Solutions (https://cloud.google.com/solutions/) Solutions

Automated image builds with Jenkins, Packer, and Kubernetes

Creating custom images to boot your Compute Engine instances or Docker containers can reduce boot time and increase reliability. By pre-installing software into a custom image, you can also reduce your dependency on the availability of 3rd party repositories that are out of your control.

You choose how much software and configuration to include in your custom images. On one end of the spectrum, a minimally-configured image—referred to as a foundation image in this document—contains a base OS image (like Ubuntu 14.10) and might also include basic software and configuration. For example, you can pre-install language runtimes like Java or Ruby, configure remote logging, or apply security patches. A foundation image provides a stable baseline image that can be further customized to serve an application.

At the other end of the spectrum, a fully-configured image—referred to as an immutable image in this document—contains not only a base OS or foundation image, but also everything required to run an application. Runtime configuration, such as database connection information or sensitive data, can be included in the image, or it can be provided via the environment, metadata, or key management service at launch time.

The process of building images has a lot in common with building software: you have code (Chef, Puppet, bash, etc.) and the people who write it; a build happens when you apply the code to a base image; a successful build process outputs an artifact; and you often want to put the artifact through some tests. Many of the best practices that apply to building software also apply to images: you can version control to manage image configuration scripts; trigger builds when changes are made to those scripts; perform image builds automatically; and version and even test the resulting image artifacts when builds complete.

What you will learn

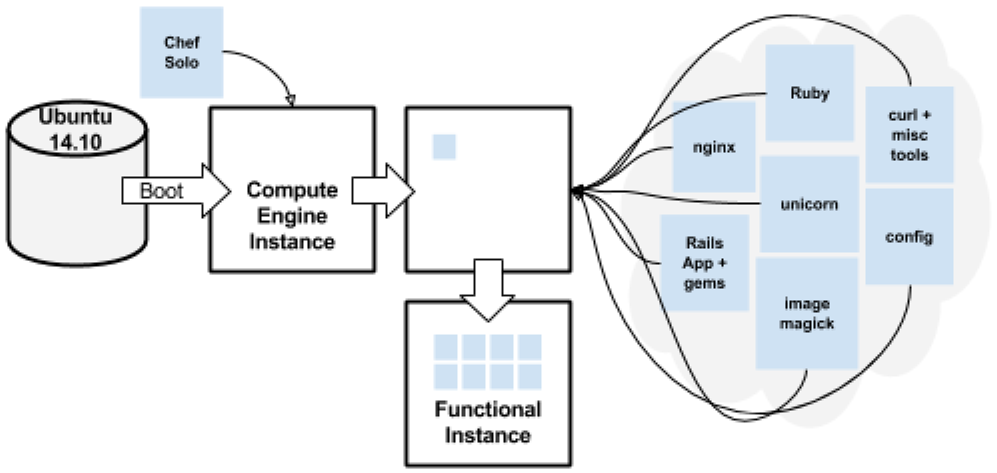
In this solution you will learn about two general approaches to building custom images and how to use several popular open source tools—including Jenkins, Packer, Docker, and Kubernetes—to create an automated pipeline to continuously build images. This pipeline integrates with the Cloud Source Repositories in Google Cloud Platform (GCP) and outputs both Compute Engine images as well as Docker images.

You will learn how to build both foundation and immutable images and learn best practices for managing access to these images across multiple projects in GCP. Finally, a comprehensive tutorial at the end of the document lets you deploy and use an open-source, reference implementation of the solution.

Image types

In the Scalable and resilient web applications (https://cloud.google.com/solutions/scalable-and-resilient-apps) solution, a Ruby on Rails web application is used as a reference for running web applications on GCP. The source code for that solution (https://github.com/GoogleCloudPlatform/scalable-resilient-web-app) does not use customized images; when a Compute Engine instance boots, a startup script installs Chef Solo which then installs everything required to run the application. This includes nginx, Ruby 2, cURL and other system tools, Unicorn, the Rails app and all of its gems, imagemagick, and the app config.

The following diagram describes the boot process.



The process isn’t fast, taking 10-15 minutes *for each instance* to boot depending on the download speed of the various repositories required for the packages—and that’s assuming every repository hosting those packages is online and available. In the following sections, you will consider how a foundation image and immutable image could improve the performance and reliability of the instance boot process.

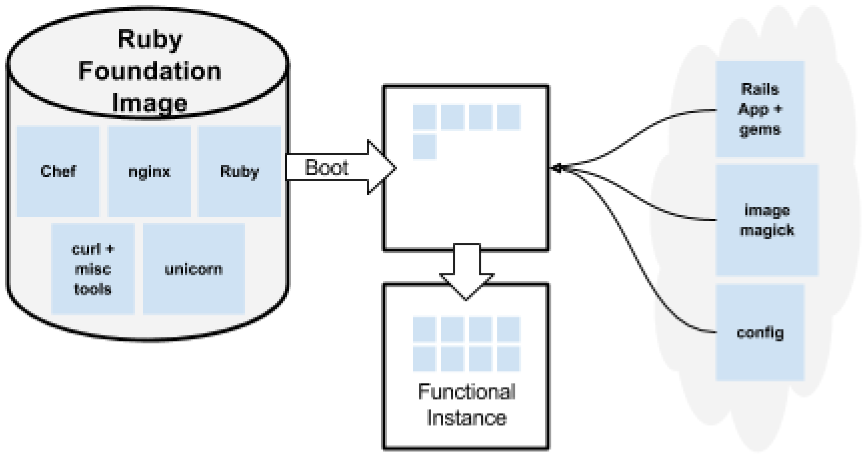
Foundation images

When creating a foundation image, you decide which software and packages to include in the image. Here are a few things to consider when making that decision:

- **Installation Speed**. Big packages can be slow to download; software that has to be built from source can be time consuming; and packages with many dependencies compound the problem. Consider including these types of software and packages in your foundation image.
- **Reliability of Remote Repository**. If you don't include the software in your foundation image and instead download it at boot time, do you trust the availability of the remote repository? If that repository is unavailable during boot, will it prevent your application from functioning? To reduce your reliance on remote repositories that may be out of your control, consider including critical dependencies in a foundation image.
- **Rate of Change**. Does the software or package change very frequently? If so, consider leaving it out of a foundation image and instead storing it in a reliable, accessible location like a Cloud Storage bucket.
- **Required or Security Mandated**. If certain packages (like logging, OSSEC, etc) are mandated to run—with a specific configuration—on every instance in your organization, those packages should be installed in a foundation image that all other images extend. A security team may use a more advanced tool like Chef or Puppet to build a Docker foundation image, while downstream developers could use a Dockerfile to easily extend the foundation.

These criteria suggest that a foundation image for the Ruby on Rails application from the Scalable and Resilient Web Applications solution could include Chef Solo, nginx, Ruby, cURL and other system tools, and Unicorn. The other dependencies would be installed at boot time.

The following diagram describes the boot process with a foundation image:

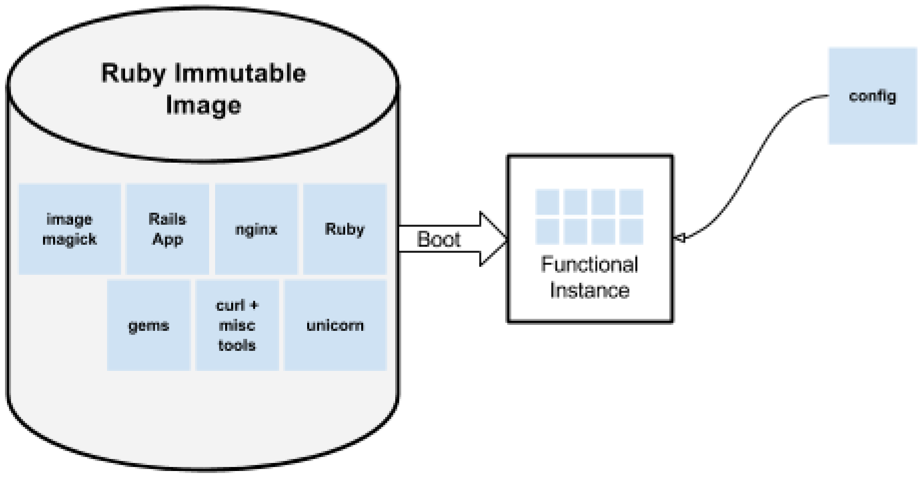


The functional instance in this example retrieves its configuration (for example, database connection string, API keys, etc.) from the Compute Engine metadata service (https://cloud.google.com/compute/docs/metadata). You may choose to use different service like etcd (https://github.com/coreos/etcd) or a simple Cloud Storage bucket to manage configuration.

Upcoming sections focus on the tools used to automate the build of the Ruby Foundation Image pictured here.

Immutable images

Unlike a foundation image, an immutable image has all of its software included on the image. When an instance or container is launched from the image, there are no packages to download or software to install. An immutable image for the Ruby on Rails application from the Scalable and Resilient Web Applications solution would include all software, and the instance would be ready to serve traffic when booted.



Configuration and immutable images

You can choose to have your application access the configuration data it needs from a configuration service or you can include all configuration in the immutable image. If you choose the latter approach, be sure to consider the security implications of including secrets in your images. If you are pushing immutable images to public repositories in the Docker Hub, they are accessible to everyone and should not contain any sensitive or secret information.

Immutable images as the unit of deployment

Using immutable images as your unit of deployment eliminates the possibility of configuration drift, where one or more instances is in a different state than expected. For example, this can happen when you apply a security patch to 100 running containers and some of them fail to update. The image becomes what you deploy when *any* change is made. If the OS requires a software patch or the logging configuration should be updated, you build a new image to include those changes and roll it out by launching new instances or containers and replacing all old ones. If you choose to bundle application configuration in an immutable image, even a simple change like updating a database connection string means creating and releasing a new image.

Architecture and implementation of an automated image building pipeline

This section includes implementation details of an automated image building pipeline that uses Jenkins, Packer, Docker, and Google Kubernetes Engine (GKE) to automatically build custom images. Each section includes an introduction, architecture diagram, and detailed analysis of the components in that diagram.

Software and services used

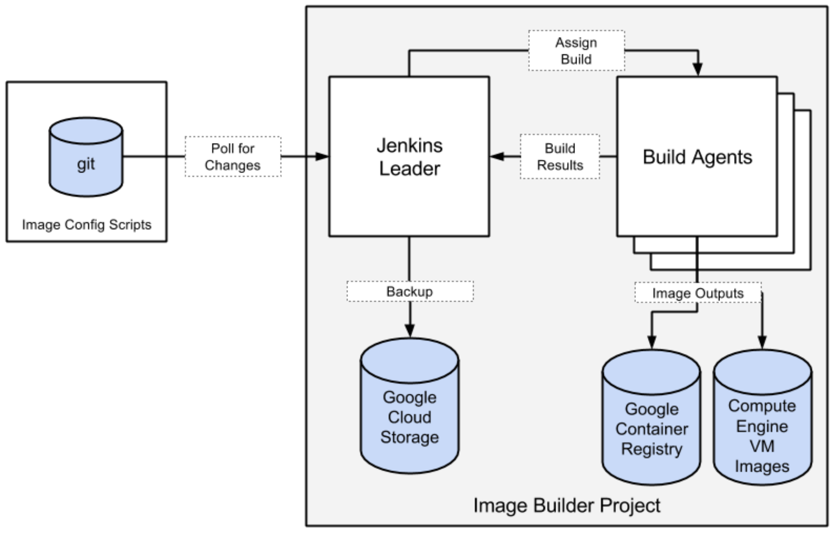
These software and services are used to create the automated image builder.

Software	Use

Software	Use
Jenkins	Jenkins is a popular open source continuous integration (CI) server. You'll use Jenkins to poll Git repositories in other projects that contain image configuration scripts, then to build images based on those repositories.
Packer	Packer is a tool for creating identical machine images for multiple platforms from a single source configuration. It supports many different configuration sources including Shell, Chef, Puppet, Ansible, and Salt, and can output images for Compute Engine, Docker, and others. Packer is used by Jenkins agents to build images from configuration in Git repositories.
Docker	Docker is an open source tool for packaging and deploying applications as containers. The Jenkins deployment (including the leader node and build agents) in this architecture and tutorial are deployed as Docker containers. The build agents also output Docker images as one of their architectures.
GKE	GKE, powered by the open source technology Kubernetes, enables you to run and manage Docker containers on GCP virtual machines.
Container Registry	Container Registry provides secure, private Docker image storage on GCP. It runs on GCP and is accessed through an HTTPS endpoint.
Compute Engine	GKE uses Compute Engine VMs to run Kubernetes and to host the Jenkins leader and build-agent containers. The Jenkins build process also outputs Compute Engine VM images in addition to Docker images.
Cloud Storage	You'll use Cloud Storage to store backups of your Jenkins configuration.
Nginx	Nginx provides reverse proxy functionality; forwarding incoming requests to the Jenkins leader web interface. It can be configured to terminate SSL connections and provide basic authentication.

Image builder overview

The following diagram shows how various components interact to create a system that automatically builds VM and Docker images.



You define a job on the Jenkins leader for each image you want to build. The job polls a source code repository, Git in this illustration, that contains configuration scripts and a Packer template describing how to build an image. When the polling process detects a change, the Jenkins leader assigns the job to a build agent. The agent uses Packer to run the build, which outputs a Docker image to the Container Registry and a VM image to Compute Engine.

Packer and configuration scripts

A Packer template and associated configuration scripts together define how to build an image. They are treated like software and stored in their own Git repository. Each image you build will have its own repository with a Packer template and config scripts.

This section provides an overview of one possible Packer configuration that uses Chef to customize Ubuntu 14.04 by adding Ruby and rbenv. For complete coverage of Packer, check out its excellent documentation at https://www.packer.io/docs (https://www.packer.io/docs).

Image naming and packer variables

The image builder builds an image any time a change is made to the Git repository containing the image's Packer template and config scripts. It's a good idea to name or tag images with the

Git branch and commit ID from which they were built. Packer templates allow you to define variables and provide values for them at runtime:

```
{
...
  "variables": {
      "Git_commit": "",
      "Git_branch": "",
      "ruby_version_name": "212",
      "project_id": "null"
  }
...
}
```

The Jenkins build agent can find the Git branch and commit ID and provide them as variables to the Packer command line tool. You'll see this in action later in the tutorial section of this document.

Programmatic configuration with provisioners

A Packer template defines one or more provisioners that describe how to use a tool like Chef, Puppet, or shell scripts to configure an instance. Packer supports many provisioners; see the table of contents in the Packer documentation (https://www.packer.io/docs) for a complete list. This snippet defines a **chef-solo** provisioner with cookbook paths and recipes to run to configure an image:

```
{
  ...
  "provisioners": [
    {
      "type": "chef-solo",
      "install_command": "apt-get install -y curl && curl -L https://www.opscode.com
      "cookbook_paths": ["chef/site-cookbooks"],
      "run_list": [{{
        "recipe[ruby]",
        "recipe[ruby::user]",
        "recipe[ruby::ruby212]"
      ]
    }
  ],
  ...
}
```

The chef cookbook and recipes are stored in the same Git repository as the Packer template.

Defining image outputs with builders

The `builders` section of the template defines where provisioners will run to create new images. To build both a Compute Engine image and a Docker image, define two builders:

```
{
  "variables": {...},
  "provisioners": [...],
  "builders": [
    {
      "type": "googlecompute",
      "project_id": "{{user `project_id`}}",
      "source_image": "ubuntu-1410-utopic-v20150202",
      "zone": "us-central1-a",
      "image_name": "{{user `ruby_version_name`}}-{{user `Git_branch`}}-{{user `Git_
    },
    {
      "type": "docker",
      "image": "ubuntu:14.10",
      "commit": "true"
    }
  ],
 ...
}
```

The `googlecompute` builder includes a `project_id` attribute that indicates where the resulting image will be stored. The `image_name` attribute, which assigns a name to the resulting image, concatenates variables to create a name with information about the image: the version of Ruby, the Git branch, and the Git commit ID that was used to build the image. A sample URI for an image created by the `googlecompute` builder can look like the following:

```
https://www.googleapis.com/compute/v1/projects/image-builder-project-name/global/ima
```

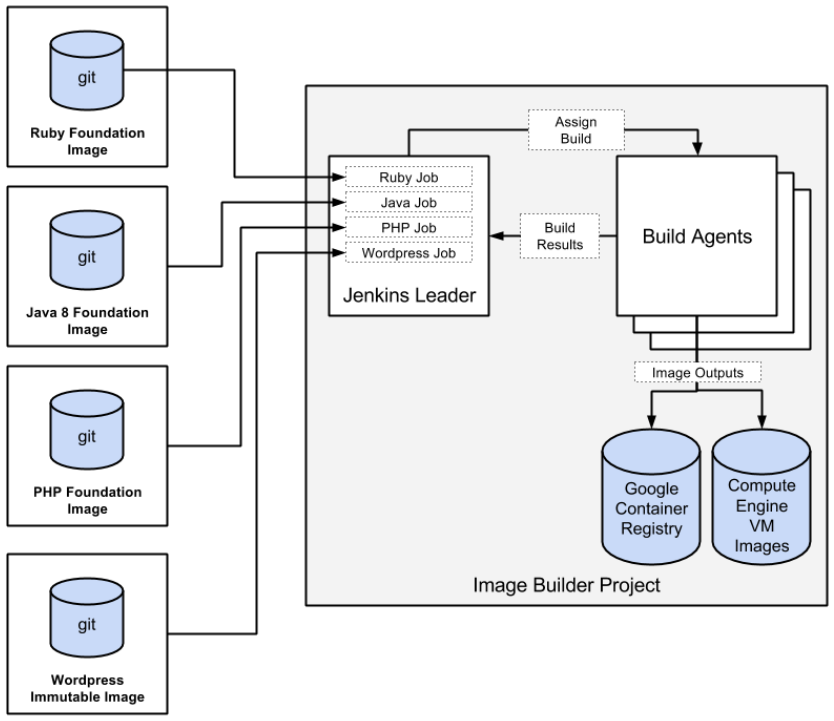
The `docker` builder should include a `post-processors` attribute to tag the image with the Docker registry and repository it will be pushed to:

```
{
  "variables": {...},
  "provisioners": [...],
  "builders": [...],
  "post-processors": [
    [
      {
        "type": "docker-tag",
        "repository": "gcr.io/{{user `project_id`}}/ruby212",
        "tag": "{{user `Git_branch`}}-{{user `Git_commit`}}",
        "only": ["docker"]
      }
    ]
  ]
}
```

This post-processor will tag the image for storage in Container Registry using the `project_id` provided when the build is run. After this Docker image is pushed, you could retrieve it:

```
docker pull gcr.io/image-builder-project-name/ruby212:master-9909043
```

Each image you want to build will have a Packer template and config scripts in its own source repository, and the Jenkins leader will have a job defined for each, as shown in the following diagram.



One advantage of using Jenkins and Packer together is that Jenkins can detect and respond to any updates you make to your Packer templates or configuration scripts. For example, if you update the version of Ruby installed in your Ruby Foundation Image, the Jenkins leader responds by assigning an agent to clone the repository, run Packer against the template, and build the images.

The tutorial at the end of this solution will cover in detail the process of configuring a Jenkins job to execute the Packer build.

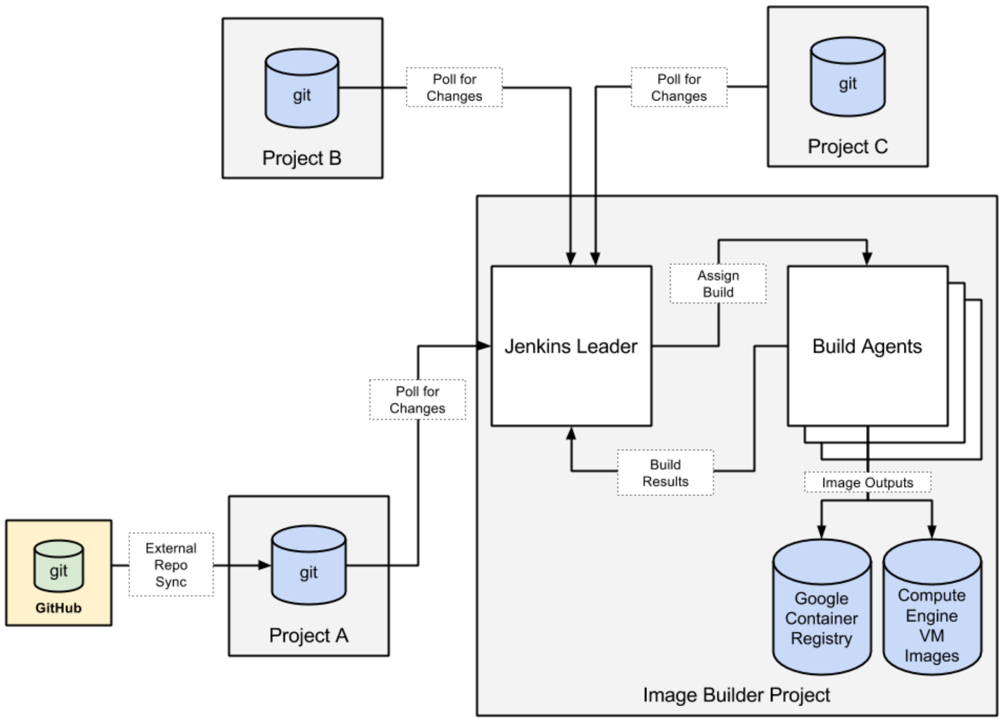
Project isolation

The Jenkins leader and build agents run together in the same Cloud Platform project, and the images they create are stored in this project. Projects allow you to isolate applications by function. There is no charge for a project; you are only charged for the resources you use. In this solution the Jenkins infrastructure will run in its own project, separated from the source control repositories it uses. Jenkins backups—discussed in an upcoming section—are stored in a Google Cloud Storage bucket inside the project. This allows Jenkins to act as an "image hub", sharing images out to other projects, while allowing other projects to maintain their own code repositories with separate access controls.

Building and sharing images across an organization

To facilitate the sharing of images, this solution places each build image stored in Git into a separate image configuration project. This separation provides project isolation between the image builder project and the build images. With this hub-and-spoke architecture, where the image builder project is the hub and the image configuration projects are the spokes, separate teams can more easily own and manage the image configurations.

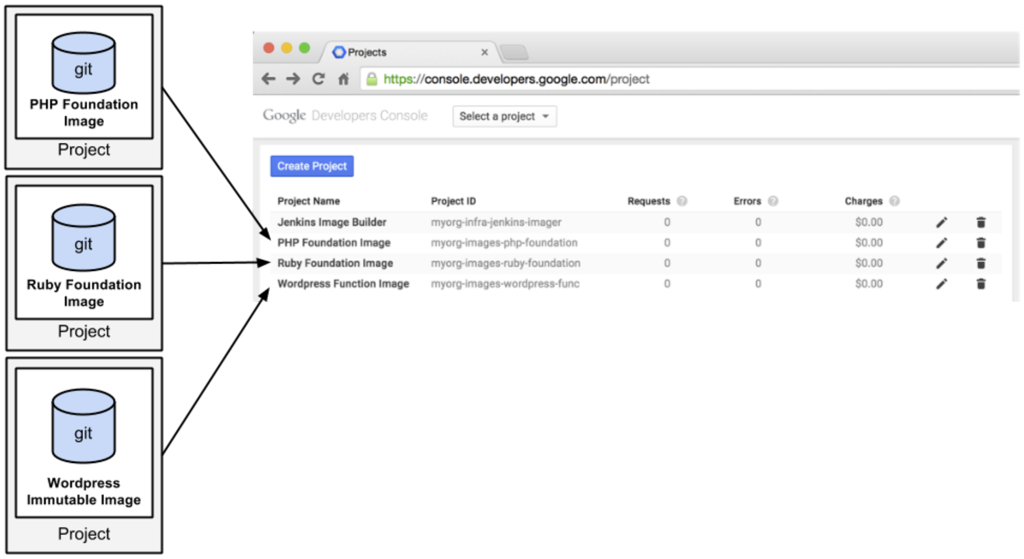
This hub-and-spoke architecture is illustrated in the following diagram.



Access control (granting the Jenkins cluster access to each image project, and granting other projects access to the images built by Jenkins) will be discussed below.

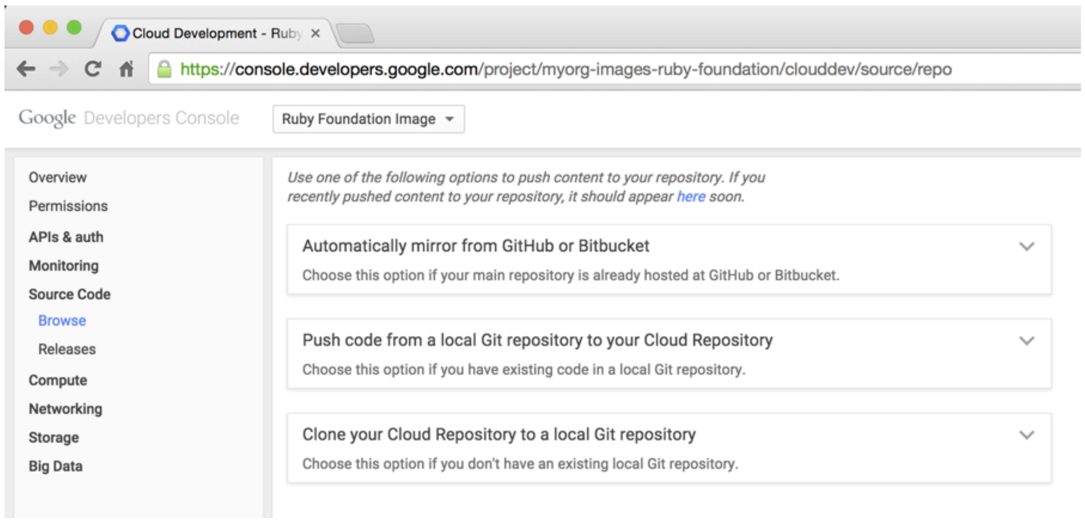
One project per image

Each project you create has a dedicated Git-based Cloud Repository. There is no limit to the number of projects you create, and you only pay for the resources, such as Compute Engine instances, that you use in a project. For example, if you have PHP, Ruby, and Wordpress images, each would have its own project visible in the Google Cloud Platform Console, as shown in the following diagram.



Project Name	Project ID	Requests	Errors	Charges
Jenkins Image Builder	myorg-infra-jenkins-imager	0	0	$0.00
PHP Foundation Image	myorg-images-php-foundation	0	0	$0.00
Ruby Foundation Image	myorg-images-ruby-foundation	0	0	$0.00
Wordpress Function Image	myorg-images-wordpress-func	0	0	$0.00

A project's Cloud Repository is accessible from the **Source Code** menu item. For new projects, you choose how to initialize the repository: you can mirror an existing GitHub or Bitbucket repository, push an existing local Git repository, or create a new local Git repository from Cloud Source Repositories, as shown in the following image.



Use one of the following options to push content to your repository. If you recently pushed content to your repository, it should appear here soon.

Automatically mirror from GitHub or Bitbucket

Choose this option if your main repository is already hosted at GitHub or Bitbucket.

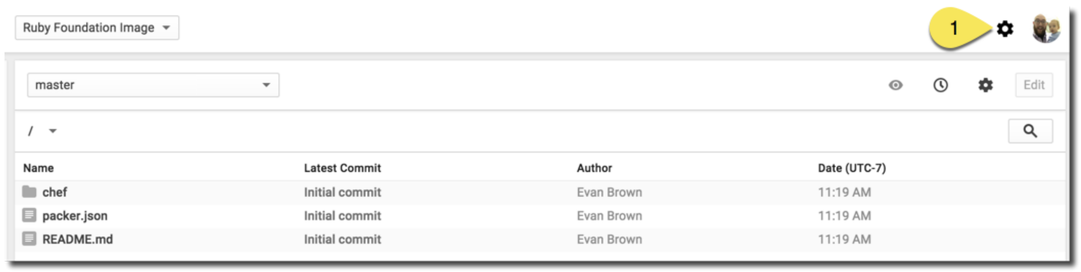
Push code from a local Git repository to your Cloud Repository

Choose this option if you have existing code in a local Git repository.

Clone your Cloud Repository to a local Git repository

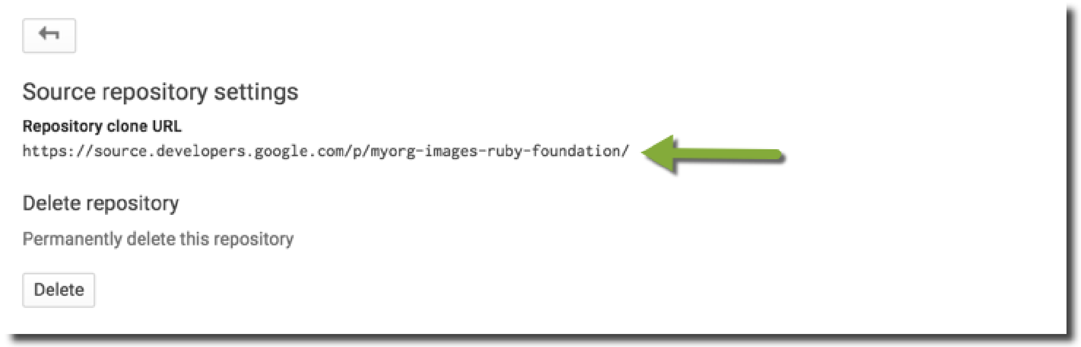
Choose this option if you don't have an existing local Git repository.

The following image shows the Ruby Foundation Image project initialized with a Packer template and Chef recipes defining the build.



Name	Latest Commit	Author	Date (UTC-7)
chef	Initial commit	Evan Brown	11:19 AM
packer.json	Initial commit	Evan Brown	11:19 AM
README.md	Initial commit	Evan Brown	11:19 AM

To view the URL for the repository, click **Settings**. You'll need this URL when creating a build job for the repository on the Jenkins leader, as shown in the following image.



Source repository settings

Repository clone URL

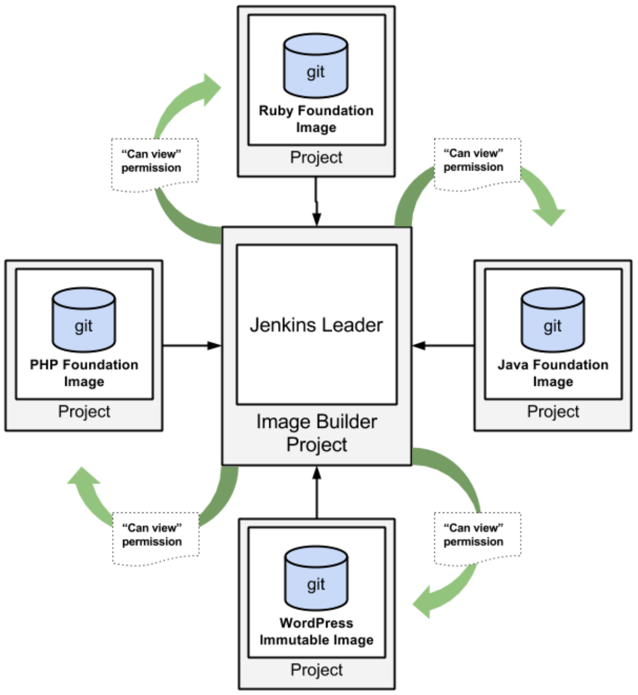
https://source.developers.google.com/p/myorg-images-ruby-foundation/

Delete repository

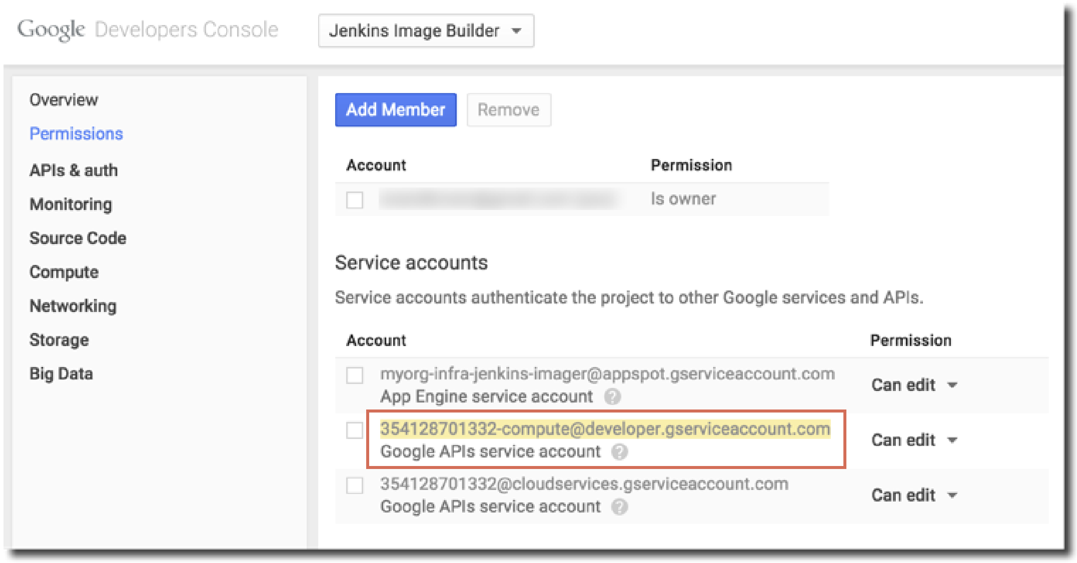
Permanently delete this repository

Cloud repository access control

The Jenkins image builder needs **Can view** permissions to each image configuration project's Cloud Repository. The following diagram shows a simplified view of the hub-and-spoke architecture shown earlier.



Each project must grant access to the Jenkins image builder project using the image builder project's compute service account email address. That address format is `\{PROJECT_ID\}-compute@developer.gserviceaccount.com` and is available to copy in the Permissions section of that project in the GCP Console, as shown in the following image.



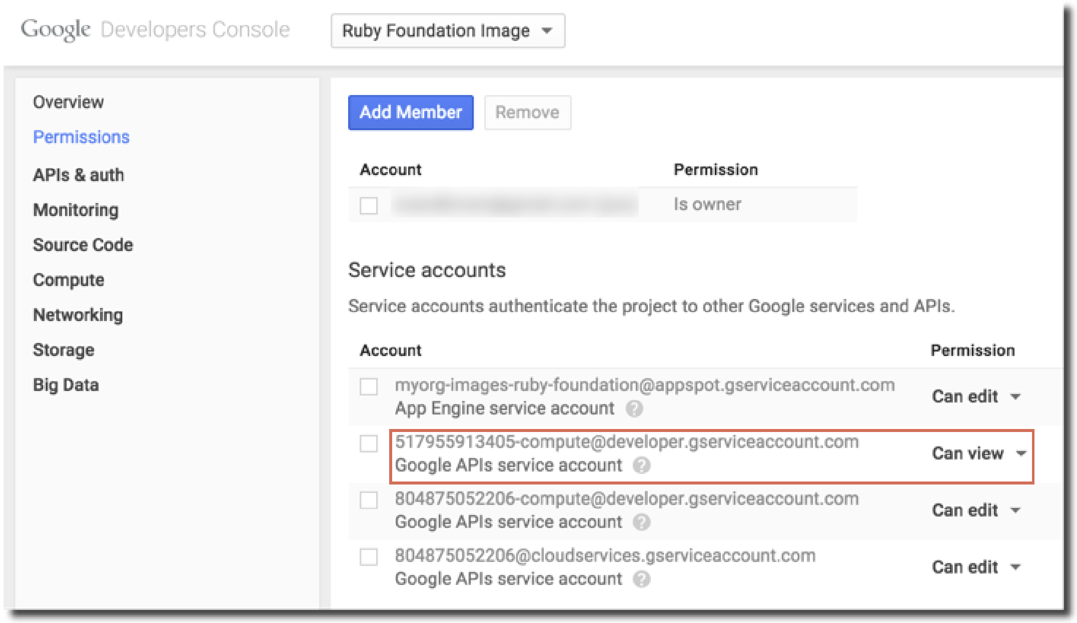
Account	Permission
[redacted]	Is owner

Service accounts

Service accounts authenticate the project to other Google services and APIs.

Account	Permission
myorg-infra-jenkins-imager@appspot.gserviceaccount.com App Engine service account	Can edit
354128701332-compute@developer.gserviceaccount.com Google APIs service account	Can edit
354128701332@cloudservices.gserviceaccount.com Google APIs service account	Can edit

After you have the compute service account email address for the project running the Jenkins image builder, go to the **Permissions** section of each project with a Cloud Repository that you want to build images from, select **Add Member**, and grant **Can view** permission, as shown in the following image.



Account	Permission
[redacted]	Is owner

Service accounts

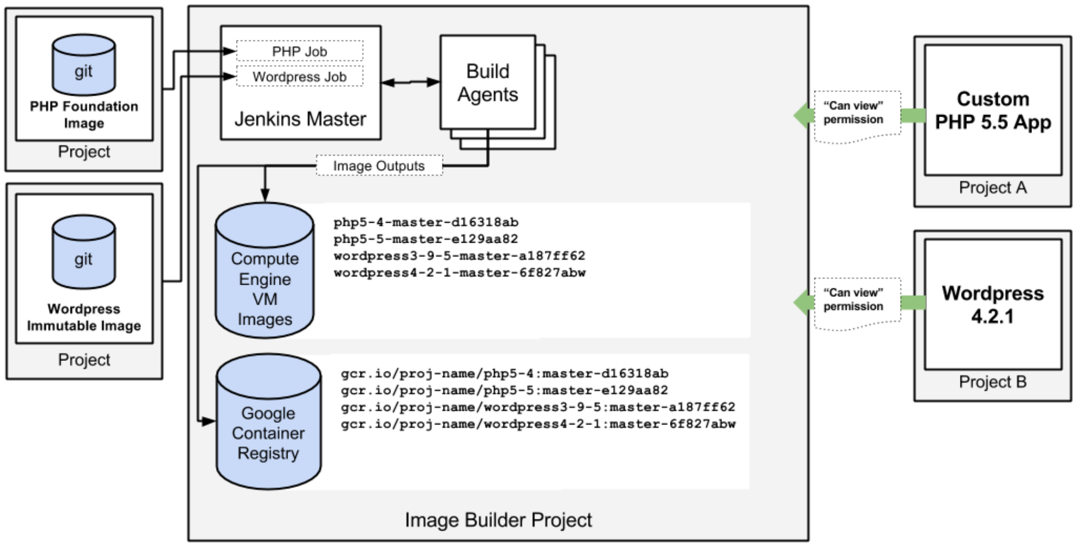
Service accounts authenticate the project to other Google services and APIs.

Account	Permission
myorg-images-ruby-foundation@appspot.gserviceaccount.com App Engine service account	Can edit
517955913405-compute@developer.gserviceaccount.com Google APIs service account	Can view
804875052206-compute@developer.gserviceaccount.com Google APIs service account	Can edit
804875052206@cloudservices.gserviceaccount.com Google APIs service account	Can edit

The Jenkins leader running in the image builder project will now be able to poll and pull from the Cloud Repository in these projects, and build new images as changes are committed.

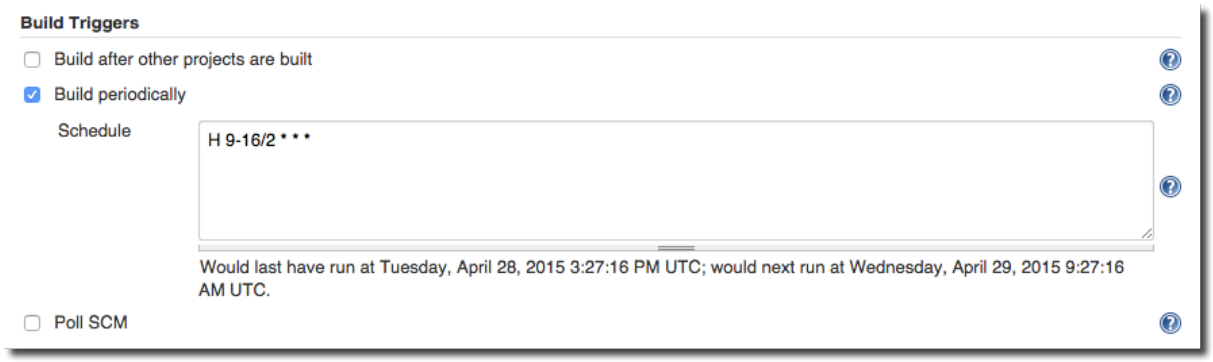
Sharing Compute Engine and Docker images

The Compute Engine and Docker images created by the image builder are stored in the same project as the image builder. The images will be used by applications in other projects to launch Compute Engine instances and Docker containers, and each application project that wishes to access these images must have **Can view** permission to the image builder project. Follow the process defined in the previous section, this time locating the compute service account of each application project and adding it as a member with **Can view** permissions to the image builder project, as shown in the following diagram.


git
PHP Foundation Image
Project
git
Wordpress Immutable Image
Project
PHP Job
Wordpress Job
Jenkins Master
Build Agents
Image Outputs
Compute Engine VM Images
php5-4-master-d16318ab
php5-5-master-e129aa82
wordpress3-9-5-master-a187ff62
wordpress4-2-1-master-6f827abw
Google Container Registry
gcr.io/proj-name/php5-4:master-d16318ab
gcr.io/proj-name/php5-5:master-e129aa82
gcr.io/proj-name/wordpress3-9-5:master-a187ff62
gcr.io/proj-name/wordpress4-2-1:master-6f827abw
Image Builder Project
"Can view" permission
Custom PHP 5.5 App
Project A
"Can view" permission
Wordpress 4.2.1
Project B

Jenkins backup and restore

The Jenkins leader includes a pre-defined job for periodic backup of the Jenkins configuration and job history to Google Cloud Storage. By default the job runs periodically (once every two hours, every weekday), as shown in the following image.



Build Triggers

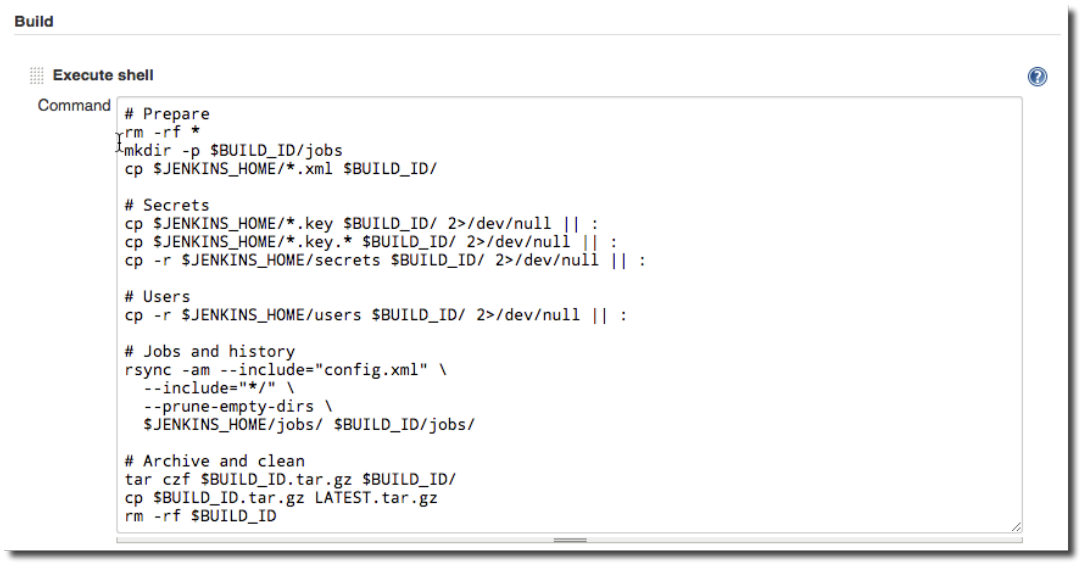
- [ ] Build after other projects are built
- [x] Build periodically

Schedule: H 9-16/2 * * *

Would last have run at Tuesday, April 28, 2015 3:27:16 PM UTC; would next run at Wednesday, April 29, 2015 9:27:16 AM UTC.

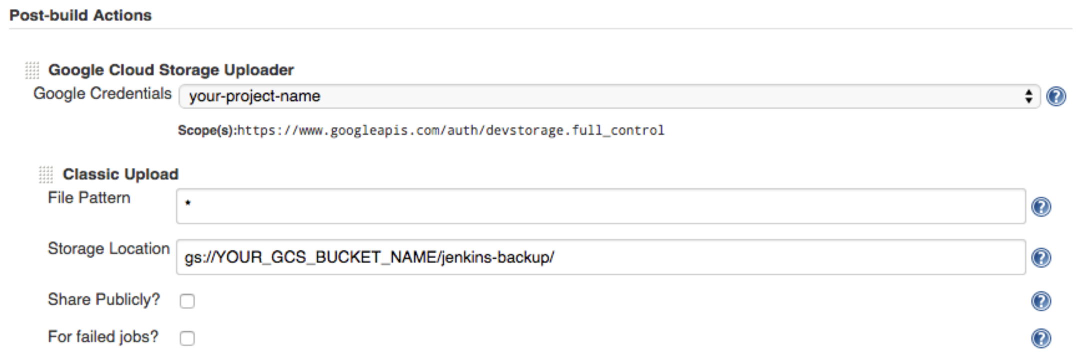
- [ ] Poll SCM

The build step of the job executes a shell script that archives secrets, users, jobs, and history into a tarball. There are two copies of the archive created: one is named with a date stamp, the other is named LATEST, allowing you to easily and automatically restore the most recent backup. You can customize this step to add or remove items to be backed up, as shown in the following image.



```
# Prepare
rm -rf *
mkdir -p $BUILD_ID/jobs
cp $JENKINS_HOME/*.xml $BUILD_ID/

# Secrets
cp $JENKINS_HOME/*.key $BUILD_ID/ 2>/dev/null || :
cp $JENKINS_HOME/*.key.* $BUILD_ID/ 2>/dev/null || :
cp -r $JENKINS_HOME/secrets $BUILD_ID/ 2>/dev/null || :

# Users
cp -r $JENKINS_HOME/users $BUILD_ID/ 2>/dev/null || :

# Jobs and history
rsync -am --include="config.xml" \
  --include="*/" \
  --prune-empty-dirs \
  $JENKINS_HOME/jobs/ $BUILD_ID/jobs/

# Archive and clean
tar czf $BUILD_ID.tar.gz $BUILD_ID/
cp $BUILD_ID.tar.gz LATEST.tar.gz
rm -rf $BUILD_ID
```

A post-build action uses the Cloud Storage plugin and Google metadata credential you created to interact with Google APIs and upload the backup archive to Cloud Storage. It uploads both the datestamp and LATEST archives. The following image shows the step definition.



Google Credentials: your-project-name

Scope(s):https://www.googleapis.com/auth/devstorage.full_control

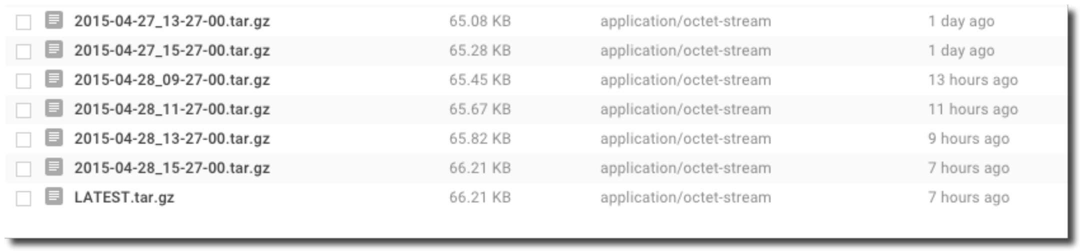
File Pattern: *

Storage Location: gs://YOUR_GCS_BUCKET_NAME/jenkins-backup/

Share Publicly? ☐

For failed jobs? ☐

The following image shows a bucket with some backups that have accumulated:



2015-04-27_13-27-00.tar.gz	65.08 KB	application/octet-stream	1 day ago
2015-04-27_15-27-00.tar.gz	65.28 KB	application/octet-stream	1 day ago
2015-04-28_09-27-00.tar.gz	65.45 KB	application/octet-stream	13 hours ago
2015-04-28_11-27-00.tar.gz	65.67 KB	application/octet-stream	11 hours ago
2015-04-28_13-27-00.tar.gz	65.82 KB	application/octet-stream	9 hours ago
2015-04-28_15-27-00.tar.gz	66.21 KB	application/octet-stream	7 hours ago
LATEST.tar.gz	66.21 KB	application/octet-stream	7 hours ago

Restoring a backup

In the same way that you use environment variables to enable SSL or basic auth on the Nginx reverse proxy (#enabling_ssl_or_basic_access_authentication) in an earlier section, you can use an environment variable to configure the Jenkins leader's replication controller definition so that it restores a backup when the service starts. The following code is a snippet from the replication controller's definition:

```
{
  "kind": "ReplicationController",
  ...
  "spec": {
    ...
    "template": {
      "spec": {
        "containers": [
            {
              "name": "jenkins",
              "env": [
                {
                  "name": "GCS_RESTORE_URL",
                  "value": "gs://your-backup-bucket/jenkins-backup/LATEST.tar.gz"
                }
              ],
             ...
           }
        ]
      }
    }
  }
}
```

The Jenkins leader Docker image checks for the existence of the `GCS_RESTORE_URL` environment variable when it starts. If found, the value is assumed to be the URL of the backup (including the **gs://** scheme) and the script uses the **gsutil** command line tool that is installed on the Jenkins leader image to securely download and restore the backup.

The restore process only happens when a container is launched. To restore a backup after you've launched a Jenkins leader, resize its replication controller to `0`, update the definition of the controller to point to the URL of the backup, then set the size back to `1`. This is covered in the tutorial.

Tutorial

The complete contents of the tutorial, including instructions and source code, are available on GitHub at https://github.com/GoogleCloudPlatform/kube-jenkins-imager (https://github.com/GoogleCloudPlatform/kube-jenkins-imager).

Last updated October 3, 2019.