## Deploy and Manage Cloud Environments with Google Cloud

* Learn how to: 1. Deploy and manage a containerized application using Google Kubernetes Engine(GKE) and kubectl. 2. Launch a VM-based application using Cloud Deployment Manager, and then monitor and stress the application. 3. Deploy continuous delivery pipelines using GKE and Spinnaker. 4. Create and control access to multiple VPC networks. 5. Configure and use Cloud Monitoring to troubleshoot breaks in applications.
* 1.Orchestrating the Cloud with Kubernetes
  + In this lab you will learn how to provision a complete Kubernetes cluster using Google Container Engine; deploy and manage Docker containers using kubectl; and break an application into microservices using Kubernetes’ Deployments and Services.
* 2.Continuous Delivery Pipelines with Spinnaker and Kubernetes Engine
  + Create a Kubernetes Engine cluster, deploy an application, and use Spinnaker to continuously deploy the application when changes are made to the application.
* Multiple VPC Networks
  + In this lab, you create several VPC networks and VM instances and test connectivity across networks.
* 3.Troubleshooting Workloads on GKE for Site Reliability Engineers
  + The Cloud Operations Sandbox is intended to make it easy for you to deploy and run a non-trivial application that lets you explore the Google Cloud Platform services, particularly the Cloud Operations (formerly Stackdriver) product suite.
* 4.Deploy and Manage Cloud Environments with Google Cloud: Challenge Lab

## 1.Orchestrating the Cloud with Kubernetes - https://www.cloudskillsboost.google/focuses/557?parent=catalog

* In this lab you will learn how to:
  + Provision a complete [Kubernetes](http://kubernetes.io/) cluster using [Kubernetes Engine](https://cloud.google.com/container-engine).
  + Deploy and manage Docker containers using kubectl.
  + Break an application into microservices using Kubernetes' Deployments and Services.
* Kubernetes is all about applications. In this part of the lab you will use an example application called "app".
  + [App](https://github.com/kelseyhightower/app) is hosted on GitHub and provides an example 12-Factor application. During this lab you will be working with the following Docker images:
    - [kelseyhightower/monolith](https://hub.docker.com/r/kelseyhightower/monolith) - Monolith includes auth and hello services.
    - [kelseyhightower/auth](https://hub.docker.com/r/kelseyhightower/auth) - Auth microservice. Generates JWT tokens for authenticated users.
    - [kelseyhightower/hello](https://hub.docker.com/r/kelseyhightower/hello) - Hello microservice. Greets authenticated users.
    - [nginx](https://hub.docker.com/_/nginx) - Frontend to the auth and hello services.
* Kubernetes is an open source project (available on [kubernetes.io](http://kubernetes.io/)) which can run on many different environments, from laptops to high-availability multi-node clusters, from public clouds to on-premise deployments, from virtual machines to bare metal.
* For this lab, using a managed environment such as Kubernetes Engine allows you to focus on experiencing Kubernetes rather than setting up the underlying infrastructure.

### **Google Kubernetes Engine**

* + In the cloud shell environment type the following command to set the zone:
    - gcloud config set compute/zone us-central1-b
  + After you set the zone, start up a cluster for use in this lab:
    - gcloud container clusters create io
  + Get the sample code
    - Clone the GitHub repository from the Cloud Shell cmd line:
      * gsutil cp -r gs://spls/gsp021/\* .
    - Change into the directory needed for this lab:
      * cd orchestrate-with-kubernetes/kubernetes
    - List the files to see what you're working with:
      * ls
* Quick Kubernetes Demo
  + The easiest way to get started with Kubernetes is to use the kubectl create command. Use it to launch a single instance of the nginx container:
    - kubectl create deployment nginx --image=nginx:1.10.0
  + Kubernetes has created a deployment -- more about deployments later, but for now all you need to know is that deployments keep the pods up and running even when the nodes they run on fail.
  + In Kubernetes, all containers run in a pod. Use the kubectl get pods command to view the running nginx container:
    - kubectl get pods
  + Once the nginx container has a Running status you can expose it outside of Kubernetes using the kubectl expose command:
    - kubectl expose deployment nginx --port 80 --type LoadBalancer
      * So what just happened? Behind the scenes Kubernetes created an external Load Balancer with a public IP address attached to it. Any client who hits that public IP address will be routed to the pods behind the service. In this case that would be the nginx pod.
  + List our services now using the kubectl get services command:
    - kubectl get services
  + Add the External IP to this command to hit the Nginx container remotely:
    - curl http://<External IP>:80
* 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.
    - Graphical user interface, diagram, application

      Description automatically generated
  + 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.
  + Creating Pods
    - Pods can be created using pod configuration files. Take a moment to explore the monolith pod configuration file. Run the following: cat pods/monolith.yaml
      * The output shows the open configuration file:
      * Text

        Description automatically generated
      * There's a few things to notice here. You'll see that:
        + your pod is made up of one container (the monolith).
        + you're passing a few arguments to our container when it starts up.
        + you're opening up port 80 for http traffic.
    - Create the monolith pod using kubectl:
      * kubectl create -f pods/monolith.yaml
    - Examine your pods. Use the kubectl get pods command to list all pods running in the default namespace:
      * kubectl get pods
    - 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!
  + 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. Now you have two terminals, one to run the kubectl port-forward command, and the other to issue curl commands.
    - In the **2nd terminal**, run this command to set up port-forwarding:
      * kubectl port-forward monolith 10080:80
    - Now in the **1st terminal** start talking to your pod using curl:
      * curl http://127.0.0.1:10080
      * Yes! You got a very friendly "hello" back from your container.
    - Now use the curl command to see what happens when you hit a secure endpoint:
      * curl http://127.0.0.1:10080/secure
      * Uh oh.
    - 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.
    - 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 pwd "password" again when prompted for the host password.
    - 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.
    - Use the kubectl logs command to view the logs for the monolith Pod.
      * kubectl logs monolith
    - **Open a 3rd terminal** and use the -f flag to get a stream of the logs happening in real-time:
      * kubectl logs -f monolith
    - 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
    - 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
    - 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.
* Services
  + 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.
    - Diagram

      Description automatically generated
  + 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
  + Creating a Service
    - Before you can create our services, first create a secure pod that can handle https traffic.
    - If you've changed directories, make sure you return to the ~/orchestrate-with-kubernetes/kubernetes directory: cd ~/orchestrate-with-kubernetes/kubernetes
    - Explore the monolith service configuration file: cat pods/secure-monolith.yaml
    - Create the secure-monolith pods and their configuration data:
      * kubectl create secret generic tls-certs --from-file tls/
      * kubectl create configmap nginx-proxy-conf --from-file nginx/proxy.conf
      * kubectl create -f pods/secure-monolith.yaml
    - Now that you have a secure pod, it's time to expose the secure-monolith Pod externally.To do that, create a Kubernetes service.
    - Explore the monolith service configuration file: cat services/monolith.yaml
    - Text

      Description automatically generated
      * Things to note:
        + There's a selector which is used to automatically find and expose any pods with the labels app: monolith and secure: enabled.
        + Now you have to expose the nodeport here because this is how you'll forward external traffic from port 31000 to nginx (on port 443).
    - Use the kubectl create command to create the monolith service from the monolith service configuration file: kubectl create -f services/monolith.yaml
    - You're using a port to expose the service. This means that it's possible to have port collisions if another app tries to bind to port 31000 on one of your servers.
    - Normally, Kubernetes would handle this port assignment. In this lab you chose a port so that it's easier to configure health checks later on.
    - Use the gcloud compute firewall-rules command to allow traffic to the monolith service on the exposed nodeport:
      * gcloud compute firewall-rules create allow-monolith-nodeport --allow=tcp:31000
    - Now that everything is set up you should be able to hit the secure-monolith service from outside the cluster without using port forwarding.
    - First, get an external IP address for one of the nodes.
      * gcloud compute instances list
    - Now try hitting the secure-monolith service using curl:
      * curl -k https://<EXTERNAL\_IP>:31000
        + Uh oh! That timed out. What's going wrong?
    - It's time for a quick knowledge check.Use the following commands to answer the questions below.
      * kubectl get services monolith
      * kubectl describe services monolith
      * **Questions:**
        + Why are you unable to get a response from the monolith service?
        + How many endpoints does the monolith service have?
        + What labels must a Pod have to be picked up by the monolith service?
      * Hint: it has to do with labels. You'll fix the issue in the next section.
  + Adding Labels to Pods
    - Currently the monolith service does not have endpoints. One way to troubleshoot an issue like this is to use the kubectl get pods command with a label query.
    - You can see that you have quite a few pods running with the monolith label.
      * kubectl get pods -l "app=monolith"
    - But what about "app=monolith" and "secure=enabled"?
      * kubectl get pods -l "app=monolith,secure=enabled"
    - Notice this label query does not print any results. It seems like you need to add the "secure=enabled" label to them.
    - Use the kubectl label command to add the missing secure=enabled label to the secure-monolith Pod. Afterwards, you can check and see that your labels have been updated.
      * kubectl label pods secure-monolith 'secure=enabled'
      * kubectl get pods secure-monolith --show-labels
    - Now that your pods are correctly labeled, view the list of endpoints on the monolith service:
      * kubectl describe services monolith | grep Endpoints
    - And you have one!
    - Test this out by hitting one of our nodes again.
      * gcloud compute instances list
      * curl -k https://<EXTERNAL\_IP>:31000
* Deploying Applications with Kubernetes
  + The goal of this lab is to get you ready for scaling and managing containers in production. That's where [Deployments](http://kubernetes.io/docs/user-guide/deployments/#what-is-a-deployment) come in. Deployments are a declarative way to ensure that the number of Pods running is equal to the desired number of Pods, specified by the user.
    - Graphical user interface, diagram

      Description automatically generated
  + The main benefit of Deployments is in abstracting away the low level details of managing Pods. Behind the scenes Deployments use [Replica Sets](http://kubernetes.io/docs/user-guide/replicasets/) to manage starting and stopping the Pods. If Pods need to be updated or scaled, the Deployment will handle that. Deployment also handles restarting Pods if they happen to go down for some reason.
  + Look at a quick example:
    - Graphical user interface, application

      Description automatically generated
  + Pods are tied to the lifetime of the Node they are created on. In the example above, Node3 went down (taking a Pod with it). Instead of manually creating a new Pod and finding a Node for it, your Deployment created a new Pod and started it on Node2.
  + That's pretty cool!
  + It's time to combine everything you learned about Pods and Services to break up the monolith application into smaller Services using Deployments.
  + Creating Deployments
    - You're going to break the monolith app into three separate pieces:
      * **auth** - Generates JWT tokens for authenticated users.
      * **hello** - Greet authenticated users.
      * **frontend** - Routes traffic to the auth and hello services.
    - You are ready to create deployments, one for each service. Afterwards, you'll define internal services for the auth and hello deployments and an external service for the frontend deployment. Once finished you'll be able to interact with the microservices just like with Monolith only now each piece will be able to be scaled and deployed, independently!
    - Get started by examining the auth deployment configuration file.
      * cat deployments/auth.yaml
      * Text

        Description automatically generated
      * The deployment is creating 1 replica, and you're using version 2.0.0 of the auth container.
    - When you run the kubectl create command to create the auth deployment it will make one pod that conforms to the data in the Deployment manifest. This means you can scale the number of Pods by changing the number specified in the Replicas field.
    - Anyway, go ahead and create your deployment object:
      * kubectl create -f deployments/auth.yaml
    - It's time to create a service for your auth deployment. Use the kubectl create command to create the auth service:
      * kubectl create -f services/auth.yaml
    - Now do the same thing to create and expose the hello deployment:
      * kubectl create -f deployments/hello.yaml
      * kubectl create -f services/hello.yaml
    - And one more time to create and expose the frontend Deployment.
      * kubectl create configmap nginx-frontend-conf --from-file=nginx/frontend.conf
      * kubectl create -f deployments/frontend.yaml
      * kubectl create -f services/frontend.yaml
    - There is one more step to creating the frontend because you need to store some configuration data with the container.
    - Interact with the frontend by grabbing its External IP and then curling to it:
      * kubectl get services frontend
      * curl -k https://<EXTERNAL-IP>
        + And you get a hello response back!

## 2.Continuous Delivery Pipelines with Spinnaker and Kubernetes Engine

* **Objectives**
  + This hands-on lab shows you how to create a CD pipeline using Google Kubernetes Engine, Google Cloud Source Repositories, Google Cloud Container Builder, and Spinnaker. After you create a sample application, you configure these services to automatically build, test, and deploy it. When you modify the application code, the changes trigger the continuous delivery pipeline to automatically rebuild, retest, and redeploy the new version.
* **Objectives**
  + Set up your environment by launching [Google Cloud Shell](https://cloud.google.com/shell/), creating a Kubernetes Engine cluster, and configuring your identity and user management scheme.
  + Download a sample application, create a Git repository then upload it to a Google Cloud Source Repository.
  + Deploy Spinnaker to Kubernetes Engine using [Helm](https://github.com/kubernetes/helm).
  + Build your Docker image.
  + Create triggers to create Docker images when your application changes.
  + Configure a Spinnaker pipeline to reliably and continuously deploy your application to Kubernetes Engine.
  + Deploy a code change, triggering the pipeline, and watch it roll out to production.
* Pipeline architecture
  + To continuously deliver application updates to your users, you need an automated process that reliably builds, tests, and updates your software. Code changes should automatically flow through a pipeline that includes artifact creation, unit testing, functional testing, and production rollout. In some cases, you want a code update to apply to only a subset of your users, so that it is exercised realistically before you push it to your entire user base. If one of these [canary](https://martinfowler.com/bliki/CanaryRelease.html) releases proves unsatisfactory, your automated procedure must be able to quickly roll back the software changes.
  + Graphical user interface, application

    Description automatically generated
  + With Kubernetes Engine and Spinnaker you can create a robust continuous delivery flow that helps to ensure your software is shipped as quickly as it is developed and validated. Although rapid iteration is your end goal, you must first ensure that each application revision passes through a gamut of automated validations before becoming a candidate for production rollout. When a given change has been vetted through automation, you can also validate the application manually and conduct further pre-release testing.
  + After your team decides the application is ready for production, one of your team members can approve it for production deployment.
* **Application delivery pipeline**
  + In this lab you build the continuous delivery pipeline shown in the following diagram.
  + Timeline

    Description automatically generated
* Set up your environment
  + Configure the infrastructure and identities required for this lab. First you'll create a Kubernetes Engine cluster to deploy Spinnaker and the sample application.
  + Set the default compute zone:
    - gcloud config set compute/zone us-central1-f
  + Create a Kubernetes Engine cluster using the Spinnaker tutorial sample application:
    - gcloud container clusters create spinnaker-tutorial --machine-type=n1-standard-2
  + Cluster creation takes between **five and ten minutes** to complete. Wait for your cluster to finish provisioning before proceeding.
  + When completed you see a report detailing the name, location, version, ip-address, machine-type, node version, number of nodes and status of the cluster that indicates the cluster is running.
* **Configure identity and access management**
  + Create a Cloud Identity Access Management (Cloud IAM) [service account](https://cloud.google.com/iam/docs/service-accounts) to delegate permissions to Spinnaker, allowing it to store data in Cloud Storage. Spinnaker stores its pipeline data in Cloud Storage to ensure reliability and resiliency. If your Spinnaker deployment unexpectedly fails, you can create an identical deployment in minutes with access to the same pipeline data as the original.
  + Upload your startup script to a Cloud Storage bucket by following these steps:
  + Create the service account:
    - gcloud iam service-accounts create spinnaker-account --display-name spinnaker-account
  + Store the service account email address and your current project ID in environment variables for use in later commands:
    - export SA\_EMAIL=$(gcloud iam service-accounts list --filter="displayName:spinnaker-account" --format='value(email)')
    - export PROJECT=$(gcloud info --format='value(config.project)')
  + Bind the storage.admin role to your service account:
    - gcloud projects add-iam-policy-binding $PROJECT --role roles/storage.admin --member serviceAccount:$SA\_EMAIL
  + Download the service account key. In a later step, you will install Spinnaker and upload this key to Kubernetes Engine:
    - gcloud iam service-accounts keys create spinnaker-sa.json --iam-account $SA\_EMAIL
* Set up Cloud Pub/Sub to trigger Spinnaker pipelines
  + Create the Cloud Pub/Sub topic for notifications from Container Registry.
    - gcloud pubsub topics create projects/$PROJECT/topics/gcr
  + Create a subscription that Spinnaker can read from to receive notifications of images being pushed.
    - gcloud pubsub subscriptions create gcr-triggers --topic projects/${PROJECT}/topics/gcr
  + Give Spinnaker's service account permissions to read from the gcr-triggers subscription.
    - export SA\_EMAIL=$(gcloud iam service-accounts list --filter="displayName:spinnaker-account" --format='value(email)')
    - gcloud beta pubsub subscriptions add-iam-policy-binding gcr-triggers --role roles/pubsub.subscriber --member serviceAccount:$SA\_EMAIL
* Deploying Spinnaker using Helm
  + In this section you use [Helm](https://github.com/kubernetes/helm) to deploy Spinnaker from the [Charts](https://github.com/kubernetes/charts) repository. Helm is a package manager you can use to configure and deploy [Kubernetes applications](http://kubeapps.com/).
  + **Helm** is already installed in your **Cloud Shell**.

### **Configure Helm**

* + - Grant Helm the cluster-admin role in your cluster:
      * kubectl create clusterrolebinding user-admin-binding --clusterrole=cluster-admin --user=$(gcloud config get-value account)
    - Grant Spinnaker the cluster-admin role so it can deploy resources across all namespaces:
      * kubectl create clusterrolebinding --clusterrole=cluster-admin --serviceaccount=default:default spinnaker-admin
    - Add the stable charts deployments to Helm's usable repositories (includes Spinnaker):
      * helm repo add stable https://charts.helm.sh/stable
      * helm repo update

### **Configure Spinnaker**

* + - Still in Cloud Shell, create a bucket for Spinnaker to store its pipeline configuration:
      * export PROJECT=$(gcloud info --format='value(config.project)')
      * export BUCKET=$PROJECT-spinnaker-config
      * gsutil mb -c regional -l us-central1 gs://$BUCKET
    - Run the following command to create a spinnaker-config.yaml file, which describes how Helm should install Spinnaker:

export SA\_JSON=$(cat spinnaker-sa.json)

export PROJECT=$(gcloud info --format='value(config.project)')

export BUCKET=$PROJECT-spinnaker-config

cat > spinnaker-config.yaml <<EOF

gcs:

enabled: true

bucket: $BUCKET

project: $PROJECT

jsonKey: '$SA\_JSON'

dockerRegistries:

- name: gcr

address: https://gcr.io

username: \_json\_key

password: '$SA\_JSON'

email: 1234@5678.com

# Disable minio as the default storage backend

minio:

enabled: false

# Configure Spinnaker to enable GCP services

halyard:

spinnakerVersion: 1.19.4

image:

repository: us-docker.pkg.dev/spinnaker-community/docker/halyard

tag: 1.32.0

pullSecrets: []

additionalScripts:

create: true

data:

enable\_gcs\_artifacts.sh: |-

\$HAL\_COMMAND config artifact gcs account add gcs-$PROJECT --json-path /opt/gcs/key.json

\$HAL\_COMMAND config artifact gcs enable

enable\_pubsub\_triggers.sh: |-

\$HAL\_COMMAND config pubsub google enable

\$HAL\_COMMAND config pubsub google subscription add gcr-triggers \

--subscription-name gcr-triggers \

--json-path /opt/gcs/key.json \

--project $PROJECT \

--message-format GCR

EOF

### **Deploy the Spinnaker chart**

* + - Use the Helm command-line interface to deploy the chart with your configuration set:
      * helm install -n default cd stable/spinnaker -f spinnaker-config.yaml --version 2.0.0-rc9 --timeout 10m0s --wait
    - The installation typically takes **5-8 minutes** to complete.
    - After the command completes, run the following command to set up port forwarding to Spinnaker from Cloud Shell:
      * export DECK\_POD=$(kubectl get pods --namespace default -l "cluster=spin-deck" -o jsonpath="{.items[0].metadata.name}")
      * kubectl port-forward --namespace default $DECK\_POD 8080:9000 >> /dev/null &
    - To open the Spinnaker user interface, click the **Web Preview** icon at the top of the Cloud Shell window and select **Preview on port 8080**.
      * Graphical user interface, text, application, chat or text message

        Description automatically generated
    - The welcome screen opens, followed by the Spinnaker user interface:
      * A picture containing text

        Description automatically generated
    - Leave this tab open, this is where you'll access the Spinnaker UI.
* Building the Docker image
  + In this section, you configure Cloud Build to detect changes to your app source code, build a Docker image, and then push it to Container Registry.

### **Create your source code repository**

* + - In Cloud Shell tab, download the sample application source code:
      * gsutil -m cp -r gs://spls/gsp114/sample-app.tar .
    - Unpack the source code:
      * mkdir sample-app
      * tar xvf sample-app.tar -C ./sample-app
    - Change directories to the source code:
      * cd sample-app
    - Set the username and email address for your Git commits in this repository. Replace [USERNAME] with a username you create:
      * git config --global user.email "$(gcloud config get-value core/account)"
      * git config --global user.name "[USERNAME]"
    - Make the initial commit to your source code repository:
      * git init
      * git add .
      * git commit -m "Initial commit"
    - Create a repository to host your code:
      * gcloud source repos create sample-app
    - Disregard the "you may be billed for this repository" message.
      * git config credential.helper gcloud.sh
    - Add your newly created repository as remote:
      * export PROJECT=$(gcloud info --format='value(config.project)')
      * git remote add origin https://source.developers.google.com/p/$PROJECT/r/sample-app
    - Push your code to the new repository's master branch:
      * git push origin master
    - Check that you can see your source code in the Console by clicking **Navigation Menu** > **Source Repositories**.
      * Graphical user interface, application

        Description automatically generated
    - Click **sample-app**.
      * Graphical user interface, text, application, email

        Description automatically generated
* **Configure your build triggers**
  + Configure Container Builder to build and push your Docker images every time you push [Git tags](https://git-scm.com/book/en/v2/Git-Basics-Tagging) to your source repository. Container Builder automatically checks out your source code, builds the Docker image from the Dockerfile in your repository, and pushes that image to Google Cloud Container Registry.
    - Diagram

      Description automatically generated
  + In the Cloud Platform Console, click **Navigation menu** > **Cloud Build** > **Triggers**.
  + Click **Create trigger**.
    - Graphical user interface, text, application

      Description automatically generated
  + Set the following trigger settings:
    - **Name**: sample-app-tags
    - **Event**: Push new tag
    - Select your newly created sample-app repository.
    - **Tag**: v1.\*
    - **Configuration**: Cloud Build configuration file (yaml or json)
    - **Cloud Build configuration file location**: /cloudbuild.yaml
  + Click **CREATE**.
    - Graphical user interface, application

      Description automatically generated
  + From now on, whenever you push a Git tag prefixed with the letter "v" to your source code repository, Container Builder automatically builds and pushes your application as a Docker image to Container Registry.
* **Prepare your Kubernetes Manifests for use in Spinnaker**
  + Spinnaker needs access to your Kubernetes manifests in order to deploy them to your clusters. This section creates a Cloud Storage bucket that will be populated with your manifests during the CI process in Cloud Build. After your manifests are in Cloud Storage, Spinnaker can download and apply them during your pipeline's execution.
  + Create the bucket:
    - export PROJECT=$(gcloud info --format='value(config.project)')
    - gsutil mb -l us-central1 gs://$PROJECT-kubernetes-manifests
  + Enable versioning on the bucket so that you have a history of your manifests:
    - gsutil versioning set on gs://$PROJECT-kubernetes-manifests
  + Set the correct project ID in your kubernetes deployment manifests:
    - sed -i s/PROJECT/$PROJECT/g k8s/deployments/\*
  + Commit the changes to the repository:
    - git commit -a -m "Set project ID"
* **Build your image**
  + Push your first image using the following steps:
  + In Cloud Shell, still in the sample-app directory, create a Git tag:
    - git tag v1.0.0
  + Push the tag:
    - git push --tags
  + Go to the Cloud Console. Still in Cloud Build, click **History** in the left pane to check that the build has been triggered. If not, verify that the trigger was configured properly in the previous section.
    - Table

      Description automatically generated
  + **Stay on this page and wait** for the build to complete before going on to the next section.
  + **Note:** If the Build fails, then click on the Build ID to open Build details page and then click **RETRY**.
* Configuring your deployment pipelines
  + Now that your images are building automatically, you need to deploy them to the Kubernetes cluster.
  + You deploy to a scaled-down environment for integration testing. After the integration tests pass, you must manually approve the changes to deploy the code to production services.

### **Install the spin CLI for managing Spinnaker**

* + - [spin](https://spinnaker.io/docs/guides/spin/) is a command-line utility for managing Spinnaker's applications and pipelines.
    - Download the 1.14.0 version of spin:
      * curl -LO https://storage.googleapis.com/spinnaker-artifacts/spin/1.14.0/linux/amd64/spin
    - Make spin executable:
      * chmod +x spin

### **Create the deployment pipeline**

* + - Use spin to create an app called sample in Spinnaker. Set the owner email address for the app in Spinnaker:
      * ./spin application save --application-name sample --owner-email "$(gcloud config get-value core/account)" --cloud-providers kubernetes --gate-endpoint http://localhost:8080/gate
    - Next, you create the continuous delivery pipeline. In this tutorial, the pipeline is configured to detect when a Docker image with a tag prefixed with "v" has arrived in your Container Registry.
    - From your sample-app source code directory, run the following command to upload an example pipeline to your Spinnaker instance:
      * export PROJECT=$(gcloud info --format='value(config.project)')
      * sed s/PROJECT/$PROJECT/g spinnaker/pipeline-deploy.json > pipeline.json
      * ./spin pipeline save --gate-endpoint http://localhost:8080/gate -f pipeline.json

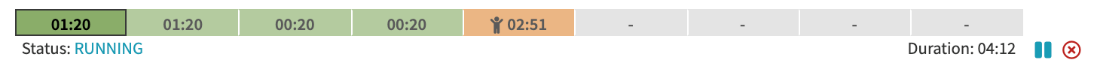
### **Manually Trigger and View your pipeline execution**

* + - The configuration you just created uses notifications of newly tagged images being pushed to trigger a Spinnaker pipeline. In a previous step, you pushed a tag to the Cloud Source Repositories which triggered Cloud Build to build and push your image to Container Registry. To verify the pipeline, manually trigger it.
    - Switch to your browser tab displaying your Spinnaker UI.
    - If you are unable to find it, you can get to this tab again by selecting **Web Preview** > **Preview on Port 8080** in your Cloud Shell window.
    - In the Spinnaker UI, click **Applications** at the top of the screen to see your list of managed applications.
    - **sample** is your application. If you don't see **sample**, try refreshing the Spinnaker Applications tab.
      * Graphical user interface, application, email

        Description automatically generated
    - Click **sample** to view your application deployment.
    - Click **Pipelines** at the top to view your applications pipeline status.
    - Click **Start Manual Execution** and then click **Run** to trigger the pipeline this first time.
      * Graphical user interface, application, website

        Description automatically generated
    - Click **Execution Details** to see more information about the pipeline's progress.
    - Graphical user interface, text, application, email

      Description automatically generated
    - The progress bar shows the status of the deployment pipeline and its steps.
      * Graphical user interface, text, application, website

        Description automatically generated
      * 
    - Steps in blue are currently running, green ones have completed successfully, and red ones have failed.
    - Click a stage to see details about it.
    - After **3 to 5 minutes** the integration test phase completes and the pipeline requires manual approval to continue the deployment.
    - Hover over the yellow "person" icon and click **Continue**.
      * A picture containing timeline

        Description automatically generated
    - Your rollout continues to the production frontend and backend deployments. It completes after a few minutes.
    - To view the app, select **Infrastructure** > **Load Balancers** in the top of the Spinnaker UI.
      * Timeline

        Description automatically generated
    - Scroll down the list of load balancers and click Default, under service sample-frontend-production. You will see details for your load balancer appear on the right side of the page. If you do not, you may need to refresh your browser.
    - Scroll down the details pane on the right and copy your app's IP address by clicking the clipboard button on the **Ingress** IP. The ingress IP link from the Spinnaker UI may use HTTPS by default, while the application is configured to use HTTP.
      * Graphical user interface, application

        Description automatically generated
    - Paste the address into a new browser tab to view the application. You might see the canary version displayed, but if you refresh you will also see the production version.
      * Table

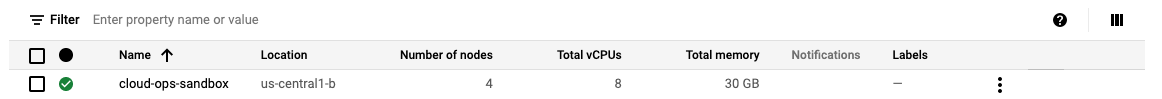
        Description automatically generated
    - You have now manually triggered the pipeline to build, test, and deploy your application.
* Triggering your pipeline from code changes
  + Now test the pipeline end to end by making a code change, pushing a Git tag, and watching the pipeline run in response. By pushing a Git tag that starts with "v", you trigger Container Builder to build a new Docker image and push it to Container Registry. Spinnaker detects that the new image tag begins with "v" and triggers a pipeline to deploy the image to canaries, run tests, and roll out the same image to all pods in the deployment.
  + From your sample-app directory, change the color of the app from orange to blue:
    - sed -i 's/orange/blue/g' cmd/gke-info/common-service.go
  + Tag your change and push it to the source code repository:
    - git commit -a -m "Change color to blue"
    - git tag v1.0.1
    - git push --tags
  + In the Console, in **Cloud Build** > **History**, wait a couple of minutes for the new build to appear. You may need to refresh your page. Wait for the new build to complete, before going to the next step.
  + **Note:** If the Build fails, please click on Build ID and then click **RETRY**.
  + Return to the Spinnaker UI and click **Pipelines** to watch the pipeline start to deploy the image. The automatically triggered pipeline will take a few minutes to appear. You may need to refresh your page.
    - Graphical user interface, text, application, email

      Description automatically generated
* Observe the canary deployments
  + When the deployment is paused, waiting to roll out to production, return to the web page displaying your running application and start refreshing the tab that contains your app. Four of your backends are running the previous version of your app, while only one backend is running the canary. You should see the new, blue version of your app appear about every fifth time you refresh.
  + When the pipeline completes, your app looks like the following screenshot. Note that the color has changed to blue because of your code change, and that the **Version** field now reads canary.
    - Table

      Description automatically generated
  + You have now successfully rolled out your app to your entire production environment!
  + Optionally, you can roll back this change by reverting your previous commit. Rolling back adds a new tag (v1.0.2), and pushes the tag back through the same pipeline you used to deploy v1.0.1:
    - git revert v1.0.1
    - Press **CTRL+O**, **ENTER**, **CTRL+X**.
    - git tag v1.0.2
    - git push --tags
  + When the build and then the pipeline completes, verify the roll back by clicking **Infrastructure** > **Load Balancers**, then click the **service sample-frontend-production** **Default** and copy the Ingress IP address into a new tab.
  + Now your app is back to orange and you can see the production version number.
    - Table

      Description automatically generated

## 3.Troubleshooting Workloads on GKE for Site Reliability Engineers

* Introduction
  + Site Reliability Engineers (SRE) have a broad set of responsibilities, and managing incidents is a critical part of their role. You will learn how to take advantage of the integrated capabilities of Google Cloud's operations suite that includes logging, monitoring, and rich, out-of-the-box dashboards.
  + The troubleshooting process is an “iterative” approach where SREs form a hypothesis about the potential root cause of an incident, then filter, search, and navigate through large volumes of telemetry data collected from their systems to validate or invalidate their hypothesis. If a hypothesis is invalid, SREs will form another hypothesis and perform another iteration until they can isolate a root cause.
  + In this lab, you will learn how to navigate that iterative journey efficiently and effectively using Google Cloud's operations tools!
* **Lab Objectives**
  + In this lab, you will learn how to:
  + Navigate resource pages of Google Kubernetes Engine (GKE)
  + Leverage the GKE dashboard to quickly view operational data
  + Create logs-based metrics to capture specific issues
  + Create a Service Level Objective (SLO)
  + Define an Alert to notify SRE staff of incidents
* Scenario
  + Your organization has deployed a multi-tier microservices application. It is a web-based e-commerce application called "Hipster Shop", where users can browse for vintage items, add them to their cart and purchase them. Hipster Shop is composed of many microservices, written in different languages, that communicate via gRPC and REST APIs. The architecture of the deployment is optimized for learning purposes and includes modern technologies as part of the stack: Kubernetes, Istio, Cloud Operations, App Engine, gRPC, OpenTelemetry, and similar cloud-native technologies.
  + As a member of the Site Reliability Engineering (SRE) team, you are contacted when end users report issues viewing products and adding them to their cart. You will explore the various services deployed to determine the root cause of the issue and set up a Service Level Objective (SLO) to prevent similar incidents from occuring in the future.
* Navigating Google Kubernetes Engine (GKE) Resource Pages
  + For the first part of this lab, you will view Google Kubernetes Engine (GKE) resource pages, then navigate to various metrics dashboards to investigate the issue reported by end users in more detail.
  + In Cloud Console, from the **Navigation menu** go to **Kubernetes Engine** > **Clusters**.
  + Confirm that you see the following Kubernetes cluster available: cloud-ops-sandbox. Validate that each cluster has a green checkbox next to it to indicate it is up and running.
    - 
  + Click on the cloud-ops-sandbox link under the **Name** column to navigate to the cluster's **Details** tab.
  + Click on the **Nodes** tab to see all the nodes in the cluster. Validate that there is a single node pool.
  + Under the **Nodes** section of the **Nodes** tab, click on the link for the first node in the table under the **Name** column to view more details about the node.
    - A picture containing text

      Description automatically generated
  + On the node details page, note the metrics of the node that are available. These should be listed under the **Summary** tab and include CPU and Memory Usage as well as Disk I/O among others. This lab generates load during provisioning and you should see metrics activity but no obvious spikes or metrics above the "requested" limit for the graphs displayed on the **Summary** tab.
  + The metrics plotted in the image below during your lab run will not appear exactly as pictured
    - Graphical user interface, application, table, Excel

      Description automatically generated
  + To investigate further, rather than navigating to each individual node to view its metrics, click on the three dots in the top right corner of the **CPU** tile and select **View in Metrics Explorer**.
    - Graphical user interface, text, application, chat or text message

      Description automatically generated
  + On the **Metrics Explorer** page, you will see the metrics associated with the node that you just navigated from. There are three filters configured in the **Metrics explorer** under the **Filters** section.
  + Remove the filter for the nodename by expanding the filter and clicking the **delete** icon.
  + Under the **How do you want to view that data** section, set **Group by** to node\_name.
    - Graphical user interface, text, application, email

      Description automatically generated
  + Once the filters are set the visualization will update and you will be able to view the same metrics for all of the nodes in the node pool of the cloud-ops-sandbox cluster.
    - Graphical user interface, application

      Description automatically generated
  + You may notice two additional metrics are displayed as well, these are limit cores and request cores. The limit cores is the CPU cores limit of the container running on the node while the request cores metric is the number of CPU cores requested by the container running on the node. You can find out more about Kubernetes metrics on the following documentation page: [Kubernetes metrics](https://cloud.google.com/monitoring/api/metrics_kubernetes)
* Accessing Operational Data Through GKE Dashboards
  + In this next section, you will explore how to quickly navigate to detailed operational data of various resources deployed to GKE via the GKE Dashboard.
  + Recall that website users have reported that they cannot view product details or add items to their cart. You can verify this by opening the website. Navigate to **Left Menu** > **Kubernetes Engine** > **Services & Ingress**. Click on the IP address next to the service frontend-external. Click on any product that is displayed on the landing page to reproduce the error reported.
    - Table

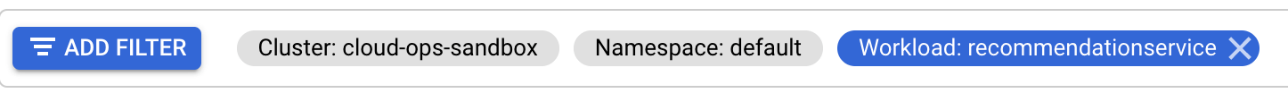
      Description automatically generated
  + Upon reproducing the error you notice that the stack trace mentions the application "failed to get product recommendations" and decide to investigate the **recommendationservice** deployed to GKE as a result.
    - Graphical user interface, text, application

      Description automatically generated
  + Navigate to Cloud Monitoring from Cloud Console, from the **Navigation Menu** go to **Monitoring** > **Dashboards**.
  + When on the **Dashboards** landing page select **GKE**.
  + To avoid scrolling down the left menu to reach the **Monitoring** section for the rest of this lab, hover over the **Monitoring** section of the left menu and select the pin icon that appears. This will place the **Monitoring** section at the top of the left menu for future navigation.
    - Graphical user interface, text, application, email

      Description automatically generated
  + You should see a dashboard view that provides relevant Cluster, Namespace, Workload, Service, Pod and Container related metrics for GKE resources found in the project in an aggregated view similar to the following:
    - Table

      Description automatically generated
  + For this lab's scenario, you will want to view logs and metrics related to the recommendationservice as end users are seeing errors related to product recommendations when viewing a product's landing page. You will create filters for the cloud-ops-sandbox cluster to investigate possible symptoms and diagnose the issue further.
  + Add the following filters to your GKE Dashboard. Click on the **Add Filter** button at the top of the GKE Dashboard page.
    - Graphical user interface, text, application, chat or text message

      Description automatically generated
  + From the available filters, select **Workloads** > **recommendationservice**.
    - Graphical user interface, text, application

      Description automatically generated
  + Click on the **Apply** button once the correct filter is selected. The Filter section of the GKE Dashboard page should look similar to the following image.
    - 
  + This view allows you to focus your attention on the problematic recommendationservice microservice.
  + Under the **Workloads** section, click on the recommendationservice to reveal the **Deployment details** pane. This view presents details on Alerts, Service Level Objectives (SLOs), Events, Metrics and Logs. At this point in the lab, no SLOs are present. You will add an SLO here in the next part of this lab.
  + Click on the **Metrics** tab to view metrics related to the recommendationservice. You can change the **Metrics** drop down selection to alter the visualization data provided and view different metrics available for this service.
    - Graphical user interface, text, application, email

      Description automatically generated
  + Click on the **Logs** tab to view logs related to the recommendationservice. You can filter the available logs by using the **Severity** drop down corresponding to the log level of the entries available. This is useful in an SRE context to find errors recorded in the logs and leverage the entries to troubleshoot issues.
  + Set the **Severity** to Error in order to filter the recommendationservice logs.
    - Graphical user interface, text, application, email

      Description automatically generated
  + At this point, the error related to the problematic code should be obvious. Look for the phrase invalid literal for int() with base 10: '5.0' in the items in the result set. This error appearing in the recommendationservice filters confirms that the service has a bug in the code.
  + You will re-deploy the **recommendationservice** microservice to ensure that the error is no longer present.
  + For simplicity, you will simulate deploying a new version of the application using kubectl.
  + In Cloud Shell run the following command:
    - git clone --depth 1 --branch cloudskillsboost https://github.com/GoogleCloudPlatform/cloud-ops-sandbox.git
    - cd cloud-ops-sandbox/sre-recipes
  + Navigate to **Navigation Menu** > **Kubernetes Engine** > **Clusters**. Select the three dots to the right of the cloud-ops-sandbox cluster and select the option to **Connect**.
    - Graphical user interface, text, application

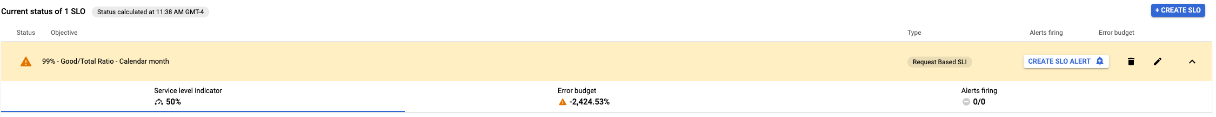
      Description automatically generated
  + On the **Connect to the cluster** modal dialog, click the **RUN IN CLOUD SHELL** button. Press **Enter** to run the command once populated in Cloud Shell.
  + Last, run the apply command to update the service:
    - ./sandboxctl sre-recipes restore "recipe3"
  + To verify the application is working correctly, navigate to **Navigation Menu** > **Kubernetes Engine** > **Services & Ingress**.
  + Click on the Endpoint listed next to the frontend-external service.
    - Table

      Description automatically generated.
  + This will take you to the Hipster Shop website used for this lab exercise. Click on any product to verify that it loads without throwing any errors.
    - Graphical user interface, website

      Description automatically generated.
  + In this section of the lab, you explored the available logs and metrics in the GKE dashboard to diagnose an issue with the application workload deployed by the DevOps team. You were able to pin point the exact cause of an issue and remediate it by re-deploying the problematic microservice with a bug fix.
* Proactive Monitoring with Logs-Based Metrics
  + To ensure that the updated recommendationservice code is working as expected, and to prevent future incidents from occuring again, you decide to create a logs-based metric to monitor the logs and notify SRE when similar incidents occur in the future.
  + In this section, you will create a logs-based metric specific to the error noticed in the previous sections.
  + Using logs-based metrics you can define a metric that tracks errors in the logs to proactively respond to similar problems and symptoms before they are noticed by end users.
  + From Cloud Console, click on the **Navigation Menu** > **Logging** > **Logs Explorer**.
  + To avoid scrolling down the left menu to reach the **Logging** section for the rest of this lab, hover over the **Logging** section of the left menu and select the pin icon that appears. This will place the **Logging** section at the top of the left menu for future navigation.
  + In the Query results section click on **Create metric**. This will open a new tab to create a logs based metric.
    - Bar chart

      Description automatically generated with low confidence
  + Enter the following options on the Create logs metric page:
    - Metric Type: **Counter**
    - Log metric name: **Error\_Rate\_SLI**
    - Filter Selection: (Copy and paste the filter below)
      * resource.labels.cluster\_name="cloud-ops-sandbox" AND resource.labels.namespace\_name="default" AND resource.type="k8s\_container" AND labels.k8s-pod/app="recommendationservice" AND severity>=ERROR
  + In the next section we will leverage a different metric centered around Availability to proactively notify the SRE team of issues, however, please note that the custom, logs-based metric defined in this section could also be utilized to generate an alert when its filter condition is met.
  + Click **Create Metric**.
* Creating a Service Level Objective (SLO)
  + After creating the logs-based metric, the SRE team decides that it will define a **Service Level Objective (SLO)** on the recommendationservice. You use an SLO to specify service-level objectives for performance metrics. An SLO is a measurable goal for performance over a period of time. See [Designing and using SLOs](https://cloud.google.com/service-mesh/docs/observability/design-slo) for more guidance on SLO design and the filters you will use below.
  + Navigate to **Navigation menu** > **Monitoring** > **Services**. The resulting page will display a list of all services deployed to GKE for the application workload.
  + Select the recommendationservice service from the list of available services which will take you to the **Service details** page.
  + Click on **+ Create SLO** on the top right of the page.
    - Graphical user interface, text, application, email

      Description automatically generated
  + On **Step 1** you will be presented with a dialog for creating a new SLO. Set the following parameters:
  + Choose a metric: **Availability**
  + Request-based or windows-based: **Request Based**
  + Click **Continue**.
* On **Step 2** Define SLI details, the Performance Metric will already be assigned to measure the availability of the service based on the percentage of successful requests.
  + - Graphical user interface, text, application

      Description automatically generated
  + Click on **Continue**.
  + On **Step 3** Set your service-level objective (SLO), the user needs to define two additional items: the Compliance period and Performance goal. Make the following selections:
    - Period type: **Calendar**
    - Period length: **Calendar month**
    - Performance Goal: **99%**
  + Click on **Continue**.
  + Click **Create SLO** on the last step of the wizard to complete the SLO creation process.
  + This will bring you back to the **Monitoring** > **Services** landing page. You should be able to see an SLO violation under the Current status of SLO section.
  + Click on the entry listed and select the **Error budget** tab once expanded.
  + The Error budget fraction represents the actual percentage of error budget remaining for the compliance period. In the SLO defined, there is a period of one calendar month and a performance goal of 99% or better.
  + As denoted by the percentage, the error preventing product pages from loading properly in this fictitious scenario severely degraded the service-level objective defined. This may not be the case in a real world scenario as this lab ran a load test against the Kubernetes cluster hosting the application workload.
* Define an Alert on the Service Level Objective (SLO)
  + To proactively notify the SRE team of any violations of the Service Level Objective set, it is a best practice to define an Alert that will trigger when the SLO is violated. The alert can invoke a notification channel of your choice, including: Email, SMS, PagerDuty, Slack, a WebHook or a subscription to a PubSub topic.
  + Navigate to **Navigation menu** > **Monitoring** > **Services**.
  + Click on the recommendationservice service from the list of services available.
  + Under the section **Current status of 1 SLO**, you should see the Service Level Objective created in the last task. Expand the **SLO** listed.
  + Click the **Create SLO Alert** button present on the SLO. This will allow you to define an Alert policy when the SLO is violated.
    - 
  + Note: your error budget percentage will be different.
  + On the **Create SLO burn rate alert policy** modal input, you will see Lookback duration and Burn rate threshold fields on Step 1 of the wizard. The lookback duration allows you to specify the duration of time from the present time to have the Alerting policy lookback for possible burn rate violations. The burn rate threshold allows you to specify the window time-slice to split the lookback duration into in order to assess whether or not the SLO has been violated.
    - Leave the default values:
    - Lookback duration: 60 minutes
    - Burn rate threshold: 10
  + Click **Next**.
  + On Step 2, you can define a notification channel to receive the alert when the violation is observed. For the purposes of this lab, you can optionally supply an email address or SMS channel to receive a notification.
    - * Graphical user interface, text, application

        Description automatically generated
  + Click **Next**.
  + Step 3 is optional and allows you to supply any information to the end user receiving the notification so that they have immediate context as to what the issue may be and ways to mitigate the problem.
  + Click **Save**.

## 4.Deploy and Manage Cloud Environments with Google Cloud: Challenge Lab

* Overview
  + In a challenge lab you’re given a scenario and a set of tasks. Instead of following step-by-step instructions, figure out how to complete the tasks on your own!.
* Topics tested:
  + Complete the production application environment.
  + Ensure monitoring and alerts enabled on key development components.
  + Test the Spinnaker CI/CD deployed environment is working as expected.
* Challenge scenario
  + You have started a new role as a Cloud Architect for Jooli Inc. You are expected to help design and manage the infrastructure at Jooli. Common tasks revolve around designing environments for the various projects within the Jooli Inc. family but also include provisioning resources for projects.
  + You are expected to have the skills and knowledge for these tasks, so don't expect step-by-step guides.
  + You have been asked to assist the kraken team complete setting up their product development environment. The previous Cloud Architect working with the kraken team was unfortunately too curious about if krakens were real or not, and has gone missing after venturing out into the open sea last weekend in search of such a beast.
  + Jooli Inc. management has supreme faith in your abilities, don't let them down! (Seriously, they don't need the dates to slip further).
  + The kraken team is building a next generation tool and they will host the application on Kubernetes. The project source code is stored in Cloud Source Repositories, with Spinnaker building and deploying any changes into the build Kubernetes environment.
* **Your challenge**
  + As soon as you sit down at your desk and open your new laptop you receive the following request to complete these tasks. Good luck!
  + Task 1: Create the production environment
    - The previous Cloud Architect had written the **Deployment Manager** configuration to build the network for kraken's production environment. You can find the DM configuration on your **jumphost** in /work/dm. Create the network using the Deployment Manager configuration (/work/dm/prod-network.yaml and /work/dm/prod-network.jinja). Make sure you review the configuration before deploying it.
    - You will also need to create the Kubernetes environment. The application is already created and in the Container Repository.
    - Create a two (2) node cluster called Cluster Name in the kraken-prod-vpc (remember to use --num-nodes to create 2 nodes only).
    - Use kubectl with the files in /work/k8s to create the frontend and backend deployments and services (which will expose the frontend service via a load balancer).
    - The architecture is simple, the diagram below describes the environment. The new work you need to complete (VPC and Kubernetes cluster with services) is in the red box.
      * Graphical user interface, timeline

        Description automatically generated with medium confidence
    - Solution
      * The first step is to create a production environment.
      * Navigation > Compute Engine > VM Instance > Click the jumphost instance SSH button.
      * In the SSH window, go to the cd /work/dm directory.
      * Edit the configuration file prod-network.yaml, and replace SET\_REGION to us-east1
        + 
      * OR edit through command.
        + sed -i s/SET\_REGION/us-east1/g prod-network.yaml
        + gcloud deployment-manager deployments create prod-network --config=prod-network.yaml

Text

Description automatically generated

* + - * Now create a Kubernetes cluster in the new network
        + gcloud config set compute/zone us-east1-b
        + gcloud container clusters create kraken-prod --num-nodes 2 --network kraken-prod-vpc --subnetwork kraken-prod-subnet

Text

Description automatically generated

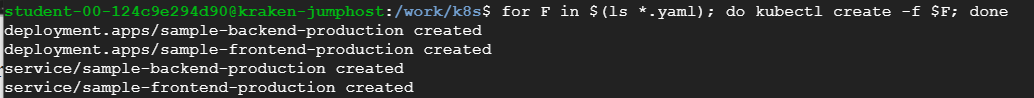
* + - * Checking Credentials of kraken-pod
        + gcloud container clusters get-credentials kraken-prod



* + - * Go to Directory
        + cd /work/k8s



* + - * + for F in $(ls \*.yaml); do kubectl create -f $F; done



* + - * Note: All commands written in SSH
  + Task 2: Setup the Admin instance
    - You need to set up an admin machine for the team to use.
    - Once you create the **kraken-prod-vpc**, you will need to add an instance called Instance Name , a network interface in **kraken-mgmt-subnet** and another in **kraken-prod-subnet**.
    - You need to monitor **Instance Name**and if **CPU utilization** is over **Threshold**for more than a minute you need to send an email to yourself, as admin of the system.
    - Solution
      * Run through jumpshot SSH (the above SSH)
        + gcloud config set compute/zone us-east1-b
        + gcloud compute instances create kraken-admin --network-interface="subnet=kraken-mgmt-subnet" --network-interface="subnet=kraken-prod-subnet"

Text

Description automatically generated

* + - * Create a Monitoring workspace. Navigation > Monitoring> Alerting > Click Create Policy.
        + Click Add Condition, and then set up the Metrics with the following parameters:

Resource Type : VM Instance

Metric : CPU utilization

Filter : instance\_name

Value : kraken-admin

Condition : is above

Threshold : 80%

For : 1 minute

* + - * + Click ADD
        + Then add an email in the Notification setting.
  + Task 3: Verify the Spinnaker deployment
    - The previous architect set up Spinnaker in **kraken-build-vpc**. Please connect to the Spinnaker console and verify that the built resources are working.
    - To access the Spinnaker console use Cloud Shell and kubectl to **port forward** the **spin-deck** pod from port **9000** to **8080** and then use Cloud Shell's **web preview**.
    - You must test that a change to the source code will result in the automated deployment of the new build. You should pull the **sample-app** repository to make the changes. Make sure you push a **new, updated, tag**.
    - Solution
      * Through Cloud Shell and kubectl to port forward the spin-deck pod from port 9000 to 8080.
        + gcloud config set compute/zone us-east1-b
        + gcloud container clusters get-credentials spinnaker-tutorial
        + DECK\_POD=$(kubectl get pods --namespace default -l "cluster=spin-deck" -o jsonpath="{.items[0].metadata.name}")
        + kubectl port-forward --namespace default $DECK\_POD 8080:9000 >> /dev/null &
      * OR Through Navigation Menu >Kubernetes Engine > Services & Ingress > Go to spin-deck.
        + Click Port Forward.
        + Cloud Shell will open with the port forwarding command.
      * Click on the web preview icon on cloud shell open port 8080.
        + Go to applications -> sample
        + Open pipelines and manually run the pipeline if it has not already running.
        + Approve the deployment to production.
        + Check the production frontend endpoint (use http, not the default https)
      * Back to cloudshell, run to push a change
        + Clone your source code repository and commit it by following the command.

gcloud config set compute/zone us-east1-b

gcloud source repos clone sample-app

cd sample-app

touch a

git config --global user.email "$(gcloud config get-value account)"

git config --global user.name "Student"

git commit -a -m "change"

git tag v1.0.1

git push –tags

* + - * Spinnaker, deploy manually to production all piepliens.
  + prod-network.yaml

imports:

- path: prod-network.jinja

resources:

- name: prod-network

type: prod-network.jinja

properties:

region: us-east1

* + cat prod-network.jinja

resources:

{# Network #}

- name: kraken-prod-vpc

type: gcp-types/compute-v1:networks

properties:

description: "Kraken Production VPC"

autoCreateSubnetworks: false

{# Subnet #}

- name: kraken-prod-subnet

type: gcp-types/compute-v1:subnetworks

properties:

ipCidrRange: 192.168.12.0/24

network: $(ref.kraken-prod-vpc.selfLink)

region: {{ properties['region'] }}

privateIpGoogleAccess: true

enableFlowLogs: true

logConfig:

enable: true

flowSampling: 1

{# Firewall Rule #}

- name: kraken-prod-fw-ssh

type: gcp-types/compute-v1:firewalls

properties:

network: projects/{{ env['project'] }}/global/networks/kraken-prod-vpc

sourceRanges: [35.235.240.0/20]

priority: 100

allowed:

- IPProtocol: tcp

ports: [22]

targetTags:

- ssh-ingress

metadata:

dependsOn:

- kraken-prod-vpc

* + deployment-prod-backend.yaml

kind: Deployment

apiVersion: apps/v1

metadata:

name: sample-backend-production

spec:

replicas: 2

selector:

matchLabels:

app: sample

template:

metadata:

name: backend

labels:

app: sample

role: backend

env: production

spec:

containers:

- name: backend

image: gcr.io/qwiklabs-resources/sample-app:v1.0.0

resources:

limits:

memory: "500Mi"

cpu: "100m"

imagePullPolicy: Always

readinessProbe:

httpGet:

path: /health

port: 8080

env:

- name: COMPONENT

value: backend

- name: VERSION

value: production

ports:

- name: backend

containerPort: 8080

* + cat deployment-prod-frontend.yaml

kind: Deployment

apiVersion: apps/v1

metadata:

name: sample-frontend-production

spec:

replicas: 2

selector:

matchLabels:

app: sample

template:

metadata:

name: frontend

labels:

app: sample

role: frontend

env: production

spec:

containers:

- name: frontend

image: gcr.io/qwiklabs-resources/sample-app:v1.0.0

resources:

limits:

memory: "500Mi"

cpu: "100m"

imagePullPolicy: Always

readinessProbe:

httpGet:

path: /health

port: 8080

env:

- name: COMPONENT

value: frontend

- name: BACKEND\_URL

value: http://sample-backend-production:8080/metadata

- name: VERSION

value: production

ports:

- name: frontend

containerPort: 8080

* + service-prod-backend.yaml

apiVersion: v1

kind: Service

metadata:

labels:

app: sample-backend-production

name: sample-backend-production

spec:

ports:

- name: 8080-8080

port: 8080

protocol: TCP

targetPort: 8080

selector:

app: sample

role: backend

sessionAffinity: None

type: ClusterIP

* + cat service-prod-frontend.yaml

apiVersion: v1

kind: Service

metadata:

labels:

app: sample-frontend-production

name: sample-frontend-production

spec:

ports:

- name: http

port: 80

protocol: TCP

targetPort: 8080

selector:

app: sample

role: frontend

sessionAffinity: None

type: LoadBalancerstudent-00-124c9e294d90@kraken-jumphost:/work/k8s$