**Google Kubernetes Engine Qwik Start**

# Task1: Create a GKE cluster

To **create a cluster**, run the following command, replacing [CLUSTER-NAME] with the name you choose for the cluster (**for example**:my-cluster).

| gcloud container clusters create [CLUSTER-NAME] |
| --- |

# Task2: Get authentication credentials for the cluster

After creating your cluster, you need authentication credentials to interact with it. To **authenticate the cluster**, run the following command, replacing [CLUSTER-NAME] with the name of your cluster:

| gcloud container clusters get-credentials [CLUSTER-NAME] |
| --- |

# Task3: Quick Deploy an application to the cluster

You can now deploy a [containerized application](https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview) to the cluster. For this lab, you'll run hello-app in your cluster.

GKE uses Kubernetes objects to create and manage your cluster's resources. Kubernetes provides the [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) object for deploying stateless applications like web servers. [Service](https://kubernetes.io/docs/concepts/services-networking/service/) objects define rules and load balancing for accessing your application from the internet.

1. To **create a new Deployment** hello-server from the hello-app container image, run the following [kubectl create](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#create) command:

| kubectl create deployment hello-server --image=gcr.io/google-samples/hello-app:1.0 |
| --- |

This Kubernetes command creates a Deployment object that represents hello-server. In this case, --image specifies a container image to deploy. The command pulls the example image from a [Container Registry](https://cloud.google.com/container-registry/docs) bucket. gcr.io/google-samples/hello-app:1.0 indicates the specific image version to pull. If a version is not specified, the latest version is used.

1. To **create a Kubernetes Service**, which is a Kubernetes resource that lets you expose your application to external traffic, run the following [kubectl expose](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#expose) command:

| kubectl expose deployment hello-server --type=LoadBalancer --port 8080 |
| --- |

In this command:

* --port specifies the port that the container exposes.
* type="LoadBalancer" creates a Compute Engine load balancer for your container.

1. To **inspect** the hello-server Service, run [kubectl get](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get):

| kubectl get service |
| --- |

**Note:** It might take a minute for an external IP address to be generated. Run the previous command again if the EXTERNAL-IP column status is **pending**.

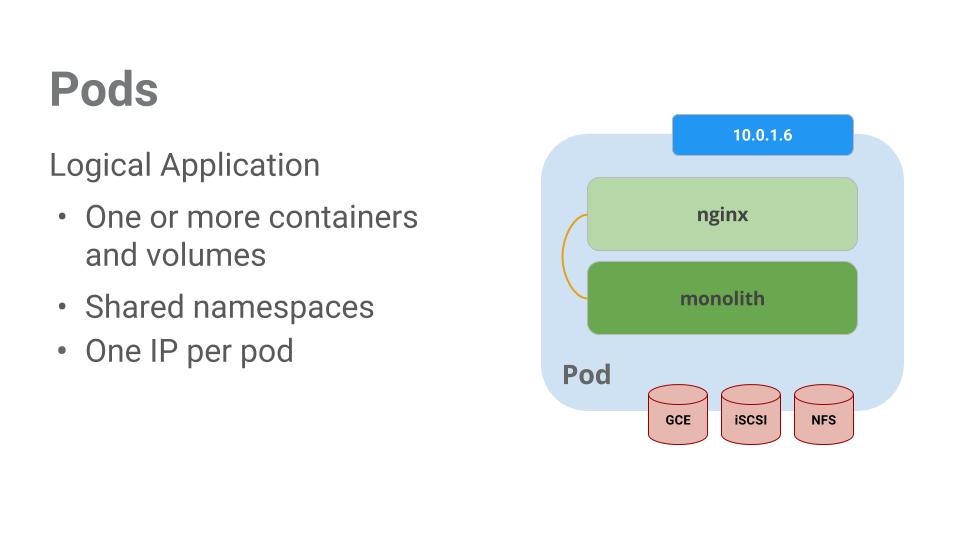
1. To view the application from your web browser, open a new tab and enter the following address, replacing [EXTERNAL IP] with the EXTERNAL-IP for hello-server.

| http://[EXTERNAL-IP]:8080 |
| --- |

# Task4: Creating Pods and Interacting with Pods

At the core of Kubernetes is the [Pod](http://kubernetes.io/docs/user-guide/pods/).

Pods represent and hold a collection of one or more containers. Generally, if you have multiple containers with a hard dependency on each other, you package the containers inside a single pod.



In this example there is a pod that contains the monolith and nginx containers.

Pods also have [Volumes](http://kubernetes.io/docs/user-guide/volumes/). Volumes are data disks that live as long as the pods live, and can be used by the containers in that pod. Pods provide a shared namespace for their contents which means that the two containers inside of our example pod can communicate with each other, and they also share the attached volumes.

Pods also share a network namespace. This means that there is one IP Address per pod.

Next, a deeper dive into pods.

## Task4-1: Creating Pods

Pods can be created using pod configuration files. Take a moment to explore the monolith pod configuration file.

1. Create the monolith pod using kubectl:

| kubectl create -f pod.yaml |
| --- |

1. Examine your pods. Use the kubectl get pods command to list all pods running in the default namespace:

| kubectl get pods |
| --- |

**Note:** It may take a few seconds before the monolith pod is up and running. The monolith container image needs to be pulled from the Docker Hub before you can run it.

Once the pod is running, use kubectl describe command to get more information about the monolith pod:

| kubectl describe pods monolith |
| --- |

You'll see a lot of the information about the monolith pod including the Pod IP address and the event log. This information will come in handy when troubleshooting.

Kubernetes makes it easy to create pods by describing them in configuration files and easy to view information about them when they are running. At this point you have the ability to create all the pods your deployment requires!

## Task4-2: Interacting with Pods

By default, pods are allocated a private IP address and cannot be reached outside of the cluster. Use the kubectl port-forward command to map a local port to a port inside the monolith pod.

From this point on the lab will ask you to work in multiple cloud shell tabs to set up communication between the pods. Any commands that are executed in a second or third command shell will be denoted in the command's instructions.

Open a second Cloud Shell terminal. One to run the kubectl port-forward command, and the other to issue curl commands.

1. In the **2nd terminal**, run this command to set up port-forwarding:

| kubectl port-forward monolith 10080:80 |
| --- |

1. Now in the **1st terminal** start talking to your pod using curl:

| curl http://127.0.0.1:10080 |
| --- |

Uh oh.

1. Try logging in to get an auth token back from the monolith:

| curl -u user http://127.0.0.1:10080/login |
| --- |

At the login prompt, use the super-secret password "password" to login.

1. Logging in caused a JWT token to print out. Since Cloud Shell does not handle copying long strings well, create an environment variable for the token.

| TOKEN=$(curl http://127.0.0.1:10080/login -u user|jq -r '.token') |
| --- |

Enter the super-secret password "password" again when prompted for the host password.

1. Use this command to copy and then use the token to hit the secure endpoint with curl:

| curl -H "Authorization: Bearer $TOKEN" http://127.0.0.1:10080/secure |
| --- |

At this point you should get a response back from our application, letting us know everything is right in the world again.

1. Use the kubectl logs command to view the logs for the monolith Pod.

| kubectl logs monolith |
| --- |

1. **Open a 3rd terminal** and use the -f flag to get a stream of the logs happening in real-time:

| kubectl logs -f monolith |
| --- |

1. Now if you use curl in the **1st terminal** to interact with the monolith, you can see the logs updating (in the **3rd terminal**):

| curl http://127.0.0.1:10080 |
| --- |

1. Use the kubectl exec command to run an interactive shell inside the Monolith Pod. This can come in handy when you want to troubleshoot from within a container:

| kubectl exec monolith --stdin --tty -c monolith /bin/sh |
| --- |

For example, once you have a shell into the monolith container you can test external connectivity using the ping command: `ping -c 3 google.com`

1. Be sure to log out when you're done with this interactive shell.

| exit |
| --- |

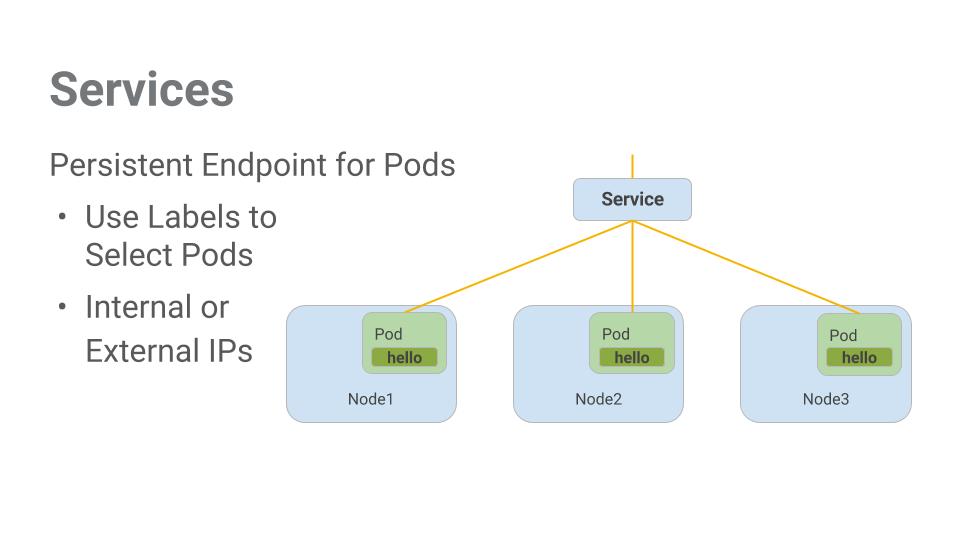
As you can see, interacting with pods is as easy as using the kubectl command. If you need to hit a container remotely, or get a login shell, Kubernetes provides everything you need to get up and going.

# Task5: Create a Service

Pods aren't meant to be persistent. They can be stopped or started for many reasons - like failed liveness or readiness checks - and this leads to a problem:

What happens if you want to communicate with a set of Pods? When they get restarted they might have a different IP address.

That's where [Services](http://kubernetes.io/docs/user-guide/services/) come in. Services provide stable endpoints for Pods.



Services use labels to determine what Pods they operate on. If Pods have the correct labels, they are automatically picked up and exposed by our services.

The level of access a service provides to a set of pods depends on the Service's type. Currently there are three types:

* ClusterIP (internal) -- the default type means that this Service is only visible inside of the cluster,
* NodePort gives each node in the cluster an externally accessible IP and
* LoadBalancer adds a load balancer from the cloud provider which forwards traffic from the service to Nodes within it.

Now you'll learn how to:

* Create a service
* Use label selectors to expose a limited set of Pods externally

## Task5-1: Create a Service

Before you can create our services, first create a secure pod that can handle https traffic.

1. Explore the monolith service configuration file:

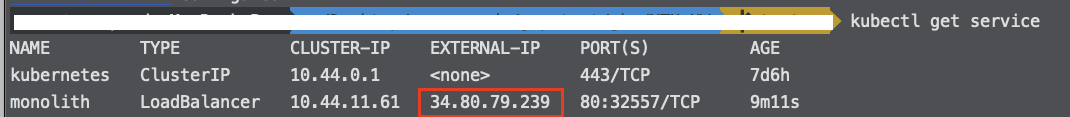
| kubectl create -f service.yaml |
| --- |

1. Examine your service. Use the kubectl get service command to list all pods running in the default namespace:

| kubectl get service |
| --- |

**Note:** It may take a few seconds before the monolith service is up and running. The monolith container image needs to be pulled from the Docker Hub before you can run it.

(output)



1. When the service is ready, you will see the monolith service’s `EXTERNAL-IP` is not none. Copy the EXTERNAL-IP and paste it in your browser and see what happens.

# Final Task: Deleting the cluster

1. To **delete** the cluster, run the following command:

| gcloud container clusters delete [CLUSTER-NAME] |
| --- |

1. When prompted, type **Y** to confirm.

Deleting the cluster can take a few minutes. For more information on deleted GKE clusters, view the [documentation](https://cloud.google.com/kubernetes-engine/docs/how-to/deleting-a-cluster).

**Installing Anthos Service Mesh on GKE**

# Before Task:

1. In Cloud Shell, verify your default **account** is configured.
2. Update **project** configuration if needed.

If the SDK does not have the default **project** set correctly, update the configuration. Replace [project\_id] with the name of the project provided in the **credential** section of the Qwiklabs instructions page.

| gcloud config set project [project\_id] |
| --- |

1. Enable APIs

In Cloud Shell, enable the APIs required to use GKE and Anthos Service Mesh.

| gcloud services enable \  container.googleapis.com \  compute.googleapis.com \  monitoring.googleapis.com \  logging.googleapis.com \  cloudtrace.googleapis.com \  meshca.googleapis.com \  meshtelemetry.googleapis.com \  meshconfig.googleapis.com \  iamcredentials.googleapis.com \  anthos.googleapis.com \  gkeconnect.googleapis.com \  gkehub.googleapis.com \  cloudresourcemanager.googleapis.com |
| --- |

1. Configure environment variables

Configure environment variables that will be used in the setup and installation commands.

| export PROJECT\_ID=$(gcloud config get-value project)  export PROJECT\_NUMBER=$(gcloud projects describe ${PROJECT\_ID} \  --format="value(projectNumber)")  export CLUSTER\_NAME=central  export CLUSTER\_ZONE=us-central1-b  export WORKLOAD\_POOL=${PROJECT\_ID}.svc.id.goog  export MESH\_ID="proj-${PROJECT\_NUMBER}" |
| --- |

**WORKLOAD\_POOL** will be used to enable Workload Identity, which is the recommended way to safely access Google Cloud services from GKE applications.

**MESH\_ID** will be used to set the mesh\_id label on the cluster, which is required for metrics to get displayed on the Anthos Service Mesh Dashboard in the Cloud Console.

1. Verify sufficient permissions

In Cloud Shell, verify that your user account has the **Owner** role assigned.

| gcloud projects get-iam-policy $PROJECT\_ID \  --flatten="bindings[].members" \  --filter="bindings.members:user:$(gcloud config get-value core/account 2>/dev/null)" |
| --- |

**Note:** you also possess **viewer** privileges.

To complete setup, you need the permissions associated with these roles:

* Project Editor
* Kubernetes Engine Admin
* Project IAM Admin
* GKE Hub Admin
* Service Account Admin
* Service Account key Admin

If you have the Owner role, you have all these permissions and more, so you're ready to proceed.

# Task1: Set up your GKE cluster

1. Create the cluster

Now run the following command in Cloud Shell to create the Kubernetes cluster central:

| gcloud config set compute/zone ${CLUSTER\_ZONE}  gcloud beta container clusters create ${CLUSTER\_NAME} \  --machine-type=n1-standard-4 \  --num-nodes=4 \  --workload-pool=${WORKLOAD\_POOL} \  --enable-stackdriver-kubernetes \  --subnetwork=default \  --release-channel=regular \  --labels mesh\_id=${MESH\_ID} |
| --- |

It will take several minutes for cluster creation to complete.

1. Register the cluster

Connect for Anthos allows you to register any of your Kubernetes clusters to Google Cloud, even clusters running on-premises or on other cloud providers. This enables access to cluster and workload management features in Cloud Console, including a unified user interface to interact with your clusters.

* 1. In Cloud Shell, check that you have the **cluster-admin** RBAC role.

| kubectl auth can-i '\*' '\*' --all-namespaces |
| --- |

You should have this role because you created the cluster.

If you need cluster-admin, create the clusterrolebinding, replace the variables with appropriate values.

| kubectl create clusterrolebinding [BINDING\_NAME] \  --clusterrole cluster-admin --user [USER] |
| --- |

* 1. Create a service account for use by GKE Connect.

| gcloud iam service-accounts create connect-sa |
| --- |

* 1. Assign the **gkehub.connect** role to the newly created service account.

| gcloud projects add-iam-policy-binding ${PROJECT\_ID} \  --member="serviceAccount:connect-sa@${PROJECT\_ID}.iam.gserviceaccount.com" \  --role="roles/gkehub.connect" |
| --- |

* 1. Create and download a service account key file.

| gcloud iam service-accounts keys create connect-sa-key.json \  --iam-account=connect-sa@${PROJECT\_ID}.iam.gserviceaccount.com |
| --- |

* 1. Use the key file to register the cluster.

| gcloud container hub memberships register ${CLUSTER\_NAME}-connect \  --gke-cluster=${CLUSTER\_ZONE}/${CLUSTER\_NAME} \  --service-account-key-file=./connect-sa-key.json |
| --- |

As part of registering your cluster, the Connect Agent is deployed to your cluster.

# Task2: Prepare to install Anthos Service Mesh

1. Initialize your project for service mesh installation.

| curl --request POST \  --header "Authorization: Bearer $(gcloud auth print-access-token)" \  --data '' \  https://meshconfig.googleapis.com/v1alpha1/projects/${PROJECT\_ID}:initialize |
| --- |

This command creates a service account for Istio components, among other things.

The command responds with empty curly braces: {}.

1. Download the installation file.

| curl -LO https://storage.googleapis.com/gke-release/asm/istio-1.6.11-asm.1-linux-amd64.tar.gz |
| --- |

1. Download the signature file and verify the signature.

| curl -LO https://storage.googleapis.com/gke-release/asm/istio-1.6.11-asm.1-linux-amd64.tar.gz.1.sig  openssl dgst -verify /dev/stdin -signature istio-1.6.11-asm.1-linux-amd64.tar.gz.1.sig istio-1.6.11-asm.1-linux-amd64.tar.gz <<'EOF'  -----BEGIN PUBLIC KEY-----  MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEWZrGCUaJJr1H8a36sG4UUoXvlXvZ  wQfk16sxprI2gOJ2vFFggdq3ixF2h4qNBt0kI7ciDhgpwS8t+/960IsIgw==  -----END PUBLIC KEY-----  EOF |
| --- |

1. Extract the content of the file.

| tar xzf istio-1.6.11-asm.1-linux-amd64.tar.gz |
| --- |

1. Change directories into the installation directory.

| cd istio-1.6.11-asm.1 |
| --- |

1. For convenience, add the tools in the **/bin** directory to your PATH.

| export PATH=$PWD/bin:$PATH |
| --- |

You can now use the **istioctl** and **asmctl** tools.

# Task3: Preparing Resource Configuration Files

When running the istioctl install command, you specify -f istio-operator.yaml on the command line. This file contains information about your project and cluster that is needed to enable the Mesh telemetry and Mesh security features. You need to download the istio-operator.yaml and other resource configuration files and set the project and cluster information.

1. Install **kpt**

| sudo apt-get install google-cloud-sdk-kpt |
| --- |

1. Create a new dierectory for the Anthos Service Mesh package resource configuration files.

| mkdir ${CLUSTER\_NAME}  cd ${CLUSTER\_NAME} |
| --- |

1. Download the Anths Service Mesh package.

| kpt pkg get https://github.com/GoogleCloudPlatform/anthos-service-mesh-packages@1.6.8-asm.9 asm |
| --- |

1. Go to the newly created asm directory

| cd asm |
| --- |

1. Use kpt to customize installation settings in the configuration files:

| kpt cfg set asm gcloud.container.cluster ${CLUSTER\_NAME}  kpt cfg set asm gcloud.project.environProjectNumber ${PROJECT\_NUMBER}  kpt cfg set asm gcloud.core.project ${PROJECT\_ID}  kpt cfg set asm gcloud.compute.location ${CLUSTER\_ZONE}  kpt cfg set asm anthos.servicemesh.profile asm-gcp  k |
| --- |

# Task4: Install Anthos Service Mesh

Auto mutual TLS (auto mTLS) is enabled by default. With auto mTLS, a client sidecar proxy automatically detects if the server has a sidecar. The client sidecar sends mTLS to workloads with sidecars and sends plain text traffic to workloads without sidecars..

1. Use istioctl to install ASM.

| istioctl install -f asm/cluster/istio-operator.yaml |
| --- |

1. Use **kubectl** to check on completion of the installation.

| kubectl wait --for=condition=available --timeout=600s deployment \  --all -n istio-system |
| --- |

It can take several minutes for installation to complete.

1. When all conditions are met, check that the control plane Pods are **running**.

| kubectl get pod -n istio-system |
| --- |

1. Validate the installation with the following command.

| asmctl validate |
| --- |

You can use the **asmctl** analysis tool to help validate your setup. **asmctl** comes with your Anthos Service Mesh installation and is located in the bin directory of your installation.

**asmctl** checks the correctness of basic configurations in your project, cluster, and workloads, and if possible, recommends solutions. **asmctl** can also verify that certificates were issued by Mesh CA. **asmctl** validates the following:

* Check the enabled APIs on the project for APIs that are required by Anthos Service Mesh.
* That the Istio-Ingressgateway is properly configured to call Mesh CA
* The general health of Istiod and Istio-Ingressgateway.

You can run **asmctl** to validate mTLS communication as well as the other validation checks. To test this, **asmctl** deploys workloads on your cluster in a test namespace, runs the mTLS communication tests, outputs the results, and deletes the test namespace.

# Task5: Enable sidecar injection

Anthos Service Mesh uses sidecar proxies to enhance network security, reliability, and observability. With Anthos Service Mesh, these functions are abstracted away from an application's primary container and implemented in a common out-of-process proxy delivered as a separate container in the same Pod.

Before you deploy workloads, make sure to configure sidecar proxy injection so that Anthos Service Mesh can monitor and secure traffic.

| kubectl label namespace default istio-injection=enabled --overwrite |
| --- |

**Note**: Sidecar proxy injection is now enabled for future workloads.

# Task6: Deploy Bookinfo, an Istio-enabled multi-service application

Bookinfo detail see [here](#_2miaroe6c3gy). We now use **kubectl** to deploy the application, not **istioctl**.

1. Look at the .yaml which describes the bookInfo application:

| cd ~/istio-1.6.11-asm.1  cat samples/bookinfo/platform/kube/bookinfo.yaml |
| --- |

Look for containers to see that each Deployment, has **one** container, for each version of each service in the Bookinfo application.

1. In **Cloud Shell**, use the following command to inject the proxy sidecar along with each application Pod that is deployed.

| kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml |
| --- |

**NOTE**: Istio uses an extended version of the open-source [Envoy proxy](https://www.envoyproxy.io/envoy/), a high-performance proxy developed in C++, to mediate all inbound and outbound traffic for all services in the service mesh.

**NOTE**: Istio leverages Envoy's many built-in features including dynamic service discovery, load balancing, TLS termination, HTTP/2 & gRPC proxying, circuit breakers, health checks, staged rollouts with %-based traffic split, fault injection, and rich metrics.

# Task7: Enable external access using an Istio Ingress Gateway

Now that the Bookinfo services are up and running, you need to make the application accessible from outside of your Kubernetes cluster, e.g. from a browser. An **Istio Gateway** is used for this purpose.

1. Configure the **ingress gateway** for the application, which exposes an *external IP* you will use later:

| kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml |
| --- |

You can look at the .yaml which describes the configuration for the application ingress gateway.

Look for the Gateway and VirtualService mesh resources which get deployed. The Gateway exposes services to users outside the service mesh, and allows Istio features such as monitoring and route rules to be applied to traffic entering the cluster.

# Task8: Verify the Bookinfo deployments

1. Confirm that the application has been deployed correctly, review services, pods, and the ingress gateway:

| kubectl get services |
| --- |

1. Review running application pods:

| kubectl get pods |
| --- |

You may need to re-run this command until you see that all six pods are in **Running** status.

**Note:** See how each pod has two READY containers? That's the application container and the Istio proxy sidecar.

1. Confirm that the Bookinfo application is running by sending a curl request to it from some pod, within the cluster, for example from ratings:

| kubectl exec -it $(kubectl get pod -l app=ratings -o jsonpath='{.items[0].metadata.name}') \  -c ratings -- curl productpage:9080/productpage | grep -o "<title>.\*</title>" |
| --- |

1. Confirm the ingress gateway has been created:

| kubectl get gateway |
| --- |

1. Get the **external IP** address of the **ingress gateway**:

| kubectl get svc istio-ingressgateway -n istio-system |
| --- |

1. Now run the following command, replacing [EXTERNAL-IP] with the external IP that was outputted from the previous command:

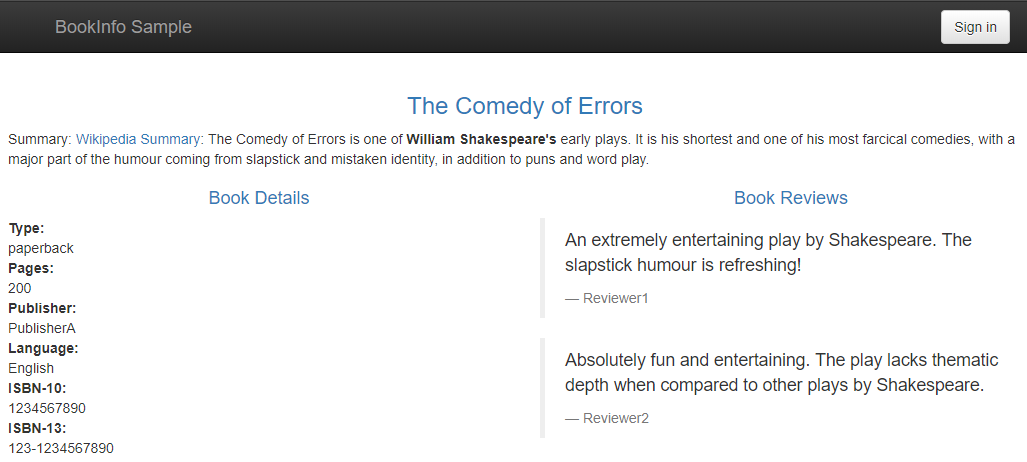
| export GATEWAY\_URL=[EXTERNAL-IP]  curl -I http://${GATEWAY\_URL}/productpage |
| --- |

# 

# Task9: Use the Bookinfo application

Try the application in your web browser

1. Point your browser to http://[$GATEWAY\_URL]/productpage to see the BookInfo web page. Don't forget to replace [$GATEWAY\_URL] with your working external IP address.



1. Refresh the page several times.  
   Notice how you see three different versions of reviews, since we have not yet used Istio to control the version routing.  
   There are three different book review services being called in a round-robin style:
   * No stars
   * Black stars
   * Red stars

Switching among the three is normal Kubernetes routing/balancing behavior.

# Task10: Generate a steady background load

Run the siege utility to simulate traffic to Bookinfo.

1. In Cloud Shell, install **siege**.

Siege is a utility for generating load against Web sites.

| sudo apt install siege |
| --- |

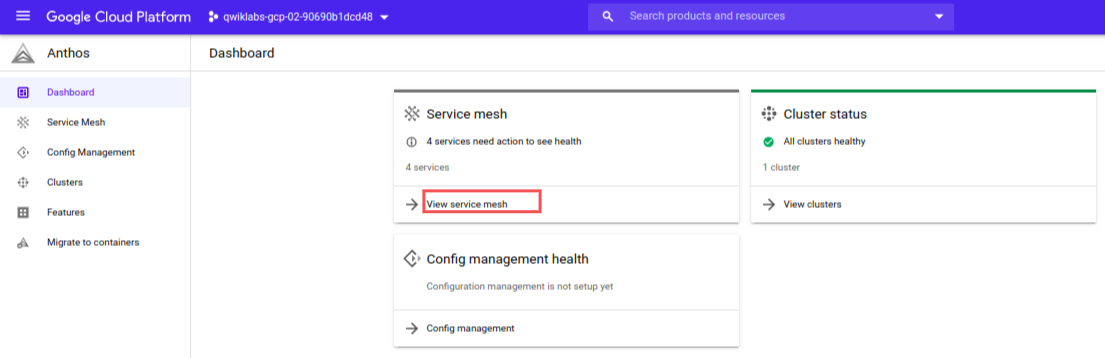
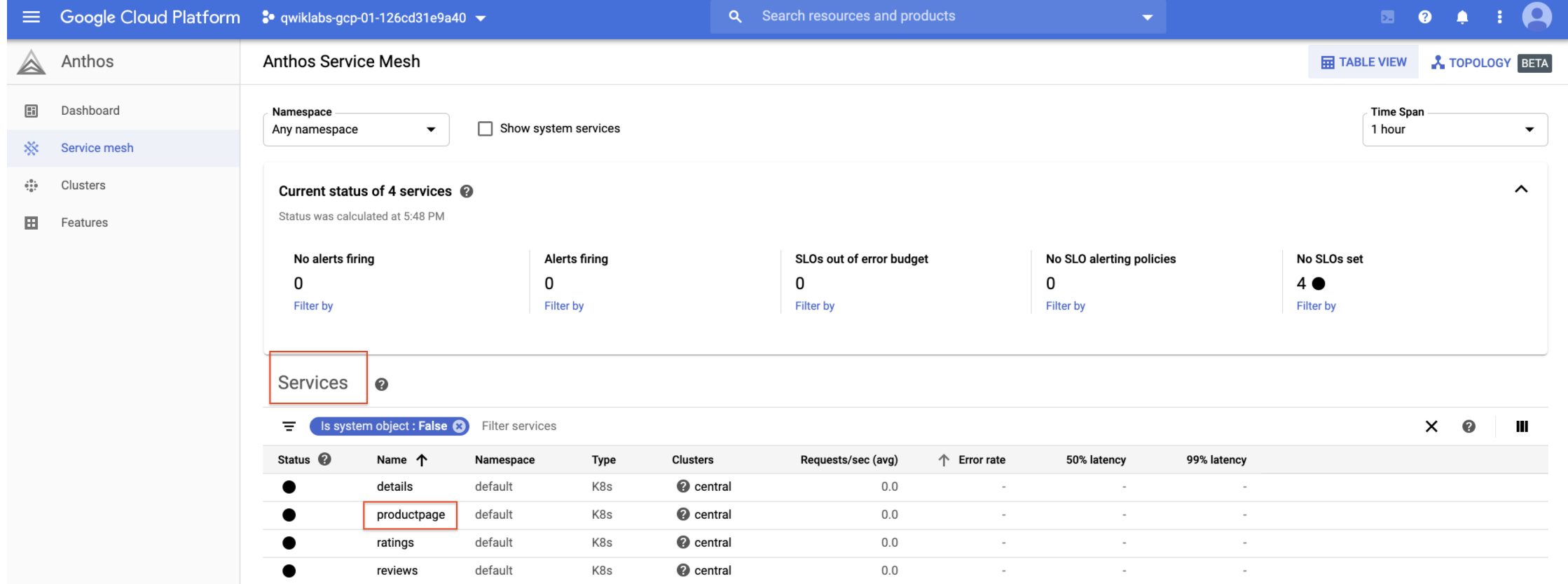
1. Use **siege** to create traffic against your services.

| siege http://${GATEWAY\_URL}/productpage |
| --- |

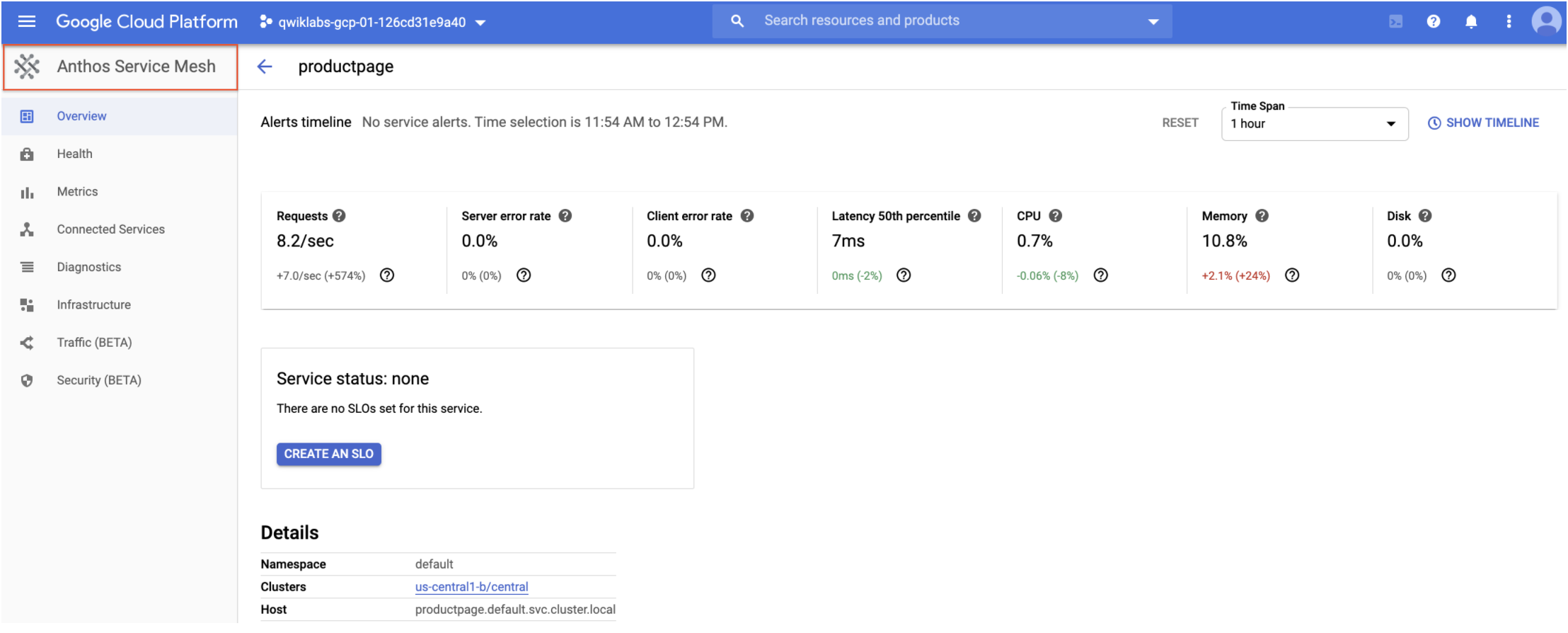
# 

# Task11: Evaluate service performance using the Anthos Service Mesh dashboard

Gather data from the services table view.

1. In the Console, go to **Navigation menu** > **Anthos** > **Dashboard**.
2. Click **View Service Mesh**.  
   
3. On the bottom half of the window, you will see a **Service** section.  
   How many services are listed?
4. Click on the **productpage** service to drill down and see more details.  
     
   Note the summary at the top detailing current requests/second, error rates, latencies, and resource usage.  
   If you don't see **Requests > 0**, try exiting and re-entering the productpage service after a few minutes.
5. On the left side of the window, click on the **Metrics** option.
   * What is the current request rate, and how has it changed over time?
   * What is the current error rate, and how has it changed over time?
   * What latencies do you see charted?
   * What is the median request size?
   * What is median response size?
   * What is the aggregate cpu usage?
6. Click on the **Connected Services** option from the left side.  
   This lists other services that make inbound requests of the productpage, and services the productpage makes outbound requests to.
   * What services make calls to the **productpage** service?
   * What services does the **productpage** service call?
   * Is mTLS enforced on inter-service calls?

**Note:** the **ratings** service does not normally send requests to the productpage service. The ratings service is shown here because you used the ratings service to issue a test request as part of this lab.

1. Return to the Anthos Service Mesh dashboard by clicking on the **Anthos Service Mesh** logo in the upper left corner.  
   

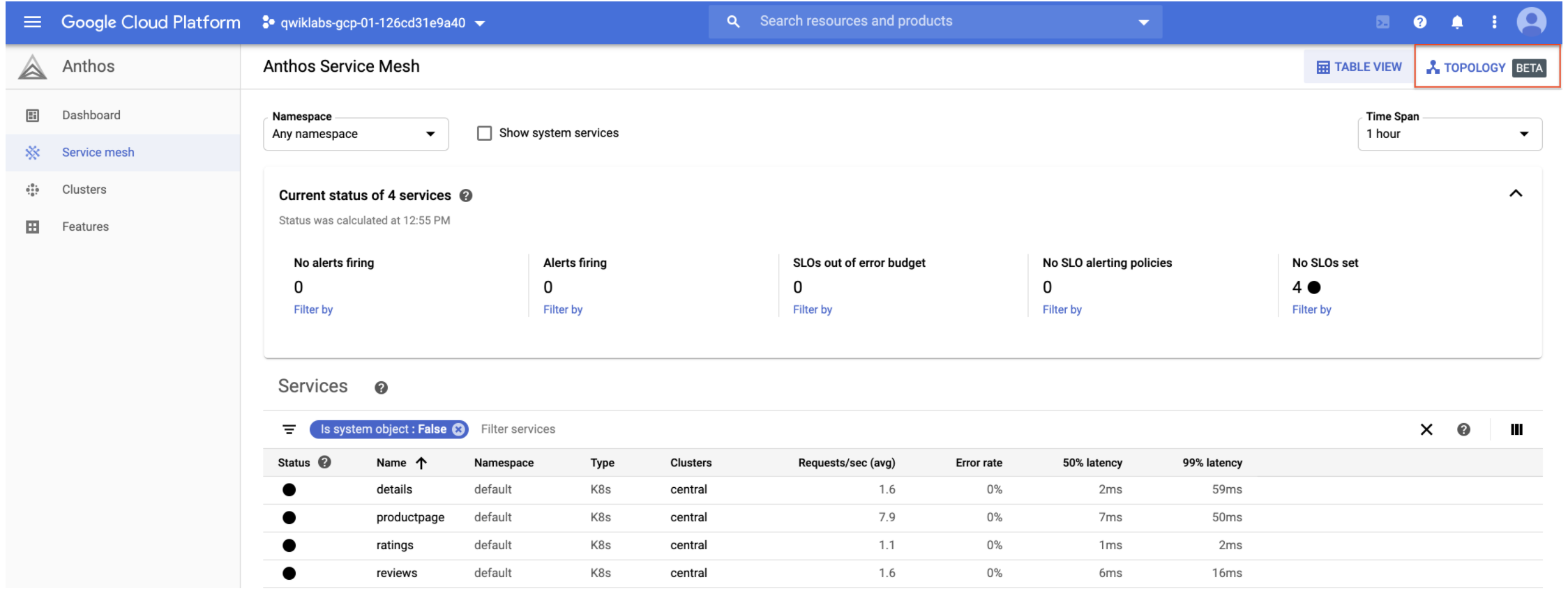
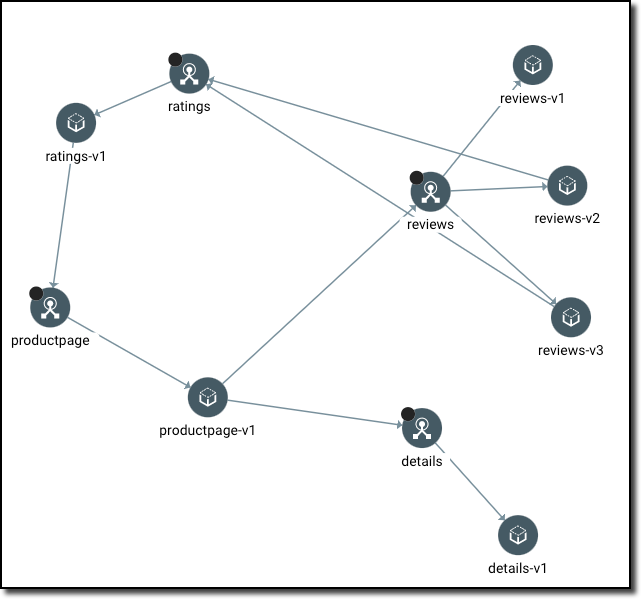
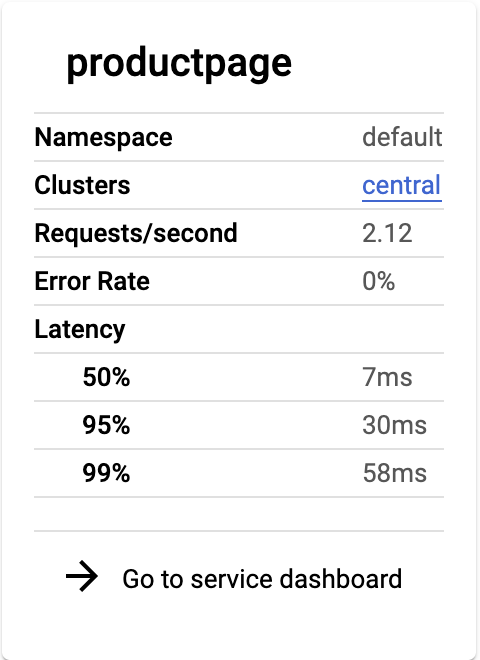
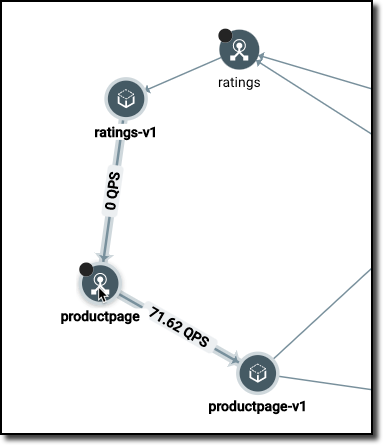
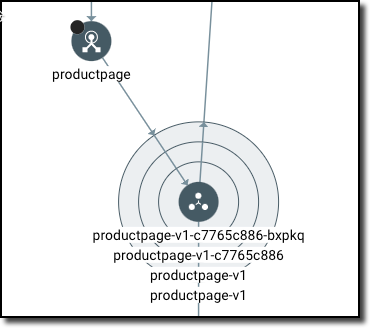
At this time, you can explore or drill down on other services.

**Note:** The top section of the dashboard shows information about Service Level Objectives (SLOs) and Alerts.

SLOs are targets for Service Level Indicators (SLIs); they embody your definition of how well a service should perform. An example SLO would be 99.9% of hourly requests return a 200 response. You might define an alerting policy that pages on-call staff when your service is failing to meet its SLOs

Look for other labs where you can define and test SLOs!

# Task12: Use the Topology view to better visualize your mesh

1. On the Anthos Service Mesh dashboard, click on the **TOPOLOGY** link in the upper-right corner.  
     
   Here you may need to wait few minutes for the topology graph to appear.
2. Rearrange the nodes in the graph until you can easily visualize the relationships between services and workloads.  
   Remember, external requests start at productpage. You can scroll back and study the Bookinfo architecture diagram at the **Bookinfo Overview**.  
   Your drawing might look something like this:  
   
3. Click on the **productpage** service node.  
   You should see a service details card:  
   
   * How many requests per second is this service receiving?
   * What are the median and 99% latencies?
4. Now, hover over each of the service nodes and notice edge stats.
   * How many requests/second is the \*\*productpage\*\* service receiving, and from where?
   * How many requests per second is the service forwarding to the \*\*productpage-v1\*\* workload?
5. You should see traffic details like this:  
   
6. Drill down on one of the workloads until you can see the deployment, the replica set, and the Pods.  
   It should look something like this:  
   

# Final Task: Deleting the cluster

1. To **delete** the cluster, run the following command:

| gcloud container clusters delete [CLUSTER-NAME] |
| --- |

1. When prompted, type **Y** to confirm.

Deleting the cluster can take a few minutes. For more information on deleted GKE clusters, view the [documentation](https://cloud.google.com/kubernetes-engine/docs/how-to/deleting-a-cluster).

**Implementing Cloud Run canary deployments with Git branches and Cloud Build**

# Before Task:

1. In Cloud Shell, create environment variables:

| export PROJECT\_ID=$(gcloud config get-value project)  export PROJECT\_NUMBER=$(gcloud projects describe $PROJECT\_ID --format='value(projectNumber)') |
| --- |

1. Enable the following APIs:
   1. Resource Manager
   2. GKE
   3. Cloud Source Repositories
   4. Cloud Build
   5. Container Registry
   6. Cloud Run

| gcloud services enable \  cloudresourcemanager.googleapis.com \  container.googleapis.com \  sourcerepo.googleapis.com \  cloudbuild.googleapis.com \  containerregistry.googleapis.com \  run.googleapis.com |
| --- |

1. Grant the Cloud Run Admin role (**roles/run.admin**) to the Cloud Build service account:

| gcloud projects add-iam-policy-binding $PROJECT\_ID \  --member=serviceAccount:$PROJECT\_NUMBER@cloudbuild.gserviceaccount.com \  --role=roles/run.admin |
| --- |

1. Grant the IAM Service Account User role (**roles/iam.serviceAccountUser**) to the Cloud Build service account for the Cloud Run runtime service account:

| gcloud iam service-accounts add-iam-policy-binding \  $PROJECT\_NUMBER-compute@developer.gserviceaccount.com \  --member=serviceAccount:$PROJECT\_NUMBER@cloudbuild.gserviceaccount.com \  --role=roles/iam.serviceAccountUser |
| --- |

1. If you haven't used Git in Cloud Shell previously, set the **user.name** and **user.email** values that you want to use:

| git config --global user.email "YOUR\_EMAIL\_ADDRESS"  git config --global user.name "YOUR\_USERNAME" |
| --- |

1. Clone and prepare the sample repository:

| git clone https://github.com/GoogleCloudPlatform/software-delivery-workshop cloudrun-progression  cd cloudrun-progression/labs/cloudrun-progression  rm -rf ../../.git |
| --- |

1. Replace placeholder values in the sample repository with your **PROJECT\_ID**:

| sed "s/PROJECT/${PROJECT\_ID}/g" branch-trigger.json-tmpl > branch-trigger.json  sed "s/PROJECT/${PROJECT\_ID}/g" master-trigger.json-tmpl > master-trigger.json  sed "s/PROJECT/${PROJECT\_ID}/g" tag-trigger.json-tmpl > tag-trigger.json |
| --- |

1. Store the code from the sample repository in Cloud Source Repositories:

| gcloud source repos create cloudrun-progression  git init  git config credential.helper gcloud.sh  git remote add gcp https://source.developers.google.com/p/$PROJECT\_ID/r/cloudrun-progression  git branch -m master  git add . && git commit -m "initial commit"  git push gcp master |
| --- |

# Task1: Creating your Cloud Run service

In this section, you build and deploy the initial production application that you use throughout this tutorial.

1. In Cloud Shell, build and deploy the application, including a service that requires authentication. To make a public service, use the **--allow-unauthenticated** flag as described in [Allowing public (unauthenticated) access](https://cloud.google.com/run/docs/authenticating/public).

| gcloud builds submit --tag gcr.io/$PROJECT\_ID/hello-cloudrun  gcloud run deploy hello-cloudrun \  --image gcr.io/$PROJECT\_ID/hello-cloudrun \  --platform managed \  --region us-central1 \  --allow-unauthenticated  --tag=prod -q |
| --- |

The output looks like the following:

| Deploying container to Cloud Run service [hello-cloudrun] in project [sdw-mvp6] region [us-central1]  ✓ Deploying new service... Done.  ✓ Creating Revision...  ✓ Routing traffic...  Done.  Service [hello-cloudrun] revision [hello-cloudrun-00001-tar] has been deployed and is serving 100 percent of traffic.  Service URL: https://hello-cloudrun-apwaaxltma-uc.a.run.app  The revision can be reached directly at https://prod---hello-cloudrun-apwaaxltma-uc.a.run.app |
| --- |

The output includes the service URL and a unique URL for the revision. Your values will differ slightly from what's indicated here.

1. After the deployment is complete, view the newly deployed service on the [Revisions page](https://console.cloud.google.com/run/detail/us-central1/hello-cloudrun/revisions) in the Cloud Console.
2. In Cloud Shell, view the authenticated service response:

| PROD\_URL=$(gcloud run services describe hello-cloudrun \  --platform managed \  --region us-central1 \  --format=json | jq \  --raw-output ".status.url")  echo $PROD\_URL  curl -H "Authorization: Bearer $(gcloud auth print-identity-token)" $PROD\_URL |
| --- |

# Task2: Enabling dynamic developer deployments

In this section, you enable a unique URL for development branches in Git. Each branch is represented by a URL that's identified by the branch name. Commits to the branch trigger a deployment, and the updates are accessible at that same URL.

1. In Cloud Shell, set up the trigger:

| gcloud beta builds triggers create cloud-source-repositories \  --trigger-config branch-trigger.json |
| --- |

1. To review the trigger, go to the [Cloud Build Triggers page](https://console.cloud.google.com/cloud-build/triggers) in the Cloud Console.
2. In Cloud Shell, create a new branch:

| git checkout -b new-feature-1 |
| --- |

1. Open the sample application in the Cloud Shell Editor:

| edit app.py |
| --- |

1. In the sample application, modify the code to indicate v1.1 instead of v1.0:

| @app.route('/')  def hello\_world():  return 'Hello World v1.1' |
| --- |

1. To return to your terminal, click **Open Terminal**.
2. In Cloud Shell, commit the change and push to the remote repository:

| git add . && git commit -m "updated" && git push gcp new-feature-1 |
| --- |

1. To review the build in progress, go to the [Cloud Build Builds page](https://console.cloud.google.com/cloud-build/builds) in the Cloud Console.
2. After the build completes, to review the new revision, go to the [Cloud Run Revisions page](https://console.cloud.google.com/run/detail/us-central1/hello-cloudrun/revisions) in the Cloud Console.
3. In Cloud Shell, get the unique URL for this branch:

| BRANCH\_URL=$(gcloud run services describe hello-cloudrun \  --platform managed \  --region us-central1 \  --format=json | jq \  --raw-output ".status.traffic[] | select (.tag==\"new-feature-1\")|.url")  echo $BRANCH\_URL |
| --- |

1. Access the authenticated URL:

| curl -H "Authorization: Bearer $(gcloud auth print-identity-token)" $BRANCH\_URL |
| --- |

The updated response output looks like the following:

| Hello World v1.1 |
| --- |

# Task3: Automating canary testing

When code is released to production, it's common to release code to a small subset of live traffic before migrating all traffic to the new code base.

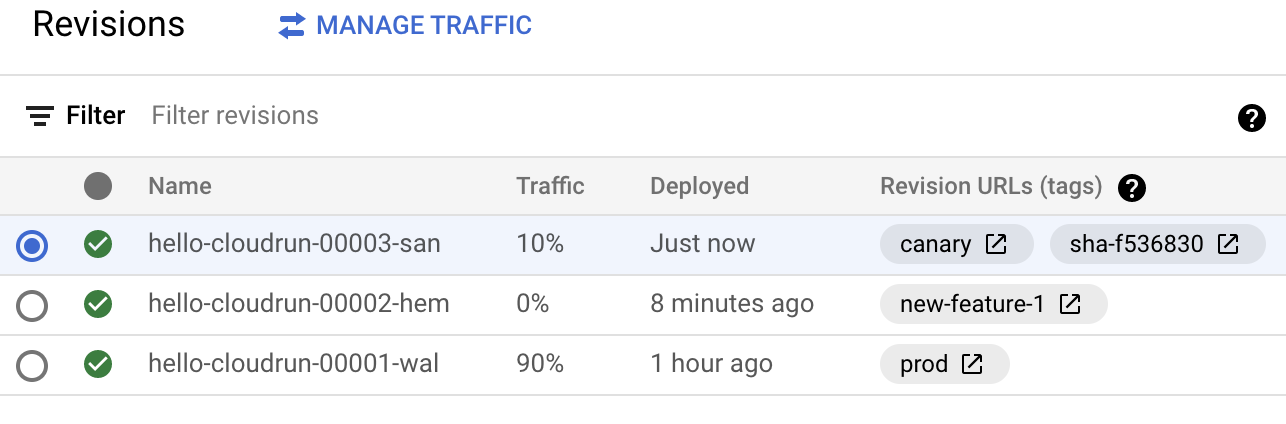
In this section, you implement a trigger that is activated when code is committed to the main branch. The trigger deploys the code to a unique canary URL and it routes 10% of all live traffic to the new revision.

1. In Cloud Shell, set up the branch trigger:

| gcloud beta builds triggers create cloud-source-repositories \  --trigger-config master-trigger.json |
| --- |

1. To review the new trigger, go to the [Cloud Build Triggers page](https://console.cloud.google.com/cloud-build/triggers) in the Cloud Console.
2. In Cloud Shell, merge the branch to the main line and push to the remote repository:

| gcloud beta builds triggers create cloud-source-repositories \  --trigger-config master-trigger.json |
| --- |

1. To review the build in progress, go to the [Cloud Build Builds page](https://console.cloud.google.com/cloud-build/builds) in the Cloud Console.
2. After the build is complete, to review the new revision, go to the [Cloud Run Revisions page](https://console.cloud.google.com/run/detail/us-central1/hello-cloudrun/revisions) in the Cloud Console.As the following screenshot shows, 90% of the traffic is routed to **prod**, 10% to **canary**, and 0% to the branch revisions.  
   
3. Review the lines of **master-cloudbuild.yaml** that implement the logic for the canary deployment.  
   The following lines deploy the new revision and use the **tag** flag to route traffic from the unique canary URL:

| gcloud run deploy ${\_SERVICE\_NAME} \  --platform managed \  --region ${\_REGION} \  --image gcr.io/${PROJECT\_ID}/${\_SERVICE\_NAME} \  --tag=canary \  --no-traffic |
| --- |

The following line adds a static tag to the revision that notes the Git short SHA of the deployment:

| gcloud beta run services update-traffic ${\_SERVICE\_NAME} --update-tags=sha-$SHORT\_SHA=$${CANARY} --platform managed --region ${\_REGION} |
| --- |

The following line updates the traffic to route 90% to production and 10% to canary:

| gcloud run services update-traffic ${\_SERVICE\_NAME} --to-revisions=$${PROD}=90,$${CANARY}=10 --platform managed --region ${\_REGION} |
| --- |

1. In Cloud Shell, get the unique URL for the canary revision:

| CANARY\_URL=$(gcloud run services describe hello-cloudrun \  --platform managed \  --region us-central1 \  --format=json | jq \  --raw-output ".status.traffic[] | select (.tag==\"canary\")|.url")  echo $CANARY\_URL |
| --- |

1. Review the canary endpoint directly:

| curl -H "Authorization: Bearer $(gcloud auth print-identity-token)" $CANARY\_URL |
| --- |

1. To see percentage-based responses, make a series of requests:

| LIVE\_URL=$(gcloud run services describe hello-cloudrun \  --platform managed \  --region us-central1 \  --format=json | jq \  --raw-output ".status.url")  for i in {0..20};do  curl -H "Authorization: Bearer $(gcloud auth print-identity-token)" $LIVE\_URL; echo \n  done |
| --- |

# Task4: Releasing to production

After the canary deployment is validated with a small subset of traffic, you release the deployment to the remainder of the live traffic.

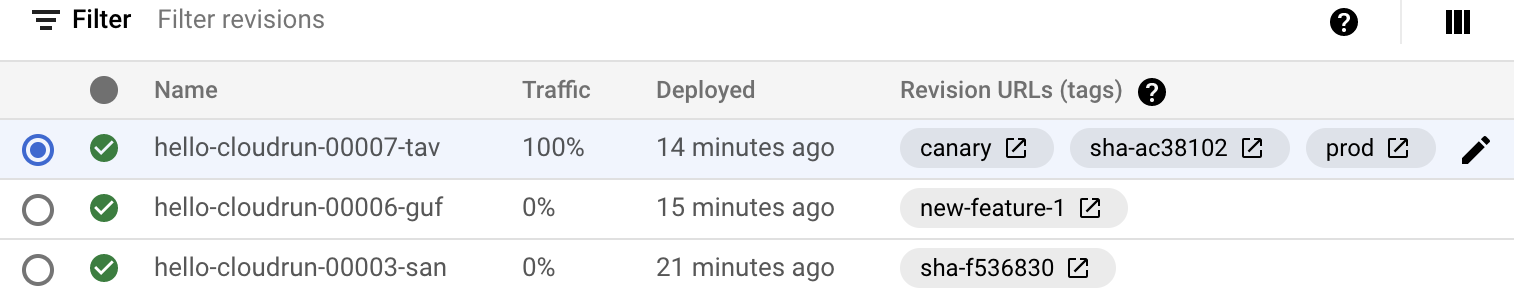
In this section, you set up a trigger that is activated when you create a tag in the repository. The trigger migrates 100% of traffic to the already deployed revision based on the commit SHA of the tag. Using the commit SHA ensures the revision validated with canary traffic is the revision used for the remainder of production traffic.

1. In Cloud Shell, set up the tag trigger:

| gcloud beta builds triggers create cloud-source-repositories \  --trigger-config tag-trigger.json |
| --- |

1. To review the new trigger, go to the [Cloud Build Triggers page](https://console.cloud.google.com/cloud-build/triggers) in the Cloud Console.
2. In Cloud Shell, create a new tag and push to the remote repository:

| git tag 1.1  git push gcp 1.1 |
| --- |

1. To review the build in progress, go to the [Cloud Build Builds page](https://console.cloud.google.com/cloud-build/builds) in the Cloud Console.
2. After the build is complete, to review the new revision, go to the [Cloud Run Revisions page](https://console.cloud.google.com/run/detail/us-central1/hello-cloudrun/revisions) in the Cloud Console.As the following screenshot shows, the revision is updated to indicate the **prod** tag and it is serving 100% of live traffic.  
   
3. In Cloud Shell, to see percentage-based responses, make a series of requests:

| LIVE\_URL=$(gcloud run services describe hello-cloudrun \  --platform managed \  --region us-central1 \  --format=json | jq \  --raw-output ".status.url")  for i in {0..20};do  curl -H "Authorization: Bearer $(gcloud auth print-identity-token)" $LIVE\_URL; echo \n  Done |
| --- |

1. Review the lines of **tag-cloudbuild.yaml** that implement the production deployment logic.  
   The following line updates the canary revision adding the **prod** tag. The deployed revision is now tagged for both **prod** and **canary**:

| gcloud beta run services update-traffic ${\_SERVICE\_NAME} --update-tags=prod=$${CANARY} --platform managed --region ${\_REGION} |
| --- |

The following line updates the traffic for the base service URL to route 100% of traffic to the revision tagged as prod:

| gcloud run services update-traffic ${\_SERVICE\_NAME} --to-revisions=$${NEW\_PROD}=100 --platform managed --region ${\_REGION} |
| --- |

# Final Task: Clean Up

You can delete the project that contains the resources, or keep the project and delete the individual resources.

1. **I**n the Cloud Console, go to the **Manage resources** page.  
   [Go to Manage resources](https://console.cloud.google.com/iam-admin/projects)
2. In the project list, select the project that you want to delete, and then click **Delete**.
3. In the dialog, type the project ID, and then click **Shut down** to delete the project.