ConductorOne

Implementing Cloud IAM for Cloud Functions with a Least Privilege Approach

Google Cloud's Cloud Functions is a serverless platform that offers a powerful and scalable way to build event-driven applications. Despite the platform's strengths, it's important to carefully manage the access permissions of your cloud functions to ensure they are secure. Lack of secure practices when building and deploying cloud functions can result in unauthorized access and data leaks. This is where the principle of least privilege (PoLP) comes in.

The principle of least privilege is a security concept that ensures users and non-human identities (e.g., service accounts) only have the minimum permissions needed to perform their tasks. This reduces the potential damage if an account is compromised.

In this article, you'll learn how to implement Cloud Identity and Access Management (IAM) for Cloud Functions with a least privilege approach. You'll first delve into the core concepts of least privilege and explore how Cloud IAM helps enforce PoLP for your serverless functions. Then you'll learn how to create and bind unique service accounts to your functions, granting them only the most restricted set of permissions needed for their specific workloads.

What is the principle of least privilege (PoLP)?

PoLP is a fundamental security concept that dictates granting users and non-human identities only the bare minimum permissions required to perform their designated tasks.

By following this principle, software teams can reap many benefits:

→ **Minimized attack surface:** Granting only essential permissions reduces the number of entry points a potential attacker can exploit. A compromised low-privilege account is far less dangerous than one with broad access.

→ **Safeguards against human error:** Accidental mistakes can also lead to security breaches. Least privilege minimizes the impact of such errors by limiting the damage an authorized user with the wrong permissions can cause.

→ **Prevention of the spread of malware:** Malware often relies on elevated privileges to propagate through a system. Least privilege makes it more difficult for malware to gain a foothold and spread laterally.

→ **Better adherence to compliance:** Many compliance regulations mandate the implementation of least privilege to ensure data security. Following this principle demonstrates your commitment to data protection and regulatory compliance.

In the context of Cloud Functions, implementing least privilege ensures your serverless functions operate with the bare minimum permissions needed to access resources and complete their tasks. This significantly strengthens your cloud security posture.

The principle of least privilege in Google Cloud Platform

PoLP can be implemented on Google Cloud Platform (GCP) through service accounts in Cloud IAM. Service accounts are special types of accounts designed to represent non-human users (typically applications or virtual machines). Unlike regular user accounts, service accounts are not tied to individual people.

Each service account has a unique email address that acts as its identifier. This email address is used for authentication and authorization purposes. Service accounts are primarily used by applications running on GCP to access Google Cloud services and APIs programmatically. This eliminates the need for human intervention and streamlines automated tasks.

Service accounts offer a secure way to manage access control. You can grant specific permissions (IAM roles) to a service account, allowing it to interact with only the authorized resources. This enhances security by minimizing the potential damage if a service account is compromised.

There are multiple types of service accounts. GCP creates some service accounts automatically when you enable certain APIs and services. You can also create custom service accounts. You can read more about the different kinds of service accounts here.

By default, 1st gen Cloud Functions use the App Engine default service account, and 2nd gen Cloud Functions use the Compute Engine default service account. These service accounts give the functions access to a wide range of Google Cloud services. Having access to a large number of Google Cloud services from the get-go can improve the speed of your Cloud Functions development process. However, Google recommends using the default service accounts for development and testing only, as they can be a security risk in production.

This is why you need to create custom service accounts for your functions in GCP and grant them access to only the resources that they interact with. You'll learn how to do so via the web console and the terminal in the next section.

Implementing the principle of least privilege

There are multiple ways you can implement PoLP in Cloud Functions:

→ Some Google Cloud services, such as Cloud Storage buckets, can be restricted through IAM policies. This allows you to create a custom service account for your Cloud Functions to set up fine-grained access to these services.

→ Other services, like Compute Engine VMs, can't be restricted through IAM policies. For these, you'll need to use Google-signed identity tokens to help identify requests coming from allowed sources.

→ When restricting access to other functions, you'll use the same Google-signed identity tokens.

You'll learn more about all three scenarios in the following sections.

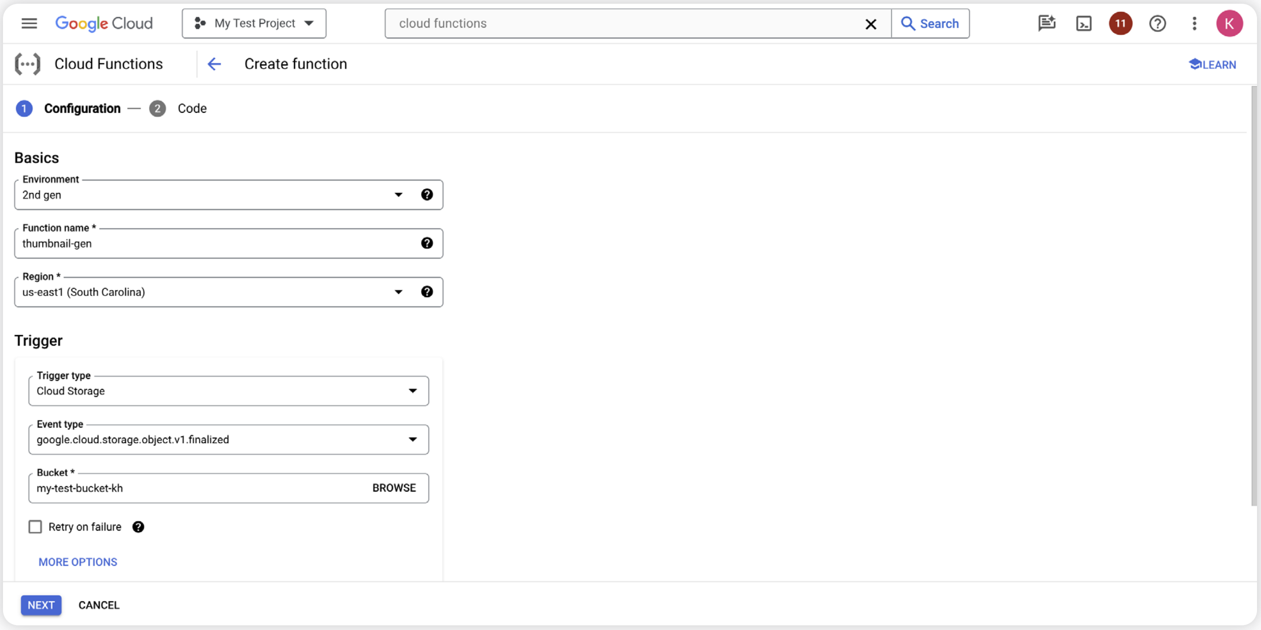
When calling GCP services

When calling GCP services such as Cloud Storage buckets through Cloud Functions, you can use custom service accounts to restrict access.

To try it out for yourself, you'll first need to set up a sample function and a bucket. This tutorial teaches you how to create a custom service account with the right permissions and bind it to the Cloud Function.

Setting up the function and the bucket

First, go to Cloud Storage on the GCP console and create a new bucket. Once you've created the new bucket, go to the Create function page on your Google Cloud console. Provide a name for your new function, set the **Trigger type** to **Cloud Storage**, and select the bucket you just created. You can leave all the default values for all the other fields. This is what the page should look like when you've entered all the information:



Environment: 2nd gen
Function name *: thumbnail-gen
Region *: us-east1 (South Carolina)
Trigger type: Cloud Storage
Event type: google.cloud.storage.object.v1.finalized
Bucket *: my-test-bucket-kh BROWSE

Click the **Next** button. Now, paste the following code into the `index.js` file to define the function:

```
const functions = require('@google-cloud/functions-framework');
const { Storage } = require('@google-cloud/storage');
const path = require("path")
const sharp = require('sharp');

functions.cloudEvent('helloGCS', async cloudEvent => {

  const file = cloudEvent.data;

  // Don't run for thumbnails
  if (file.name.includes("_thumb")) {
      return
  }

  const storage = new Storage();
  const bucketRef = storage.bucket(file.bucket);
  const fileRef = bucketRef.file(file.name);
  const name = file.name.split("/")[1];

  try {
    await fileRef.download({ destination: name });
    console.log(`Downloaded file: ${name}`);

    sharp(name)
      .resize(320, 240)
      .toFile(name.split(".")[0] + "_thumb." + name.split(".")[1], async
(err, info) => {
        await bucketRef.upload(
            name.split(".")[0] + "_thumb." + name.split(".")[1],
            {
                destination: file.name.split("/")[0] + "/" + name.
split(".")[0] + "_thumb." + name.split(".")[1]
            });

        console.log(`Uploaded thumbnail: ${name.split(".")[0] + "_thumb."
+ name.split(".")[1]}`);
      });

  } catch (error) {
      console.error(`Error generating thumbnail: ${error}`);
  }
});
```

Paste the following code into the `package.json` file:

```
{
  “dependencies”: {
    “@google-cloud/functions-framework”: “^3.0.0”,
    “@google-cloud/storage”: “7.8.0”,
    “sharp”: “0.33.2”
  }
}
```

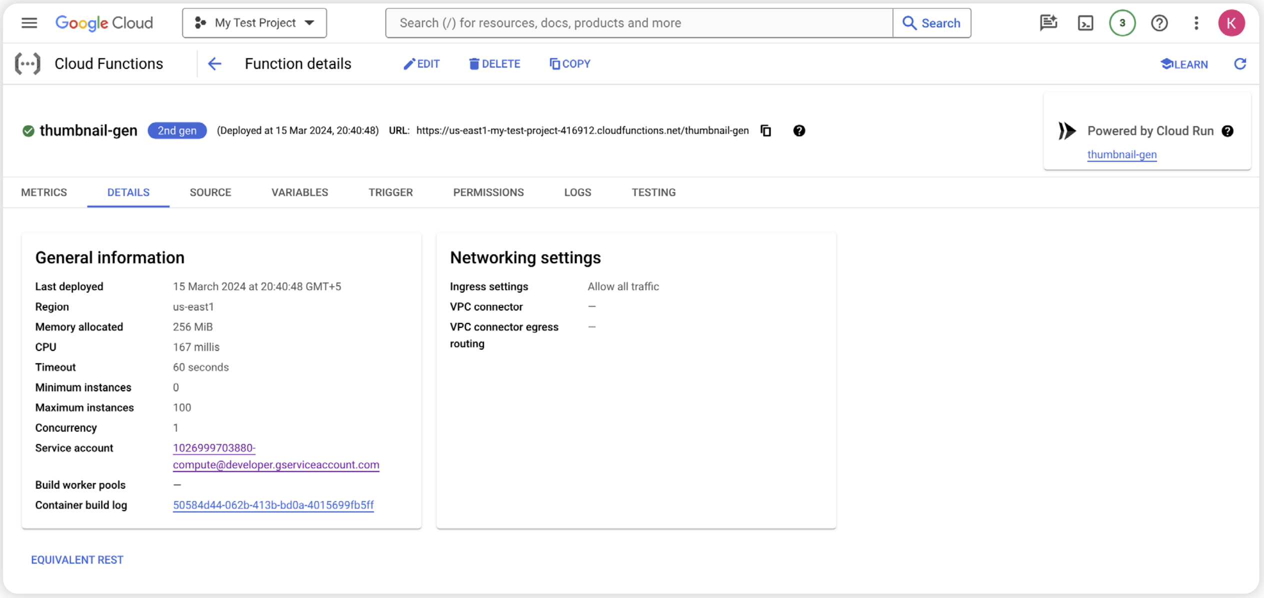
This is a simple function that listens for events in your chosen bucket. When a new photo is uploaded to any folder in the bucket, the function downloads it locally, creates a thumbnail for it using sharp, and then uploads it to the same bucket at the same location.

Click the blue **Deploy** button at the bottom to deploy the function. GCP will start to deploy your Cloud Function. Once deployed, you can try creating a folder and adding files to it. After a short time, a thumbnail of that photo will be available in the same folder. This demonstrates that the function works and is doing its job correctly.

Implementing PoLP through a custom service account

Let’s now look at the major security issue associated with this function and how you can implement PoLP to mitigate it.

Currently, it uses the Compute Engine default service account, which you can confirm by going to the **Details** tab on the **Function details** page:



thumbnail-gen 2nd gen (Deployed at 15 Mar 2024, 20:40:48) URL: https://us-east1-my-test-project-416912.cloudfunctions.net/thumbnail-gen

Powered by Cloud Run
thumbnail-gen

General information

Last deployed	15 March 2024 at 20:40:48 GMT+5
Region	us-east1
Memory allocated	256 MiB
CPU	167 millis
Timeout	60 seconds
Minimum instances	0
Maximum instances	100
Concurrency	1
Service account	1026999703880-compute@developer.gserviceaccount.com
Build worker pools	—
Container build log	50584d44-062b-413b-bd0a-4015699fb5ff

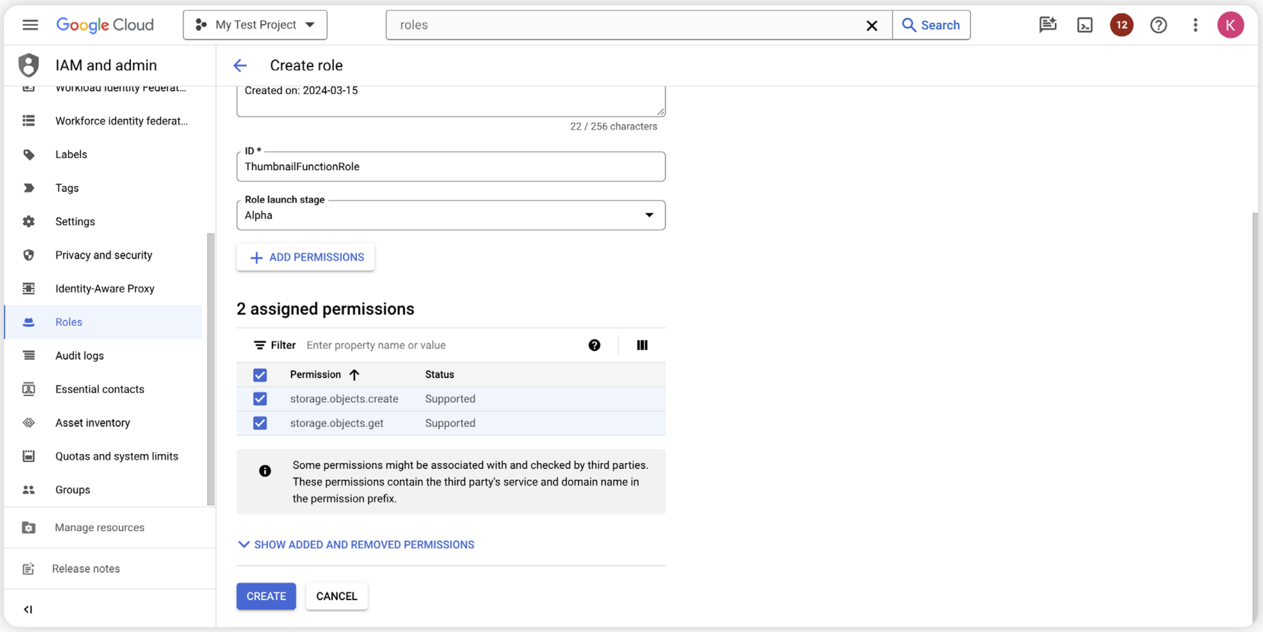
Networking settings

Ingress settings	Allow all traffic
VPC connector	—
VPC connector egress routing	—

This means it has assumed an Editor role within its GCP project. This role comes with around 8,000 permissions throughout your GCP project, but this function only needs the permissions `storage.objects.get` and `storage.objects.create`.

However, you cannot assign permissions to service accounts directly. You can only assign roles to them, which in turn are granted permissions. To limit this function's permissions, you could assign it the Cloud Storage Object Admin role, but this role also contains five additional permissions on top of the two you need. While it's better than having over 8,000 unnecessary permissions, it's not exactly the "least privilege" approach.

To solve this problem and adhere to PoLP, you'll need to create a custom role with only these two permissions. You can do that by navigating to the Create role page. Give your new role a name and an ID, and click the **+ Add Permissions** button to add the two permissions you need for your Cloud Function. Once done, the page should display your two permissions with checkboxes under the **2 assigned permissions** label:



Create role

Created on: 2024-03-15

22 / 256 characters

ID *: ThumbnailFunctionRole

Role launch stage: Alpha

2 assigned permissions

Filter Enter property name or value

☑	Permission	Status
☑	storage.objects.create	Supported
☑	storage.objects.get	Supported

Some permissions might be associated with and checked by third parties. These permissions contain the third party's service and domain name in the permission prefix.

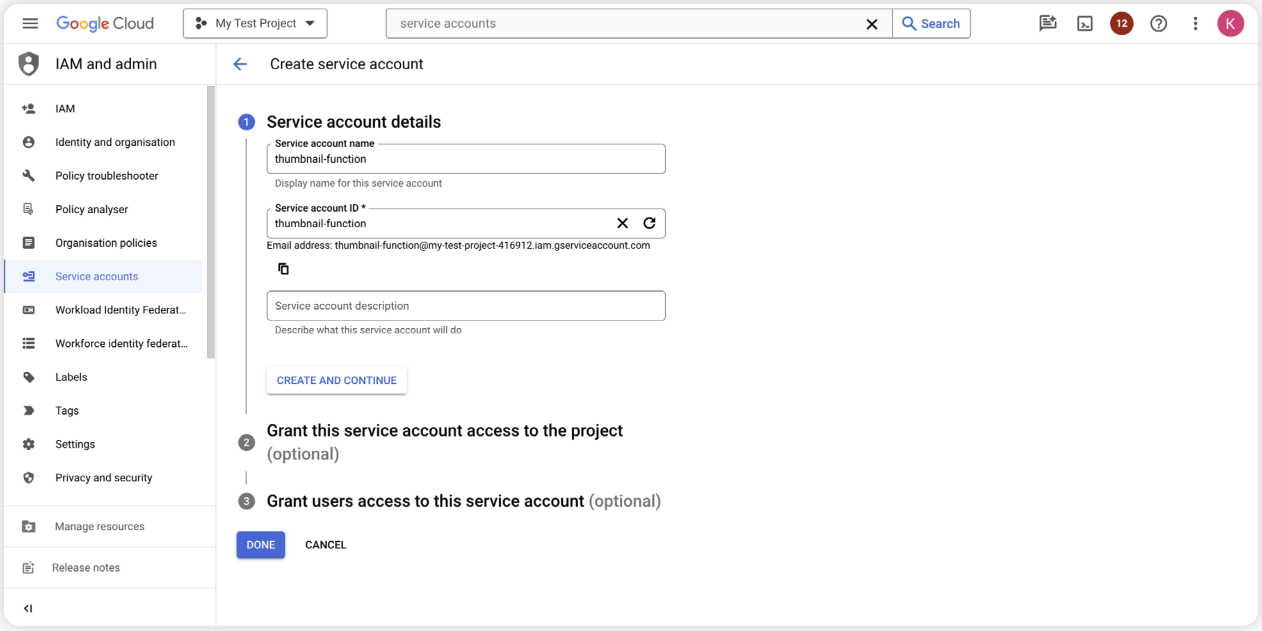
Click the **Create** button to create the role.

Alternatively, you could also use the following `gcloud` command to create the role through the CLI:

```
gcloud iam roles create ThumbnailFunctionRole --project=”PROJECT_
ID” --title=”thumbnail-function-role” --permissions=”storage.
objects.create,storage.objects.get”
```

You can now navigate to the Create service account page to start creating a new account.

Start by providing a name for your new service account. A service account ID will automatically be generated for you from the name you enter. Click **Create and continue**.



1 Service account details

Service account name: thumbnail-function
Display name for this service account

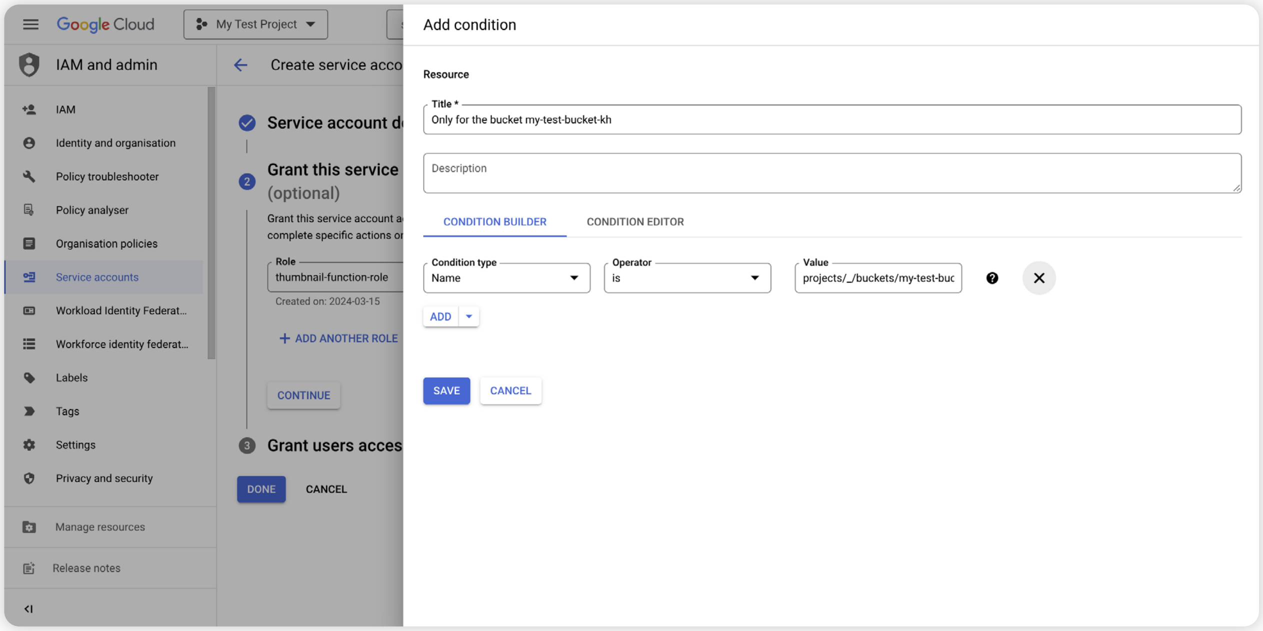
Service account ID *: thumbnail-function
Email address: thumbnail-function@my-test-project-416912.iam.gserviceaccount.com

Service account description
Describe what this service account will do

2 Grant this service account access to the project (optional)

3 Grant users access to this service account (optional)

Next, you need to grant this service account the role that you need your function to have. Click the **Select a role** dropdown menu. Search for the thumbnail function role and select it. Click **+ Add IAM condition** to add a condition that checks for the bucket name before allowing access to it. Enter a title like `Only for the bucket [BUCKET_NAME]`, choose the condition type `Name` and operator `is`, and enter the value as `projects/_/buckets/[BUCKET_NAME]`:



Grant this service account a... complete specific actions or...

Role: thumbnail-function-role

Add condition

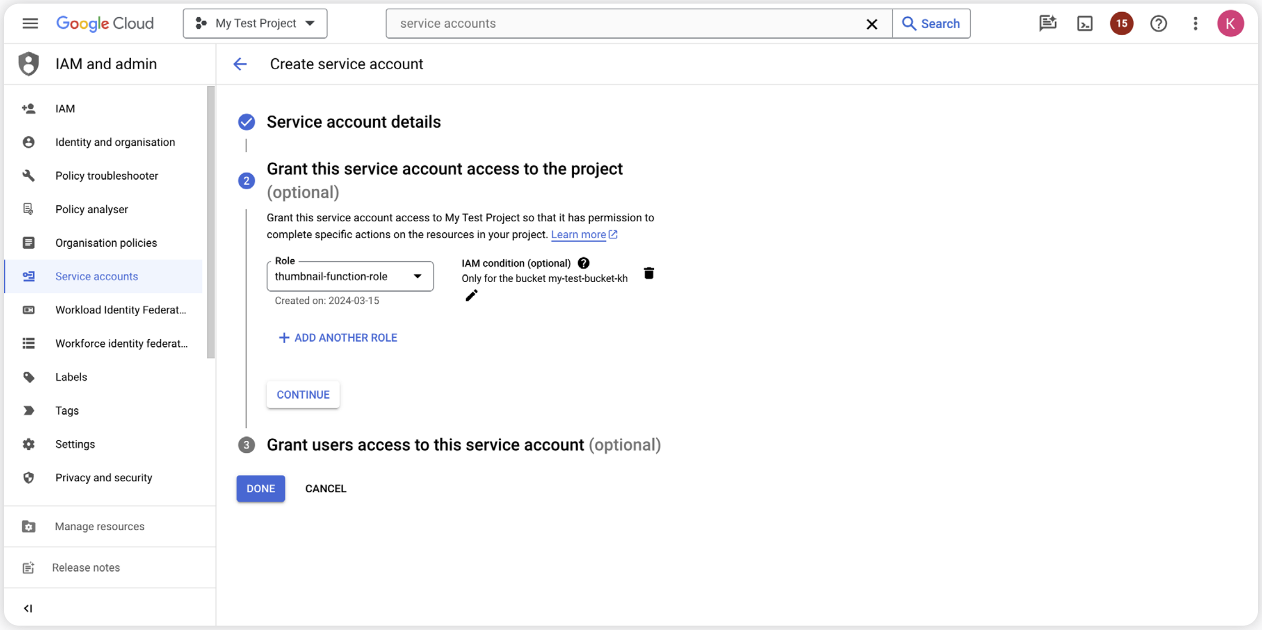
Resource

Title *: Only for the bucket my-test-bucket-kh

Description

Condition type	Operator	Value
Name	is	projects/_/buckets/my-test-buc

Click the **Save** button. You'll be taken to the **Create service account** page, which has some optional configurations that you can ignore for this tutorial.



Service account details

Grant this service account access to the project (optional)

Grant this service account access to My Test Project so that it has permission to complete specific actions on the resources in your project. Learn more

Role: thumbnail-function-role
Created on: 2024-03-15

IAM condition (optional)
Only for the bucket my-test-bucket-kh

Grant users access to this service account (optional)

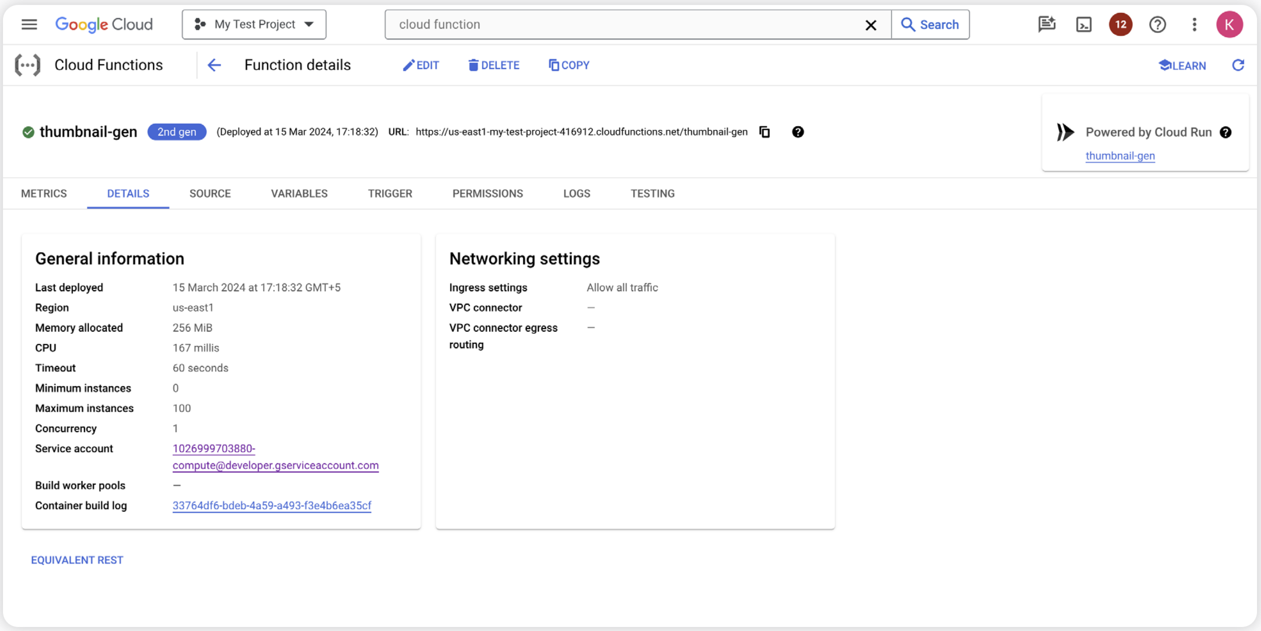
Click the **Continue** button and click **Done** on the last step. Your new service account with just the right permissions is ready.

Alternatively, you could use the following `gcloud` commands to create the service account and add the roles:

```
gcloud iam service-accounts create thumbnail-function
gcloud projects add-iam-policy-binding [PROJECT_ID]
--member=”serviceAccount:[SERVICE_ACCOUNT_NAME]@[PROJECT_ID].
iam.gserviceaccount.com” --role=”projects/[PROJECT_ID]/roles/
ThumbnailFunctionRole” --condition=’expression=resource.name ==
“projects/_/buckets/[BUCKET_NAME]”,title=Only for the bucket my-
test-bucket-kh’
```

Your service account is now ready. You can go back to your Cloud Function and redeploy it while binding it to the new service account.

Go back to the **Function details** page and click the **Edit** button:



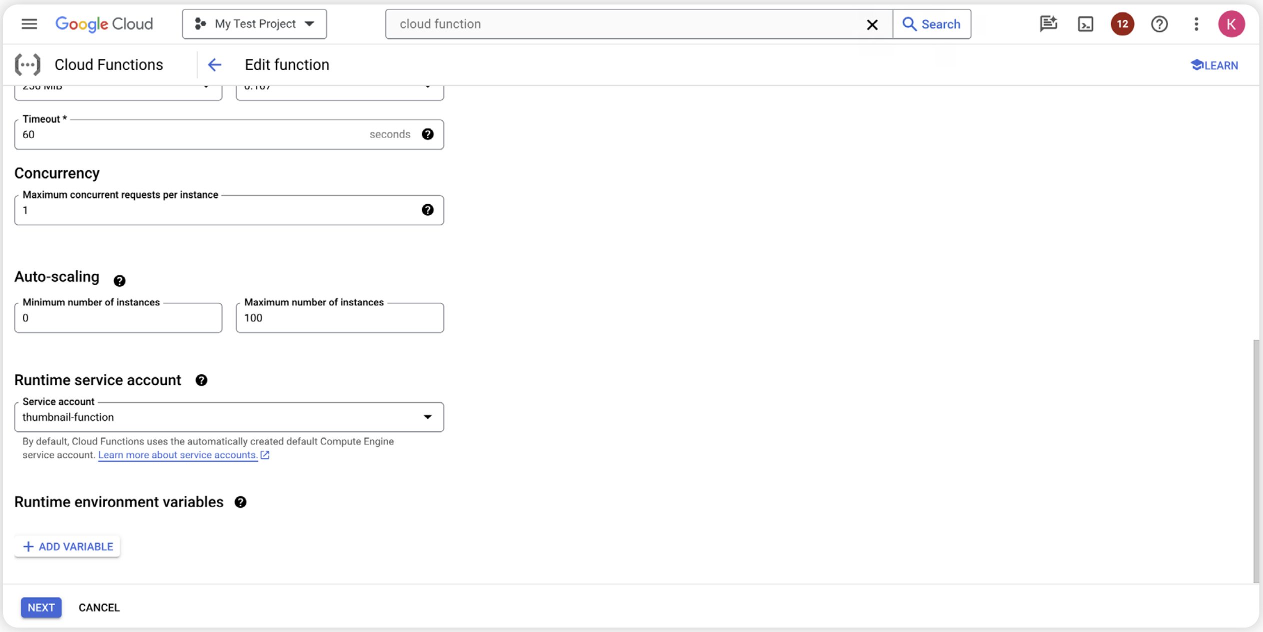
General information

Last deployed	15 March 2024 at 17:18:32 GMT+5
Region	us-east1
Memory allocated	256 MiB
CPU	167 millis
Timeout	60 seconds
Minimum instances	0
Maximum instances	100
Concurrency	1
Service account	1026999703880-compute@developer.gserviceaccount.com
Build worker pools	—
Container build log	33764df6-bdeb-4a59-a493-f3e4b6ea35cf

Networking settings

Ingress settings	Allow all traffic
VPC connector	—
VPC connector egress routing	—

Scroll down and expand the **Runtime, build, connections and security settings** section. In that section, scroll down to the **Runtime service account** subsection and choose your new service account in the **Service account** dropdown menu:



Timeout *: 60 seconds

Concurrency

Maximum concurrent requests per instance: 1

Auto-scaling

Minimum number of instances: 0

Maximum number of instances: 100

Runtime service account

Service account: thumbnail-function

By default, Cloud Functions uses the automatically created default Compute Engine service account. Learn more about service accounts.

Runtime environment variables

Click **Next** at the bottom of the page, then click **Deploy** on the next page.

Alternatively, you could use the following `gcloud` command to redeploy the function with the new service account:

```
gcloud functions deploy thumbnail-gen \
  --service-account “[SERVICE_ACCOUNT_NAME]@[PROJECT_ID].iam.
gserviceaccount.com”
```

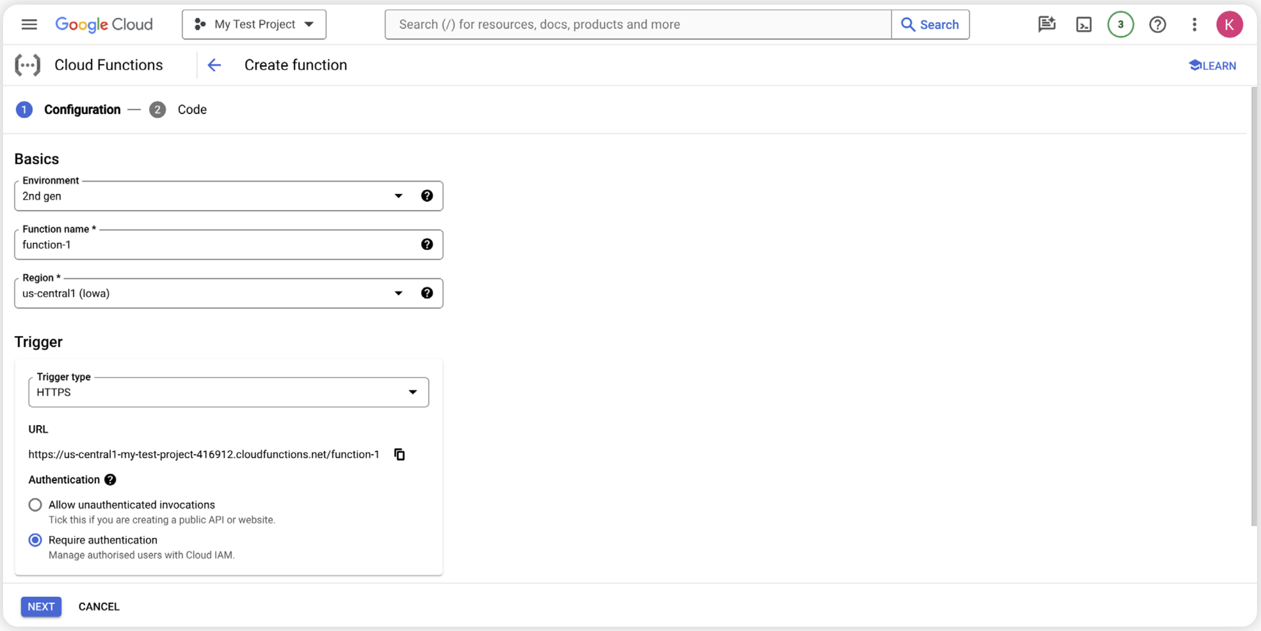
Once deployed, the function will now only have the two permissions `storage.objects.get` and `storage.objects.create`. You can try uploading a photo to the bucket once again to see if it works correctly.

When calling other functions

Cloud Functions are often designed to call other functions to modularize business logic and implement a better separation of concerns. In such cases, the function to be called will have an HTTP trigger, since event-driven functions like the one you saw above can only be invoked by the event sources.

When one Cloud Function, let’s say Function A, calls another Cloud Function, let’s say Function B, there are multiple things you need to do to restrict access to only the necessary permissions.

First of all, you need to set Function B to require authentication. You can usually do that when creating the function by clicking the radio button next to **Require authentication** under **Authentication** in the **Trigger** section:



Environment: 2nd gen
Function name *: function-1
Region *: us-central1 (Iowa)

Trigger
Trigger type: HTTPS
URL
https://us-central1-my-test-project-416912.cloudfunctions.net/function-1
Authentication
( ) Allow unauthenticated invocations
Tick this if you are creating a public API or website.
(●) Require authentication
Manage authorised users with Cloud IAM.

Additionally, you could also edit an existing function and set the same value.

Next, you need to provide the calling function (Function A) with the `cloudfunctions.functions.invoke` permission if the called function (Function B) is a 1st gen function, or the `run.routes.invoke` permission if the called function (Function B) is a 2nd gen function. Without these permissions, no principal can invoke authenticated Cloud Functions.

You can attach the `roles/cloudfunctions.invoker` role to your custom service account for 1st gen functions or the `roles/run.invoker` role for 2nd gen functions.

Finally, you need to pass in a Google-signed identity token when calling Function B. You can follow the steps outlined here to use one of Google's Auth SDKs to generate the ID token, or add the following lines to your function (for Node.js) to fetch the ID token manually:

```
// Install and import node-fetch
// const fetch = require('node-fetch');

const headers = new fetch.Headers({
  'Metadata-Flavor': 'Google'
});

const functionBUrl = 'https://[REGION]-[PROJECT_ID].cloudfunctions.
net/functionB;

// Set up metadata server request to get a token
const metadataServerTokenURL = 'http://metadata/computeMetadata/v1/
instance/service-accounts/default/identity?audience=';

const response = await fetch(metadataServerTokenURL +
otherFunction, {
    method: "POST",
    headers: headers
})
```

You can now pass the retrieved token in the header of the call to Function B as `Authorization: Bearer [ID_TOKEN]`. Requests that do not contain such a token will automatically be rejected by the called function (Function B).

When calling GCP services not restricted by Cloud IAM policies

When calling GCP services that do not work with Cloud IAM policies, you can use the same Google-signed ID tokens to authenticate your calls. For instance, when making calls to a Compute Engine VM, you can generate a Google-signed ID token using the same method mentioned in the previous section—replace the `functionBUrl` with the URL to which you are sending your request—and pass it as a bearer token in your HTTP requests.

On the service side, you can validate the token yourself to make sure the request is coming from a verified source. Google provides detailed instructions to validate the token here.

Conclusion

In this article, you learned about the importance of the principle of least privilege and how to implement it for Cloud Functions. By following a least privilege approach, you can minimize the attack surface of your Cloud Functions and improve the overall security of your project.

To simplify Cloud Function IAM management at scale, consider using ConductorOne, a comprehensive cloud IAM platform that helps you enforce least privilege and automate IAM policies across your GCP resources.