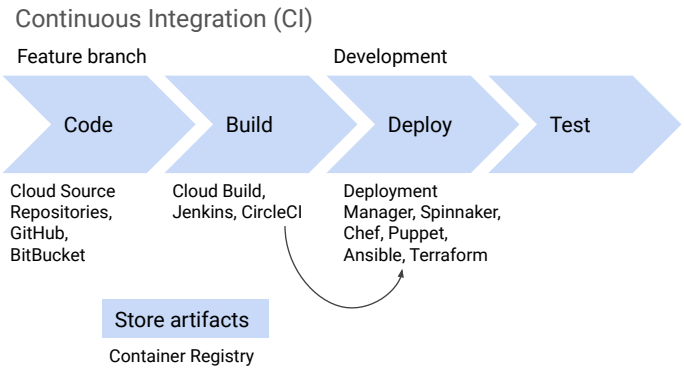
Deploying Applications

Mike Truty
Technical Curriculum Lead, Cloud Application Development

To run reliable services, you must have reliable release processes. When you have many engineers building components of an application, it's crucial that they are able to run unit tests, integration tests, and other tests quickly.

With most user-facing software, teams usually want to release software often with new features and bug fixes. To enable high release velocity, build, test and release processes must be automated as much as possible. In the module Deploying Applications, you'll learn the components of a continuous integration and delivery pipeline. You'll also learn how to build container images for your application by using Cloud Build and push the images to container registry.

Implement continuous integration and delivery for reliable releases

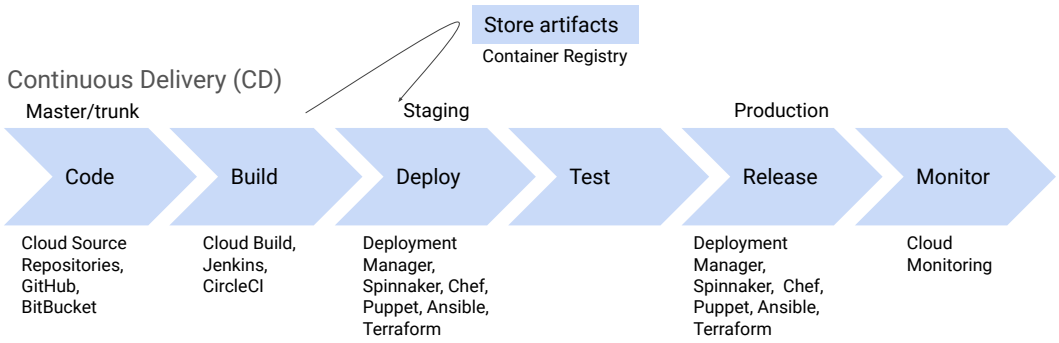


Continuous integration is a developer workflow in which developers frequently pull from the master and commit their changes into a feature branch, into a source code repository such as Cloud Source Repositories or GitHub. These commit triggers are built in a build system such as Jenkins or CircleCI.

The build process creates a new application image by using Cloud Build and stores it in an artifact repository such as container registry. A deployment system such as Spinnaker deploys the artifacts in your Cloud environment.

You can use Deployment Manager to stand up to resources for the managed services that your application needs. For example, you can use deployment manager to create Cloud Storage buckets and Pub/Sub topics. After your application is deployed in your development environment, you can automatically run tests to verify your code. If all the tests pass, you can modify changes from the feature branch to the master.

Implement continuous integration and delivery for reliable releases



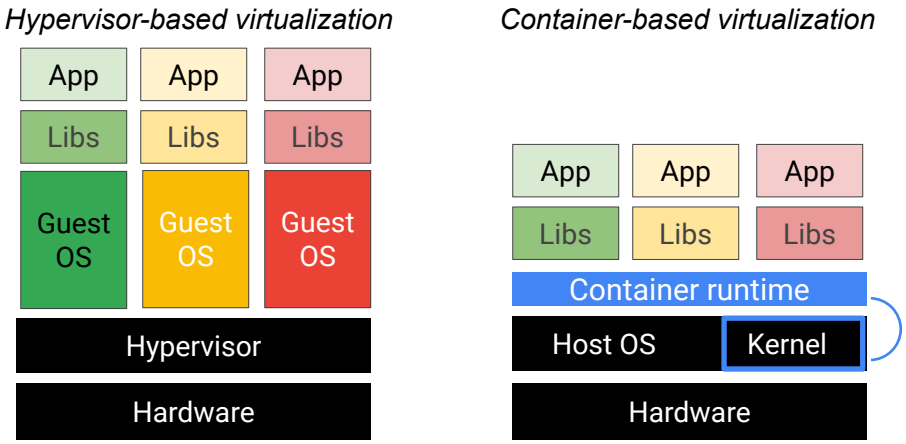
Continuous delivery is a workflow that is triggered when changes are pushed to the master repository. The build system builds the code and creates application images. The deployment system deploys the application images to the staging environment and runs integration tests, performance tests, and more. If all tests pass, the build is tagged as a release candidate build. You can manually approve a release candidate build. This approval can trigger deployment to production environments as a canary or blue-green release.

You can monitor the performance of your application and print in the production environment by using monitoring services such as Cloud Monitoring. If the new deployment functions optimally, you can switch over your entire traffic to this new release. But if you discover problems, you can also quickly roll back to the last stable release.

The continuous deployment workflow varies slightly in that there is no manual approval process. The deployment system automatically deploys release candidates to the production environment.

In the rest of this presentation, you'll consider two key aspects of your Cloud native application: the container image for your application and the Google Cloud resources required by your application.

Containers: an efficient way to isolate code and manage workloads



Let's do a quick refresher on Containers.

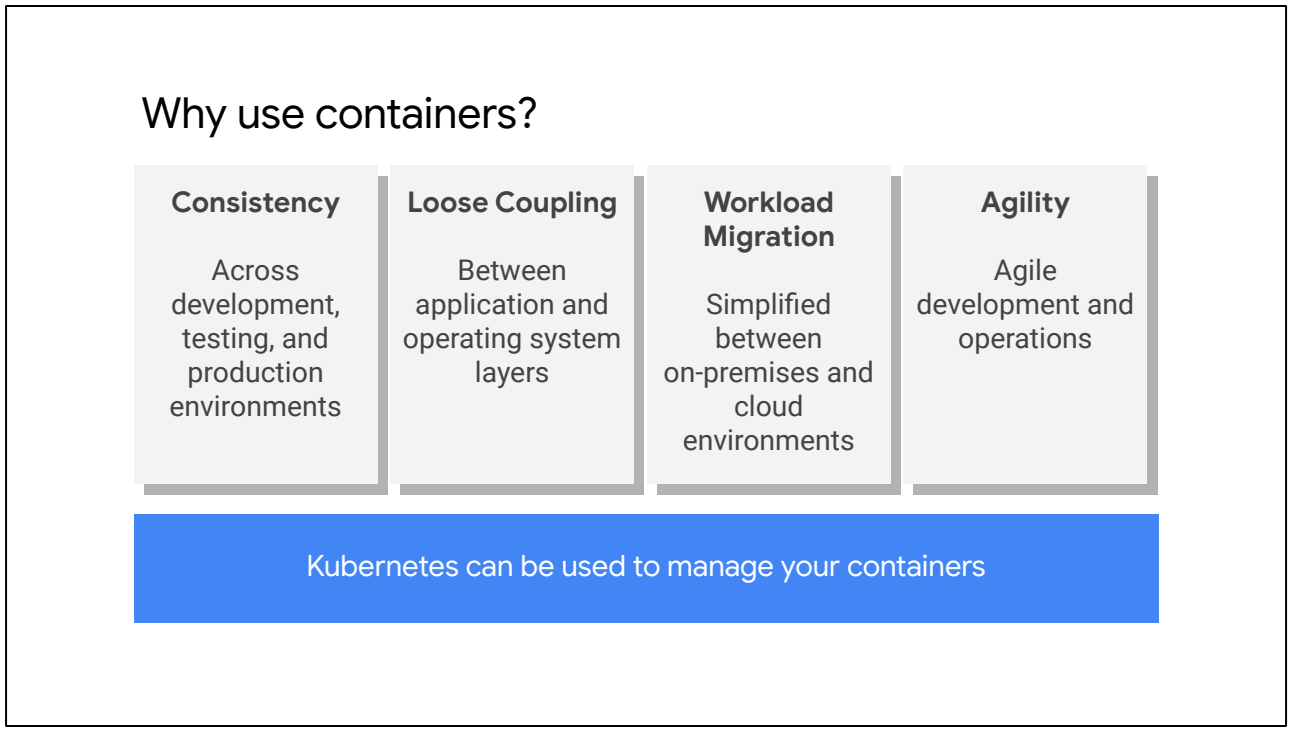
Container-based virtualization is an alternative to hardware virtualization, as in traditional virtual machines. Virtual machines are isolated from one another in part by having each virtual machine have its own instance of the operating system. But operating systems can be slow to boot and can be resource-heavy. Containers respond to these problems by using modern operating systems' built-in capabilities to isolate environments from one another.

A process is a running program. In Linux and Windows, the memory address spaces of running processes have long been isolated from one another. Popular implementations of software containers build on this isolation. They take advantage of additional operating-system features that give processes the capability to have their own namespaces and that give a supervisor process the ability to limit other processes' access to resources.

Containers start much faster than virtual machines and use fewer resources, because each container does not have its own instance of the operating system. Instead, developers configure each container with a minimal set of software libraries to do the job. A lightweight container runtime does the plumbing jobs needed to allow that container to launch and run, calling into the kernel as necessary. The container runtime also determines the image format.

Google Kubernetes Engine uses the Docker container runtime, and Docker

containers are what we'll focus on in this course.



So what does a container provide that a virtual machine does not?

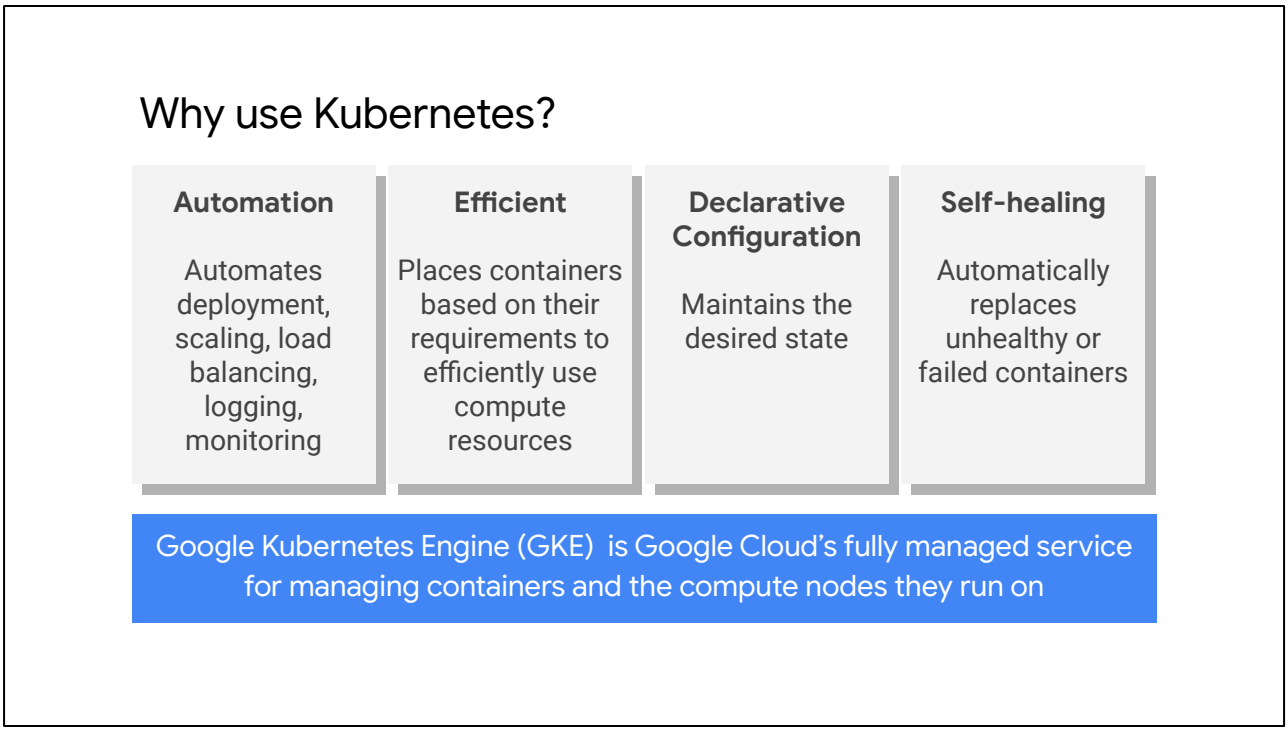
- Simple deployment: By packaging your application as a singularly addressable, registry-stored, one-command-line deployable component, a container radically simplifies the deployment of your app no matter where you're deploying it.
- Rapid availability: By abstracting just the OS instead of the whole physical computer, this package can "boot" in ~1/20th of a second, compared to a minute or so for a modern virtual machine.
- Leverage microservices: Containers allow developers and operators to further subdivide compute resources. If a micro VM instance seems like overkill for your app, or if scaling an entire VM at a time seems like a big step function, containers will make a big, positive impact in your systems.

What are the implications of some of these advantages?

- A developer has in their laptop plenty of compute power to run multiple containers, making for easier and faster development. Although it is certainly possible to run several virtual machines on a laptop, it's far from fast, easy, or lightweight.
- Similarly, release administration is easier: pushing a new version of a container is a single command. Testing is also cheaper: in a public cloud

- where a VM is billed for a minimum of 10 minutes of compute time, a single test might only cost you a small amount. But if you're running thousands of programmatically driven tests per day, this starts to add up. With a container, you could do thousands of simple tests at the same cost, amounting to large savings for your production applications.
- Another implication is the composability of application systems using this model, especially with applications using open-source software. Although it might be a daunting systems administration task for a developer to install and configure MySQL, memcached, MongoDB, Hadoop, GlusterFS, RabbitMQ, node.js, nginx, and so on together on a single box to provide a platform for their application, it is much easier and a vastly lower risk to start a few containers housing these applications with some very compact scripting. This model eliminates a significant amount of error-prone, specialized, boilerplate work.

We use Kubernetes to manage our containers.

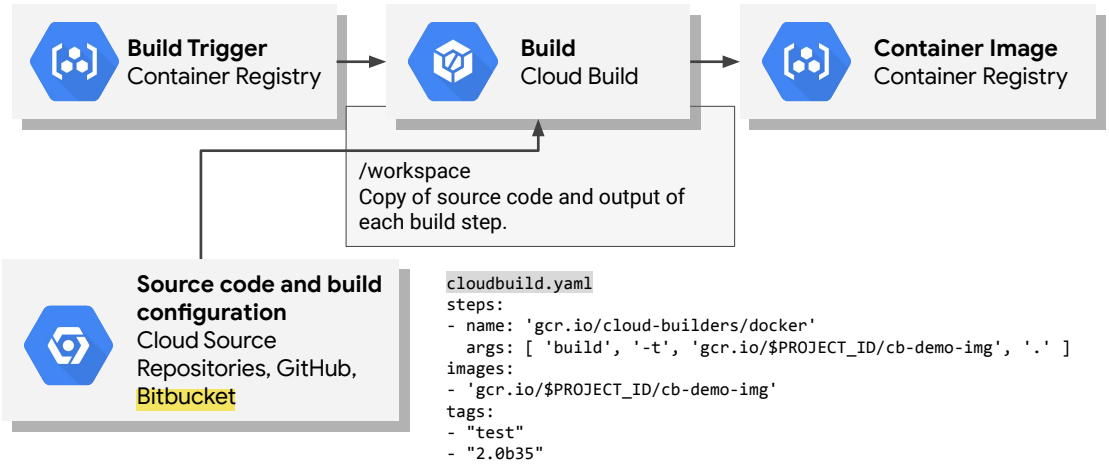


Kubernetes is an open-source tool which manages container orchestration. Benefits include:

- Automation: Kubernetes automates the deployment, scaling, load balancing, logging, monitoring, and other management features of containerized applications.
- Efficient: By specifying CPU and memory requirements of containers, Kubernetes can efficiently pack containers on nodes to efficiently use compute nodes.
- Declarative configuration: You can configure the desired state you want to achieve, rather than issuing a series of commands to achieve that desired state. Kubernetes will make your system conform to that desired state. This greatly reduces the operational complexity of large systems.
- Self-healing: Kubernetes can automatically replace containers that have failed or are failing a health check. This leads to many fewer middle-of-the-night pages for operations staff.

Google Kubernetes Engine, or GKE, is Google Cloud's fully managed solution for running containers. GKE scales, repairs, and manages your cluster of compute nodes that runs your Kubernetes-managed containers.

Use Cloud Build and Container Registry to create application images



The container image for your application is a complete package that contains the application binary and all the software that's required for the application to run. When you deploy the same container image on your development, test and production environments, you can be sure that your application will perform exactly the same way in each of these environments. Cloud Build is a fully managed service that enables you to set up build pipelines to create a Docker container image for your application and push the image to a Google Cloud container registry. You don't need to download all build tools and container images to a build machine or manage your own build infrastructure.

By using Container Registry and Cloud Build, you can create built pipelines that are automatically triggered when you commit code to a repository. In Container Registry, you can create a build trigger that is executed based on a trigger type. A trigger type specifies whether build should be triggered based on commits to a particular branch in a repository or commits that contain a particular tag.

You must also create a build configuration file that specifies the steps in the build pipeline. Steps are analogous to commands or scripts that you execute to build your application. Each build step is a Docker container that's invoked by Cloud Build when the build is executed. The step name identifies the container to invoke for the build step. The images attribute contains the name of the container image to be created by this build configuration. Cloud Build enables you to specify different types of source repositories, tag container images to enable searches, and create build steps that perform operations such as downloading and processing data, without even creating

a container image. The build configuration can be specified in a YAML or JSON format.

Cloud Build mounts your source code into the “slash workspace” directory of the Docker container associated with a build step. The artifacts produced by each build step are persisted in the “slash workspace” folder and can be used by the following build step. Cloud Build automatically pushes the built container image to Container Registry. In Container Registry, you can view the status and history of builds. Cloud Build also publishes built status notifications to Pub/Sub. You can subscribe to these notifications to take action based on build status or other attributes. For more information, see the downloads and resources pane.

For more information, see the downloads and resources pane.

For more information, see:

- Cloud Build Documentation: https://cloud.google.com/cloud-build/docs/
- Cloud Build API documentation: https://cloud.google.com/cloud-build/docs/api/reference/rest/
- Container Registry: https://cloud.google.com/container-registry/docs/

Use Deployment Manager to launch your Google Cloud resources

```
Deployment Configuration (YAML file)
imports:
- path: path/to/template.jinja
  name: my-template
- path: path/to/another/template.py

resources:
  - name: resource-a
    type: type-of-resource
    properties:
      property-a: value
      property-b: value
      ...
```

- ✓ Define template properties and environment variables.
- ✓ Define conditions to create different deployments based on a single configuration.
- ✓ Specify dependencies between resources.
- ✓ Specify startup scripts to run when VM launches.
- ✓ Define outputs.

Deployment Manager enables you to launch VM instances based on container images that are created by your build pipeline. You can also use Deployment Manager to launch other Google Cloud resources that are required by your application. The Deployment Manager API is integrated into Google Cloud.

A deployment configuration defines the Cloud resources to provision. The configuration includes the type and properties of the resources that are part of the deployment.

A deployment configuration consists of a top level-configuration file and YAML syntax, templates, and additional files. You can declare all your resources in the top-level configuration file for simple deployments. For more complex configurations or to create reusable chunks of configuration, you can create templates that create subsets of the configuration. You can then import one or more templates to construct your complete configuration. Deployment Manager will expand these templates to create your final configuration.

These reusable templates can be developed to using Jinja or Python syntax. Jinja templates and Python code are use to generate the YAML configuration.

What are the benefits of using Deployment Manager? You can reuse templates and configure resources differently for different environments. For example, you can create test and production environments by using the same deployment configuration with different template properties, environment variables, and conditions. For complex

deployments, you can specify dependencies between resources. For example, the creation of a VM might depend on, say, the creation of persistent disks. You can specify startup scripts that are run when the VM launches. These startup scripts my download software or run other commands to configure the VM. You can declare outputs for the template. For example, you can expose the IP address of a database created in a template so that other scripts can connect to the database.

For more information about Deployment Manager, see the downloads and resources pane.

For more information about Deployment Manager, see:

- Deployment Manager documentation: https://cloud.google.com/deployment-manager/docs/
- Deployment Manager examples on GitHub: https://github.com/GoogleCloudPlatform/deploymentmanager-samples

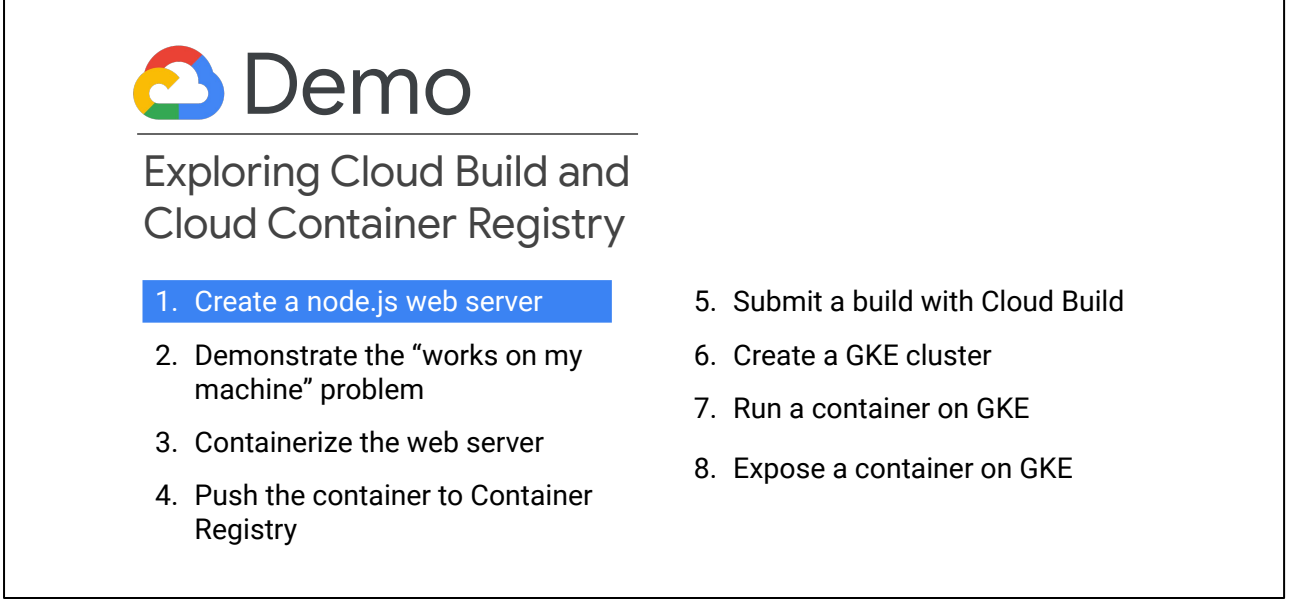
Exploring Cloud Build and Cloud Container Registry

1. Create a node.js web server
2. Demonstrate the “works on my machine” problem
3. Containerize the web server
4. Push the container to Container Registry
5. Submit a build with Cloud Build
6. Create a GKE cluster
7. Run a container on GKE
8. Expose a container on GKE

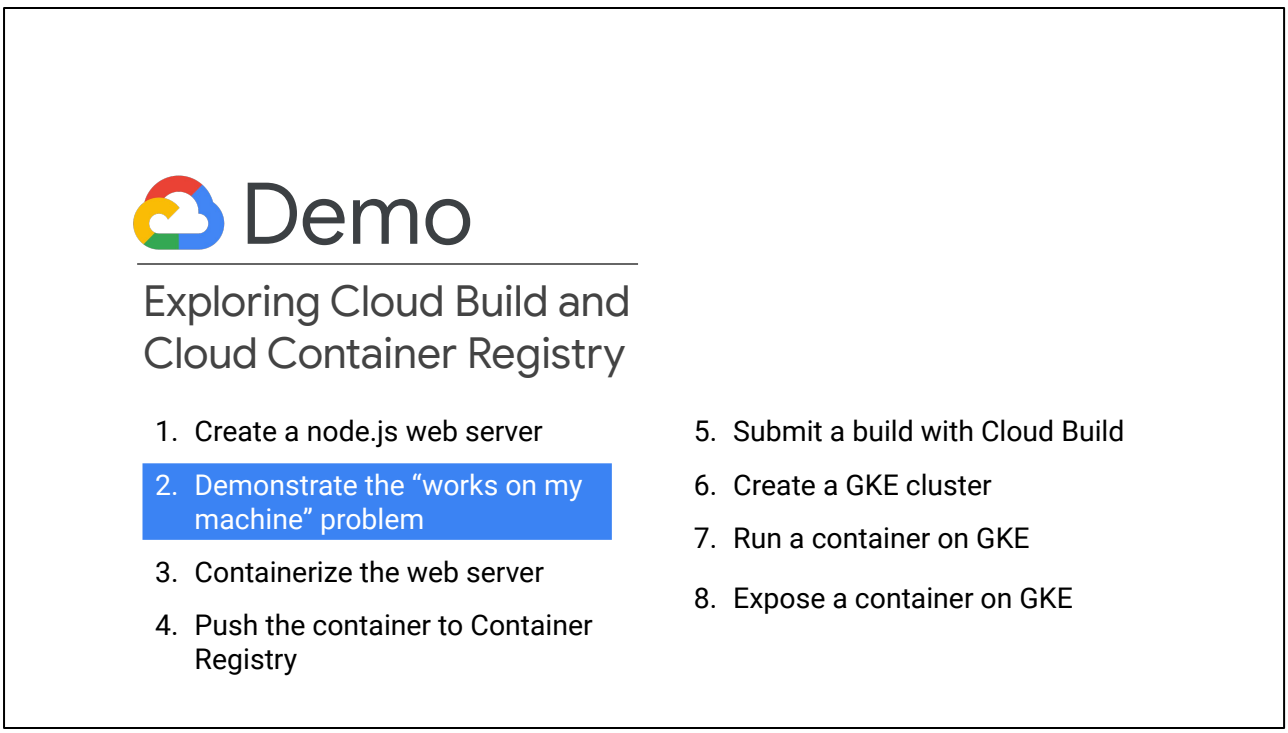
Demo 06a
https://docs.google.com/document/d/1QcM3kEIlHZGdOZO2hdFr4GKVbAaU2HoYSeJ_nZ1TyUM/edit#

In this demo, we will explain how Docker, Kubernetes, Cloud Build, Container Registry, and Google Kubernetes Engine (GKE) work. To do this, we will write a simple web application using Node.js code.

We will run it in Cloud Shell, containerize our code using Docker, save the container to Container Registry, and deploy the container to GKE. We will then expose the container via a load balancer, automate the container build, save it using Cloud Build, and automate and deploy using a deployment file.

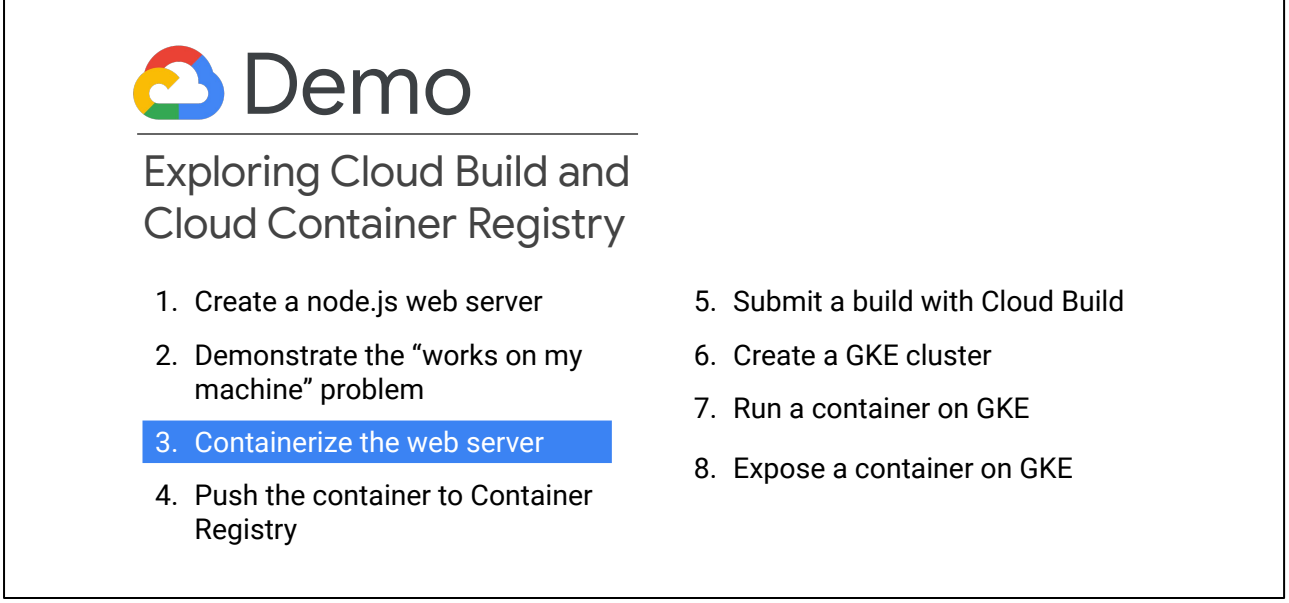


First, let's write the code to run a web server with Node.js. To write our code, we’ll open Cloud Shell. Our Node.js program will be in a file called app.js. Let's create this file now and paste in the code. This is just a simple HTTP server. When a request comes in, it’s going to log the phrase "Hello containers". Our application is going to listen on Port 8080. We’ll save and then run the program in Node. It says our server is running, so let's test it out. We know it's running on Port 8080, so we can use Web Preview from Cloud Shell to see if it works. Here we'll preview on Port 8080. In the response, we can see our phrase "Hello containers", which means our server is working as expected. Now, let’s stop the program.



Next, we’ll create a Compute Engine virtual machine and try deploying our Node.js application to it. In the Google Cloud Console, we'll go to Products and Services, then Compute Engine, and create our machine which we’ll call “Not My Machine” and set the zone to “US Central-1A”. We'll accept most of the rest of the defaults, but we'll allow HTTP traffic since this is going to be a web application. Then, we’ll click "Create" and give the machine a moment to start up. Now let's use Cloud Shell to copy our program to the virtual machine we’ve just created. Remember: our program is a Node.js application written in the file “app.js”. We can use the command “gcloud compute SCP” to copy our app.js file into our machine, which we called “Not My Machine”. Cloud Shell says our file has been copied successfully, so let’s SSH into the machine and see if we can get our program working. We can make sure our app.js file is there by running the “ls” command.

Now, let's see if we can run our app by using the command “Node app.js”. We’ve received an error because Node.js isn’t installed on this machine. We could configure the machine to run Node.js, but we're going to use a more automated method. This will involve packaging our application into a Docker container, deploying the container, and running it on GKE virtual machines. To create the Docker image, we must first create a file called “Dockerfile”. We can do this using nano. Shown here is the code we’ll need to create our image. Notice how we're starting with a base image, provided by Google, that includes Node.js. In the second command, we're going to copy our app.js file to the “slash app” folder in the container we're building. Finally, at the end of our code, we’re going to run our program just like we did before, using the command “Node app.js”. Let's save that file.

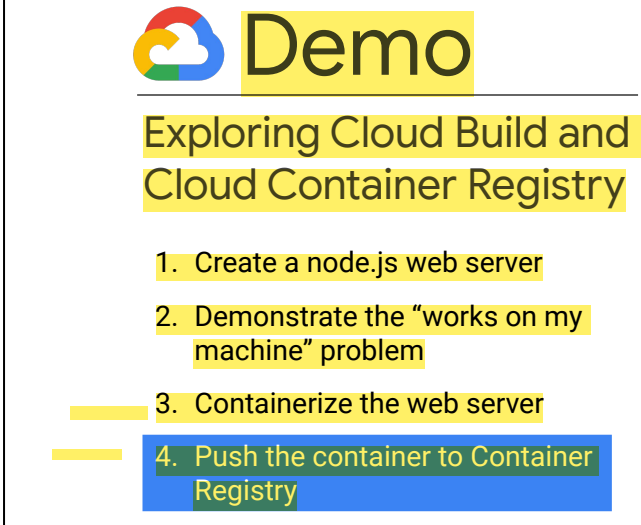


Next, we'll use Docker's “build” command to build the image. The command we'll use is “docker build” and then the name of the image: “my-web-server”. The period at the end indicates that your Docker file is in the current folder. Once our Docker image is built, we can verify that it works by typing “docker images”. In the results, we can see our image “my-web-server” along with the base image we downloaded in order to create our own.

Now, let's test our program using the “docker run” command, as shown here. We'll run our image “my-web-server”, expose port 8080, and then forward a request to that port. Remember that port 8080 is the port our application is listening on. If this works, we should be able to preview our application on port 8080. And as we can see here, our application is running and showing the phrase "Hello containers". If we want to stop the Docker image, we first need to know its ID. We can type “docker ps”. This will show all running images. The first parameter in each row is the ID of the image. To stop the image, we can use the command “docker stop” and then paste in the ID. Returning to the browser and hitting refresh, we can see that our application is no longer running.

Now, we have a container, but it's in the local machine set of Docker images. If we want to use it elsewhere, we need to send it to a well-known location. This is a Container Registry. Google runs one on GCP that is project-private; that is, your images are hosted privately. Let's upload the Docker file to the Container Registry. First, we need to tag our container. To push a container into the Container Registry, it needs to be tagged with the “gcr.io” domain followed by the Google Cloud project ID

that you're working in and the name of your container. The command shown here will do all of that for us. Notice how we're tagging “my-web-server” with the string “gcr.io”, then the project's ID, and finally the name of our image. We can verify this has worked by typing “docker images” again. This now lists three images: the first of these is the one we created originally. The second image in the list is the image we have just tagged. The third is the base image we downloaded when we created our Docker container.



Now, we're ready to push the tagged image up to the Container Registry. The command “docker push” that will push images up to Docker's website. However, we want to use the Google Cloud registry, so we're going to use the gcloud command line interface to push our Docker container. The command is shown here, with the name of the image we’ve just tagged at the end. To verify that our image was pushed into the container registry, we can return to the web console, go to Products and Services menu and scroll down to Container Registry. Now, under “Images”, we can see “my-web-server”.

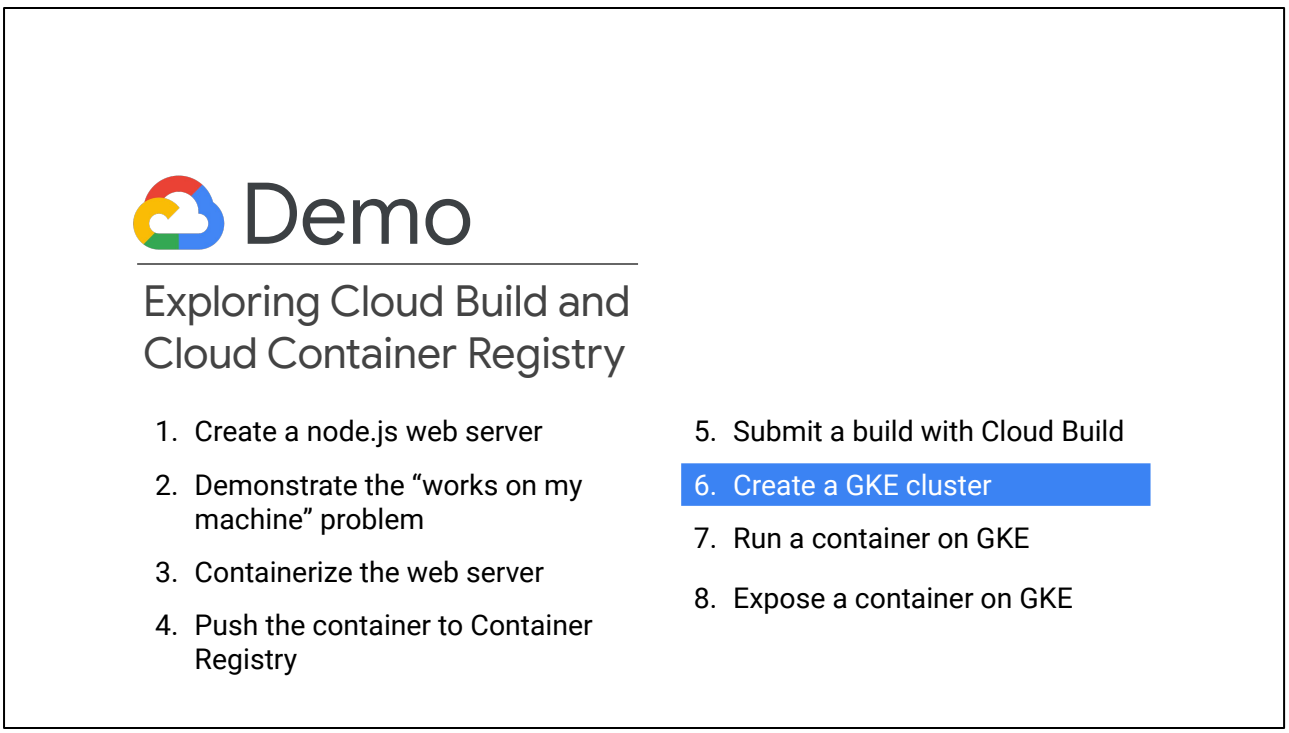
We've shown how to use Docker to build an image, and gcloud Docker to get our image into the container registry.

Exploring Cloud Build and Cloud Container Registry

1. Create a node.js web server
2. Demonstrate the “works on my machine” problem
3. Containerize the web server
4. Push the container to Container Registry
5. Submit a build with Cloud Build
6. Create a GKE cluster
7. Run a container on GKE
8. Expose a container on GKE

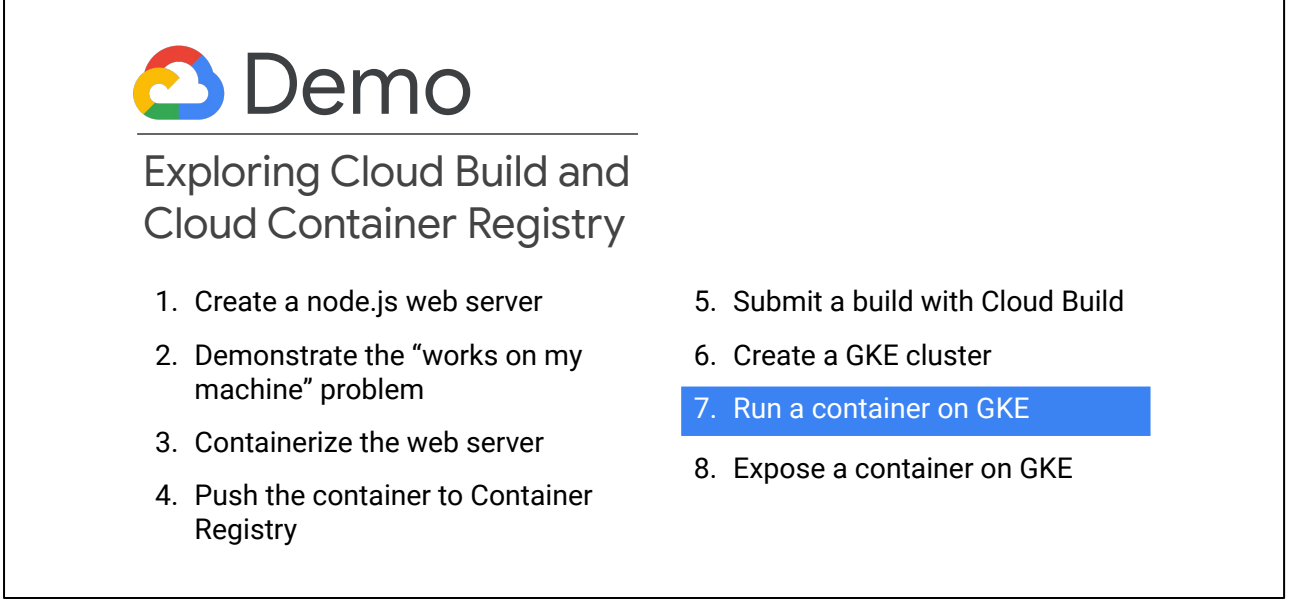
We can combine these two steps into one using Cloud Build. Let's see how that works. The command, shown here, begins with “gcloud container builds”. We’ll submit a build and follow this with the name of the image we want to build. Notice we’re using a slightly different name to the image we uploaded last time. Instead of “my-web-server”, we’ll call it “my-CB-web-server”, “CB” standing for Cloud Build, and we’ll build the image in the local folder.

Using one build command, we have built our Docker container and then uploaded it to Container Registry. Now, we'd like to run it somewhere. On Google Cloud, this is the job of GKE. We can also use Compute Engine directly, but then we would have to configure the virtual machines to run Docker. GKE provides a more automated way of running our application.



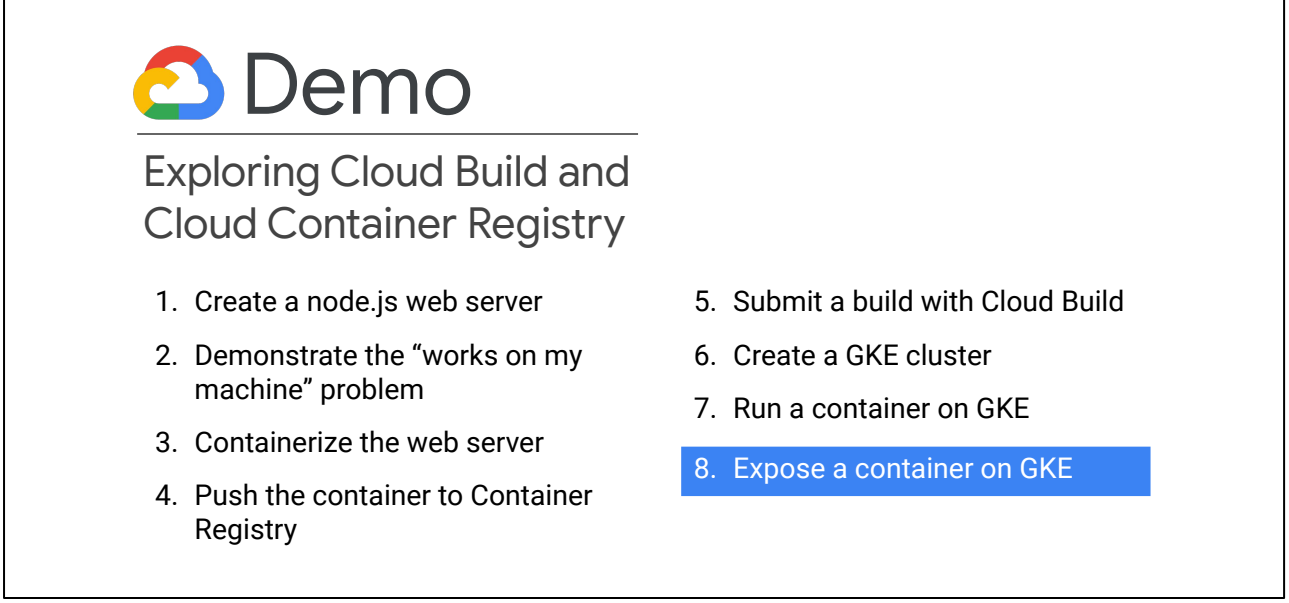
Let's start by creating a GKE cluster. We’ll go to "Products and Services" and choose GKE. First, we'll create a cluster of machines. We’ll call it demo cluster and set the zone to “US Central 1A”. Then, we'll accept all the rest of the defaults and click Create. It'll take some time for the cluster of machines to be ready.

The cluster appears to be ready, so let's click on it. Then, we can click the link that says “Connect to the cluster”. After doing this, we will see a command, as shown here, that we can enter in order to connect to that cluster and control it with Kubernetes. Let's copy the command to the clipboard. We can then return to Cloud Shell where we we’ll paste in the command.



Now that we've made our connection to the cluster, let’s enter some Kubernetes commands to see how they work. First, we'll enter “kubectl [cube controller] get pods”. A pod is like a collection of containers. At this point, we don’t have any pods. We can also type “kubectl get services”. We have one service up and running: the Kubernetes service. We can also type “kubectl get deployments”. At this point, we haven’t created any deployments, so we’ll see the message “No resources found”.

Now let's try to deploy our application. We’ll use the Kubernetes command line interface to deploy the app. To do this, we’ll type “kubectl run”, followed by the name of our deployment “my-web-server-gke”, then the image we’ve deployed to Container Registry.



Lastly, we’ll specify a port to expose. In this instance, we’ll use port 8080.

It says our deployment was created. Let's try running “kubectl get pods” again. This time, we have one pod. It's not quite ready yet, so let's wait a second and try again. Now it is ready. We can then type “kubectl get deployments” again, which will show a single deployment. We need to be able to talk to our deployment. To do this, we’ll set up a load balancer using a Kubernetes command. The beginning of the command is “kubectl expose deployment my-web-server-gke”. We then need to specify that we want to create a load balancer. The load balancer is going to listen on port 80 and will forward requests to port 8080, where our deployment is listening. Now let's run “kubectl get services” again. Notice that we now have a load balancer, but our external IP address is pending. We’ll need to wait a few seconds and then try the command again.

Now, we have an external IP address for the load balancer. To make sure our application is running, we can copy this IP address to the clipboard and paste it into the browser. As we can see, the page is displaying our “Hello containers” message, which means our application is working. Now, we can return to Cloud Shell and type in “kubectl get pods” again. We now have one of one instances ready. If we want to scale this up, we can run the command “kubectl scale deployment my-web-server-gke” and set the number of replicas equal to three. Cloud Shell says our deployment was scaled, so we can call “get pods” again and we’ll see we now have three pods, only one of which is running. The two new pods we created are still being built. Let's try running the “get pods” command again. Now, we can see that all

of our pods are ready. We will still need to make any requests through the load balancer. Returning to the browser, if we hit refresh, our application will still running, but now we have three instances of it.

Exploring Deployment Manager

1. Clone source code in Cloud Shell
2. Review and use a Deployment Manager configuration
3. Harness Deployment Manager templates

Demo 06b
https://docs.google.com/document/d/1rJtqLdTsuVW_41evZuXQYzFUIoO1uCaz2DMFLznsbYc/edit#heading=h.weq2hvwxyy3h
Welcome to this demo on Exploring Deployment Manager. Deployment Manager provides a reproducible coordination of infrastructure deployment in Google Cloud. In this demo, we will review some simple Deployment Manager examples, use Deployment Manager to create infrastructure, manage deployed infrastructure, and delete deployments. Deployment Manager supports the idea of configuration as code or reproducible infrastructure provisioning. In the cloud, you don't want to nurse infrastructure. But to be able to delete infrastructure and spin it up again, you'll need a way of guaranteeing a reproducible set of machines, networks, and code. You can use simple shell scripts to provision your infrastructure; however, for real-world production scenarios, there are complex requirements that make scripting difficult. For example, it might be useful to have templates that enable you to pass in variables to customize resource provisioning, like creating a set of Compute Engine resources in different regions to support different territories. Or you may want to coordinate multiple resources like capturing the IP address of a Cloud SQL instance so it can be passed to Compute Engine metadata. There are lots of products in this space: Chef, Puppet, Ansible, and Terraform are examples. However, Deployment Manager is the native product from Google that allows you to perform infrastructure-as-code-based resource provisioning on Google Cloud. Deployment Manager supports a wide variety

of Google Cloud resources including Compute Engine, GKE, App Engine, Cloud Functions, Pub/Sub and more. It supports simple configuration using YAML files and more complex templates via Python code and Jinja templates. Let's review a simple Deployment Manager configuration file, then deploy the resources described in the configuration. From the Google Cloud management console, let's go into the Deployment Manager service.

You'll notice that, at this point, we have no current deployments, so let's go ahead and create one. To do this, we'll open Cloud Shell.

A simple Deployment Manager template has already been created for us. Let's clone the Git repository that contains that template.

Now we'll cd into the folder that contains our template. If we take a look at the template, we can see that it is just a text file in YAML format. We're going to configure a set of resources. The name of our template will be "my-dep-man-vm", as shown here. We'll be configuring a virtual machine, so the type is "compute instance". Shown below the type are the properties of the virtual machine. For the purposes of this demo, we'll create our virtual machine in the zone "us-central1-a". The machine type is "n1-standard-1". We're going to create a Debian machine, so this should be specified in the "source image" property. The machine also needs a network interface, which we can configure in this file.

Now, let's deploy the environment using the command "gcloud deployment-manager deployments create" followed by the name of our deployment "my-deployment" and the name of the YAML file we have just looked at that stores its configuration. Let's try it out. It'll take some time for the deployment to be ready. While we're waiting, let's go back to Google Cloud and refresh the Deployment Manager page. We have one deployment that's in the process of starting up. When the deployment is ready, click on it. Shown here is an overview of our deployment. Let's click on the virtual machine we have just deployed. We can see that it's a Compute Engine instance. If we click on MANAGE RESOURCE, we will be brought directly to the VM Instances section of Compute Engine. This page shows us information about that particular instance. From here, we can even SSH into the instance.

Now let's see how easy it is to remove the deployment. Going back to Google Cloud Shell, we'll enter the following command: "gcloud deployment-manager deployments delete" followed by the name of the deployment we just created.

Again, this may take some time, so while we're waiting, let's go back to the

management console and click Refresh. In the VM Instances section, we can see the message “This machine is being stopped” if we hover over our VM instance. And if we return to the Deployment Manager service, we'll see that this deployment is being deleted.

Now let's look at a slightly more complex example. Deployment Manager allows you to create dynamic templates that combine Deployment Manager YAML files with Python code and Jinja templates. These templates work like functions. You can pass arguments to them and they will prepare a resource based on the parameters. Let's return to Cloud Shell and see how it works. As before, some demo files have been created for us. Let's cd into the folder that contains our files and open the YAML file. Notice how the YAML file now looks simpler. It refers to our Python file “vm_config.py”. If we take a look at the Python file, we’ll see that it defines a number of different parameters that can be passed into the template.

Finally, there's the schema file, which we will look at now. In this file, we're setting the properties dynamically, so we can specify the zone we want and also the startup script that we want to run. Let's run this deployment with the command "gcloud deployment-manager deployments" followed by “create my-second-deployment” and then "--config vm.yaml". This may take a few minutes. Returning to Deployment Manager, we can click Refresh and see that our deployment is complete. As we did previously, we can click on our deployment and see an overview of what happened. When we deployed the virtual machine, it ran a startup script, which was one of the parameters we passed to it.

Let's take a look at that virtual machine by clicking on MANAGE RESOURCE. This takes us into the VM instances section. Since our startup script deployed a web server, we can check in our web browser to see if it worked. And here’s our web server.

Thanks for watching this demo on Cloud Deployment Manager. In this demo, we reviewed some simple Deployment Manager examples, we used Deployment Manager to create infrastructure, we saw how to manage deployed infrastructure, and we saw how to delete deployments.

Deploying the Application into Google Kubernetes Engine

Duration: 45 minutes

In this lab, you will deploy an application into Google Kubernetes Engine.

Lab objectives

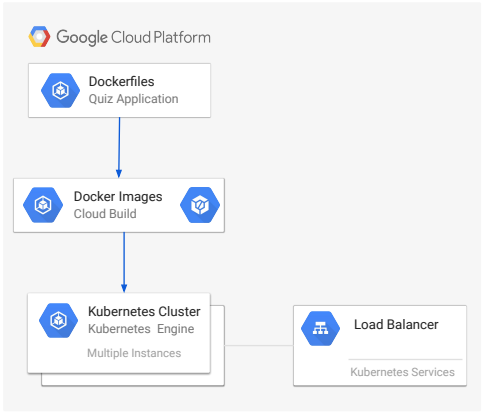
Create Dockerfiles to package up the Quiz application frontend and backend code for deployment

Harness Cloud Build to produce Docker images

Provision a GKE cluster to host the Quiz application

Employ Kubernetes deployments to provision replicated pods into GKE

Leverage a Kubernetes service to provision a load balancer for the Quiz frontend



Google Cloud

In the lab, you will create Docker files to describe how your application's container image should be built, you'll use Cloud Build to build Docker images and store the images in Container Registry, you'll launch a Kubernetes cluster in GKE and deploy your application's container image in the cluster. Finally, you'll expose the front-end service by using a load balancer.

Summary

Mike Truty
Technical Curriculum Lead, Cloud Application Development

To summarize, implement continuous integration and delivery pipelines to enable repeatable and reliable deployments. Treat your infrastructure as code. This approach will enable you to create and store versions of your infrastructure. In case of unexpected problems, you can roll back to a previous version of the infrastructure and application image that works. Using Cloud Build, you can build Docker images for your application from source code located in Cloud Storage, Cloud Source Repositories, GitHub, or Bitbucket. Cloud Build automatically pushes these images to Container Registry. Your application's binaries as well as dependencies will all be packaged and released as one unit. Your application can then function consistently in any environment. You can use Deployment Manager to stand up your Google Cloud infrastructure including Compute Engine VMs, GKE, and more.


Google Cloud