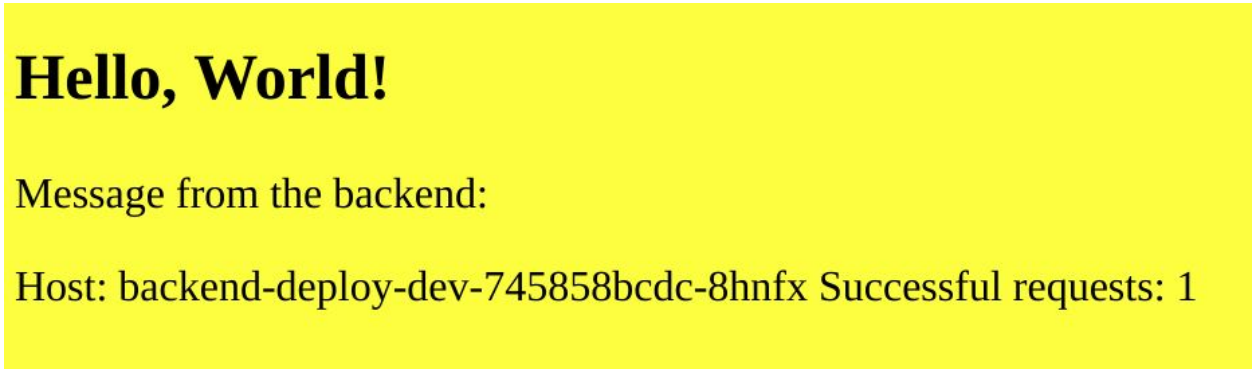
Get the `frontend-dev` service *EXTERNAL-IP* address.

```
kubectl get svc -n dev --context gke-central
```

Output (Do not copy)

```
NAME              TYPE           CLUSTER-IP   EXTERNAL-IP     PORT(S)        AGE
backend-svc-dev   ClusterIP      10.2.13.66   <none>          80/TCP         3h
frontend-svc-dev  LoadBalancer   10.2.4.239   35.224.34.160   80:32430/TCP   3h
3m
```

Open a new Chrome tab and copy/paste the *EXTERNAL-IP* address in the new tab.



You can see the current frontend web page. The frontend shows a “hello world” page with the hostname of the backend it is currently talking to, as well as the number of successful requests (if you refresh the page a few times **CTRL-R**, you can see that number increase).

For this lab, let’s change the background color from yellow to green.

You can either use “vi” or the Cloud Shell built in Code Editor (IDE style editor) to edit the `index.html` file in the content folder. To open the Code Editor from Cloud Shell, click on the pencil icon on the right hand side of the Cloud Shell top bar.

Alternatively, you change the color in the background using the following command.

```
sed -i -e s/yellow/green/g
$HOME/advanced-kubernetes-workshop/services/frontend/content/index.html
```

Switch back to Cloud Shell window #1 (where `skaffold dev` is running). Notice that the build and deploy process has restarted when the file `index.html` changed. Whenever skaffold detects a file system change (for instance changing the `index.html` file), it builds and deploys the new change instantly without the developer having to manually delete the old deployment, rebuild the new image and redeploy the manifest.

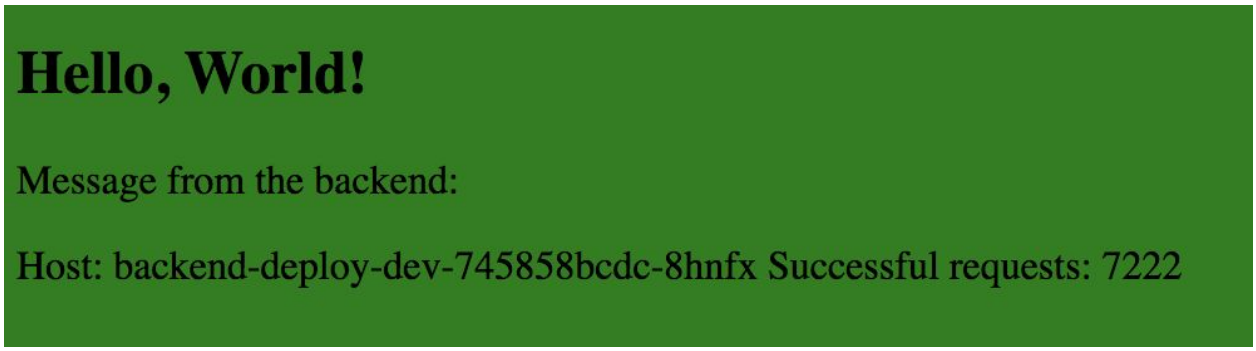
Confirm the change in color.

```
cat $HOME/advanced-kubernetes-workshop/services/frontend/content/index.html
```

Output (Do not copy)

```
<!DOCTYPE html>
<html>
  <body style="background-color:green">
    <h2>Hello, World!</h2>
    <p>Message from the backend:</p>
    <p>{{.Message}}</p>
    <p>{{.Feature}}</p>
  </body>
</html>
```

Confirm the change by refreshing the frontend web page.



This change is only applied to the dev version of the `frontend`.

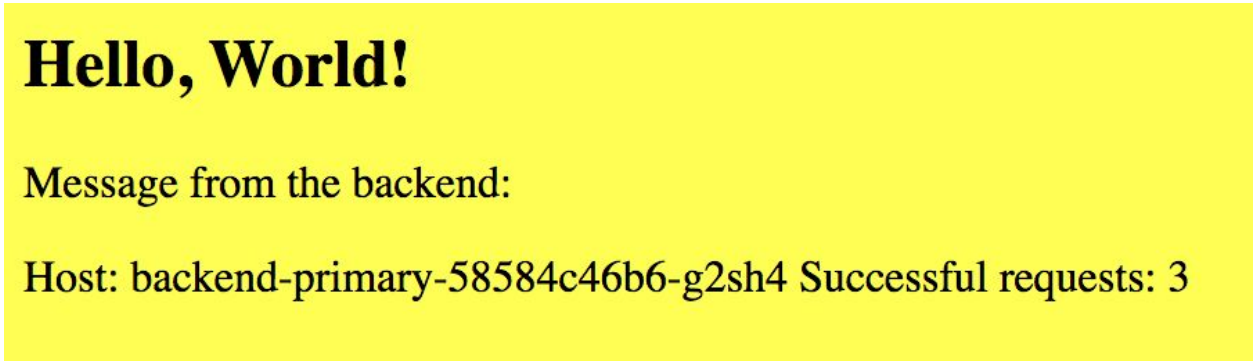
To confirm this, get the ingress IP addresses for the production `frontend` by running the following command.

```
workshop_get-ingress
```

Output (Do not copy)

```
gke-central ingress gateway:
35.202.138.224

gke-west ingress gateway:
35.236.250.232
```

Open a new Chrome tab and type the IP address on the `gke-central` ingress gateway in the new tab.



The production `frontend` still shows yellow background. You can also see the hostname of the backend which is pointing to the `primary` backend deployment.

Note: You can confirm the hostname by looking at the pods in the production namespace of the gke-central cluster.
`kubectl get pods -n production --context gke-central`

Optionally, you can change the color from green to something else (i.e. blue, purple, magenta, cyan etc) and watch skaffold rebuild and redeploy the new change instantly.

Once you are satisfied with your choice of color, exit out of `skaffold dev` command using **`CTRL-C`** key combination.

Exiting Skaffold deletes any artifacts deployed as part of `skaffold dev`.

Output (Do not copy)

```
Cleaning up...
configmap "frontend-config-dev" deleted
deployment "frontend-deploy-dev" deleted
```

```
Cleanup complete in 3.820695597s
```

Confirm by running the following command.

```
kubectl config set-context $(kubectl config current-context) --namespace=default
kubectl get pods -n dev --context gke-central
```

Output (Do not copy)

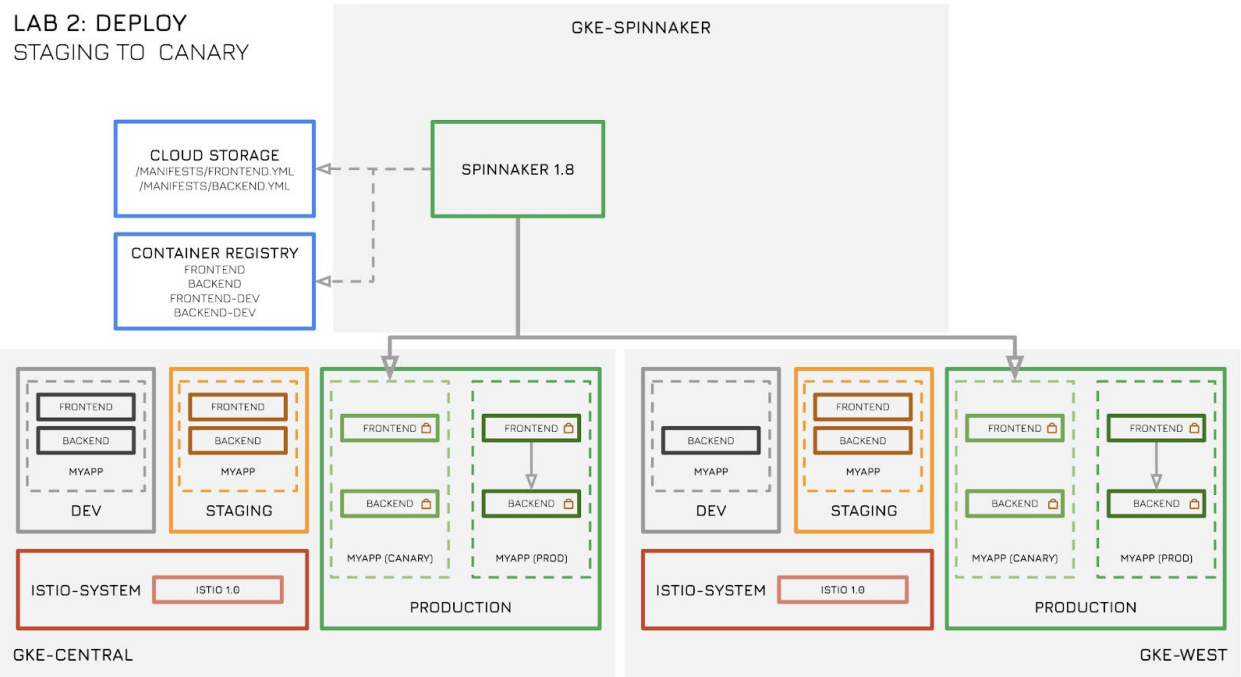
```
NAME                                  READY     STATUS    RESTARTS   AGE
backend-deploy-dev-745858bcdc-8hnfx   1/1       Running   0          9h
```

Close the Chrome tab for the `frontend-dev` webpage.

Congratulation! You can successfully iterate applications live on Kubernetes using Skaffold local development tool.

END OF LAB 1 - Please STOP and wait for instructions before proceeding!

Lab 2: Deploy (45 mins)



In this lab, you:

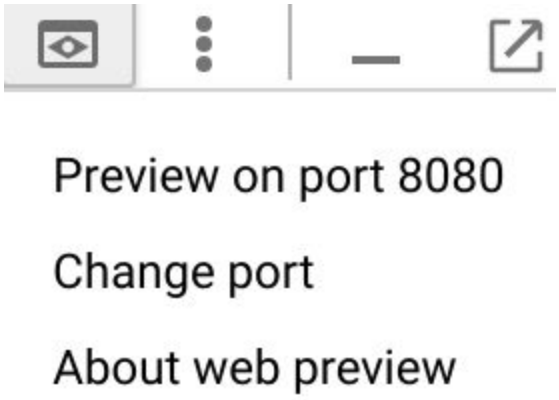
- Build the new image using Cloud Build.
- Use PubSub to trigger the Spinnaker pipeline.
- Deploy the new application `frontend` via Spinnaker to the `gke-west` and `gke-central` clusters.
- Deploy the new application `frontend` in `staging` namespace.
- Deploy the new application `frontend` as `canary` in the `production` namespace.

Configuring Spinnaker pipelines

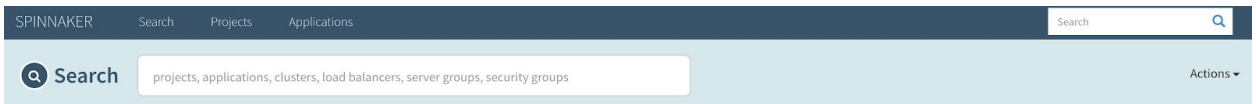
Ensure that you are port-forwarding all the required services using the `workshop_connect` command. If you are unsure, re-run the script again by running the following command.

```
workshop_connect
```

One of the services as part of Spinnaker is `Deck`. Deck is the web frontend for Spinnaker. To access the Spinnaker user interface, click **Web Preview** in Cloud Shell (top bar, right hand side) and click **Preview on port 8080**.

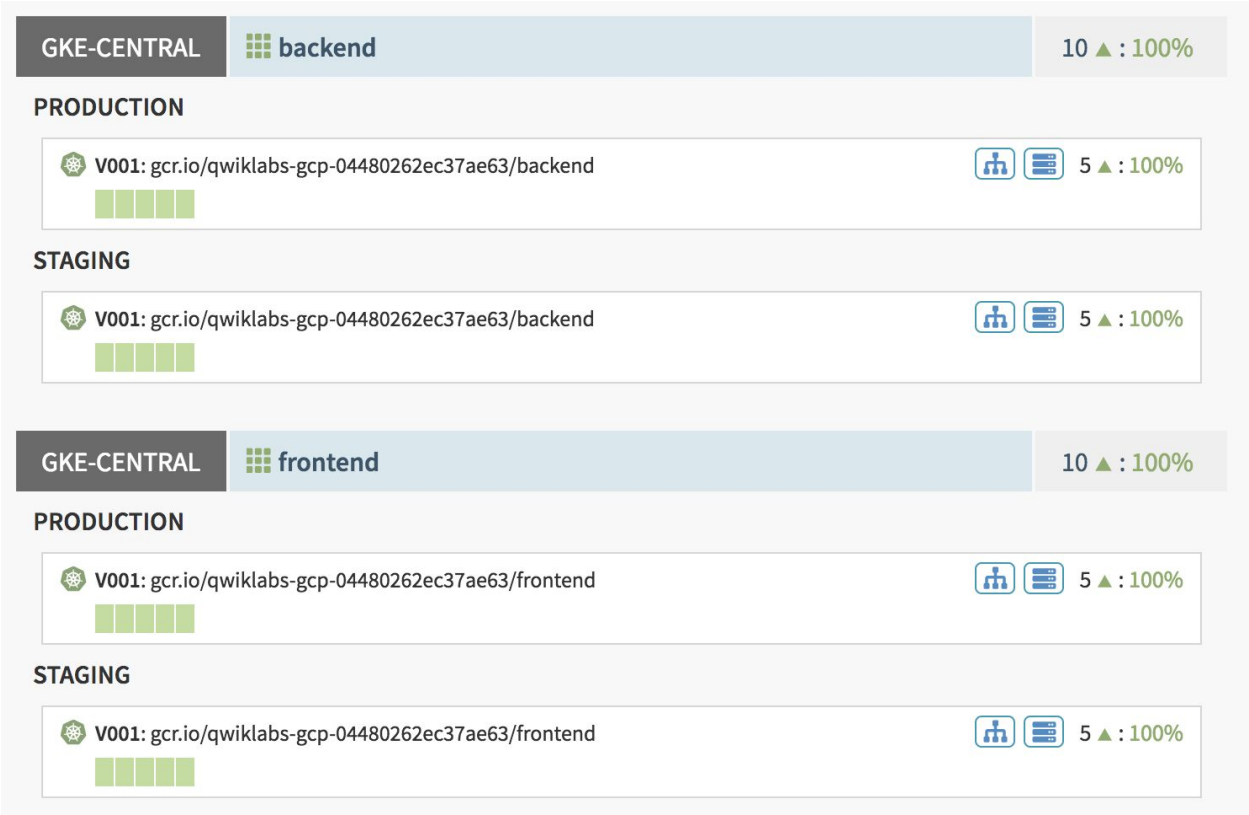


You get the Spinnaker GUI with the header as shown.



Click on **Applications** in the header and select the application labeled **myapp**. Note that **myapp** might be on page 2 of the Applications list.

You can see the two clusters `gke-central` and `gke-west`, with `frontend` and `backend` deployments (5 replicas each) in the `production` and `staging` namespaces. Each green rectangle represents a pod.



Click on **PIPELINES** in the header. You see no pipelines currently deployed.

Deploy Spinnaker pipelines by running the following commands in Cloud Shell.

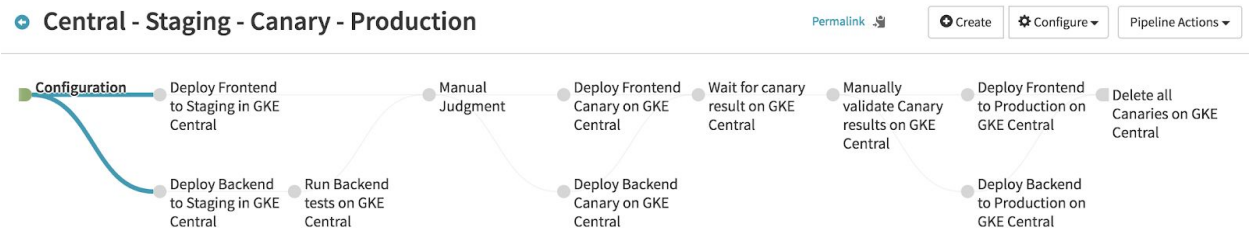
```
workshop_create-pipelines
```

Refresh the Spinnaker pipelines page using **CTRL-R** key combination. You see two pipelines.

Both pipelines are identical for the two clusters: one for `gke-central` and one for `gke-west`.

Inspect the pipeline for the `gke-central` cluster. Click on the **Configure** link for the **Central - Staging - Canary - Production** pipeline.

Inspect the pipelines by clicking on various stages.



Configuration

The initial **Configuration** stage defines the triggers and expected artifacts for the pipeline. You see two triggers of type Google PubSub: one for **GCR** and one for **GCS**. Whenever there is a change either to the GCR repositories (for `frontend` or `backend`), or to the kubernetes manifests files stored in GCS, the pipeline is triggered. The **Expected Artifacts** section describes which artifacts will be used in the current pipeline. You see the two images: for `frontend` and `backend`. You also see the two manifests files `frontend.yml` and `backend.yml` (stored in a GCS bucket).

Deploy to Staging

The next two stages update the `frontend` and `backend` deployment in the `staging` namespace. These stages use the respective manifests files from the GCS bucket. It is recommended to store the Kubernetes manifests in a storage bucket instead of hard coding them in Spinnaker so that they may be version controlled as you make changes to them. Inspect the manifests files by running the following command in Cloud Shell.

```
export PROJECT=$(gcloud info --format='value(config.project)')
gsutil cat gs://$PROJECT-spinnaker/manifests/frontend.yml
gsutil cat gs://$PROJECT-spinnaker/manifests/backend.yml
```

The output shows template stylized kubernetes manifests using Spinnaker Pipeline Expressions. Expressions allow you to use the same manifests files for various stages / phases of the application lifecycle.

Testing Backend

The following stage tests the `backend` service by performing a `curl` to the service FQDN. This stage uses a hard coded text box (instead of GCS stored manifests) for the Kubernetes job object. Unlike the Kubernetes manifests, this job object may not change which makes it ok to be hard coded as text.

In addition to the Job object, you also see a Policy object. This is a Kubernetes CRD (Custom Resource Definition) defined by Istio (discussed in more detail in the Lab 3). The current implementation of Istio injects a sidecar proxy in each pod which encrypts all traffic (service to service) using mTLS (mutual TLS). You can disable mTLS on a per-service basis using the Policy object. This allows the `staging backend` service to not use mTLS. Furthermore, the Job object has the following annotation:

```
sidecar.istio.io/inject: 'false'
```

This prohibits the Istio sidecar to be injected in this pod. Bypassing mTLS using Policy object coupled with the annotated Job object allows for the Job to do a simple `curl` to the `backend` service.

Manual Judgement

After both the `frontend` and `backend` deployments have been updated in the `staging` namespace, there is a **manual judgement** stage which waits for admin intervention to proceed to the next step.

Deploy Canary to Production

The next stage updates the `frontend` and `backend` deployment in `production` namespace as canary. Canary Deployments (or Canary Testing) is a way to test software in production by deploying the new (updated) version of a service in production environment and routing a subset of traffic to it. This stage deploys one replica of the `frontend` and `backend` each (as opposed to five for production). Using Spinnaker pipeline expressions, you can use the same manifest files as before, and change certain fields like labels and replicas to adjust for the current stage.

Manual Judgement

After the canary stage, there is yet another **manual judgement** stage prior to promoting to production.

Promote to Production

Once the canary release has been tested and verified, you promote the canary release to production, making it the new production version, using the same manifest files and taking advantage of Spinnaker pipeline expressions.

Delete all canaries

Lastly, delete all canary deployments in the `production` namespace.

Once you inspect each stage, exit out of the pipeline configuration screen by clicking the back arrow next to the pipeline name.

Building the image

Next you build the new `frontend` image (from Lab 1) using **Cloud Build** by running the following commands in Cloud Shell.

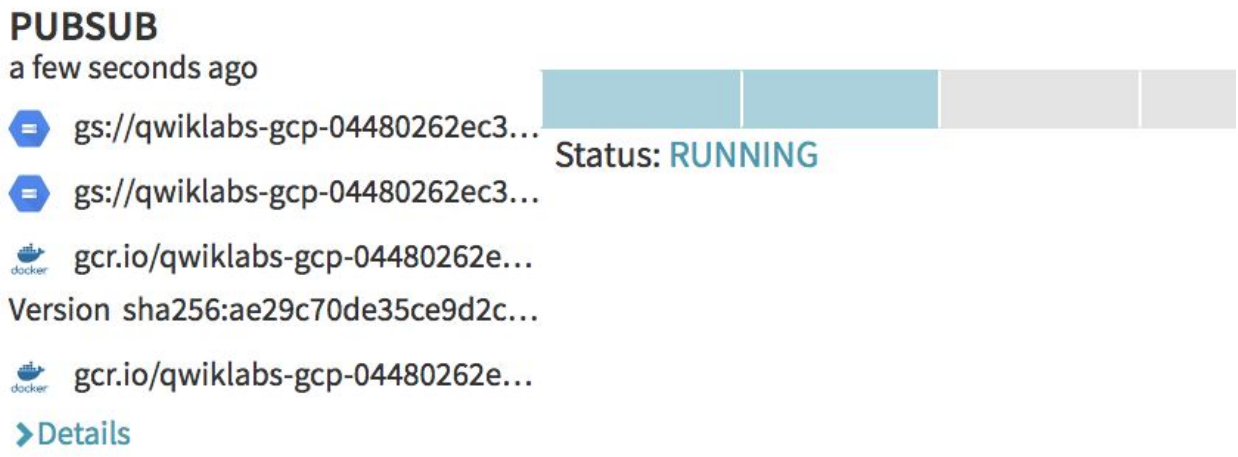
```
cd ~/advanced-kubernetes-workshop/services/frontend/
./build.sh
```

The script builds the new image and stores it in GCR.

Deploying to staging

The build from the previous step causes Spinnaker pipelines to be triggered (recall the Spinnaker pipelines are set to trigger whenever there is a new image uploaded to GCR).

Inspect the running pipelines from the Spinnaker GUI.



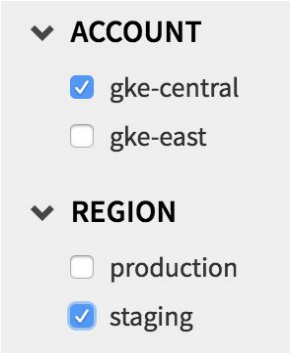
You see the cause of the trigger as **PUBSUB** and the status of each stage (represented by individual rectangles). Blue color signifies stage in progress while green means a successful completion of the stage. Hovering over the rectangle reveals the name of the stage. You can also click on the stage (rectangle) and get further details into tasks and subtasks being performed during the stage.

After a few moments, you see a person icon representing the **MANUAL JUDGEMENT** stage.

Do **NOT** click Continue yet.

Hover over the orange rectangle and you see certain instructions and an option to Continue or Stop.

Click on **INFRASTRUCTURE** in the top panel. From the left side panel, under Account, select **gke-central** and under Region, select **staging**.



You can see **V002** for both `frontend` and `backend` deployments. The **V*** tags are internal Spinnaker tags to keep track of updates. Every time the pipeline is triggered, the **V*** versions tags are incremented. You can also see the SHA signature for the new `frontend` image. `Backend` image remained the same as no changes were made to it.

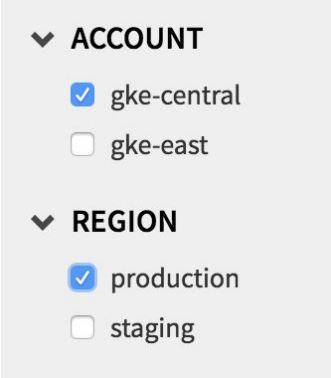
You also see **V001** deployments (previous versions of the staging deployments prior to the current execution of the pipeline). Both the `frontend` and `backend` **V001** deployments have **0** replicas however, they are still present in case you want to roll back.

Deploying canaries to production

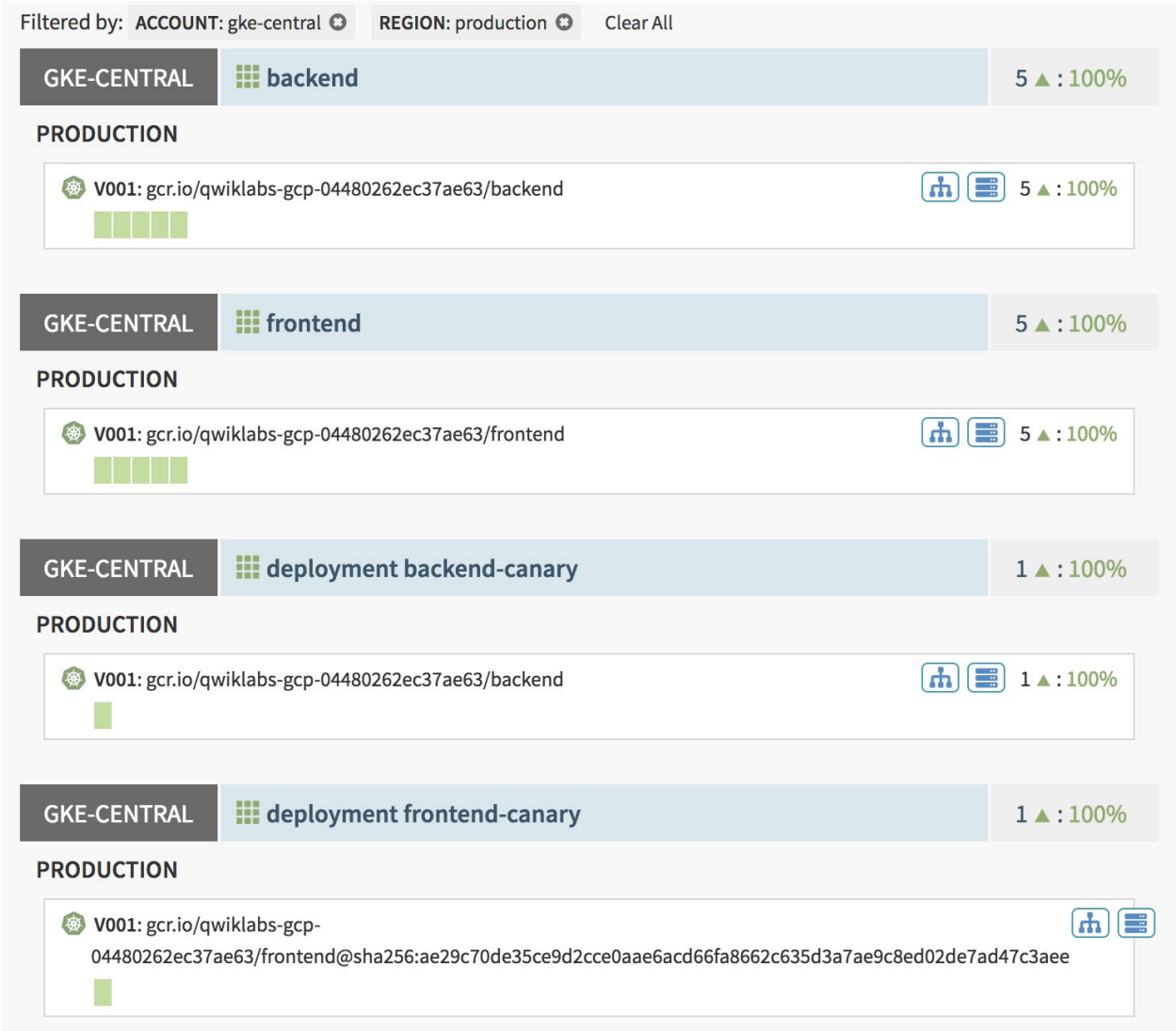
Click on the **PIPELINES** link from the top bar. Hover over the orange rectangles and click **Continue**. This causes the pipelines to continue to the next stage. The next stage deploys the `frontend` and `backend` in the `production` namespace as canaries. The successful completion of the following few stages will culminate with yet another **Manual Judgement** stage.

Do **NOT** click Continue yet!

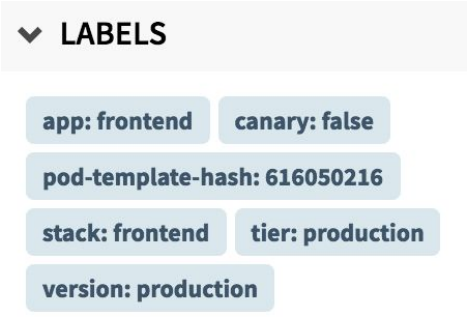
Inspect the deployment by clicking on **INFRASTRUCTURE** link in the top bar. From the left hand side bar, under Account, select **gke-central** and under Region, select **production**.



You now see that in addition to the `primary production` deployment, there are two new canary deployments each containing a single replica. The **V001** (for the two canary deployments) indicate this is the first iteration of the canary releases for this pipeline.



Click on any of the green rectangles of the **frontend** deployment and inspect the labels on the right hand side bar.



The two labels to take note of are **app: frontend** and **version: production**. Likewise, click on the single green rectangle of the **deployment frontend-canary** deployment. Inspect the labels from the right hand side bar.

The two labels to take note of are **app: frontend** and **version: canary**. You use these labels and Istio in the next Lab for controlling traffic to the two versions.

Congratulations! You have successfully used Cloud Build to build the new image, deploy a Spinnaker pipeline, trigger the pipeline using PubSub and deploy the new image as canary.

Troubleshooting Lab 2

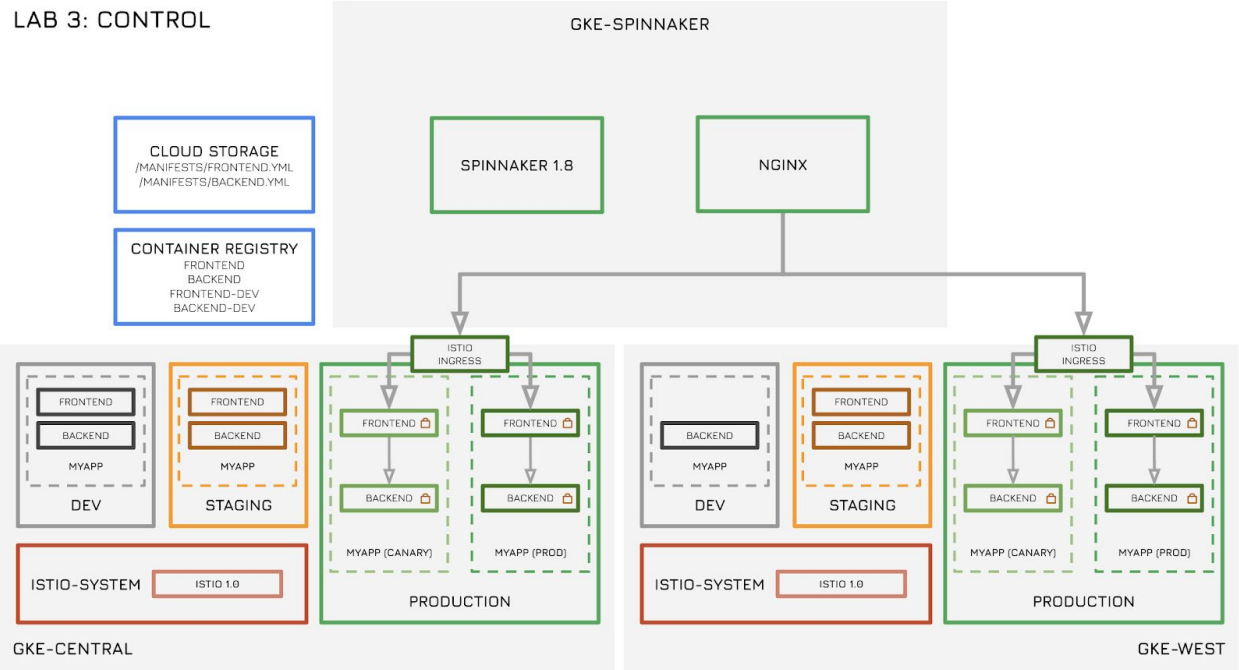
If you run into problems with spinnaker such as missing pipelines, pubsub queues, or projects, don't forget that you can always run:

```
workshop_hal-config
```

This will re-run the hal config scripts that setup spinnaker.

END OF LAB 2 - Please STOP and wait for instructions before proceeding!

Lab 3: Control (60 mins)



In this lab, you will:

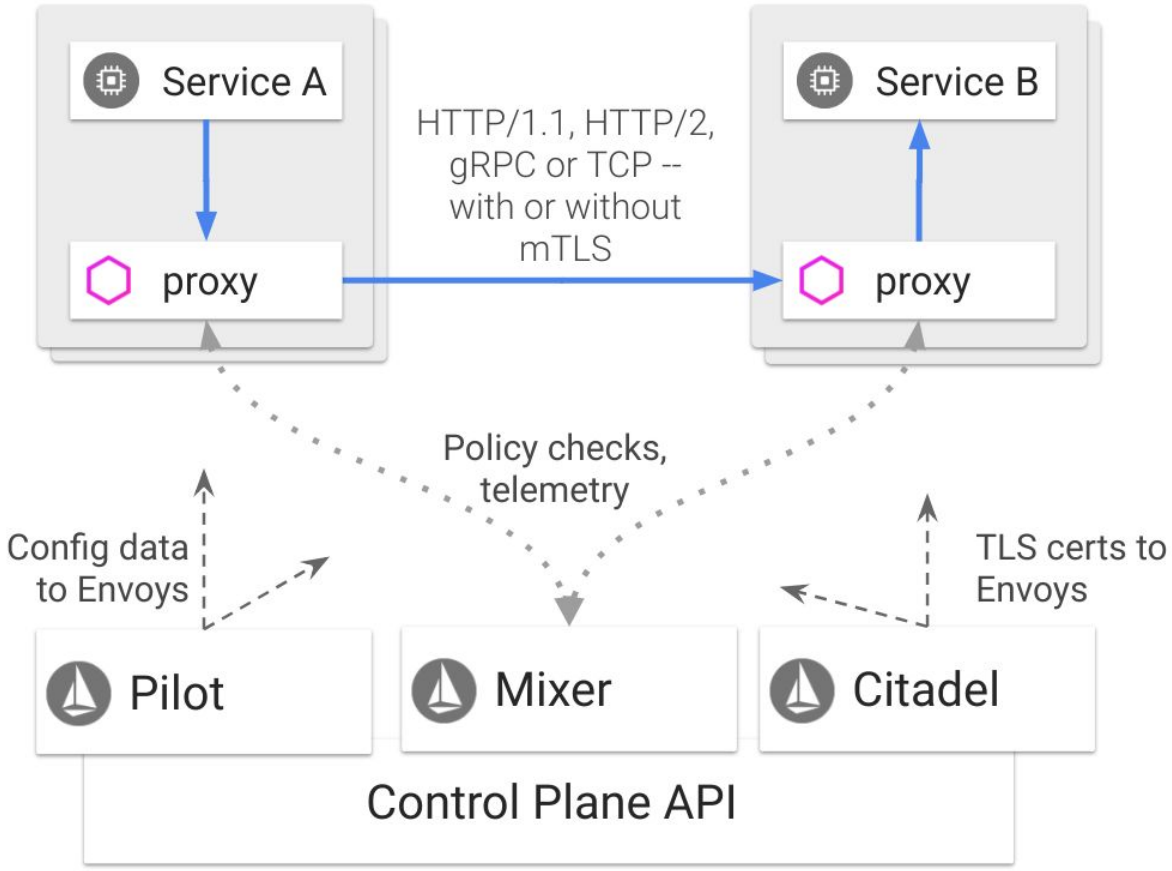
- Use Istio Service Mesh for advanced traffic routing.
- Use Istio for policy and quota management.
- Use Istio for securing pod-to-pod communication using mTLS.
- Use Prometheus and Grafana for monitoring traffic and policy changes.
- Use NGINX for global load balancing to multiple kubernetes clusters.

Istio Service Mesh

Istio is an open source framework for connecting, securing, and managing microservices, including services running on Kubernetes Engine. It lets you create a network of deployed services with load balancing, service-to-service authentication, monitoring, and more, without requiring any changes in service code.

In order to create the service mesh, Istio deploys a special sidecar proxy to each of your application's pods. This proxy intercepts all network communication between microservices and is configured and managed using Istio's control plane.

Istio's architecture looks as follows:



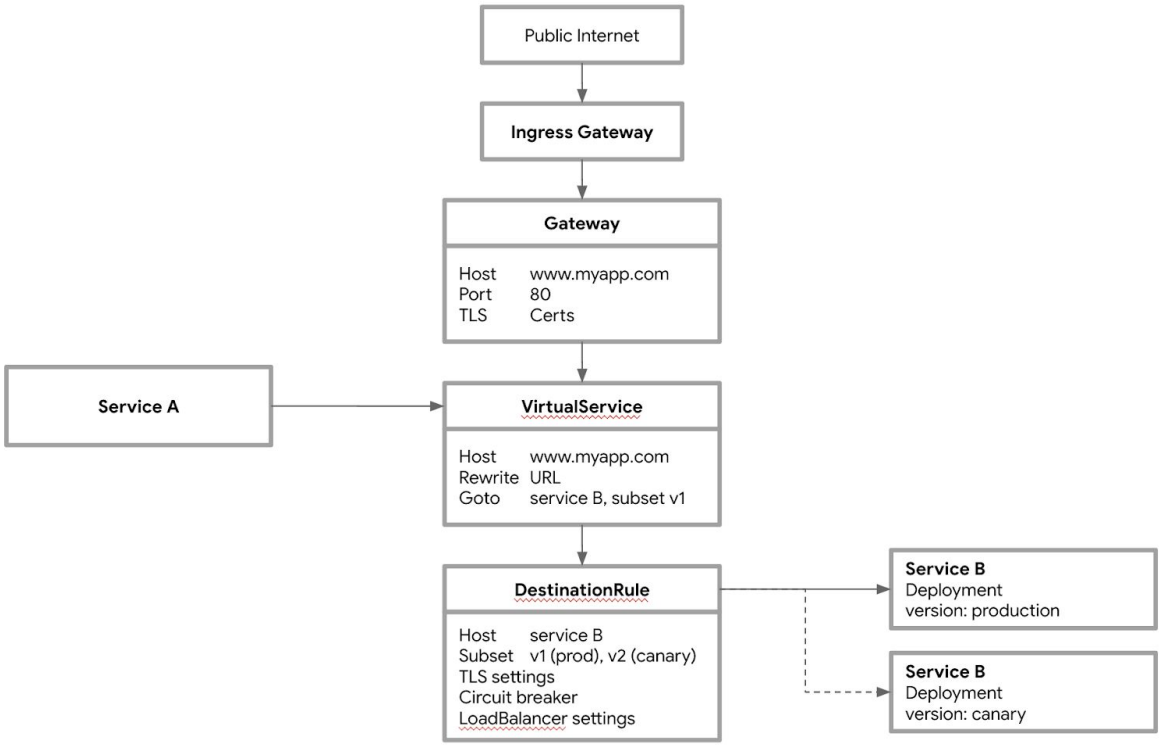
The Data plane consists of a mesh of sidecar proxies (powered by the Envoy proxy) deployed within each pod next to the service container. The sidecar proxy abstracts all of the application networking functionality (routing, security, policy and monitoring) from the service. The service has no knowledge of its underlying networking topology. It simply receives and sends traffic via its sidecar proxy. All proxies are interconnected in a full-mesh fashion thus making a service mesh. Proxies send and receive traffic from their sidecar service (also known as downstream service) to other proxies connected to other services (also known as upstream services) based on a set of configured rules. These rules are configured by the Istio Control plane.

The Istio Control plane consists of three components:

- **Pilot** is responsible for configuring the rules on the sidecar proxies within the mesh. For Kubernetes, these rules are typically configured using Istio CRDs which are in turn sent to each sidecar proxy in the mesh.
- **Mixer** is responsible for policy/quota management and enforcement as well as telemetry. All proxies synchronously check with the Mixer for allowances/policies as they communicate with other services. In Kubernetes, Mixer policies are typically configured via Istio CRDs. The proxies also asynchronously send telemetry information (metrics, traces etc) to mixer which can then be sent via plugins to another service. For example, one of the Istio add-ons is Prometheus which is a popular open source metrics collector. More on Prometheus in the next lab.
- **Citadel** is responsible for securing traffic within the service mesh. It provides a verifiable identity as well as encryption for service to service communication. It is responsible for managing, rotating and revoking certificates to each proxy within the mesh.

Ingress Gateway, VirtualServices, and DestinationRules

In Kubernetes, exposing services externally is typically done using the Kubernetes Ingress resource. Istio provides a more robust mechanism using Istio Gateway. An Istio Gateway operates at the edge of the service mesh and controls traffic into and out of the mesh. In GCP, it sets up a Google Cloud Load Balancer for ingress/egress traffic. A Gateway custom resource is used for configuration which can then be applied to a kubernetes service.



An Istio gateway is already created as part of the initial setup. Inspect the Gateway resource by running the following commands.

```
kubectl get gateway -n production --context gke-central -o yaml
```

Output (Do not copy)

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
…
  name: frontend-gateway
  namespace: production
…
spec:
  selector:
    istio: ingressgateway
  servers:
  - hosts:
    - '*'
    port:
      name: http
      number: 80
      protocol: HTTP
```

The name of the gateway is `frontend-gateway`. The `selector` spec selects which proxy the rule applies to. In this case, it applies to the istio `ingressgateway` which is deployed in the `istio-system` namespace as part of the istio installation.

The `hosts` field is set to wildcard '*' which means this applies to all hostnames. This is done to make it easier for the workshop. In production, you would have actual hostnames defined in each gateway for example, **www.company.com** or **www.company.com/svcA** etc. In this case, the gateway will only be `applied for the defined hosts.

The gateway can then be applied to a service. In Istio, you use the **VirtualService** concept, which corresponds to a Kubernetes service.

Inspect the `frontend` VirtualService by running the following command.

```
kubectl get virtualservice frontend -n production --context gke-central -o yaml
```

Output (Do not copy)

```
...
spec:
  gateways:
  - frontend-gateway
  hosts:
  - '*'
  http:
  - route:
    - destination:
        host: frontend
        port:
          number: 80
```

The VirtualService has a single rule which send all traffic destined for HTTP port 80 to kubernetes service `frontend`. The `gateways` field is used to select the gateway defined in the previous step - `frontend-gateway`.

The last piece of the puzzle is the **DestinationRule**. A DestinationRule configures the set of policies to be applied to a request after VirtualService routing has occurred. They are intended to be authored by service owners, describing the circuit breakers, load balancer settings, TLS settings, and other settings.

A DestinationRule also defines addressable **subsets**, meaning named versions, of the corresponding destination host. These subsets are used in VirtualService route specifications when sending traffic to specific versions of the service. More on this in later sections.

Inspect the DestinationRule by running the following command.

```
kubectl get destinationrule frontend -n production --context gke-central -o yaml
```

Output (Do not copy)

```
...
spec:
  host: frontend
  subsets:
  - labels:
      version: canary
    name: canary
  - labels:
      version: production
    name: production
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
    tls:
      mode: ISTIO_MUTUAL
```

The DestinationRule is defining two components for the `frontend` VirtualService. First is **subsets**, which define versions within a service. In Kubernetes, this can be thought of as multiple deployments running under a single service with different labels. For example, in this case, you have two deployments for `frontend` in the `production` namespace, one for primary (or production) traffic and one deployed as canary. These two deployments are distinguished by labels. Subset names `canary` uses the label selector `"version": "canary"` and likewise subset named `production` uses the label selector `"version": "production"`. The DestinationRule also defines a **trafficPolicy** which sets the load balancing policy to *ROUND-ROBIN* and TLS to use *mTLS*.

In the next section, you use VirtualServices to control traffic routing to specific versions of `frontend` and `backend` service.

Request Routing

In a typical multi-tier microservices architecture, you may have numerous services each running multiple versions. In this workshop, you have a two tier application and each tier is currently running two versions. You have two versions of `frontend` service - `primary` and `canary` (running as two deployments) and two versions of `backend` service - `primary` and `canary` (running as two deployments).

Inspect the deployments in the `gke-central` cluster by running the following command.

```
kubectl get deploy -n production --context gke-central
```

Output (Do not copy)

```
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
backend-canary      1         1         1            1           16h
backend-primary     5         5         5            5           1d
frontend-canary     1         1         1            1           16h
frontend-primary    5         5         5            5           1d
```

There are five pods for the `primary` (or production) deployments versus one pod for the `canary` deployments. Kubernetes uses basic round robin logic for load balancing traffic to different pods within a service. Since both deployments are under a single service, the two services send traffic to all 6 pods in a round robin fashion.

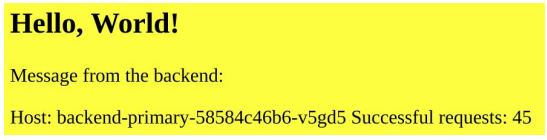
Get the Ingress gateway external IP address for the cluster by running the following script.

```
workshop_get-ingress
```

Output (Do not copy)

```
gke-central ingress gateway:
35.202.138.224

gke-west ingress gateway:
35.236.250.232
```

Open a new Chrome tab and copy/paste the `gke-central` ingress gateway IP address. Continually refresh the page using **CTRL-R** and you can see both the `primary` (yellow background) and the `canary` deployments (green background).



Create some load on the frontend using the **hey** utility.

For this task, open two Cloud Shell windows. In Cloud Shell window #1, start the **hey** traffic generator by running the following command.

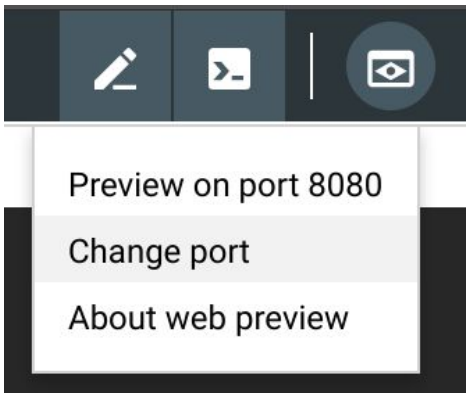
```
export GKE_ONE_ISTIO_GATEWAY=$(kubectl get svc istio-ingressgateway -n istio-system
-o jsonpath="{.status.loadBalancer.ingress[0].ip}" --context gke-central)

hey -z 30m http://$GKE_ONE_ISTIO_GATEWAY
```

This command starts the hey test with default settings for 10 minutes.

Note: The **hey** traffic generator runs for 30 minutes. If you are still in this section, re-run the hey command to keep generating more traffic.

In order to view the service metrics, you can use **Grafana**. All metrics from Istio Mixer are fed into **Prometheus** (a metrics collector) and Grafana is a visualization tool used for Prometheus. More on this in the next lab.

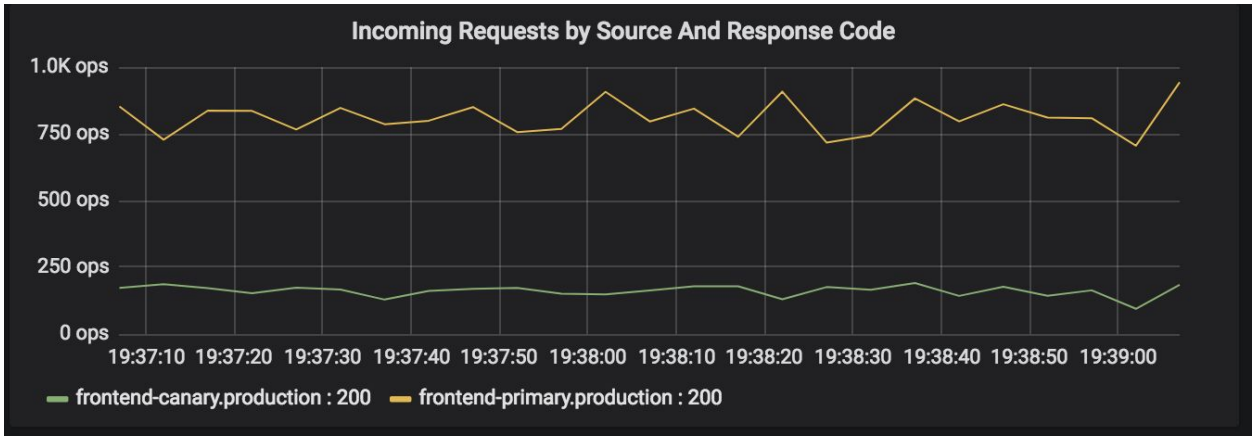
Click **Web Preview** in Cloud Shell (top bar, right hand side) and click **Change port**.



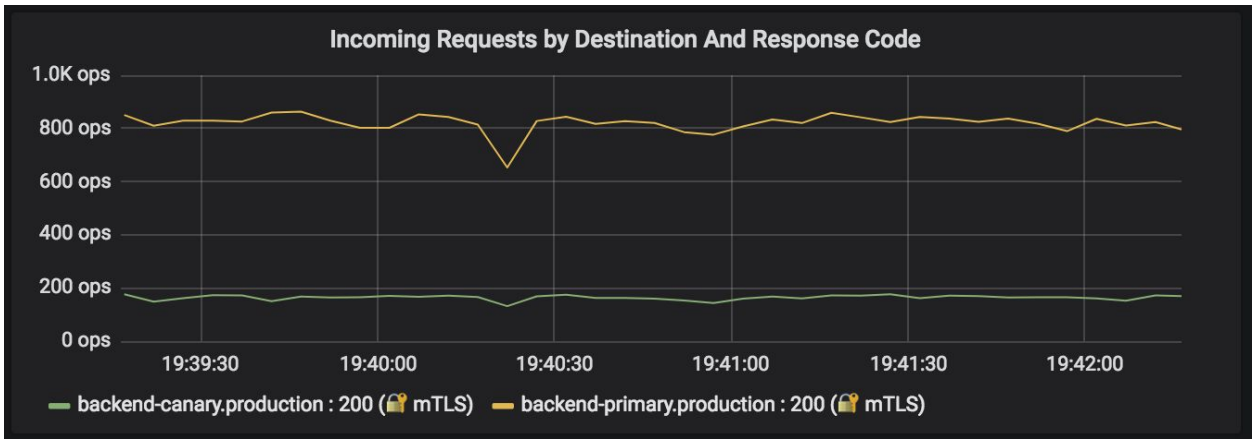
Change the port to **3000** and click **Change and Preview**. This opens a new tab in Chrome for Grafana.

Click on the **Home** button in the top left corner and then the **Istio Service Dashboard** link from the left hand side.

In the **Service** dropdown menu (top left corner), ensure that **backend.production.svc.cluster.local** is selected. Leave everything else as default. Inspect the **Incoming Requests by Source and Response Code** graph.



This shows traffic by `frontend` deployments. By default, traffic is being sent to both `primary` and `canary` versions of the `frontend` service. There are five primary pods and one canary pod hence more traffic (about five times more) is being sent to the primary pods vs canary. Scroll down and look at the **Incoming Requests by Destination and Response Code** graph. This shows the `backend` service and you can see a similar pattern to the `frontend`.



Currently, all `frontend` pods (primary and canary) can talk to all `backend` pods (primary and canary). There is no way of knowing which `frontend` will serve the client and also which `backend` will be served for every request.
Apply a rule to send 50 percent of the client traffic to the `primary frontend`, and 50 percent of the client traffic to the `canary frontend`. Run the following command from the second Cloud Shell window.

```
kubectl apply -f
~/advanced-kubernetes-workshop/services/manifests/frontend-vs-50prod-50canary.yml
--context gke-central
```

Output (Do not copy)

```
Updated config virtual-service/production/frontend to revision 276546
```

Inspect the new `frontend` VirtualService.

```
kubectl get virtualservice frontend -n production --context gke-central -o yaml
```

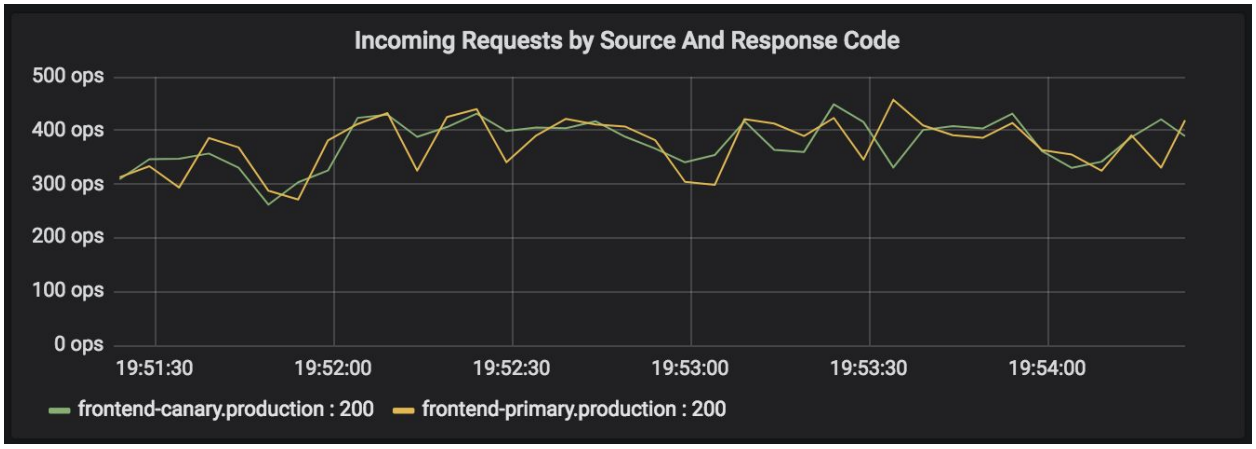
Output excerpt (Do not copy)

```
...
spec:
```

```
gateways:
- frontend-gateway
hosts:
- '*'
http:
- route:
  - destination:
      host: frontend
      port:
        number: 80
      subset: canary
    weight: 50
  - destination:
      host: frontend
      port:
        number: 80
      subset: production
    weight: 50
```

Notice the subsets (defined in the DestinationRule) "`canary`" and "`production`" are getting 50% of the traffic each, defined by the "`weight`" field.

Confirm the rule is in effect by inspecting the **Incoming Requests by Source and Response Code** graph on Grafana.



Notice that the **Incoming Requests by Destination and Response Code** graph remains the same. Can you guess why?

Regardless of which `frontend` version serves the client, the `backend` service is being load balanced in the default round robin manner. Apply a VirtualService rule for the backend service to only send `primary` (or production) `frontend` traffic to `primary` (or production) `backend` and `canary frontend` traffic to `canary backend`. Run the following command in the second Cloud Shell.

```
kubectl apply -f
~/advanced-kubernetes-workshop/services/manifests/backend-vs-can-to-can-prod-to-prod
.yml --context gke-central
```

Output (Do not copy)

```
Updated config virtual-service/production/backend to revision 277689
```

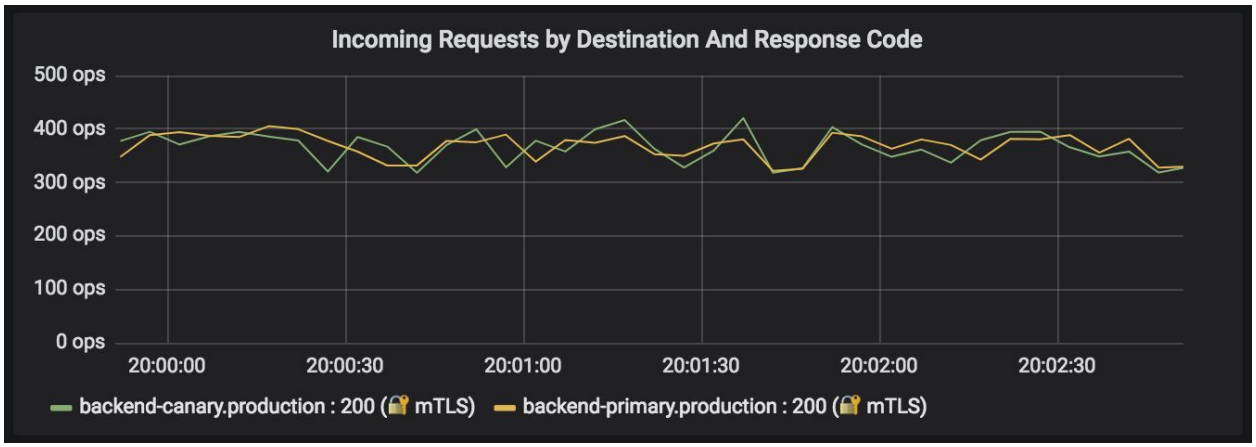
Inspect the new backend VirtualService by running the following command.

```
kubectl get virtualservice backend -n production --context gke-central -o yaml
```

Output excerpt (Do not copy)

```
...
spec:
  hosts:
  - backend
  http:
  - match:
    - sourceLabels:
        app: frontend
        version: canary
    route:
    - destination:
        host: backend
        port:
          number: 80
        subset: canary
  - match:
    - sourceLabels:
        app: frontend
        version: production
    route:
    - destination:
        host: backend
        port:
          number: 80
        subset: production
```

The rule matches the `sourceLabel` of `app: frontend` and `version: canary` and routes it to the subset `canary` of the `backend` service. Likewise, `sourceLabel` of `app: frontend` and `version: production` is routed of subset `production` of the `backend` service. Inspect the **Incoming Requests by Destination and Response Code** graph.



The incoming graph now looks identical to the **Incoming Requests by Source** graph.

There is still no way to be sure that the `frontend` production is going to `backend` production and vice versa.

Apply a `frontend` VirtualService rule to send 100% of the traffic to production frontend. Run the following command.

```
kubectl apply -f
~/advanced-kubernetes-workshop/services/manifests/frontend-vs-all-to-prod.yml
--context gke-central
```

Output (Do not copy)

```
Updated config virtual-service/production/frontend to revision 278549
```

Inspect the new `frontend` VirtualService and confirm 100% of the traffic is now being sent to the production frontend..

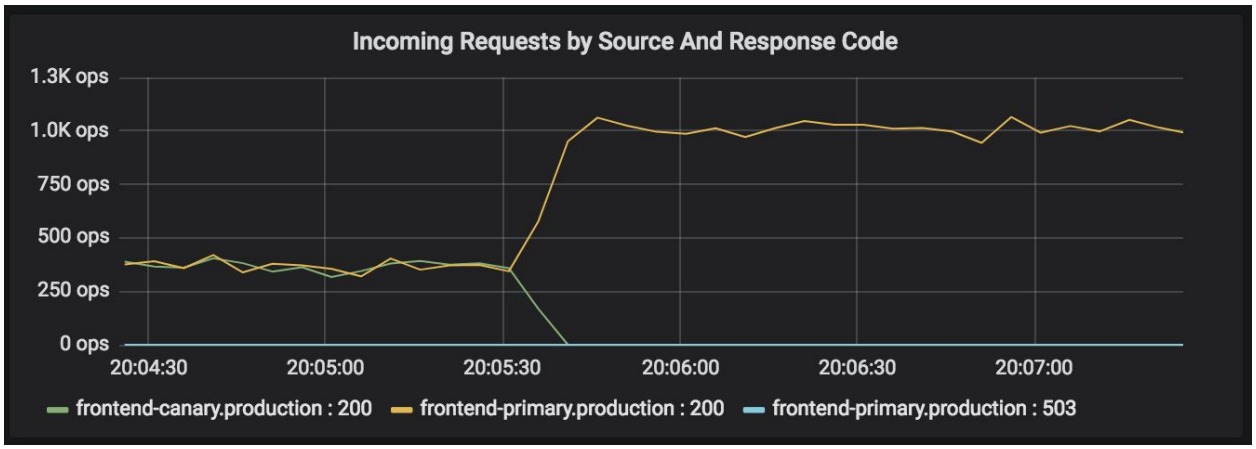
```
kubectl get virtualservice frontend -n production --context gke-central -o yaml
```

Output excerpt (Do not copy)

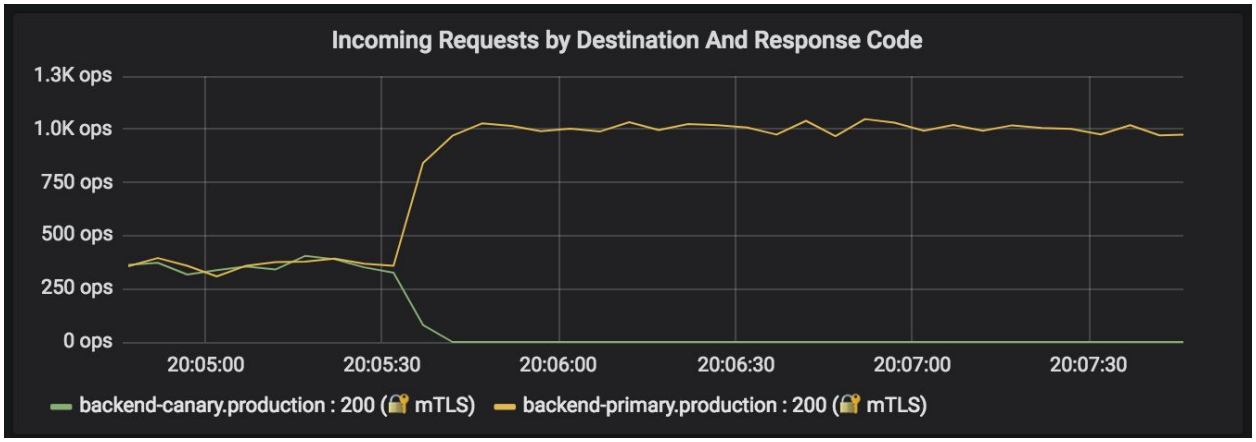
```
...
spec:
...
  http:
  - route:
    - destination:
        host: frontend
        port:
          number: 80
        subset: canary
    - destination:
```

```
      host: frontend
      port:
        number: 80
      subset: production
    weight: 100
```

Inspect both the Grafana charts to confirm that all traffic is going to the production frontend.



Since the `backend` VirtualService rule sends all `production frontend` traffic to `production backend` deployment, the **Incoming Requests by Destination** shows the same behavior.



Finally, apply a `frontend` VirtualService rule to send 100% of the traffic to the canary frontend. Run the following command.

```
kubectl apply -f
~/advanced-kubernetes-workshop/services/manifests/frontend-vs-all-to-canary.yml
--context gke-central
```

Output (Do not copy)

```
Updated config virtual-service/production/frontend to revision 279182
```

Inspect the new `frontend` VirtualService.

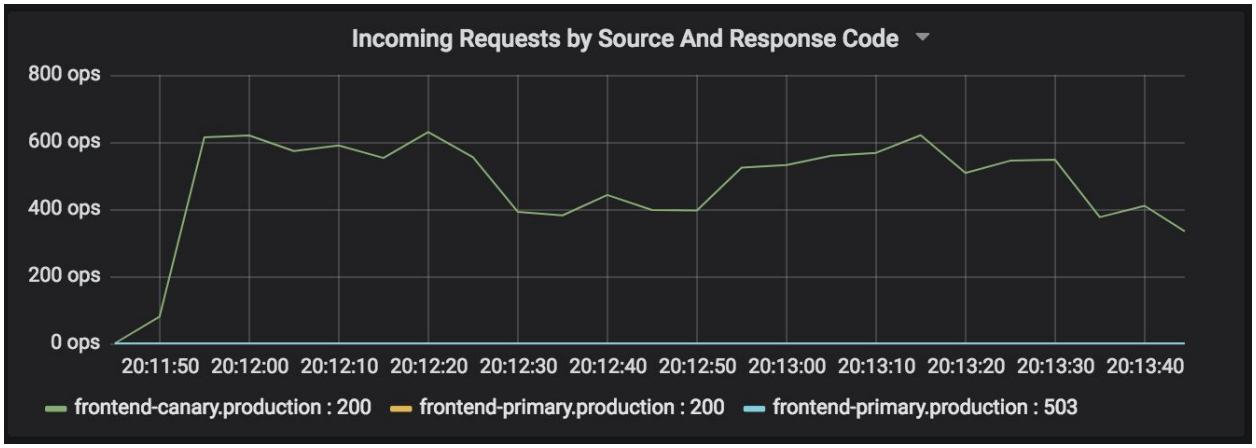
```
kubectl get virtualservice frontend -n production --context gke-central -o yaml
```

Output excerpt (Do not copy)

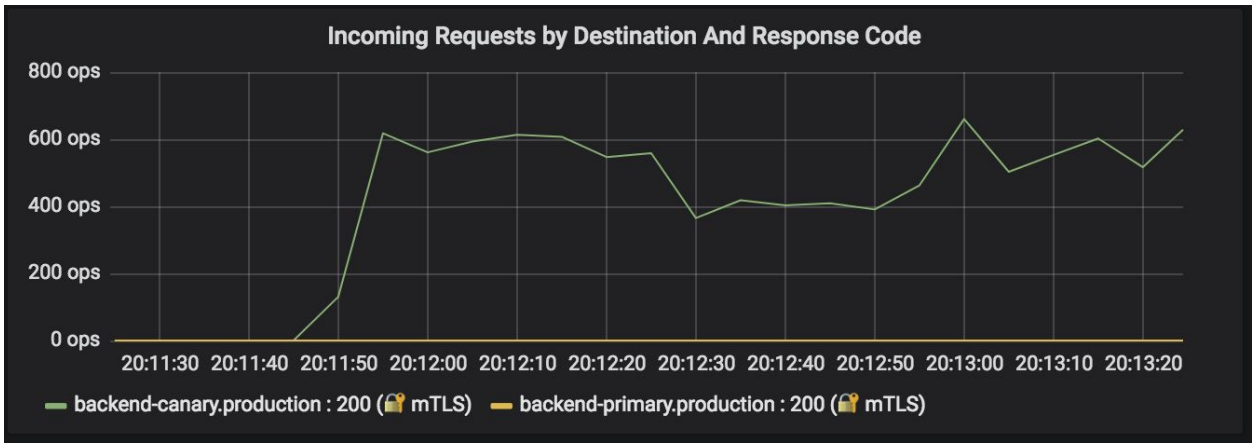
```
...
spec:
...
  http:
  - route:
    - destination:
        host: frontend
        port:
          number: 80
        subset: canary
      weight: 100
    - destination:
        host: frontend
        port:
          number: 80
        subset: production
```

Inspect the two Grafana graphs and confirm the behavior as expected.

100% of the `frontend` traffic is now going to the `canary frontend`.



As expected, 100% of the backend traffic to going to the `canary backend`.



Using match filters and subset weights, you have full control over service to service traffic routing. Learn more about advanced Request Routing use cases on the Istio site.

Rate Limiting

Using mixer policy and quota management, you can rate limit traffic to various services and subsets of subsets. Prior to applying rate limiting policies, reset the `frontend` VirtualService to its default state. Run the following command.

```
kubectl apply -f ~/advanced-kubernetes-workshop/services/manifests/myapp-vs-base.yml
--context gke-central
```

Output (Do not copy)

```
virtualservice.networking.istio.io/frontend
configuredvirtualservice.networking.istio.io/backend configured
```

Ensure that you are generating traffic using **hey**.

```
export GKE_ONE_ISTIO_GATEWAY=$(kubectl get svc istio-ingressgateway -n istio-system
-o jsonpath="{.status.loadBalancer.ingress[0].ip}" --context gke-central)

hey -z 30m http://$GKE_ONE_ISTIO_GATEWAY
```

Inspect the Grafana service graphs.

Apply a rate limit rule for the production and canary version of the `frontend` service. Run the following command.

```
kubectl apply -f ~/advanced-kubernetes-workshop/lb/rate-limit-frontend.yaml
--context gke-central
```

Output (Do not copy)

```
memquota.config.istio.io/handler created
quota.config.istio.io/requestcount created
rule.config.istio.io/quota created
quotaspec.config.istio.io/request-count
createdquotaspecbinding.config.istio.io/request-count created
```

The rule applies five resources. Inspect the `memquota` resource by running the following command.

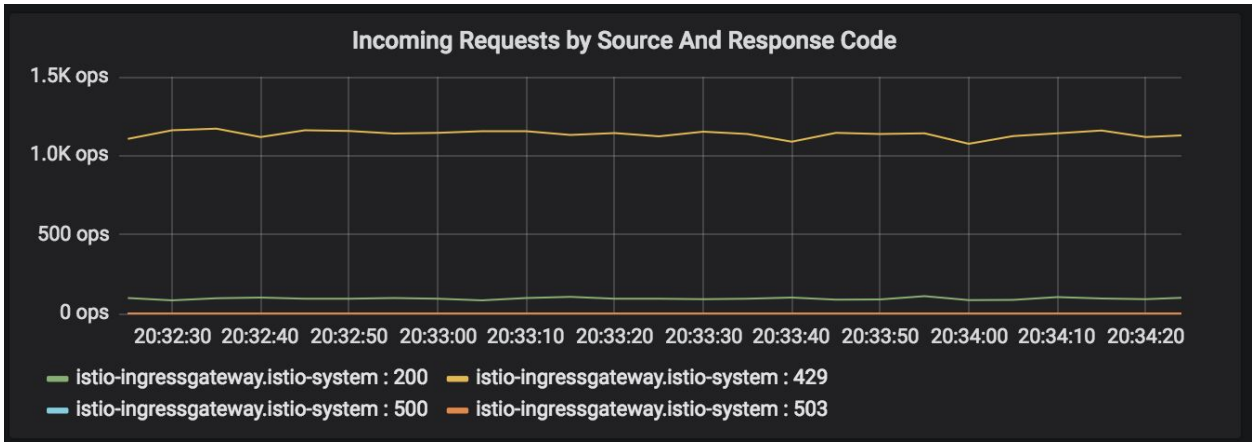
```
kubectl get memquota -n istio-system --context gke-central -o yaml
```

Output excerpt (Do not copy)

```
...
  spec:
    quotas:
    - maxAmount: 900
      name: requestcount.quota.istio-system
      overrides:
      - dimensions:
          destination: frontend
          destinationVersion: canary
        maxAmount: 20
        validDuration: 5s
      - dimensions:
          destination: frontend
          destinationVersion: production
        maxAmount: 75
        validDuration: 5s
      validDuration: 1s
```

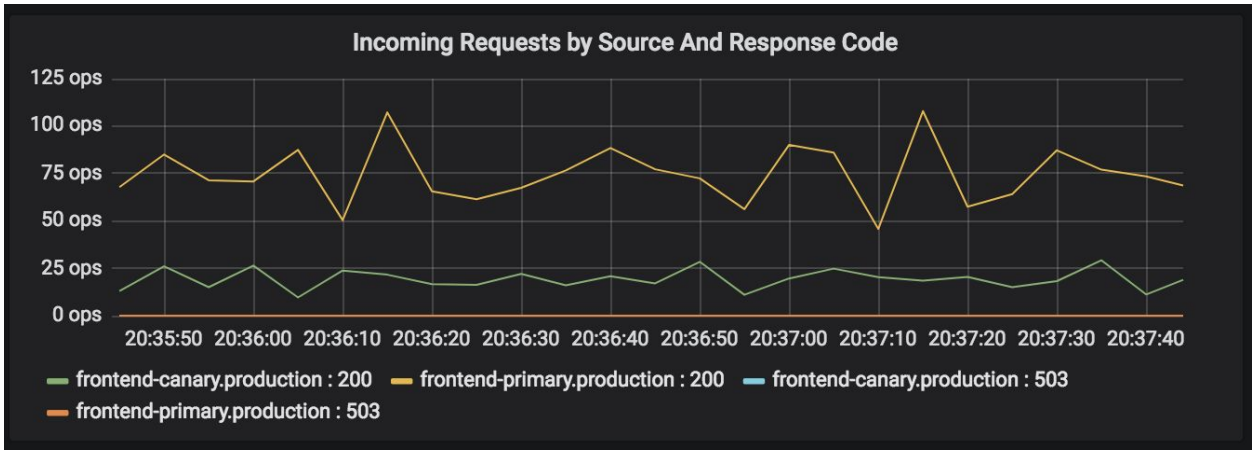
The `canary frontend` is being rate limited to **20** requests for every **5** seconds. The `production frontend` is being rate limited to **75** requests for every **5** seconds.

Inspect the Grafana service graph for the `frontend` service but this time switch the **Service** dropdown (top left corner) to the **frontend.production.svc.cluster.local**.



You can see large number of **429** (**Too Many Requests**) messages.

Switch the **Service** dropdown back to the **backend.production.svc.cluster.local** service and inspect the service graph.



While the load is being generated, switch to the Chrome tab with the `gke-central` ingress gateway IP address and refresh the page a few times using the **CTRL-R** key combination. You often see this message.

```
RESOURCE_EXHAUSTED:Quota is exhausted for: RequestCount
```

Learn more about Rate Limiting on the Istio website.

Before proceeding to the next step, delete the rate limit rule by running the following command.

```
kubectl delete -f ~/advanced-kubernetes-workshop/lb/rate-limit-frontend.yaml
--context gke-central
```

Output (Do not copy)

```
Deleted config: quota-spec/istio-system/request-count
Deleted config: quota-spec-binding/istio-system/request-count
Deleted config: memquota/istio-system/handler
Deleted config: quota/istio-system/requestcount
Deleted config: rule/istio-system/quota
```

Stop the **hey** traffic generator prior to next section. In the **hey** Cloud Shell window, use the **CTRL-Z** key combination if the **hey** traffic generator is still running.

Circuit Breaking

Circuit breaking is an important pattern for creating resilient microservice applications. Circuit breaking allows you to write applications that limit the impact of failures, latency spikes, and other undesirable effects of network peculiarities.

In this task, you configure circuit breaking rules for the backend service and then test the configuration by intentionally "tripping" the circuit breaker.

Create a destination rule to apply circuit breaking settings when calling the `backend` service. Run the following command.

```
kubectl apply -f ~/advanced-kubernetes-workshop/lb/circuit-breaker-backend.yml
--context gke-central
```

Output (Do not copy)

```
Updated config destination-rule/production/backend to revision 286108
```

Inspect the `trafficPolicy` section of the new `backend` DestinationRule. Run the following command.

```
kubectl get destinationrule backend -n production --context gke-central -o yaml
```

Output excerpt (Do not copy)

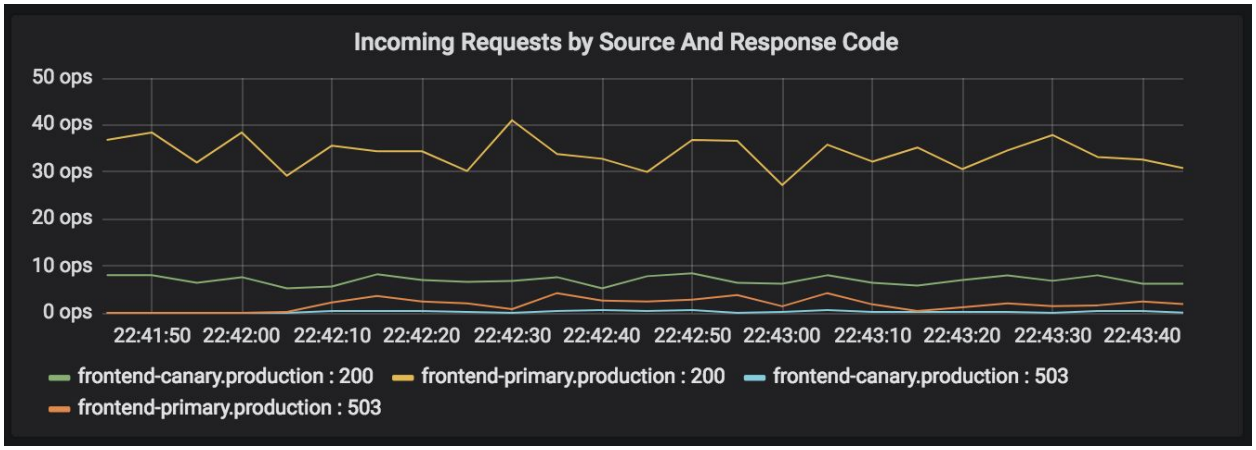
```
...
spec
...
  trafficPolicy:
    connectionPool:
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
      tcp:
        maxConnections: 1
    loadBalancer:
      simple: ROUND_ROBIN
    outlierDetection:
      baseEjectionTime: 180.000s
      consecutiveErrors: 1
      interval: 1.000s
      maxEjegraphctionPercent: 100
    tls:
      mode: ISTIO_MUTUAL
```

In the DestinationRule settings, you specified **maxConnections: 1** and **http1MaxPendingRequests: 1**. These rules indicate that if you exceed more than one connection and request concurrently, you should see some failures when the istio-proxy opens the circuit for further requests and connections.

Run the **hey** test with **2** concurrent connections for **5** minutes. Run the following command.

```
export GKE_ONE_ISTIO_GATEWAY=$(kubectl get svc istio-ingressgateway -n istio-system
-o jsonpath="{.status.loadBalancer.ingress[0].ip}" --context gke-central)

hey -c 5 -z 5m http://$GKE_ONE_ISTIO_GATEWAY
```

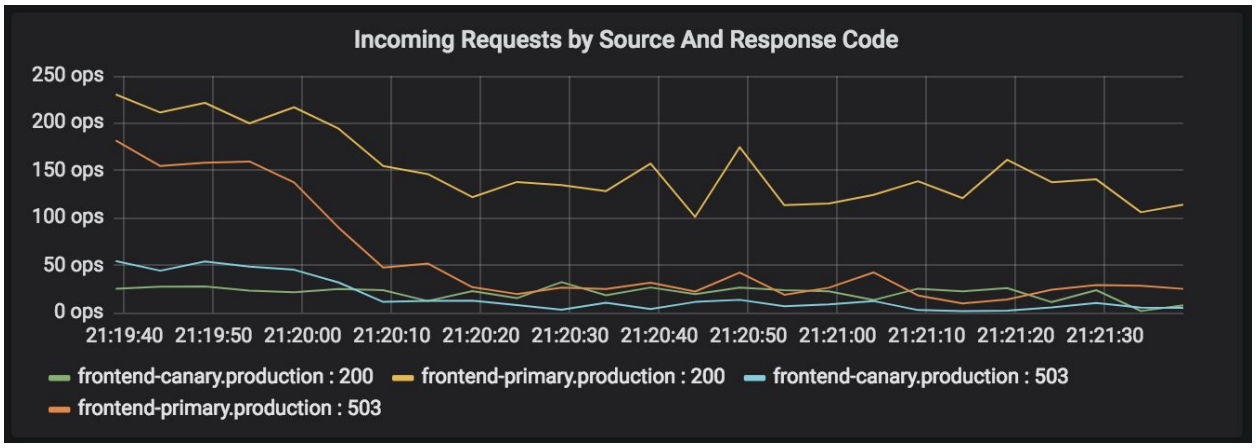
After 5 mins, inspect the Grafana service for the `frontend.production` service.. You can see that almost all requests make it through as Istio allows for some leeway.



Let the **hey** traffic generator complete. Re-run the test with **3** concurrent connections. Run the following command.

```
hey -c 3 -z 5m http://$GKE_ONE_ISTIO_GATEWAY
```

After 5 mins, inspect the Grafana service graph for the `frontend.production` service..



You now see a lot more **503** (**Service Unavailable**) messages.

Query the `istio-proxy` stats from one of the frontend pods. You can log in to the `istio-proxy` of one of the frontend pods to see more details. Run the following command.

```
export FRONTEND_POD=$(kubectl get pods -l app=frontend -n production --context
gke-central | awk 'NR == 3 {print $1}')
kubectl exec -it $FRONTEND_POD -n production --context gke-central -c istio-proxy
-- sh -c 'curl localhost:15000/stats' | grep  "||backend.production" | grep pending
```

Output (Do not copy)

```
cluster.outbound|80||backend.production.svc.cluster.local.upstream_rq_pending_activ
e: 0
cluster.outbound|80||backend.production.svc.cluster.local.upstream_rq_pending_failu
re_eject: 0
cluster.outbound|80||backend.production.svc.cluster.local.upstream_rq_pending_overf
low: 6674
cluster.outbound|80||backend.production.svc.cluster.local.upstream_rq_pending_total
: 14979
```

You see **6674** for the `upstream_rq_pending_overflow` value which means 6674 calls so far have been flagged for circuit breaking.

Remove the circuit breaker prior to the next section. Run the following commands to delete the circuit breaker destination rule and reconfigure the original destinationrule for the backend service.

```
kubectl delete -f ~/advanced-kubernetes-workshop/lb/circuit-breaker-backend.yml
--context gke-central
kubectl create -f ~/advanced-kubernetes-workshop/lb/backend-destination-rule.yml
--context gke-central
```

Output (Do not copy)

```
Deleted config: destination-rule/production/backend
Created config destination-rule/production/backend at revision 122084
```

Securing the mesh

In addition to advanced routing and policy and quota management, another benefit Istio provides is end to end security between services running inside the service mesh. In this workshop, there are `frontend` and `backend` deployments talking to each other. In real world scenarios, there can be hundreds of services (with different versions) talking to each other. Security is a top priority in this situation.

You can install Istio with or without mutual TLS (mTLS) turned on mesh wide. For this workshop, the Istio implementations have mTLS turned on and thus by default all pod-to-pod communication is encrypted using certificates provided and managed by Citadel.

This workshop covers Transport authentication, also known as service-to-service authentication (for which Istio uses mTLS for service to service communication).

Inspect the default `MeshPolicy` which is a policy that configures mTLS. Run the following command.

```
kubectl get meshpolicy -n production --context gke-central -o yaml
```

Output excerpt (Do not copy)

```
...
spec:
  peers:
  - mtls: {}
```

The peers spec dictates that mTLS is turned on mesh wide.

Recall that the DestinationRule for both `frontend` and `backend` services also have TLS settings defined. Run the following command to inspect the DestinationRule for both services.

```
kubectl get destinationrule frontend -n production --context gke-central -o yaml
kubectl get destinationrule backend -n production --context gke-central -o yaml
```

Output excerpt for both services (Do not copy)

```
...
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
    tls:
      mode: ISTIO_MUTUAL
```

Istio automatically installs necessary keys and certificates for mutual TLS authentication in all sidecar containers. Run command below to confirm key and certificate files exist under `/etc/certs` for one of the frontend pods.

```
export FRONTEND_POD=$(kubectl get pods -l app=frontend -n production --context
gke-central | awk 'NR == 3 {print $1}')
kubectl exec $FRONTEND_POD -n production --context gke-central -c istio-proxy -- ls
/etc/certs
```

Output (Do not copy)

```
cert-chain.pem
key.pem
```

```
root-cert.pem
```

`cert-chain.pem` is Envoy's cert that needs to be presented to the other side. `key.pem` is Envoy's private key paired with Envoy's cert in `cert-chain.pem`. `root-cert.pem` is the root cert to verify the peer's cert. In this example, we only have one Citadel in a cluster, so all Envoys have the same `root-cert.pem`.

Use the `openssl` tool to check if certificate is valid (current time should be in between `Not Before` and `Not After`). Run the following command.

```
kubectl exec $FRONTEND_POD -n production --context gke-central -c istio-proxy -- cat /etc/certs/cert-chain.pem  | openssl x509 -text -noout  | grep Validity -A 2
```

Output (Do not copy)

```
Validity
            Not Before: Sep  7 05:16:35 2018 GMT
            Not After : Dec  6 05:16:35 2018 GMT
```

You can also check the identity of the client certificate. Run the following command.

```
kubectl exec $FRONTEND_POD -n production --context gke-central -c istio-proxy -- cat /etc/certs/cert-chain.pem  | openssl x509 -text -noout  | grep 'Subject Alternative Name' -A 1
```

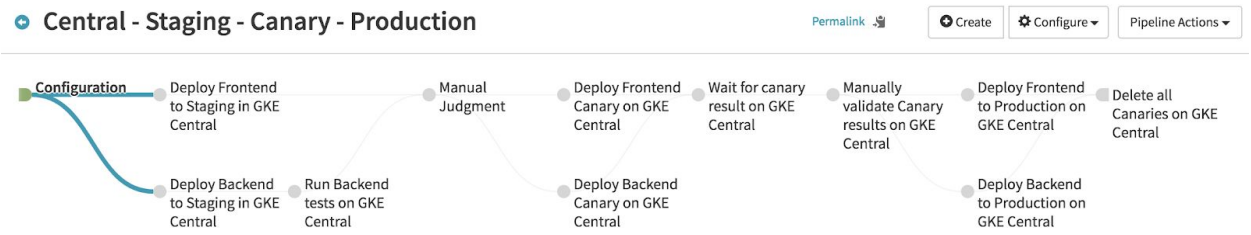
Output (Do not copy)

```
X509v3 Subject Alternative Name:
                URI:spiffe://cluster.local/ns/production/sa/default
```

The `default` service account is utilized in the absence of any defined service account. Istio uses SPIFFE identity framework. Istio and SPIFFE share the same identity document: SVID (SPIFFE Verifiable Identity Document). For example, in Kubernetes, the X.509 certificate has the URI field in the format of "spiffe://<domain>/ns/<namespace>/sa/<serviceaccount>". This enables Istio services to establish and accept connections with other SPIFFE-compliant systems.

Even though the MeshPolicy seen earlier dictates that every service must use mutual TLS, you can disable mTLS on a per service basis.

Recall your Spinnaker pipeline from the previous lab.



The stage labeled **Run Backend tests on GKE Central** runs a Kubernetes job. Currently, with Istio, Kubernetes jobs do not function perfectly well with a sidecar container. Since the sidecar container has no visibility to when the job container finished successfully (or unsuccessfully for that matter), it continues to run. The job never finishes. You need the sidecar container due to mTLS since the `istio-proxy` is where the certificates are stored. In order to bypass this, you implemented an authentication policy to simply disable mTLS just for the backend service running in the `staging` namespace. Inspect the authentication policy for the `backend` service in the `staging` namespace.

```
kubectl get policy -n staging --context gke-central -o yaml
```

Output excerpt (Do not copy)

```
...
spec:
  peers:
  - mtls:
      mode: PERMISSIVE
  targets:
  - name: backend
```

This configures the `backend.staging` service to accept both types of traffic: plain text and TLS. Thus, no request is dropped. This setting in combination with disabling the automatic sidecar injection via annotations allow you to complete the test job successfully.

Describe the job used to test the `staging backend` service by running the following command.

```
kubectl describe job -n staging --context gke-central
```

Output excerpt (Do not copy)

```
...
  Annotations:
...
                sidecar.istio.io/inject=false
  Containers:
   call-curl:
    Image:  gcr.io/spinnaker-marketplace/halyard:stable
    Port:   <none>
```

```
    Command:
      /bin/sh
    Args:
      -c
      curl http://backend.staging/
```

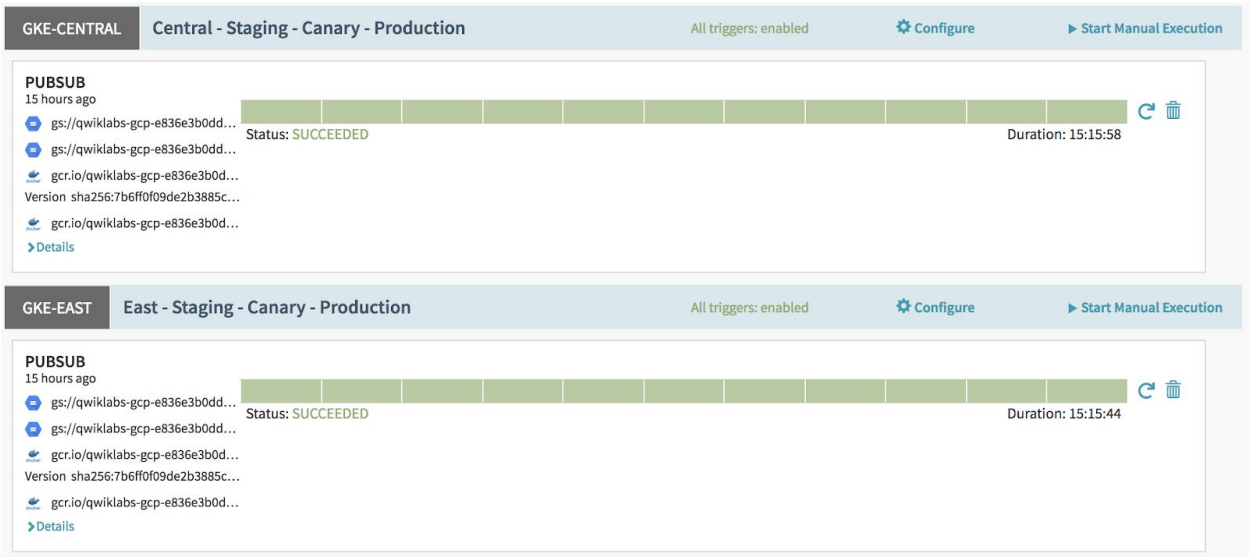
The annotation `sidecar.istio.io/inject=false` disables automatic sidecar injection for this workload. The `Args` field displays the job which is a simple `curl` to the `backend.staging` service. Even though the `backend.staging` service has an `istio-proxy` sidecar with certificates, it can accept both mTLS and plain text requests.

Global Load Balancing

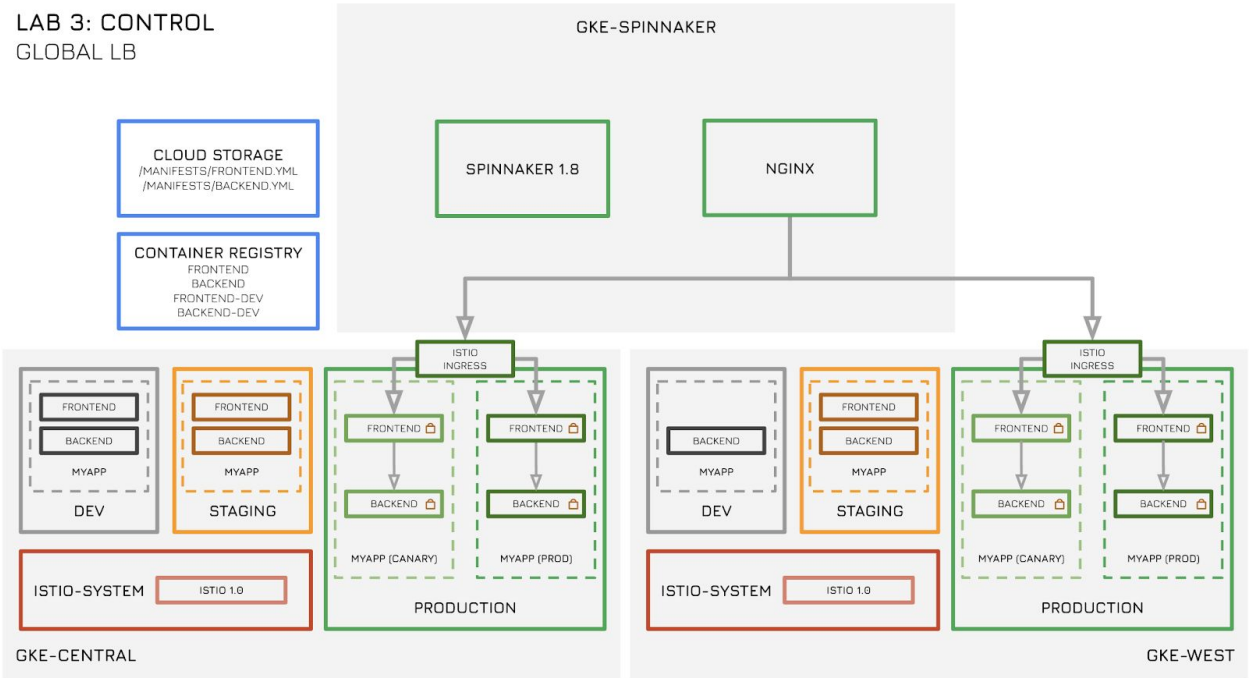
Prior to this section, complete the spinnaker pipeline.

From the **Spinnaker GUI > PIPELINES** page. Hover over the orange rectangles for both pipelines and click **Continue**.

Ensure both pipelines complete successfully before proceeding.



For this workshop, you use `NGINX` load balancer to direct traffic to the web application running in both `gke-central` and `gke-west` clusters. In production environments, you can use a third party provider for this service. CloudFlare, Akamai or backplane.io all provide this functionality.



Store the Ingress IP addresses for the two clusters in variables.

```
export GKE_ONE_ISTIO_GATEWAY=$(kubectl get svc istio-ingressgateway -n istio-system
-o jsonpath="{.status.loadBalancer.ingress[0].ip}" --context gke-central)

export GKE_TWO_ISTIO_GATEWAY=$(kubectl get svc istio-ingressgateway -n istio-system
```

```
-o jsonpath="{.status.loadBalancer.ingress[0].ip}" --context gke-west)
```

Creating NGINX ConfigMap

You can use `gke-spinnaker` for global load balancing. Create an NGINX ConfigMap in `gke-spinnaker`. This ConfigMap provides the `load-balancer.conf` file to the NGINX deployment used to globally load balance traffic to both `gke-central` and `gke-west` clusters. The ConfigMap template file uses placeholder values for backend IPs (i.e. cluster Ingress IPs) which need to be replaced before applying it.

```
kubectx gke-spinnaker
cd ~/advanced-kubernetes-workshop/lb
sed -e s/CLUSTER1_INGRESS_IP/$GKE_ONE_ISTIO_GATEWAY\ weight=1/g -e
s/CLUSTER2_INGRESS_IP/$GKE_TWO_ISTIO_GATEWAY\ weight=1/g glb-configmap-var.yaml >
glb-configmap.yaml
```

Confirm that the Ingress IP addresses are in the output file.

```
cat glb-configmap.yaml
```

Output excerpt (Do not copy)

```
...
  load-balancer.conf: |
    upstream example.com {
      server 35.188.96.218 weight=1;
      server 35.231.81.175 weight=1;
...
```

Apply the configmap.

```
kubectl apply -f glb-configmap.yaml
```

Creating NGINX Deployment and Service

Create the NGINX deployment and service

```
kubectl apply -f nginx-dep.yaml
kubectl apply -f nginx-svc.yaml
```

Ensure that the `global-lb-nginx` Service has a public IP address. You can run the following commands a few times or watch it using the `-w` option in the command line.

```
kubectl get service global-lb-nginx
```

Accessing NGINX load balancer frontend

Once you have the public IP address, store it in a variable.

```
export GLB_IP=$(kubectl get service global-lb-nginx -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
```

Do a loop-curl to the NGINX frontend IP address to ensure traffic is being sent to both `gke-central` and `gke-west` clusters.

```
for i in `seq 1 20`; do curl -s $GLB_IP | grep gke | sed -e 's/<p>/ /g' -e 's/<\/p>/
/g'; done
```

Output (Do not copy)

```
gke-west
gke-central
gke-west
gke-central
gke-west
gke-central
gke-west
gke-central
gke-west
gke-central
gke-west
gke-central
gke-west
gke-central
gke-west
gke-central
gke-west
gke-central
gke-west
gke-central
```

Traffic to the two clusters is being split 50/50. This ratio can be controlled by the `weight` field in the ConfigMap generated earlier. Recall that you set the `weight` fields for both backends (or Istio Ingress gateway IPs) in the NGINX ConfigMap to `1`.

Controlling global traffic to both clusters using NGINX

Adjust the `weight` fields in the ConfigMap. Change the `weight` for `gke-west` (or cluster-2) to `4`. Set the `weight` for `gke-central` (or cluster-1) to `1` (same as before). Apply the new configmap and recreate the deployment for the new ConfigMap.

```
sed -e s/CLUSTER1_INGRESS_IP/$GKE_ONE_ISTIO_GATEWAY\ weight=1/g -e
```

```
s/CLUSTER2_INGRESS_IP/$GKE_TWO_ISTIO_GATEWAY\ weight=4/g glb-configmap-var.yaml >
glb-configmap-2.yaml
kubectl delete -f glb-configmap.yaml
kubectl delete -f nginx-dep.yaml
kubectl apply -f glb-configmap-2.yaml
kubectl apply -f nginx-dep.yaml
```

Review the configmap by running the following command.

```
kubectl get configmap nginx --context gke-spinnaker -o yaml | head -7
```

Output (Do not copy)

```
apiVersion: v1
data:
  load-balancer.conf: |
    upstream example.com {
      server 35.225.163.174 weight=1;
      server 35.230.67.55 weight=4;
    }
```

Notice the `weight` ratio for the two clusters. `gke-central` cluster has a `weight` of **1** and `gke-west` cluster has a `weight` of **4**. This means **4** times more traffic is sent to the `gke-west` cluster compared to `gke-central` cluster.

Wait a few moments for the NGINX service to come up. Do a for loop curl on the `GLB_IP` and you can see more traffic going to `gke-west` due to higher `weight` (4 versus 1).

```
for i in `seq 1 20`; do curl -s $GLB_IP | grep gke | sed -e 's/<p>/ /g' -e 's/<\/p>/
/g'; done
```

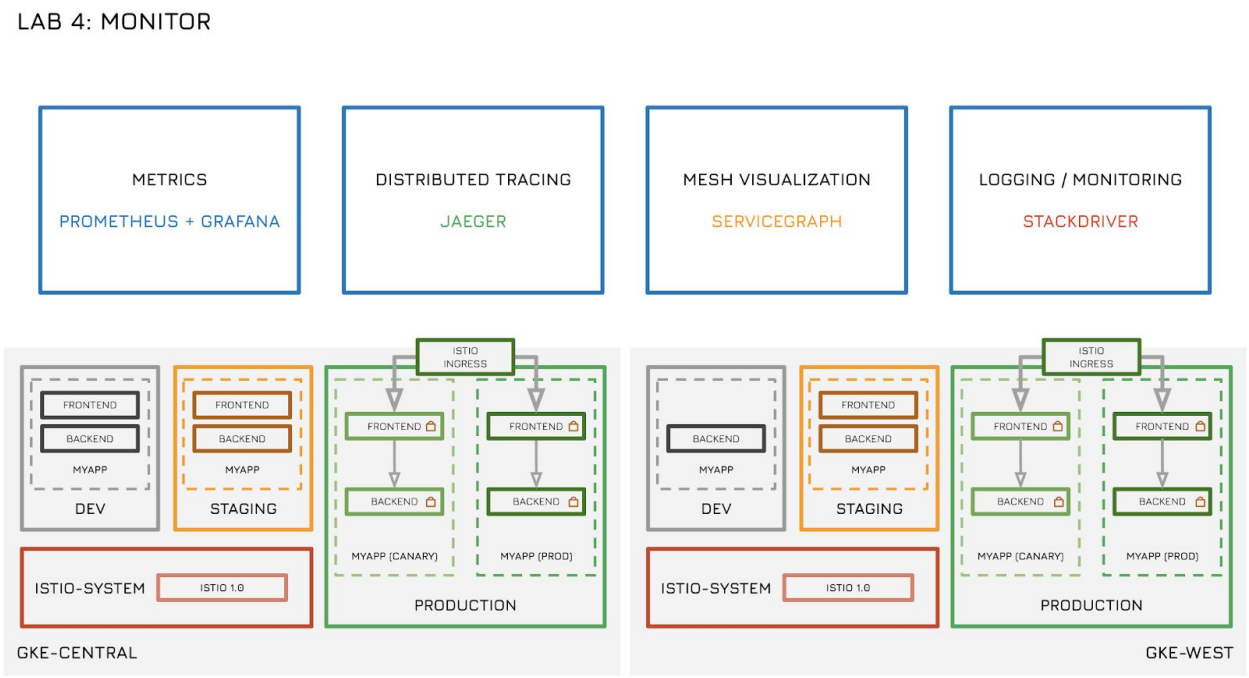
Output (Do not copy)

```
gke-west
gke-west
gke-central
gke-west
gke-west
gke-west
gke-west
gke-central
gke-west
gke-west
gke-west
gke-west
gke-central
gke-west
gke-west
gke-west
gke-west
gke-central
```

```
gke-west
gke-west
```

END OF LAB 3 - Please STOP and wait for instructions before proceeding!

Lab 4: Monitor (45 mins)



In this lab, you:

- Use Istio addons to monitoring and troubleshooting.
- Use Prometheus and Grafana for metrics and visualization.
- Use Jaeger for distributed tracing.
- Use Google Stackdriver for monitoring multiple Kubernetes Engine clusters and logging.
- Use Servicegraph for service mesh visualization.

Collecting metrics with Prometheus and Grafana

Mixer comes with a built-in Prometheus adapter that exposes an endpoint serving generated metric values. The Prometheus add-on is a Prometheus server that comes preconfigured to scrape Mixer endpoints to collect the exposed metrics. It provides a mechanism for persistent storage and querying of Istio metrics.

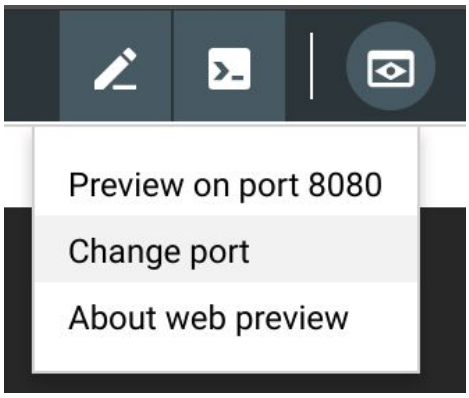
The configured Prometheus add-on scrapes three endpoints:

- `istio-mesh` (`istio-mixer.istio-system:42422`): all Mixer-generated mesh metrics.
- `mixer` (`istio-mixer.istio-system:9093`): all Mixer-specific metrics. Used to monitor Mixer itself.
- `envoy` (`istio-mixer.istio-system:9102`): raw stats generated by Envoy (and translated from Statsd to Prometheus).

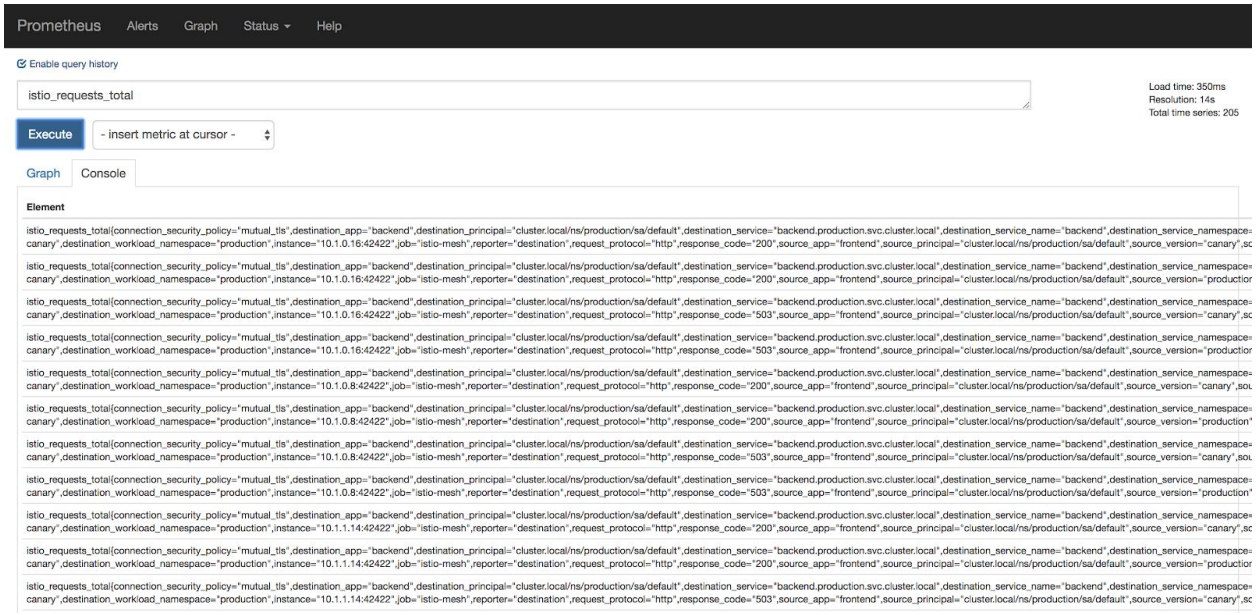
Ensure that Prometheus port 9090 (and 9091 for `gke-west` cluster) is exposed using the `workshop_connect` script. If in doubt, re-run the script by running the following command.

```
workshop_connect
```

Click **Web Preview** in Cloud Shell (top bar, right hand side) and click **Change port**.



Change the port to **9090** and click **Change and Preview**. This opens a new tab in Chrome for Prometheus. Execute a Prometheus query. In the "`Expression`" input box at the top of the web page, enter the text: `istio_requests_total`. Then, click the **Execute** button.



Try a few other queries and inspect the values. Generate some traffic using hey to get interesting metrics.

Total count of all requests to the `backend` service.

```
istio_requests_total{destination_service="backend.production.svc.cluster.local"}
```

Total count of all requests to canary version of the `frontend` service.

```
istio_requests_total{destination_service="frontend.production.svc.cluster.local",
destination_version="canary"}
```

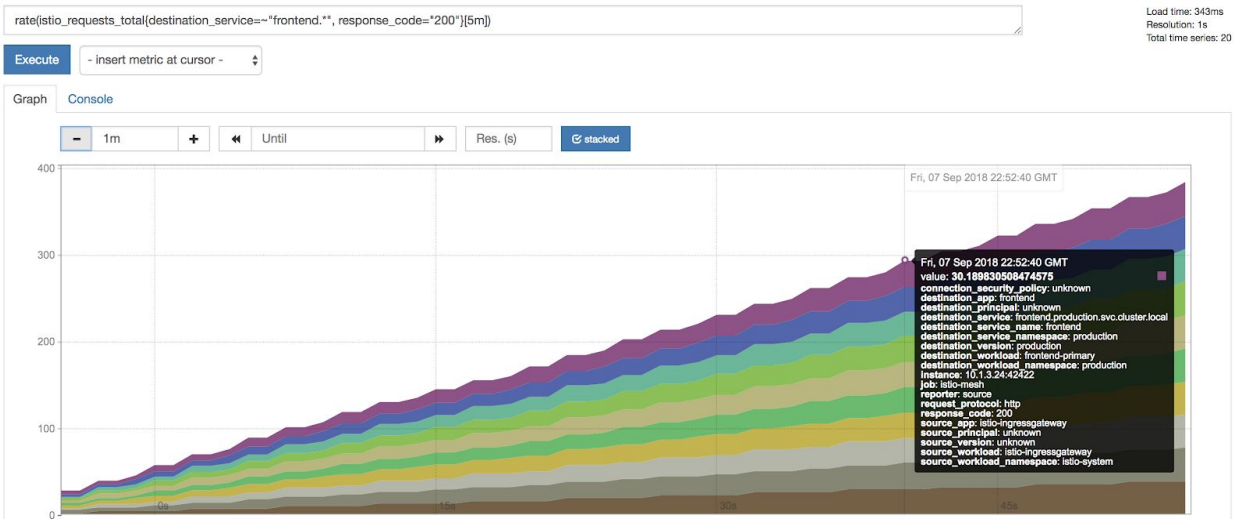
Rate of requests over the past **5** minutes to all instances of the `frontend` service.

```
rate(istio_requests_total{destination_service=~"frontend.*",
response_code="200"}[5m])
```

Generate some traffic using **hey**. Run the following command.

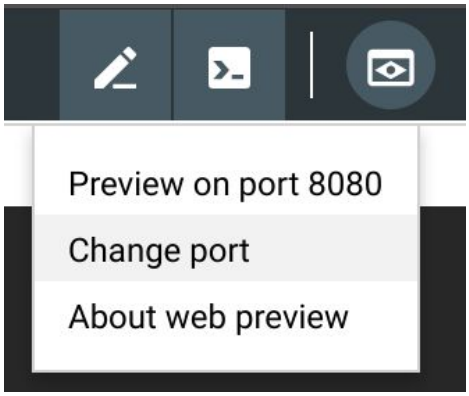
```
export GKE_ONE_ISTIO_GATEWAY=$(kubectl get svc istio-ingressgateway -n istio-system
-o jsonpath="{.status.loadBalancer.ingress[0].ip}" --context gke-central)

hey -z 10m http://$GKE_ONE_ISTIO_GATEWAY
```

Click on the Graph tab (under the Execute button). Change the time granularity to **1m** by clicking the minus button. Check the `stacked` checkbox. Hovering over the graph you get additional metrics per pod.

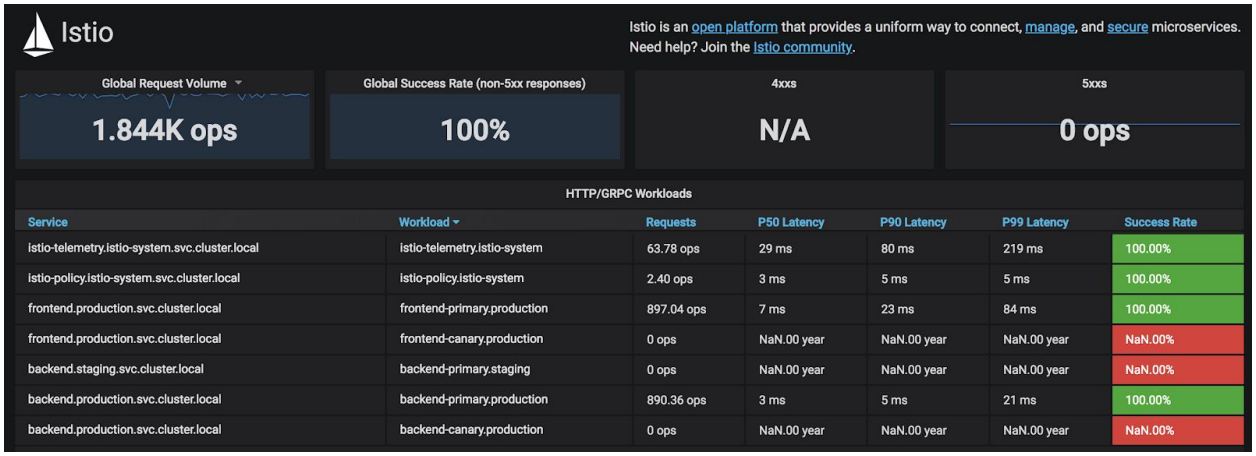


Most of the time, you use Grafana to visualize Prometheus metrics. Open Grafana Chrome tab (if its not already open).

Click **Web Preview** in Cloud Shell (top bar, right hand side) and click **Change port**.

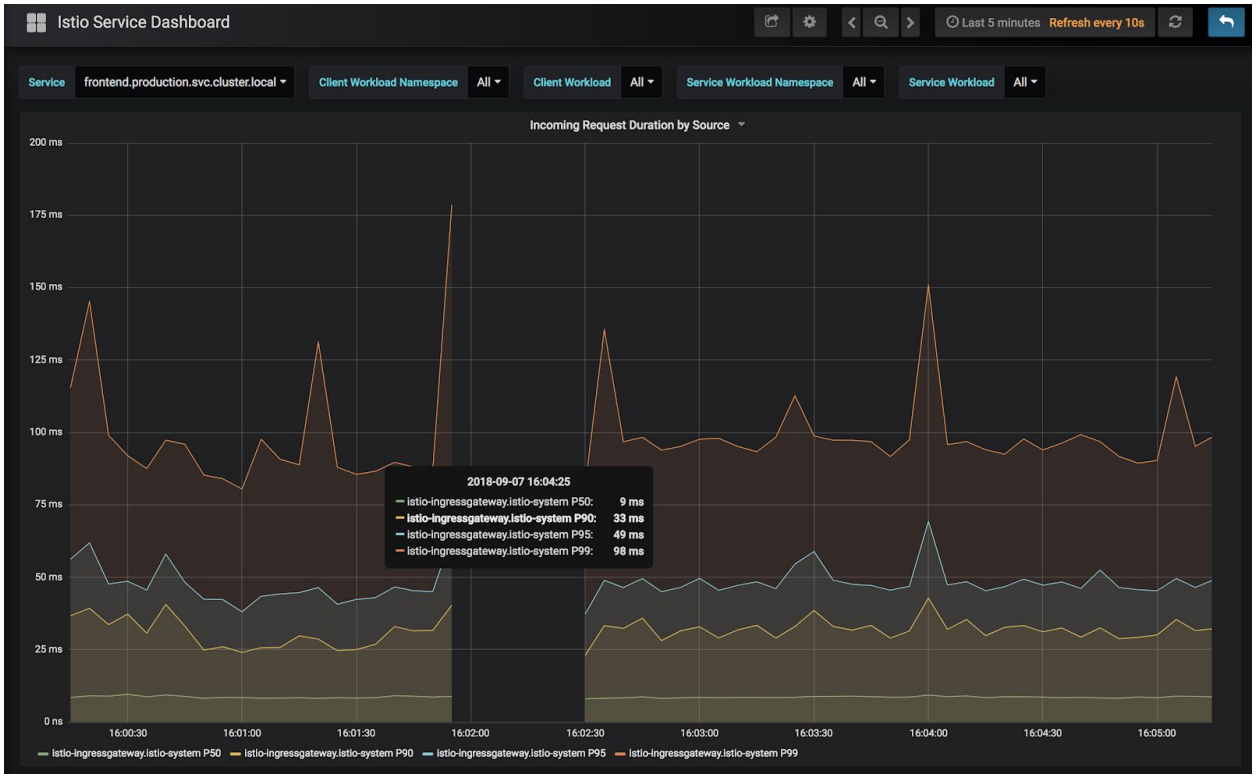


Change the port to **3000** and click **Change and Preview**. This opens a new tab in Chrome for Grafana. Click on **Home** at the top and then **Istio Mesh Dashboard**.



This table shows you all the services currently running on the `gke-central` cluster, global request volume and success rate.

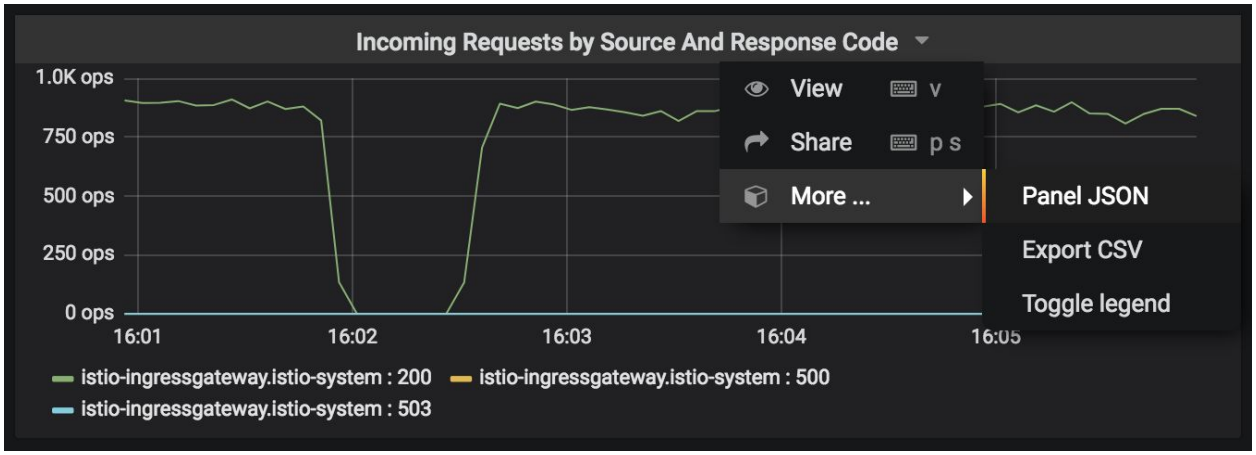
Click on the **frontend.production.svc.cluster.local** service, which takes you to the **Istio Service Dashboard**. This is the same dashboard used in Lab 3. Hovering over a single chart and pushing **V** makes it full screen.



Use **ESC** key to go back.

Note: You may notice no traffic is being sent to either the `frontend` or the `backend` canary. Recall that you completed the Spinnaker pipeline which after promoting the current canary to production, deletes the canaries. This is why all traffic is going to the new production/primary `frontend` and `backend` services.

Click the **Incoming Request by Source and Response Code** > **More** and **Panel JSON**.

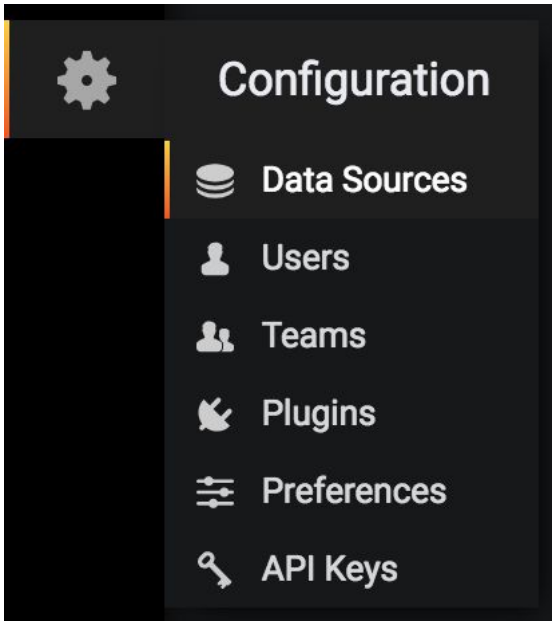


In the JSON Panel, scroll down to the `targets` object, which has information regarding the Prometheus metric being used.

At the top of the Istio Service Dashboard, from the **Service** dropdown, you can select all services running on the cluster and inspect metrics for a particular service.

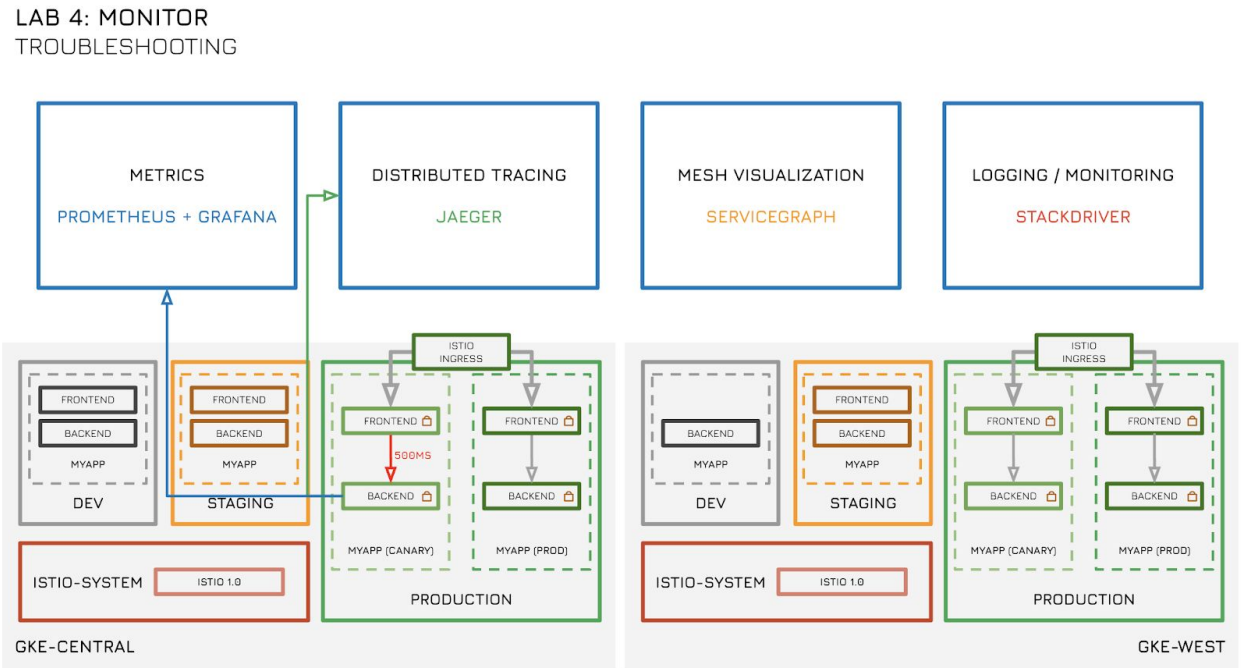
Click the **Istio Service Dashboard** at the top and inspect all the other built-in dashboards.

Click on the **Settings** icon in the left hand side column and then select **Data Sources**.

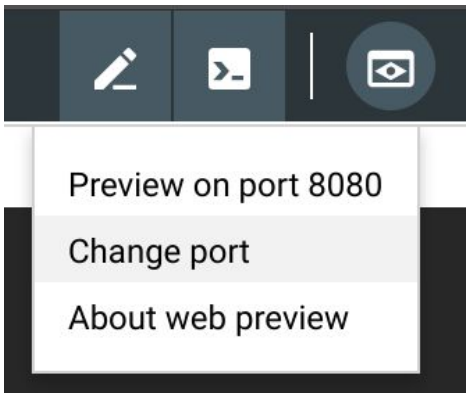


Click on the **Prometheus** data source and inspect the settings. Grafana is accessing Prometheus on port 9090.

Troubleshooting via distributed tracing with Jaeger

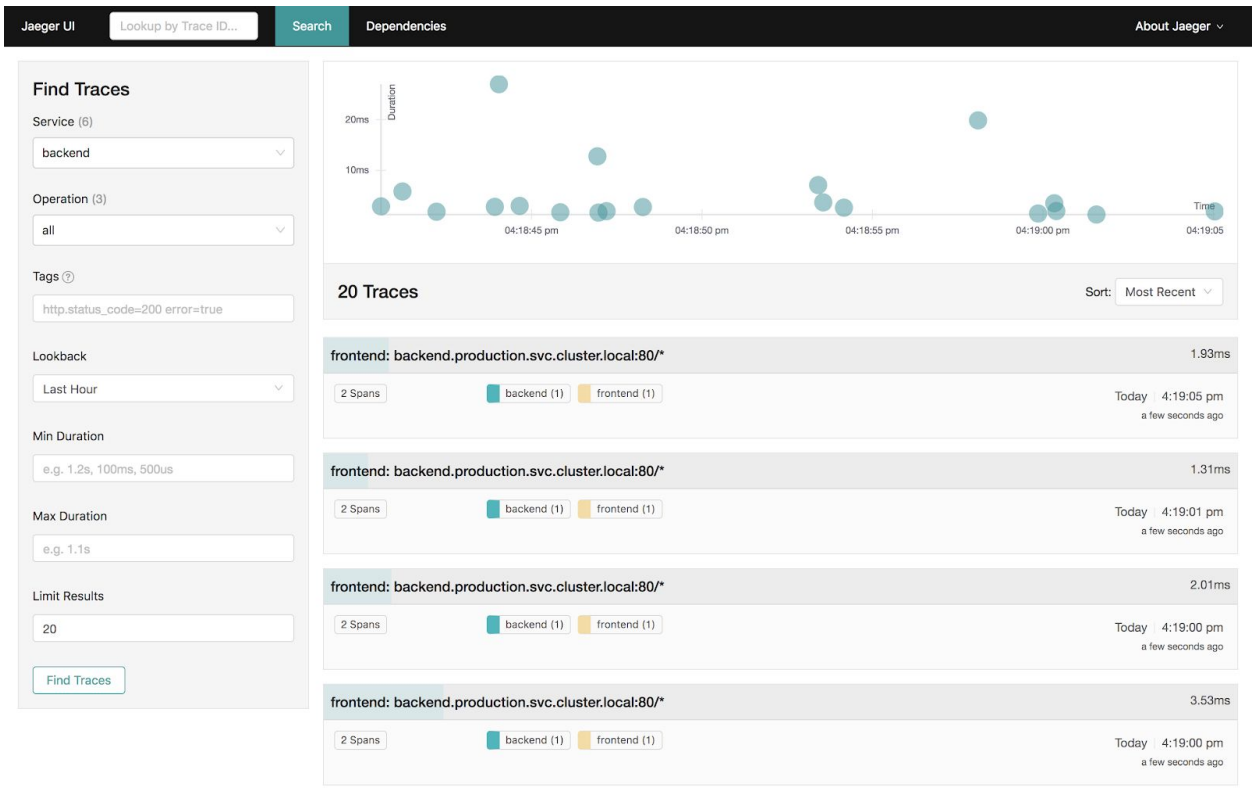


Click **Web Preview** in Cloud Shell (top bar, right hand side) and click **Change port**.

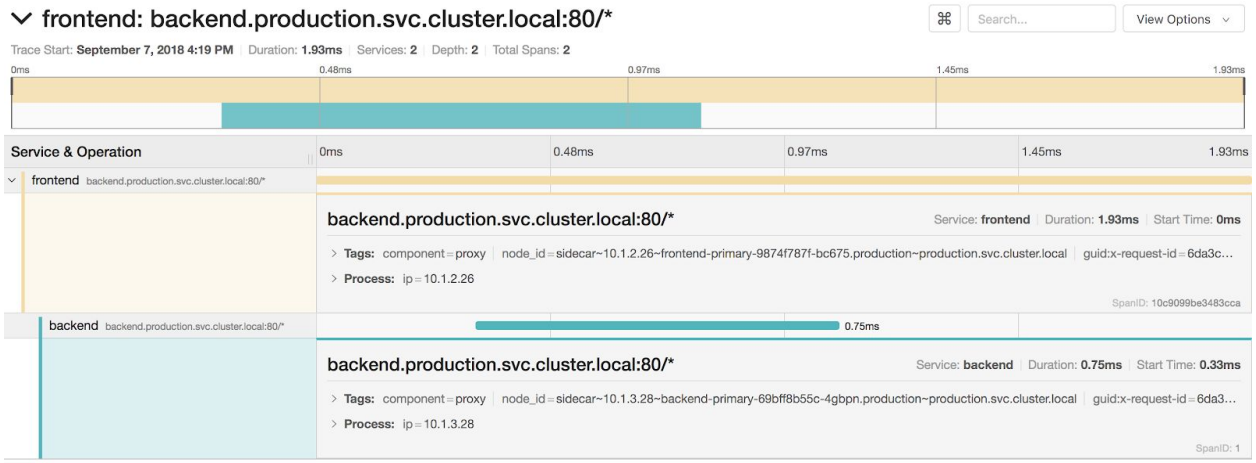


Change the port to **16686** and click **Change and Preview**. This opens a new tab in Chrome for Jaeger.

In the **Service** dropdown, select `backend` and click **Find Traces** at the bottom.



Select one of the traces and click individual operations to open more details.



In this case, you see two services (`frontend` and `backend`) and the amount of time each operation takes. In the example above, the total time of the trace is **1.93ms**. The `backend` operation takes **0.75ms**.

Lets inject an artificial delay to the `backend` service to simulate faulty code. Prior to this, ensure no traffic is being sent to the application.

Inspect the `main.go` file for the `backend` service. Don't worry if you don't speak Go. Run the following command.

```
cat ~/advanced-kubernetes-workshop/services/backend/main.go
```

Output excerpt (Do not copy)

```
func index(w http.ResponseWriter, r *http.Request) {

//      duration := time.Duration(500)*time.Millisecond
//      time.Sleep(duration)

        fmt.Printf("Handling %+v\n", r);

        host, err := os.Hostname()

        if err != nil {
                http.Error(w, fmt.Sprintf("Error retrieving hostname: %v", err),
500)
                return
        }

        msg := fmt.Sprintf("Host: %s\nSuccessful requests: %d", host, count)
        count += 1

        io.WriteString(w, msg)
```

```
}
```

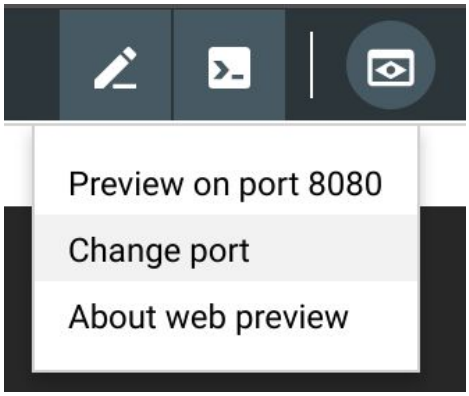
The backend service consists of a function named index. The `index` function prints the pod's hostname and number of successful requests, which is displayed as messages on the frontend webpage. Inspect the first two lines of the function. They are currently commented out. These two lines of code inject a **500ms** delay when the function is called. Uncomment these lines by running the following commands.

```
sed -i  -e 's|//||g' ~/advanced-kubernetes-workshop/services/backend/main.go
```

Build the app using the following command.

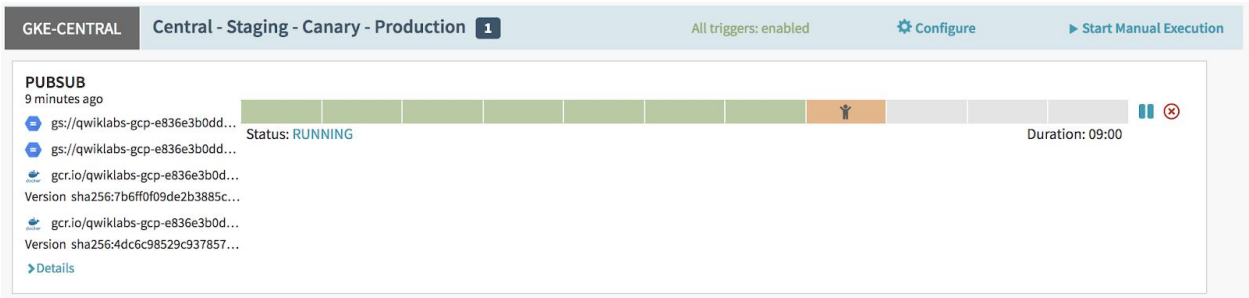
```
cd ~/advanced-kubernetes-workshop/services/backend/
./build.sh
```

Open the Spinnaker GUI. Click **Web Preview** in Cloud Shell (top bar, right hand side) and click **Change port**.



Change the port to **8080** and click **Change and Preview**. This opens a new tab in Chrome for Spinnaker.

Navigate to the **PIPELINES** page. You see the pipelines triggered. Click through the first manual judgement stage. Wait at the second manual judgement stage.

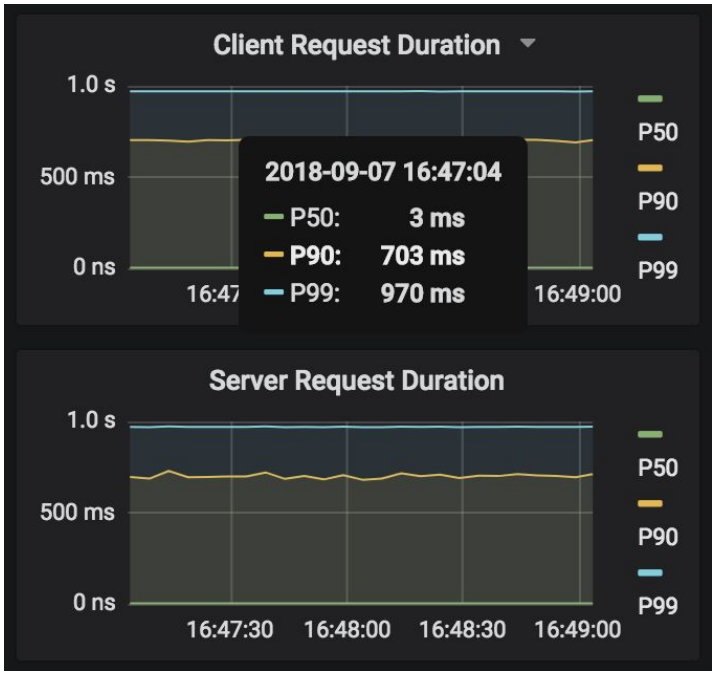


Generate some traffic using **hey**. Run the following command.

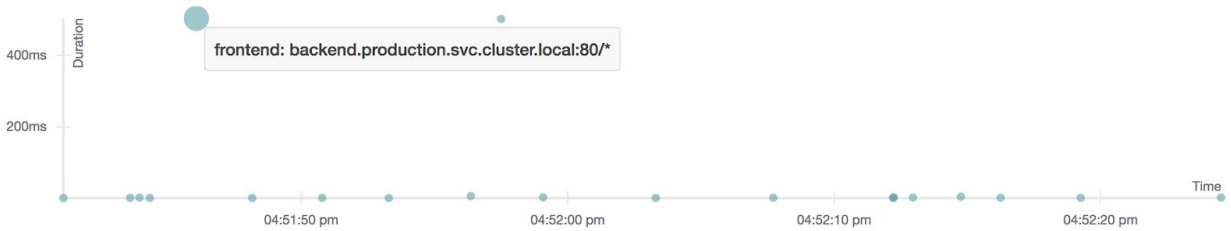
```
export GKE_ONE_ISTIO_GATEWAY=$(kubectl get svc istio-ingressgateway -n istio-system
-o jsonpath="{.status.loadBalancer.ingress[0].ip}" --context gke-central)

hey -z 10m http://$GKE_ONE_ISTIO_GATEWAY
```

Inspect the Grafana **Istio Service Dashboard** for the `backend` service. Look at the **Client and Server Request Duration** charts. Notice the `P90` and `P99` durations have considerably increased to above **500ms**. This is due to the delay injected in the previous step.

Note: Only the `canary backend` release is exhibiting the additional 500ms delay while the `production (primary) backend` is serving traffic without the additional delay.



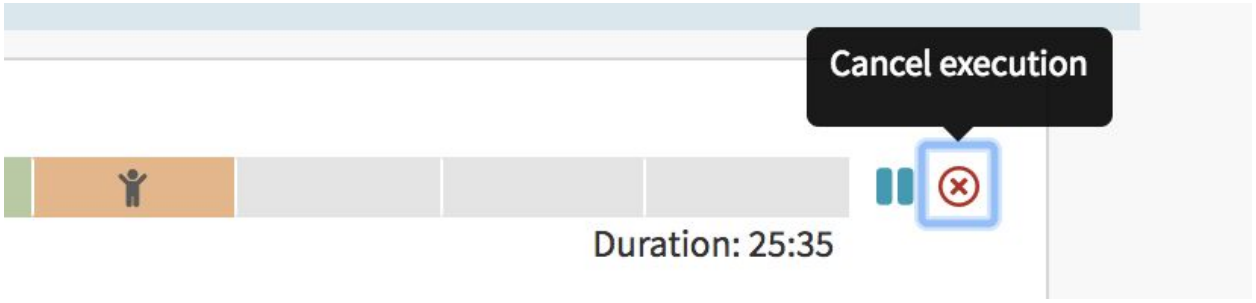
In the **Jaeger UI** tab home page, select the `backend` service and click on **Find Traces** at the bottom. Note the dots over 500ms.



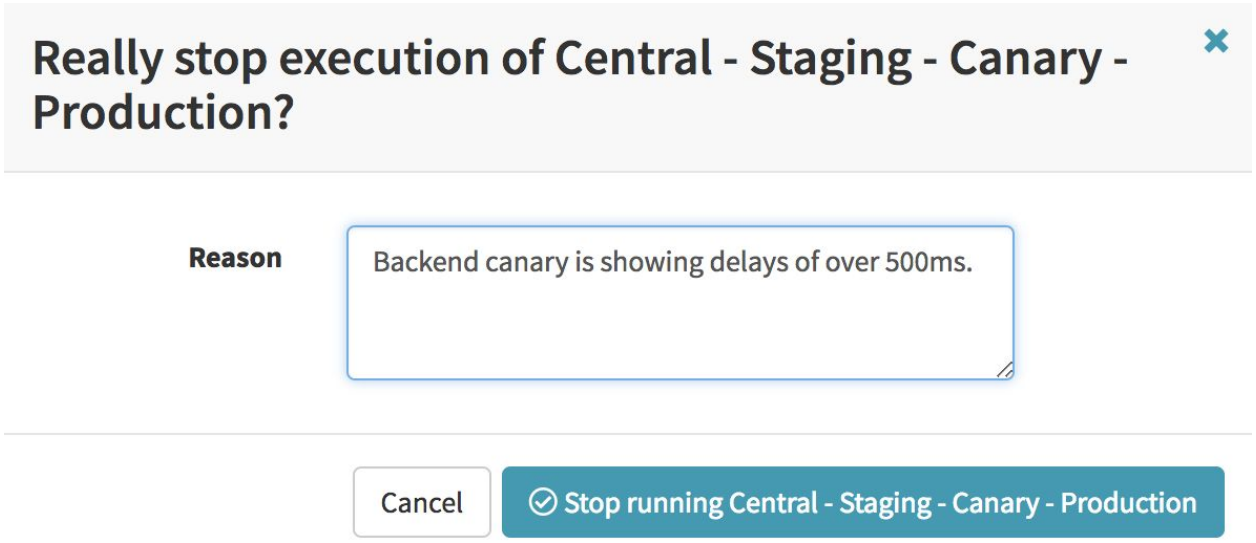
Click on one of these dots to open the details of this trace.

Note the start time on the `backend` service. In this case, the `backend` service start time is **0.33ms**, with a duration of **501.57ms**. This shows that the issue is with the `backend` service. You can also see the pod hostname in `node_id` as well as the IP address. In this case, the pod hostname is `backend-canary-645f8d79c7` with IP address of 10.1.2.30.

In the Spinnaker GUI, click on the cancel button to the right of the pipeline.



Optionally make a note of why you are canceling the pipeline.



Note that canceling a Spinnaker pipeline does not revert back any changes. It simply ceases the pipeline at the current step. Anything deployed up to this point will remain. You may create another pipeline to delete the canaries that can be triggered automatically upon cancellation of another pipeline.

Logging and Monitoring with Google Stackdriver

Google Stackdriver allows collection of Kubernetes Engine cluster metrics. The Kubernetes Engine clusters installed as part of this workshop already have this feature enabled. To verify, check the resources in the kube-system namespaces for each cluster and ensure that appropriate resources are installed. Run the following command.

```
kubectl get pods -n kube-system --context gke-central
kubectl get pods -n kube-system --context gke-west
kubectl get pods -n kube-system --context gke-spinnaker
```

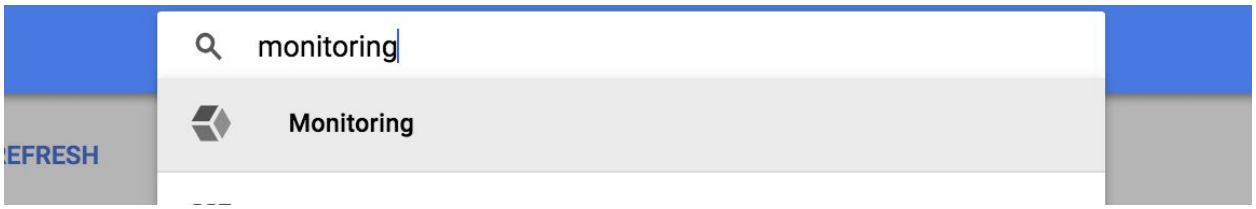
Output excerpt (Do not copy)

```
...
NAME                                                  READY     STATUS
RESTARTS   AGE
fluentd-gcp-scaler-7c5db745fc-ww7f7                   1/1       Running   0
1h
...
heapster-v1.5.3-779d86c765-292qr                      3/3       Running   0
1h
...
metadata-agent-47ffj                                  1/1       Running   11
1h
```

Ensure you have these resources running in each cluster.

- fluentd-gcp-...: the Stackdriver Logging agent.
- heapster-...: the Monitoring agent.
- metadata-... : the Stackdriver Metadata agent.

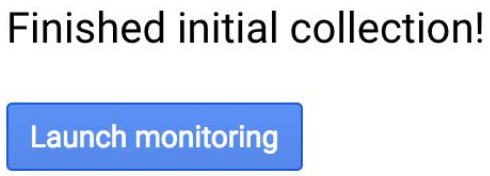
In the Cloud Console web GUI search bar, search for “monitoring”. Click Monitoring.



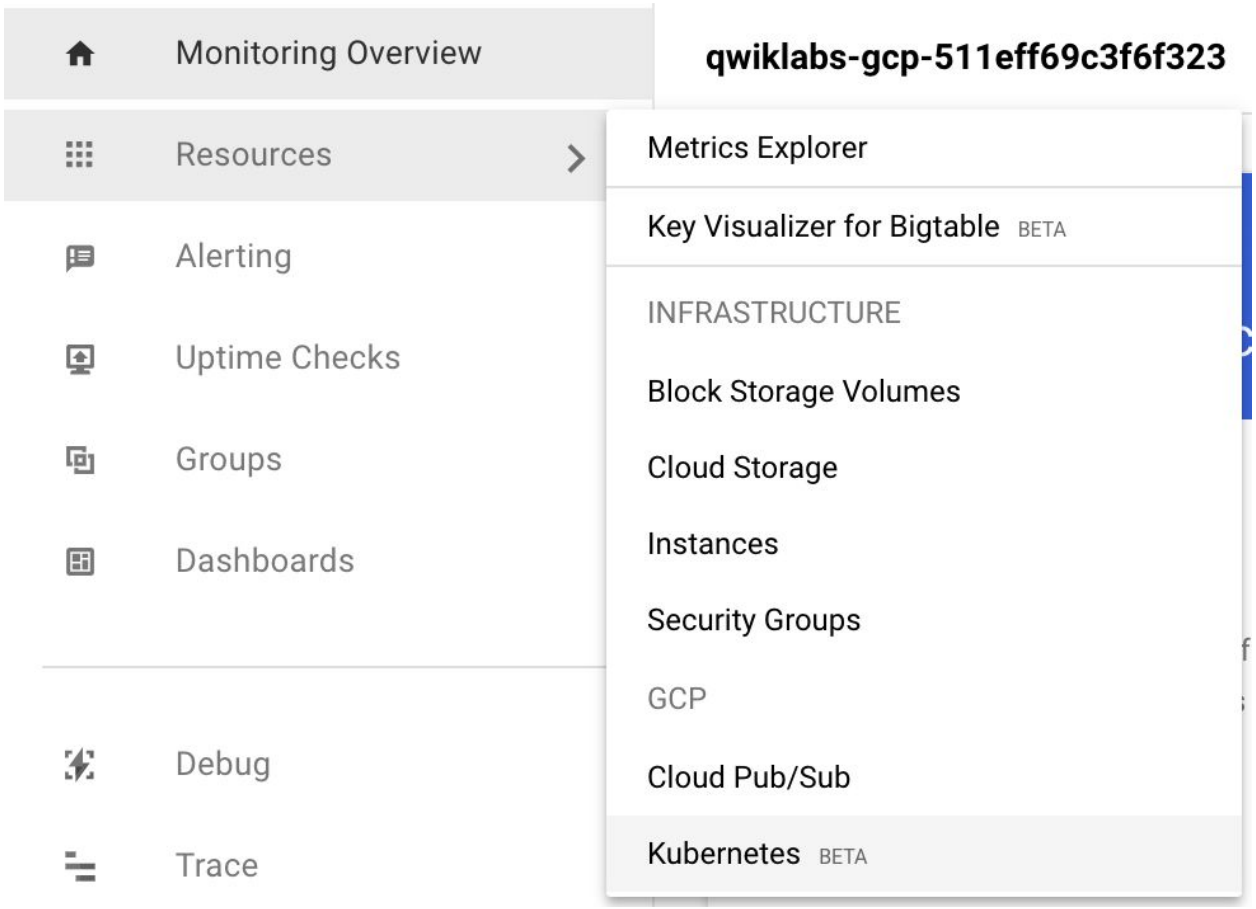
Create an account.

Click **Continue** on the next screen and **Skip AWS Setup** on the following screen. Click **Continue** again on the next screen. Select **No Reports** radio button on the next screen and click **Continue**. Wait a few moments for account creation to be completed successfully.

Click **Launch Monitoring**.

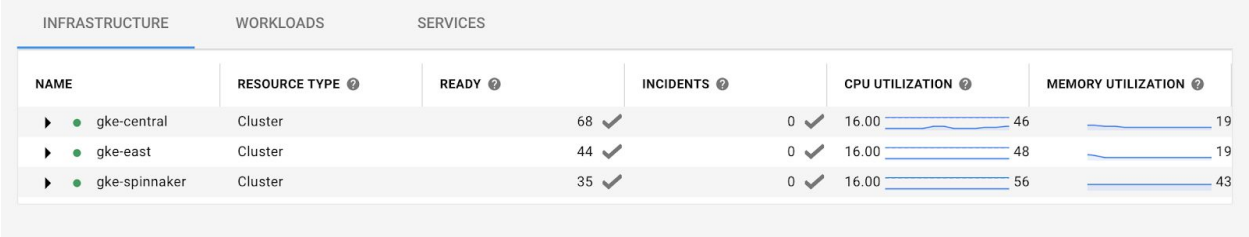


Click on **Resources** > **Kubernetes (BETA)** link.



Note: If the `Kubernetes (BETA)` link does not show up right away. Wait a few moments and refresh the Monitoring page.

You can see the three Kubernetes Engine clusters.



The main table consists of three tabs.

- Infrastructure
- Workloads

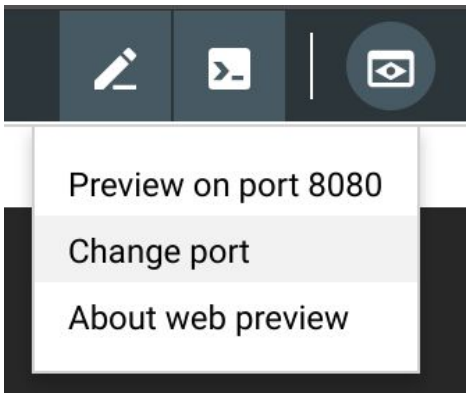
- Services

Navigate through the three tabs by opening up the clusters and looking through various metrics presented as part of the Kubernetes logging and monitoring agent.

Learn more about observing your Kubernetes clusters via Stackdriver.

Service Mesh Visualization with Kiali

Click **Web Preview** in Cloud Shell (top bar, right hand side) and click **Change port**.

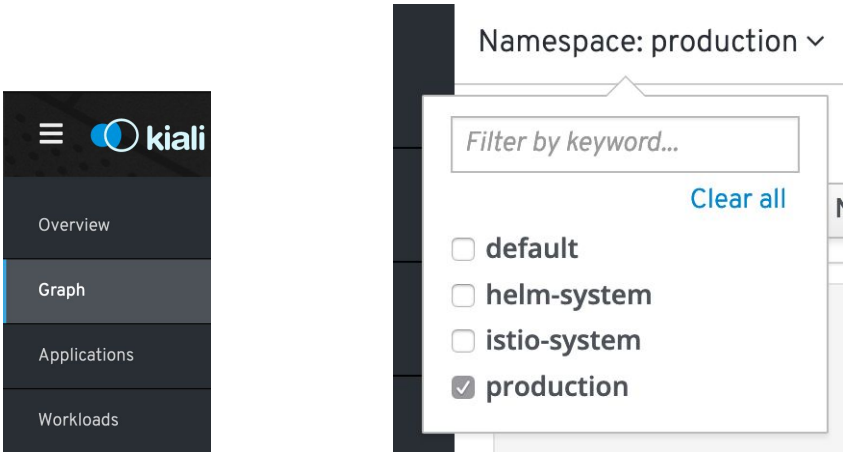


Change the port to **20001** and click **Change and Preview**. This opens a new tab in Chrome for Kiali. You will login with the username **admin** and the password **admin.**

Kiali is an open source dashboard that provides visualization and insight into the service mesh. It provides information about Istio configuration as well. Using Kiali an operator can see which services are in the mesh, how traffic flows between those services, and even access tracing for this traffic. Kiali is intended to be a one-stop dashboard for inspecting and understanding the service mesh.

Kiali Service Graph Visualization

Click **Graph** in the left bar and then in the top left dropdown, select only the **production** namespace.



You should see a graph of the existing services and the traffic between them.

Congratulations on completing the Advanced Kubernetes Workshop by Google Cloud Platform. We hope you enjoyed your experience.