

DataFlow Functions in Google Cloud Functions

Date published: 2021-04-06

Date modified: 2024-06-03

CLOUDERA

Legal Notice

© Cloudera Inc. 2024. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Creating your first Google Cloud function.....	4
Configuring your Google Cloud function.....	6
Service account.....	10
Output ports.....	11
Parameters.....	11
Cold start.....	12
Cloud storage.....	13
Data flow state.....	14
Configuring Kerberos.....	14
Handling failures.....	15
Testing your Google Cloud function.....	15
Monitoring and logs.....	16
Adjusting logs levels.....	17
Google Cloud Function triggers.....	17
Creating a Google Cloud function using CLI.....	21

Creating your first Google Cloud function

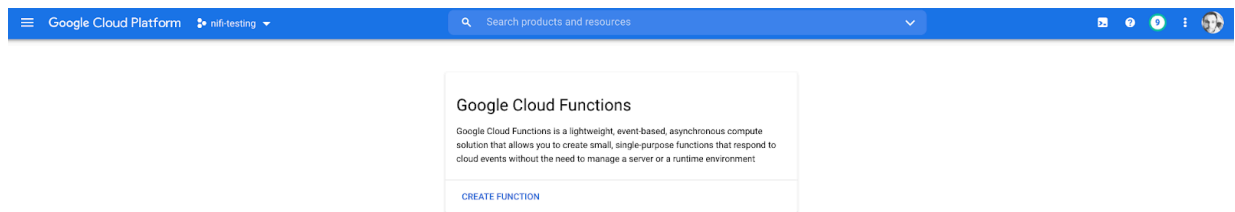
Google Cloud Functions is a serverless execution environment for building and connecting cloud services. You can write simple, single-purpose functions that are attached to events emitted from your cloud infrastructure and services. Your function is triggered when an event being watched is fired.

About this task

Follow these steps to create a Google Cloud function that is able to run :

Procedure

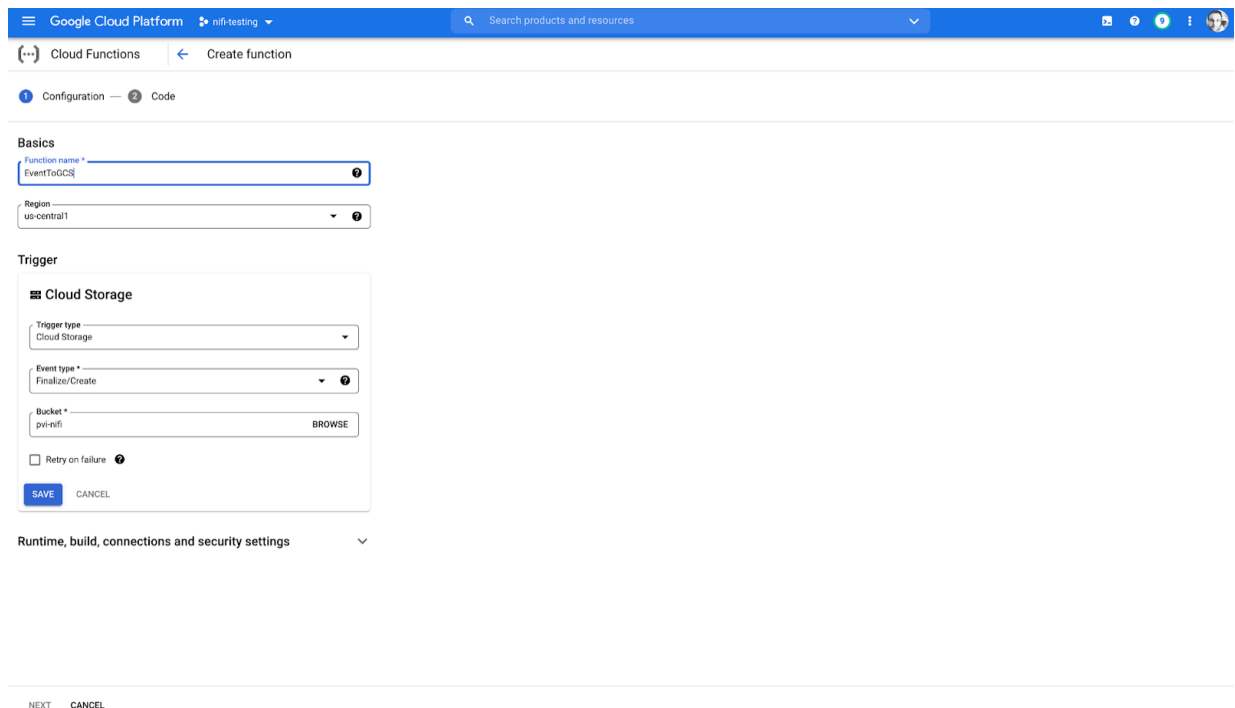
1. Navigate to the Google Cloud Function page in your cloud account in the region of your choice.



2. Click CREATE FUNCTION.

You can now create your function.

- a) Give your function a meaningful name for your use case.
- b) Select the region where your function should be running.
- c) Select the trigger that you want to use with your function and configure it.



3. Click SAVE.
4. Expand the Runtime, build, connections and security settings section.

5. Configure Memory allocation and Timeout on the Runtime tab.

For more information on recommendations, see the *Memory and timeout in runtime configuration* section in *Configuring your Google Cloud function*.

6. Select a Runtime service account.

For more information, see *Service account*.

7. Configure the Runtime environment variables. For more information, see the *Runtime environment variables* section in *Configuring your Google Cloud function*.

8. You may also want to switch to the Security tab and reference a secret for sensitive parameters.

For more information on a secure mechanism for storing parameters, see the *Google Cloud Secret Manager* section in *Parameters*.

Google Cloud Platform nifi-testing

Cloud Functions Create function

Function won't be automatically retried on failure

EDIT

Runtime, build, connections and security settings

RUNTIME BUILD CONNECTIONS SECURITY

Memory allocated * 1 GB

Timeout * 60 seconds

Runtime service account

Runtime service account nifi-test-serviceaccount

Autoscaling

Minimum number of instances 0 Maximum number of instances 3000

Runtime environment variables

Name *	Value
FLOW_CRN	VALUE
DF_PRIVATE_KEY	VALUE
DF_ACCESS_KEY	VALUE

+ ADD VARIABLE

NEXT CANCEL

9. Click NEXT.

10. In the Code section, select Java 11 for the Runtime.

If it is not already enabled, you have to enable the Cloud Build API.

11. In the Source code drop-down menu, select ZIP Upload and upload the ZIP file that was provided to you by Cloudera.

Provide a Google Cloud Storage Stage bucket where the code will be staged during the deployment of the function. Alternatively, you may choose ZIP from Cloud Storage and provide a location to the ZIP file.

12. Set the Entry point to:

For HTTP triggers:

`com.cloudera.naaf.gcp.cloud.functions.StatelessNiFiHttpFunction`

For background triggers with asynchronous calls:

`com.cloudera.naaf.gcp.cloud.functions.StatelessNiFiBackgroundFunction`

Google Cloud Platform

Cloud Functions

Create function

Configuration

Code

Runtime: Java 11

Entry point: com.cloudera.naaf.gcp.cloud.functions.StatelessNiFiBackgroundFunction

Source code: ZIP Upload

ZIP upload

ZIP file: naaf-gcp-cloud-functions-1.0.0-SNAPSHOT-bin.zip

Local file for upload

Stage bucket: pvi-nifi

PREVIOUS DEPLOY CANCEL

13. Click DEPLOY.

Once your function is deployed, it looks similar to this:

Name	Region	Trigger	Runtime	Memory allocated	Executed function	Last deployed	Authentication	Actions
EventToGCS	eu-west-1	Bucket: pvi-nifi	Java 11	1 GB	com.cloudera.naaf.gcp.cloud.functions.StatelessNiFiBackgroundFunction	Oct 28, 2021, 7:47:36 PM		

Related Information

[Memory and timeout in runtime configuration](#)

[Service account](#)

[Runtime environment variables](#)

[Google Cloud Secret Manager](#)

Configuring your Google Cloud function

After you have created a function, you can use the built-in configuration options to control its behavior. You can configure additional capabilities, adjust resources associated with your function, such as memory and timeout, or you can also create and edit test events to test your function using the console.

Memory and timeout in runtime configuration

You can configure two important elements on the Runtime Configuration tab, timeout and memory allocated to the cloud function. The appropriate values for these two settings depend on the data flow to be deployed. Typically, it is recommended to start with allocating 1 GB - 2 GB memory. You can adjust this value later as you see how much memory your function needs.



Note: Any extensions or resources added to the Google Cloud Function are placed under /tmp, which is an in-memory mount. This means that they consume memory resources provisioned for the function. For more information, see *Cloud Storage*.

While Functions performs very well during a warm start, a cold start that must source extensions may take several seconds to initialize. As a result, it is recommended to set the timeout to at least 60 seconds. If the dataflow to run reaches out to many other services or performs complex computation, it may be necessary to use a larger value (even several minutes).

Runtime environment variables

You must configure the function to specify the data flow to run and add any necessary runtime configuration using environment variables.

Procedure

1. Navigate to the Runtime tab and choose Runtime Environment variables in the left-hand menu.
2. Click Add variable to add the first environment variable.

You can add one or more environment variables to configure the function. The following environment variables are supported:

Variable Name	Description	Required	Default Value
FLOW_CRN	<p>The Cloudera Resource Name (CRN) for the data flow that is to be run.</p> <p>The data flow must be stored in the Catalog. This CRN should indicate the specific version of the data flow and as such will end with a suffix like /v.1.</p> <p>For more information, see <i>Retrieving data flow CRN</i>.</p>	true	--
DF_PRIVATE_KEY	<p>The Private Key for accessing the service.</p> <p>The Private Key and Access Key are used to authenticate with the Service and they must provide the necessary authorizations to access the specified data flow. For more information, see <i>Provisioning Access Key ID and Private Key</i>.</p>	true	--
DF_ACCESS_KEY	<p>The Access Key for accessing the service.</p> <p>The Private Key and Access Key are used to authenticate with the Service and they must provide the necessary authorizations to access the specified data flow. For more information, see <i>Provisioning Access Key ID and Private Key</i>.</p>	true	--

Variable Name	Description	Required	Default Value
INPUT_PORT	The name of the Input Port to use. If the specified data flow has more than one Input Port at the root group level, this environment variable must be specified, indicating the name of the Input Port to queue up the Cloud Function event. If there is only one Input Port, this variable is not required. If it is specified, it must properly match the name of the Input Port.	false	--
OUTPUT_PORT	The name of the Output Port to retrieve the results from. If no Output Port exists, the variable does not need to be specified and no data will be returned. If at least one Output Port exists in the data flow, this variable can be used to determine the name of the Output Port whose data will be sent along as the output of the function. For more information on how the appropriate Output Port is determined, see <i>Output ports</i> .	false	--
FAILURE_PORTS	A comma-separated list of Output Ports that exist at the root group level of the data flow. If any FlowFile is sent to one of these Output Ports, the function invocation is considered a failure. For more information, see <i>Output ports</i> .	false	--
DF_SERVICE_URL	The Base URL for the Cloudera Dataflow Service.	false	https://api.us-west-1.cdp.cloudera.com/
NEXUS_URL	The Base URL for a Nexus Repository for downloading any NiFi Archives (NARs) needed for running the data flow.	false	https://repository.cloudera.com/artifactory/cloudera-repos/
WORKING_DIR	The working directory, where NAR files will be expanded.	false	/tmp/working
EXTENSIONS_DOWNLOAD_DIR	The directory to which missing extensions / NiFi Archives (NARs) will be downloaded. For more information, see <i>Providing Custom Extensions / NARs</i> .	false	/tmp/extensions
STORAGE_BUCKET	A Cloud Storage bucket in which to look for custom extensions / NiFi Archives (NARs) and resources. For more information, see <i>Cloud storage</i> .	false	--

Variable Name	Description	Required	Default Value
STORAGE_EXTENSIONS_DIRECTORY	The directory in the Cloud Storage bucket to look for custom extensions / NiFi Archives (NARs). For more information, see <i>Cloud storage</i> .	false	extensions
STORAGE_RESOURCES_DIRECTORY	The directory in the Cloud Storage bucket to look for custom resources. For more information, see <i>Providing additional resources</i> .	false	resources
DISABLE_STATE_PROVIDER	If true, it disables the Firestore data flow state provider, even if the data flow has stateful processors.	false	--
FIRESTORE_STATE_COLLECTION	The Firestore collection name where the state will be stored.	false	nifi_state
KRB5_FILE	It specifies the filename of the krb5.conf file. This is necessary only if connecting to a Kerberos-protected endpoint. For more information, see <i>Configuring Kerberos</i> .	false	/etc/krb5.conf

The following environment variables apply only to HTTP triggers:

Variable Name	Description	Required	Default Value
HEADER_ATTRIBUTE_PATTERN	A Regular Expression capturing all flowfile attributes from the Output Port that should be returned as HTTP headers, if the trigger type is HTTP. For all other trigger types, this variable is ignored.	false	--
HTTP_STATUS_CODE_ATTRIBUTE	A flowfile attribute name that will set the response HTTP status code in a successful data flow, if the trigger type is HTTP. For all other trigger types, this variable is ignored. See the <i>HTTP trigger</i> section of <i>Google Cloud Function triggers</i> for more information.	false	--

Related Information

[Cloud storage](#)

[Retrieving data flow CRN](#)

[Provisioning Access Key ID and Private Key](#)

[Output ports](#)

[Providing custom extensions / NARs](#)

[Providing additional resources](#)

[Configuring Kerberos](#)

[HTTP trigger](#)

Service account

Cloud functions require a service account to use during runtime, granting the function permissions to use other Google Cloud Platform (GCP) services. There are some permissions commonly required for .

Creating/adding service account

When configuring your Cloud function:

1. Locate the Runtime service account section of the Runtime, build, connections and security settings.
2. If you already have a service account for this purpose, you may select it here. Otherwise, select Create new service account.
3. Specify the account name and ID.
4. Click Create service account.

You will need to add permissions to the Service Account separately.

When configuring Secrets accessible to the Cloud function, the console automatically prompts you to grant the necessary permissions to the Runtime Service Account. For more information on this, see the *Google Cloud Secret Manager* section in *Parameters*.

Adding additional role to service account

When providing custom extensions or resources from a Cloud Storage bucket, you need to add an additional role to the Service Account:

1. Navigate to the IAM & Admin GCP service, and select Roles on the left sidebar.
2. Click Create role.
3. Name the role.
4. Click Add permissions.
5. In the Filter section, add storage.buckets.get for Enter property name or value.
6. Click Add.
7. Do the same for the storage.buckets.list permission.
8. Click Create.
9. Select IAM from the left sidebar.
10. Locate your Cloud function's Runtime Service Account, and click the Edit pencil icon on the far right.
11. In the Role drop-down menu, select Custom your newly created Role on the right side .
12. Click Save.

For additional information, see *Cloud Storage*.

Your Service Account now allows your Cloud function to list and get objects from buckets.

Granting service account access to specific bucket

You need to grant your Service Account access to the specific bucket configured in the `CLOUD_STORAGE_BUCKET` environment variable:

1. Navigate to the Cloud Storage GCP service, and click your bucket name.
2. Select the Permissions tab under the bucket name.
3. In the Permissions section, click Add.
4. Provide the fully qualified name of the Runtime Service Account.

For example: `naaf@projectname.iam.gserviceaccount.com`

5. In Select a role, select Cloud Storage on the left, and Storage Object Viewer on the right.
6. Click Save.

Related Information

[Google Cloud Secret Manager](#)

[Cloud storage](#)

Output ports

Google Cloud functions may or may not allow the function to return a result.

The only Google Cloud Function trigger type that can directly return a result is an HTTP trigger. In this case, the output of the data flow, is written as the HTTP response to the trigger, as described in the *HTTP trigger* section of *Google Cloud Function triggers*.

Related Information

[HTTP trigger](#)

Parameters

The concept of parameterization is an important step in building a dataflow that you can run outside of the NiFi instance where it was built. NiFi allows you to define Processor and Controller Service properties at runtime instead of at build time by using Parameter Contexts.

Environment variables

Any parameter can be specified using the environment variables of the Google Cloud Function. When configuring the function, in the Runtime environment variables section of the Runtime, build, connections and security settings, add an environment variable with a name that matches the name of a parameter in your Parameter Context.

Google Cloud Secret Manager

A more secure mechanism for storing parameters is to use the Google Cloud Secret Manager, which is recommended for the sensitive properties of a data flow.

1. Navigate to the Secret Manager GCP service and click Create secret at the top-left.
2. Name the secret descriptively (this does not have to be the same name as a data flow Parameter), and enter the Secret value (or upload a file, which is most useful for mounted secrets).
3. Click Create secret at the bottom.
4. Repeat the above steps for all needed secrets.
5. In the Runtime, build, connections and security settings section of Cloud function configuration, select the Security tab.
6. Click Reference a secret.

This allows you to select an existing Secret. The console will prompt you to grant permissions to use the secret to the Runtime Service Account, which you should do.

7. If your secret is an uploaded file, select Mounted as volume for the Reference method.
8. For parameters, select Exposed as environment variable as the Reference method. Here you should enter the actual data flow Parameter Context name in the environment variable name, since this is how parameters are mapped.
9. Optionally, select latest from the Version drop-down menu.

This allows the secret value to be automatically updated if the function is redeployed.

**Important:**

To use this capability, the Parameter Contexts used in the flow must have names that are compatible with Google Cloud Secret Manager names and Environment Variable names. To ensure that these naming conventions are followed, Parameter Contexts and Parameters should use names that start with a letter and consist only of letters, numbers, and the underscore (_) character.

For example, instead of naming a Parameter Context GCP Parameter Context, use the name GCP_PARAMETER_CONTEXT when building the data flow.

Cold start

Cold start can be defined as the set-up time required to get a serverless application's environment up and running when it is invoked for the first time within a defined period.

What is a cold start?

The concept of cold start is very important when using Function as a Service solutions, especially if you are developing a customer-facing application that needs to operate in real time. A cold start is the first request that a new function instance handles. It happens because your function is not yet running and the cloud provider needs to provision resources and deploy your code before the processing can begin. This usually means that the processing is going to take much longer than expected.

Before execution, the cloud provider needs to:

- Provision the container that will run the code
- Initialize the container / runtime
- Initialize your function

before the trigger information can be forwarded to the function's handler.

For more information about the concept of cold start, see [Cold starts in Cloud Functions execution environment](#).

How to avoid a cold start?

Cloud providers offer the option to always have at least one instance of the function running to completely remove the occurrence of a cold start. You can read more about how Cloud Functions min instances reduce cold starts in [Minimum number of instances for Google Cloud Functions](#).

Cold start with Functions

When a cold starts happens with Functions, the following steps are executed before the trigger payload gets processed:

- The function retrieves the flow definition from the Catalog.
- The NiFi Stateless runtime is initialized.
- The flow definition is parsed to list the required NAR files for executing the flow.
- The required NAR files are downloaded from the specified repository unless the NARs are made available in the local storage attached to the function.
- The NARs are loaded into the JVM.
- The flow is initialized and components are started.

The step that might take a significant amount of time is the download of the NAR files. In order to reduce the time required for this step as much as possible, the best option is to list the NAR files required for the flow definition and make those files available to the function.

For the related instructions, see [Cloud storage](#) on page 13.

Cloud storage

Providing custom extensions / NARs

Cloudera Maven Repository

The binary that is deployed for running the Google Cloud function does not include the full NiFi deployment with all NARs / extensions, as it would result in very long startup times and would require more expensive configurations for the Cloud function. Instead, it packages only the NARs necessary to run Stateless NiFi. If any other extension is needed to run the data flow, it is automatically downloaded from the Cloudera Maven Repository by default when the Cloud function is initialized on a cold start.

Nexus repository

You can configure an alternative Nexus repository by setting the *NEXUS_URL* environment variable to the URL of the Nexus Repository where extensions should be downloaded. For example, to use Google's mirror of the Maven Central repository, set the *NEXUS_URL* environment variable to: <https://maven-central.storage-download.googleapis.com/maven2>



Note: Any configured URL must either be accessible through http or be served over https with a trusted certificate.

Cloud Storage bucket

If there is a need to provide custom extensions, you can use a Cloud Storage bucket:

1. Create a Cloud Storage bucket and create a directory with a descriptive name, like extensions.
2. Upload all NAR files that will need to be used to this directory.
3. Add the following environment variables to tell the Cloud function where to look for the extensions:
 - *STORAGE_BUCKET* containing the name of the bucket
 - *STORAGE_EXTENSIONS_DIRECTORY* with the full directory path of the above extensions directory



Note: If you do not specify this variable, the directory defaults to extensions.



Note: If you add a lot of extensions, it will take longer for the Cloud function to load on a cold start, and it will take more memory to load the additional files and classes. Although extensions are supported, Cloudera does not recommend uploading hundreds of MBs or more of extensions.

Providing additional resources

In addition to NiFi extensions, some data flows may also require additional resources to function. For example, a JDBC driver may be required for establishing a database connection, or a CSV file to provide data enrichment. The recommended approach for providing such resources to your Cloud function is using the Cloud Storage bucket.

1. Create a Cloud Storage bucket, if you do not already have one and create a directory with a descriptive name, like "resources".
2. Upload all resources required for the data flow to run to this directory.

These resources will appear in the Cloud function's `/tmp/resources` directory to be accessed by the data flow.

Because each deployment of a given data flow may need to reference files in a different location, depending on its environment, it is generally considered a best practice to parameterize all resources that need to be accessed by the data flow. For example, if the data flow contains a `DBCPCConnectionPool` controller service for accessing a database, it is recommended to use a Parameter for the "Database Driver Location(s)" property of the Controller Service. This allows each deployment to specify the location of the JDBC driver independently.

For example:

1. You can set the Database Driver Location(s) property to a value of `#{JDBC_DRIVER_LOCATION}`.
2. In the Cloud Storage, you can upload a file that contains your JDBC driver, `database-jdbc-driver.jar`.
3. You can add an Environment Variable named `JDBC_DRIVER_LOCATION` with a value of `/tmp/resources/database-jdbc-driver.jar`.

By taking this approach, your data flow is more reusable, as it can be deployed in many different environments.

Data flow state

About this task

By default, if your data flow contains any stateful processors (e.g. ListSFTP), this state is automatically stored in a Firestore collection called `nifi_state`. You can set this name using the `FIRESTORE_STATE_COLLECTION` environment variable. In order to integrate with Firestore, some additional steps are required. Without these steps, the data flow will still run, but the state will not be preserved.

Procedure

1. In the GCP console, navigate to the Firestore service.
2. Click Select Native Mode.
3. Select the appropriate region, and click Create database.
4. Navigate to the IAM service, and click the Edit (pencil) button on the service account principal that runs your Cloud Function.
5. Click Add Role, and add the Firebase Admin role.
6. Click Save.

The Firestore state provider can be disabled even if your data flow contains stateful processors by setting the `DISABLE_STATE_PROVIDER` Environment Variable to true.



Warning: Disabling the Firestore state provider will cause processor state to be lost between function executions.

Configuring Kerberos

Depending on the data flow that is used, you may need to enable Kerberos authentication to interact with some service(s).

To enable Kerberos, you must provide an appropriate `krb5.conf` file. This file tells the function where the Key Distribution Center (KDC) is located and which Kerberos Realm to use.

To specify this, make the `krb5.conf` file available to Function. You can do it by using a GCS bucket. See the corresponding section to learn more about this.

Additionally, you must tell Function where it can find this `krb5.conf` file. To do this, add an environment variable to your Lambda function. The environment variable must have the name `KRB5_FILE` and the value must be the fully qualified file name of the `krb5.conf` file. For example, it might use `/tmp/resources/krb5.conf`.

In order to use Keytabs in your data flow, it is also important to provide the necessary keytab files to the Function. You can accomplish it by using a GCS bucket. While the `krb5.conf` file is a system configuration, the keytab is not. The keytab is configured in the data flow on a per-component basis. For example, one Processor may use keytab `nifi-keytab-1` while some Controller Service makes use of keytab `hive-keytab-4`. The location of these files may change from deployment to deployment. For example, for an on-premises deployment, keytabs may be stored in `/etc/security/keytabs` while in Function, they may be made available in `/tmp/resources`.

So it is better to use Parameter Contexts to parameterize the location of the keytab files. This is a good practice for any file-based resource that needs to be made available to a deployment. In this case, because there is a parameter referencing the location, you need to provide Function with a value for that parameter. For more information, see *Parameters*.

Related Information

[Parameters](#)

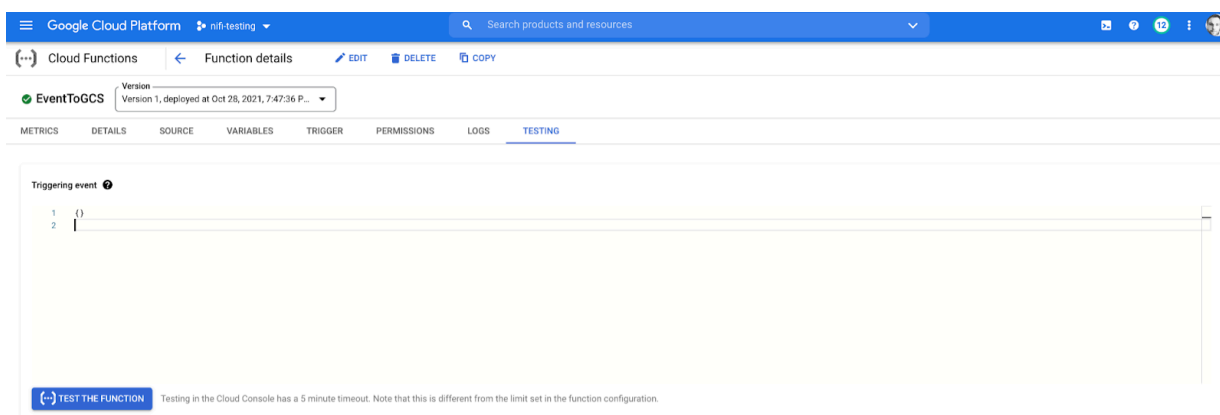
Handling failures

You can use a failure Output Port to notify the Cloud function that an invocation has failed. At that point, the Cloud function may automatically retry the invocation, if configured in the Trigger section, for up to seven days. After that, if it still fails, the Cloud function will simply drop the event notification. This could result in data loss, as it means that your Cloud function will not have a chance to process the data again.

Testing your Google Cloud function

Once you have built and verified your dataflow and uploaded it to the Catalog, you can create your Cloud function. After deploying the function, you should test that you have configured all of the settings correctly.

1. Use the Testing tab of the Cloud function to input the relevant JSON structure.



2. Click Test the function to run the data flow.

If there are problems, you can inspect the log messages to understand the problem. If the logs do not provide enough details, you may need to adjust the log levels to gain more debugging information. See the *Adjusting logs levels* for instructions.

The amount of time the cloud function takes to run depends heavily on the data flow and the number of extensions it needs. Because the cloud function may have to download some extensions from Nexus and perform initialization, a cold start may take several seconds. Even 20-30 seconds is not uncommon for a data flow with several extensions, while other data flows may complete in 10 seconds. After you run the function successfully using the Test option, it may be helpful to run several additional iterations to understand how the function will perform when a cold start is not necessary. This depends heavily on the configured data flow and other services that it may interact with. A simple data flow may complete in as a short time as 10 milliseconds, while a data flow that must perform complex transformations and reach out to one or more web services or databases for enrichment may take several seconds to complete.

It is important to adjust the Cloud function's timeout configuration in the Runtime tab of the Runtime build, connections and security settings, if the cold start takes longer than the amount of time allocated.

Related Information

[Adjusting log levels](#)

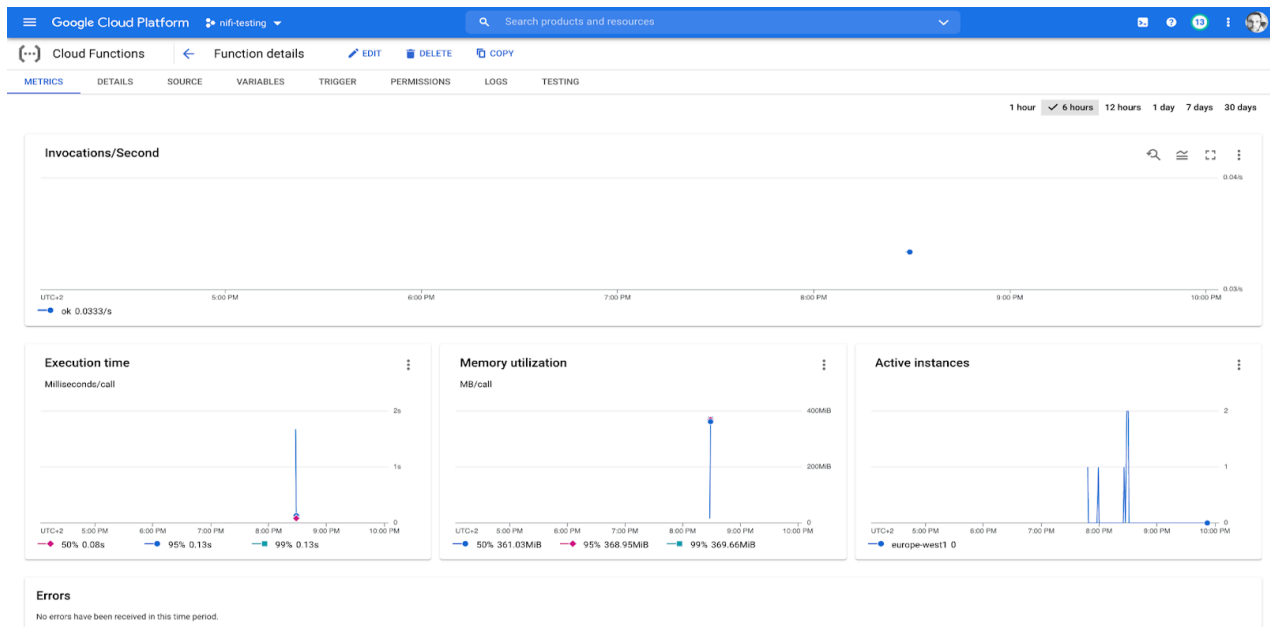
Monitoring and logs

For each invocation of your Google Cloud Function, you have access to information about the resources provisioned and used during the function's execution.



Note: With , a cold start may take a few seconds.

This view is useful in case you want to adjust the amount of memory provisioned with your function and optimize the cost charged by Google Cloud:



From this view, you can also access the logs of the function's executions:

The screenshot shows the Google Cloud Platform console interface. At the top, there's a search bar and navigation tabs for 'Cloud Functions', 'Function details', 'EDIT', 'DELETE', and 'COPY'. Below this, there are tabs for 'METRICS', 'DETAILS', 'SOURCE', 'VARIABLES', 'TRIGGER', 'PERMISSIONS', 'LOGS', and 'TESTING'. The 'LOGS' tab is selected, showing a list of log entries. The logs include timestamps, severity levels (e.g., CEST), event types (e.g., EventToGCS, EventToGCOS), and detailed messages about function execution, background tasks, and data flow processing. Some log entries are highlighted with a blue icon, indicating they are the most recent or selected.

You can enable additional monitoring capabilities in Google Cloud.

Adjusting logs levels

Functions makes use of the SLF4J Simple Logger for logging purposes. You can update the log levels by adjusting the JVM's system properties.

To do so, you need to set the `JAVA_TOOL_OPTIONS` environment variable.

- To set a log level for a given logger, you can set the JVM system property named `org.slf4j.simpleLogger.log.<logger name>` to the desired log level.

For example, if you want to enable DEBUG logging for the `StandardStatelessNiFiFunction` to view the actual data flow JSON being used, you would set the `JAVA_TOOL_OPTIONS` environment variable to `-Dorg.slf4j.simpleLogger.log.com.cloudera.naaf.StandardStatelessNiFiFunction=DEBUG`

- You can also set additional logger levels by adding multiple `-D` options separated by spaces. For example, to enable DEBUG logs on both the Functions framework and the Stateless Bootstrap class, you would set `JAVA_TOOL_OPTIONS` to a value of:

```
-Dorg.slf4j.simpleLogger.log.com.cloudera.naaf.StandardStatelessNiFiFunction=DEBUG
-Dorg.slf4j.simpleLogger.log.org.apache.nifi.stateless.bootstrap.StatelessBootstrap=DEBUG
```

Google Cloud Function triggers

All Google Cloud Function triggers are supported with . In this section you can review the most commonly used triggers and examples of the FlowFile's content that would be generated following a triggering event.

Background triggers

The Output Port data flow logic is simple with background functions.

If one Output Port is present, and its name is "failure", any data routed to that Output Port will cause the function invocation to fail.

If two Output Ports are present, and one of them is named "failure", that Output Port will behave as the "failure" port above. Any other Output Port is considered a successful invocation, but the function will do nothing with data routed here.

Any Port with the name "failure" (ignoring case) is considered a Failure Port. Additionally, if the *FAILURE_PORTS* environment variable is specified, any port whose name is in the comma-separated list will be considered a Failure Port.

Google Cloud Storage

You can use this trigger, for example, to start your function whenever a file is landing into Google Cloud Storage:

Trigger

Cloud Storage

Trigger type
Cloud Storage

Event type *
Finalize/Create

Bucket *
pvi-nifi **BROWSE**

☐ Retry on failure

SAVE **CANCEL**

The FlowFile's content would look like this:

```

1 {
2   "bucket": "pvi-nifi",
3   "contentType": "text/plain",
4   "crc32c": "LnPMaQ==",
5   "etag": "CPW+14ze7fWCEAE=",
6   "generation": "1635445725593457",
7   "id": "pvi-nifi/dummyfile.txt/1635445725593457",
8   "kind": "storageObject",
9   "md5Hash": "hKh3F8pES0116kg0N700BA==",
10  "mediaLink": "https://www.googleapis.com/download/storage/v1/b/pvi-nifi/o/dummyfile.txt?generation=1635445725593457&alt=media",
11  "metageneration": "1",
12  "name": "dummyfile.txt",
13  "selfLink": "https://www.googleapis.com/storage/v1/b/pvi-nifi/o/dummyfile.txt",
14  "size": "7842",
15  "storageClass": "STANDARD",
16  "timeCreated": "2021-10-28T18:28:45.602Z",
17  "timeStorageClassUpdated": "2021-10-28T18:28:45.602Z",
18  "updated": "2021-10-28T18:28:45.602Z"
19 }

```

Google Cloud Pub/Sub

You can use this trigger for processing events received in a Pub/Sub topic:

Trigger

 **Cloud Pub/Sub**

Trigger type
Cloud Pub/Sub

Select a Cloud Pub/Sub topic *
projects/nifi-testing-320023/topics/naaf-topic

☐ Retry on failure 

SAVE CANCEL

The FlowFile's content would look like this:

```

1 {
2   "@type": "type.googleapis.com/google.pubsub.v1.PubsubMessage",
3   "attributes": {
4     "myAttributeKey": "myAttributeValue"
5   },
6   "data": "eyJteU1lc3NhZ2U10iJUAglzIGlzIG15IG1lc3NhZ2UgcGF5bG9hZCE1fQ\u003d\u003d"
7 }
8

```



Note:

It is not possible to batch messages with this trigger. The function is executed once for each message in the Pub/Sub topic.

The payload of the message is Base 64 encoded.


HTTP trigger

This trigger can be used to expose an HTTP endpoint (with or without required authentication) allowing for clients to send data to the function.

✓ NaaF_HTTP-1

europe-west1

Trigger


 **HTTP**

Trigger type
HTTP

URL

https://europe-west1-nifi-testing-320023.cloudfunctions.net/NaaF_HTTP-1

☒ Require HTTPS

 **SAVE**

CANCEL

This trigger would have Google Cloud creating and exposing an endpoint such as: `https://europe-west1-nifi-testing-320023.cloudfunctions.net/NaaF_HTTP-1`

The FlowFile created from the client call will be as follow:

- The content of the FlowFile is the exact body of the client request
- Attributes of the FlowFile are created for each HTTP headers of the client request. Each attribute is prefixed: `gcp.http.header.<header key> = <header value>`

Since Cloud Functions with an HTTP trigger are invoked synchronously, they directly return a response.

The Output Port semantics are the following.

If no Output Port is present in the data flow's root group, no output will be provided from the function invocation, but a default success message (`DataFlow completed successfully`) is returned in the HTTP response with a status code of 200.

If one Output Port is present, and its name is "failure", any data routed to that Output Port will cause the function invocation to fail. No output is provided as the output of the function invocation, and an error message (`DataFlow completed successfully: + error message`) is returned from the HTTP response, with a status code of 500.

If two Output Ports are present, and one of them is named "failure", that Output Port will behave as the "failure" port above. The other Output Port is considered a successful invocation. When the data flow is invoked, if a single FlowFile is routed to this Port, the contents of the FlowFile are emitted as the output of the function. If more than one FlowFile is sent to the Output Port, the data flow is considered a failure, as GCP requires that a single entity be provided as its output. A MergeContent or MergeRecord Processor may be used in order to assemble multiple FlowFiles into a single output FlowFile if necessary.

Any Port with the name "failure" (ignoring case) will be considered a Failure Port. Additionally, if the *FAILURE_PORTS* environment variable is specified, any port whose name is in the comma-separated list will be considered a Failure Port.

If two or more Output Ports, other than failure ports, are present, the *OUTPUT_PORT* environment variable must be provided. In such a case, this environment variable must match the name of an Output Port at the data flow's root group (case is NOT ignored). Anything that is routed to the specified port is considered the output of the function invocation. Anything routed to any other port is considered a failure.

Note: In any successful response, the mime.type flowfile attribute is used as the Content-Type of the response.

Additionally, there are two environment variables that can be configured to customize the HTTP response:

- *HEADER_ATTRIBUTE_PATTERN* can provide a regular expression matching any flowfile attributes to be included as HTTP response headers. This means that any attributes to be included as headers should be named as the headers themselves.
- *HTTP_STATUS_CODE_ATTRIBUTE* specifies a flowfile attribute that sets the HTTP status code in the response. This allows a code other than 200 for 'success' or 500 for 'failure' to be returned. If this environment variable is specified but the attribute is not set, the status code in the HTTP response will default to 200.

In the "failure" scenario, the *HTTP_STATUS_CODE_ATTRIBUTE* is not used. So, if a non-500 status code is required, it must be provided in a successful data flow path. For example, if a 409 Conflict is desired in a particular case, you can achieve it as follows:

1. In the data flow, include an UpdateAttribute processor that sets an arbitrary attribute, for example `http.status.code`, to the value 409.
2. Optionally, add a ReplaceText processor that sets the contents of the flowfile using Always Replace to whatever HTTP response body is desired.
3. Route this part of the flow to the main Output Port used for the success case.
4. In the GCP Cloud function, set the Runtime environment variable *HTTP_STATUS_CODE_ATTRIBUTE* to `http.status.code`, and deploy it.

You may also use this same method to set a non-200 success code.

Creating a Google Cloud function using CLI

Google Cloud Functions offers two product versions: Cloud Functions 1st Generation, the original version and Google Cloud Functions 2nd Generation, a new version built on Cloud Run and Eventarc to provide an enhanced feature set. See the below examples on how to deploy a Function in both versions of Google Cloud Functions, using the Google Cloud CLI.

Before you begin

- Your NiFi flow definition is stored in the Catalog and you have the CRN corresponding to the flow version you want to execute as a function
- You have created a Machine User with the proper role and you have its Access Key and Private Key credentials
- You have installed and configured the Google Cloud CLI on your local machine

1st generation (1st gen)

HTTP Trigger

To deploy a Google Cloud Function with an unauthenticated HTTP trigger, you can use the following example:

```
gcloud functions deploy myfunctionname \
  --region=europe-west1 \
  --runtime=java11 \
  --source=gs://my-bucket/naaf-gcp-cloud-functions-1.0.0-bin.zip \
  --memory=1024MB \
```

```
--timeout=300 \
--trigger-http \
--allow-unauthenticated \
--entry-point=com.cloudera.naaf.gcp.cloud.functions.StatelessNiFiHttpFunction \
--update-labels=mylabelkey=mylabelvalue \
--set-env-vars=FLOW_CRN=crn:cdp:df:us-west-1:00000000-0000-0000-0000-000000000000:flow:my-flow/v.1,DF_PRIVATE_KEY=00000000000000000000000000000000,DF_ACCESS_KEY=00000000-0000-0000-0000-000000000000
```

GCS Trigger

To deploy a Google Cloud Function with a trigger on a GCS bucket, you can use the following example:

```
gcloud functions deploy myfunctionname \
--region=europe-west1 \
--runtime=java11 \
--source=gs://my-bucket/naaf-gcp-cloud-functions-1.0.0-bin.zip \
--memory=1024MB \
--timeout=300 \
--trigger-bucket=my-trigger-bucket
--entry-point=com.cloudera.naaf.gcp.cloud.functions.StatelessNiFiBackgroundFunction \
--update-labels=mylabelkey=mylabelvalue \
--set-env-vars=FLOW_CRN=crn:cdp:df:us-west-1:00000000-0000-0000-0000-000000000000:flow:my-flow/v.1,DF_PRIVATE_KEY=00000000000000000000000000000000,DF_ACCESS_KEY=00000000-0000-0000-0000-000000000000
```

Pub/Sub Trigger

To deploy a Google Cloud Function with a trigger on a Pub/Sub topic, you can use the following example:

```
gcloud functions deploy myfunctionname \
--region=europe-west1 \
--runtime=java11 \
--source=gs://my-bucket/naaf-gcp-cloud-functions-1.0.0-bin.zip \
--memory=1024MB \
--timeout=300 \
--trigger-topic=my-trigger-topic
--entry-point=com.cloudera.naaf.gcp.cloud.functions.StatelessNiFiBackgroundFunction \
--update-labels=mylabelkey=mylabelvalue \
--set-env-vars=FLOW_CRN=crn:cdp:df:us-west-1:00000000-0000-0000-0000-000000000000:flow:my-flow/v.1,DF_PRIVATE_KEY=00000000000000000000000000000000,DF_ACCESS_KEY=00000000-0000-0000-0000-000000000000
```

2nd generation (2nd gen)

For Google Cloud Functions 2nd gen, only HTTPS triggers are supported and a Google Eventarc trigger can be added to trigger the function through HTTPS calls according to configurable events (files added to a bucket, event published to a Pub/Sub topic, and so on).

You can use the following example:

```
gcloud functions deploy myfunctionname \
--gen2 \
--region=europe-west1 \
--runtime=java11 \
--source=gs://my-bucket/naaf-gcp-cloud-functions-1.0.0-bin.zip \
--memory=1024MB \
--timeout=300 \
--trigger-http \
```

```
--entry-point=com.cloudera.naaf.gcp.cloud.functions.StatelessNiFiHttpFunction \
--update-labels=mylabelkey=mylabelvalue \
--set-env-vars=FLOW_CRN=crn:cdp:df:us-west-1:000000000-0000-0000-0000-00000000
00000:flow:my-flow/v.1,DF_PRIVATE_KEY=00000000000000000000000000000000,DF
_ACCESS_KEY=00000000-0000-0000-0000-000000000000
```

It is then possible to create an Eventarc trigger and link it to the deployed function. This is an example with Pub/Sub:

```
gcloud eventarc triggers create pubsub \
--location=europe-west1 \
--service-account=0000000000-compute@developer.gserviceaccount.com \
--transport-topic=projects/my-project/topics/my-topic \
--destination-run-service=myfunctionname \
--destination-run-region=europe-west1 \
--destination-run-path="/" \
--event-filters="type=google.cloud.pubsub.topic.v1.messagePublished"
```