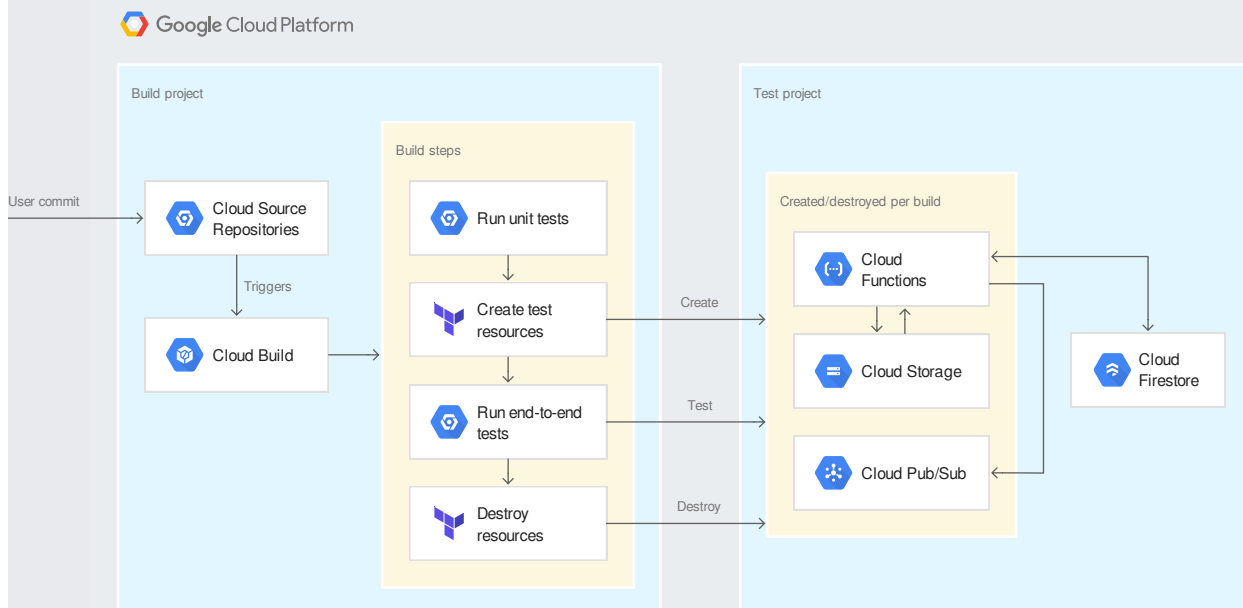


This tutorial describes how to automate end-to-end testing of an app built with [Cloud Functions](#) (/functions/). [Cloud Build](#) (/cloud-build/) runs the testing pipeline, and [HashiCorp Terraform](https://www.terraform.io/) (https://www.terraform.io/) sets up and tears down the Google Cloud resources required to run the tests. A Cloud Build trigger initiates the pipeline after each code commit.

Testability is a key consideration when you design apps and evaluate architectural choices. Creating and regularly running a comprehensive set of tests (including automated unit, integration, and end-to-end system tests) is essential to validate that your app behaves as expected. For details about how to approach each category of tests for different Cloud Functions scenarios, see the [testing best practices](#) (/functions/docs/bestpractices/testing) guide.

Creating and running unit tests is typically straightforward, because these tests are isolated and independent of the execution environment. However, integration and system tests are more complex, particularly in a cloud environment. The need for end-to-end system tests is especially relevant for apps that use serverless technologies such as Cloud Functions. These apps are often event-driven and loosely coupled, and they might be independently deployed. Comprehensive end-to-end tests are essential to validate that the functions are correctly responding to events within the Google Cloud execution environment.

The following architectural diagram shows the components you use in this tutorial.



The architecture has the following components:

- A `build` project that hosts and runs the Cloud Build pipeline.
- A `test` project that hosts Google Cloud resources for the sample app under test.
  - The app described in the [Serverless web performance monitoring](#) (/solutions/serverless-web-performance-monitoring-using-cloud-functions) tutorial is used as the sample app.
  - The Google Cloud resources for the sample app are created and destroyed for each build iteration. The Firestore database is an exception. It's created once and reused by all subsequent builds.

- Create a Cloud Build pipeline to run unit and end-to-end tests for a sample app built with Cloud Functions.
- Use Terraform from within the build to set up and destroy the Google Cloud resources that the app requires.
- Use a dedicated Google Cloud testing project to keep the test environment isolated.
- Create a [Git](http://git-scm.com) (<http://git-scm.com>) repository in Cloud Source Repositories, and add a Cloud Build trigger to run the end-to-end build after a commit.

This tutorial uses the following billable components of Google Cloud:

- [Cloud Build](/cloud-build/pricing) (/cloud-build/pricing)
- [Cloud Functions](/functions/pricing) (/functions/pricing)
- [Cloud Storage](/storage/pricing) (/storage/pricing)
- [Firestore](/firestore/pricing) (/firestore/pricing)
- [Pub/Sub](/pubsub/pricing) (/pubsub/pricing)

To generate a cost estimate based on your projected usage, use the [pricing calculator](/products/calculator) (/products/calculator). New Google Cloud users might be eligible for a [free trial](/free-trial) (/free-trial).

When you finish this tutorial, you can avoid continued billing by deleting the resources you created. For more information, see [Cleaning up](#) (#clean-up).

1. Select or create a Cloud project. This is the `test` project that hosts the sample app.

[Go to the Project selector page](https://console.cloud.google.com/projectselector2/home/dashboard) (<https://console.cloud.google.com/projectselector2/home/dashboard>)

★ **Note:** If you don't plan to keep the resources you create in this procedure, create a new project instead of selecting an existing project. After you finish following these steps, you can delete the project, removing all resources associated with the project.

2. Make a note of the Google Cloud project ID for the `test` project. You need this ID in the next section on setting up your environment.
3. Enable the Cloud Build, Cloud Functions, and Cloud Source Repositories APIs for that project.

[Enable the APIs](https://console.cloud.google.com/flows/enableapi?apiid=cloudbuild.googleapis.com,cloudfunctions.googleapis.com,sourcerepo.googleapis.com) (<https://console.cloud.google.com/flows/enableapi?apiid=cloudbuild.googleapis.com,cloudfunctions.googleapis.com,sourcerepo.googleapis.com>)

4. In the Cloud Console, go to the **Firestore** page.

[Go to the Firestore page](https://console.cloud.google.com/firestore) (<https://console.cloud.google.com/firestore>)

5. Create a Firestore database.

[Learn how to create a Firestore database](/solutions/serverless-web-performance-monitoring-using-cloud-functions#create_a_cloud_firestore_collection) (/solutions/serverless-web-performance-monitoring-using-cloud-functions#create\_a\_cloud\_firestore\_collection)

6. Select or create another Google Cloud project. This is the `build` project that hosts the Cloud Build pipeline.

[Go to the Manage Resources page](https://console.cloud.google.com/cloud-resource-manager) (<https://console.cloud.google.com/cloud-resource-manager>)

7. Make sure that billing is enabled for your Google Cloud projects.

[Learn how to enable billing](/billing/docs/how-to/modify-project) (/billing/docs/how-to/modify-project)

In this tutorial, you run commands in Cloud Shell. Cloud Shell is a shell environment with the Cloud SDK already installed, including the `gcloud` command-line tool, and with values already set for your current project. Cloud Shell can take several minutes to initialize.

1. In the Cloud Console for the `build` project, open Cloud Shell.

[Open Cloud Shell](https://console.cloud.google.com/?cloudshell=true) (<https://console.cloud.google.com/?cloudshell=true>)

2. Set a variable for the `test` Google Cloud project ID that you copied earlier:

Replace the following:

- `your-test-project-id`: The ID of your `test` Google Cloud project.

3. Set the project ID and project number of the current `build` Google Cloud project as variables:

4. Set a variable for the deployment region:

Although this tutorial uses the `us-central1` region, you can change this to [any region where Cloud Functions is available](#) (`/functions/docs/locations`).

5. Clone the repository containing the code for the sample app used in this tutorial:

6. Go to the project directory:

This tutorial uses [Terraform](https://www.terraform.io/) (<https://www.terraform.io/>) to automatically create and destroy Google Cloud resources within the test project. Creating independent resources for each build helps keep tests isolated from each other. When you isolate tests, builds can occur concurrently, and test assertions can be made against specific resources. Destroying the resources at the end of each build helps to minimize costs.

This tutorial deploys the app described in the [serverless web monitoring](#) (`/solutions/serverless-web-performance-monitoring-using-cloud-functions`) tutorial. The app consists of a set of Cloud Functions, Cloud Storage buckets, Pub/Sub resources, and a Firestore database. The Terraform configuration defines the steps required to create these resources. The Firestore database isn't deployed by Terraform; the database is created once and reused by all tests.

The following code sample from the Terraform configuration file `main.tf` shows the steps required to deploy the `trace` Cloud Function. Refer to the full file for the complete configuration.

[main.tf](https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/main.tf) (https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/main.tf)

[View on GitHub](https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/main.tf) (https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/main.tf)

In this step, you run the Terraform configuration to deploy the test resources. In a later step, Cloud Build deploys the resources automatically.

1. In Cloud Shell, initialize Terraform:

You use the public Terraform Docker image. Docker is already installed in Cloud Shell. The current working directory is mounted as a volume so the Docker container can read the Terraform configuration file.

2. Create the resources using the Terraform `apply` command:

The command includes variables that specify the Google Cloud project and region where you want the test resources created. It also includes a suffix that is used in creating named resources for this step. In a later step, Cloud Build automatically supplies a suitable suffix.

It takes a few minutes for the operation to complete.

3. Confirm that resources were created in the `test` project:

The output displays three Cloud Functions whose names end with the suffix supplied earlier.

In this section, you run end-to-end tests against the test infrastructure that you deployed in the preceding section.

The following code snippet shows the tests. The tests validate both the success and failure scenario. The test pipeline can be summarized as follows:

- First, the test calls the `trace` function. This call initiates a flow of events through the app that triggers other functions.
- Then, the test verifies the behavior of each function and confirms that objects are written to Cloud Storage, results are persisted to Firestore, and Pub/Sub alerts are generated upon failures.

[e2e/e2e\\_test.py](https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/e2e/e2e_test.py) (https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/e2e/e2e\_test.py)

[View on GitHub](https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/e2e/e2e_test.py) (https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/e2e/e2e\_test.py)

To run the end-to-end tests, complete the following steps:

1. In Cloud Shell, create a new `virtualenv` environment. The `virtualenv` utility is already installed in Cloud Shell.

2. Activate the `virtualenv` environment:

3. Install the required Python libraries:

4. Run the end-to-end tests:

You pass the Terraform state file, which contains details of the test resources created in the preceding section.

The tests can take a few minutes to complete. A message indicating that two tests passed is displayed. You can ignore any warnings.

5. Tear down the test resources by using the Terraform `destroy` command:

6. Confirm that the resources are destroyed:

There are no longer any Cloud Functions with names ending with the suffix supplied earlier.

In this section you use Cloud Build to automate the testing pipeline.

You run Cloud Build using a [Cloud Build service account](#)

([/cloud-build/docs/securing-builds/set-service-account-permissions#what\\_is\\_the\\_service\\_account](#)). The system tests executed by the build create and interact with Cloud Functions, Cloud Storage buckets, Pub/Sub resources, and Firestore documents. To do these things, Cloud Build requires the following:

- Appropriate [Cloud IAM roles](#) ([/iam/docs/understanding-roles](#)) in the test project.
- The ability to act as the [Cloud Functions runtime service account](#) ([/functions/docs/concepts/iam#runtime\\_service\\_account](#)). By default, Cloud Functions uses the App Engine service account (`your-test-project-id@appspot.gserviceaccount.com`) as a runtime service account, where `your-test-project-id` is the name of your test Google Cloud project.

In this procedure you add the appropriate roles and then add the Cloud Build service account.

1. In Cloud Shell, add the appropriate Cloud IAM roles to the default Cloud Build service account:

2. Add the Cloud Build service account as a `serviceAccountUser` of the App Engine service account within the test project:

★ **Note:** The App Engine service account has the Editor role on the project. This role has wide-ranging permissions. You can [change the roles of this service account](#) (/appengine/docs/standard/python/service-account#changing\_service\_account\_permissions\_) to limit or extend the permissions for your running functions. You can also change which service account is used by [providing a non-default service account on a per-function basis](#) (/functions/docs/securing/function-identity#per-function\_identity).

The build performs four logical tasks:

- Running unit tests
- Deploying the sample app
- Running end-to-end tests
- Destroying the sample app

Review the following code snippet from the `cloudbuild.yaml` file. The snippet illustrates the individual Cloud Build steps that deploy the sample app using Terraform and run the end-to-end tests.

[cloudbuild.yaml](https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/cloudbuild.yaml) (https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/cloudbuild.yaml)

[View on GitHub](https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/cloudbuild.yaml) (https://github.com/GoogleCloudPlatform/solutions-serverless-web-monitoring/blob/master/cloudbuild.yaml)

To submit a manual build to Cloud Build and run end-to-end tests, do the following:

- In Cloud Shell, enter the following:

The build takes several minutes to run. The following build steps occur:

- Cloud Build uses [substitutions](/cloud-build/docs/build-config#substitutions) (/cloud-build/docs/build-config#substitutions) to supply variables that specify the Google Cloud project and region for the test resources you create.
- Cloud Build runs the build in the `build` Google Cloud project. The test resources are created in the separate `test` project.

The build logs are streamed to Cloud Shell so that you can follow the build progress. The log stream terminates upon build completion. Messages are displayed that indicate the final `terraform-destroy` build step is successful and the build is done.

A key tenet of [continuous integration \(CI\)](/solutions/continuous-integration/) (/solutions/continuous-integration/) is to regularly run a set of comprehensive automated tests. Typically, the build-test pipeline runs for each commit to a shared code repository. This setup helps to confirm that each commit to the shared repository is tested and validated, allowing your team to detect problems early.

In the next sections, you perform the following actions:

- Create a Git repository in Cloud Source Repositories.
- Add a Cloud Build trigger to run the end-to-end build upon every commit.
- Push code to the repository to trigger the build.

1. In Cloud Shell, create a new Cloud Source Repository:

2. In the Cloud Console, open the Cloud Build **Triggers** page.

[Go to the Triggers page](https://console.cloud.google.com/cloud-build/triggers) (https://console.cloud.google.com/cloud-build/triggers)

3. Click **Create Trigger**.

4. On the **Create trigger** page, select **Cloud Source Repositories** as source, and then click **Continue**.

5. In the **Repository** list, select the new `serverless-web-monitoring` repository, and then click **Continue**.

6. On the **Triggers settings** page, fill out the following options:

- In the **Description** field, type `end-to-end-tests`.
- In the **Trigger type** list, select `master`.
- In the **Build configuration** list, select **Cloud Build configuration file**.
- In the **Cloud Build configuration file location** field, type `cloudbuild.yaml`.



- To add a variable substitution to specify the Google Cloud region where the test resources will be created, click **Add Item**:

- **Variable:** `_REGION`

- **Value:** `your-test-region`

Replace the following:

- `your-test-region`: The value of the `$REGION` variable in Cloud Shell.

- To add another variable substitution to specify the ID of the project that will host the test resources, click **Add Item**:

- **Variable:** `_TEST_PROJECT_ID`

- **Value:** `your-test-project`

Replace the following:

- `your-test-project`: The value of the `$TEST_PROJECT` variable in Cloud Shell.

7. Click **Create trigger**.

1. In Cloud Shell, add the repository as a new remote in your git config:

2. To trigger the build, push the code to the repository:

3. List the most recent builds:

The output displays a build in `WORKING` status, indicating that the build triggered as expected.

4. Copy the ID of the `WORKING` build for the next step.

5. Stream the build logs to the Cloud Console:

Replace the following:

- `build-id`: The ID of the `WORKING` build you copied in the preceding step.

The log stream terminates upon build completion. Messages are displayed that indicate the final `terraform-destroy` build step is successful and that the build is done.

To avoid incurring charges to your Google Cloud Platform account for the resources used in this tutorial:


**!** **Caution:** Deleting a project has the following effects:

- **Everything in the project is deleted.** If you used an existing project for this tutorial, when you delete it, you also delete any other work you've done in the project.
- **Custom project IDs are lost.** When you created this project, you might have created a custom project ID that you want to use in the future. To preserve the URLs that use the project ID, such as an `appspot.com` URL, delete selected resources inside the project instead of deleting the whole project.

If you plan to explore multiple tutorials and quickstarts, reusing projects can help you avoid exceeding project quota limits.

1. In the Cloud Console, go to the **Manage resources** page.

[Go to the Manage resources page \(https://console.cloud.google.com/iam-admin/projects\)](https://console.cloud.google.com/iam-admin/projects)

2. In the project list, select the project you want to delete and click **Delete** .

3. In the dialog, type the project ID, and then click **Shut down** to delete the project.

- Visit the Google Cloud [CI/CD developer hub \(/docs/ci-cd/\)](/docs/ci-cd/).
- Read the blog post on how [testing and CI/CD keeps bugs out of production \(/blog/products/application-development/release-with-confidence-how-testing-and-cicd-can-keep-bugs-out-of-production\)](/blog/products/application-development/release-with-confidence-how-testing-and-cicd-can-keep-bugs-out-of-production).
- Complete the related [Serverless web monitoring \(/solutions/serverless-web-performance-monitoring-using-cloud-functions\)](/solutions/serverless-web-performance-monitoring-using-cloud-functions) tutorial.
- Complete the [GitOps-style continuous delivery with Cloud Build \(/kubernetes-engine/docs/tutorials/gitops-cloud-build\)](/kubernetes-engine/docs/tutorials/gitops-cloud-build) tutorial.
- Explore how to [manage Google Cloud projects using Terraform \(/community/tutorials/managing-gcp-projects-with-terraform\)](/community/tutorials/managing-gcp-projects-with-terraform).
- Try out other Google Cloud features for yourself. Have a look at our [tutorials \(/docs/tutorials\)](/docs/tutorials).