Tips & Tricks

cloud.google.com/functions/docs/bestpractices/tips

This document describes best practices for designing, implementing, testing, and deploying Cloud Functions.

Note: Several of the recommendations in this document center around what is known as a *cold start*. Functions are stateless, and the execution environment is often initialized from scratch, which is called a cold start. Cold starts can take significant amounts of time to complete. It is best practice to avoid unnecessary cold starts, and to streamline the cold start process to whatever extent possible (for example, by avoiding unnecessary dependencies).

Correctness

This section describes general best practices for designing and implementing Cloud Functions.

Write idempotent functions

Your functions should produce the same result even if they are called multiple times. This lets you retry an invocation if the previous invocation fails partway through your code. For more information, see retrying background functions.

Ensure HTTP functions send an HTTP response

If your function is HTTP-triggered, remember to send an HTTP response, as shown below. Failing to do so can result in your function executing until timeout. If this occurs, you will be charged for the entire timeout time. Timeouts may also cause unpredictable behavior or cold starts on subsequent invocations, resulting in unpredictable behavior or additional latency.

```
from flask import

 escape


def hello_http(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
        <https://flask.palletsprojects.com/en/1.1.x/api/#incoming-request-data>
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`
        <https://flask.palletsprojects.com/en/1.1.x/api/#flask.make_response>.
    """
    request_json = request.get_json(silent=True)
    request_args = request.

args


    if request_json and 'name' in request_json:
        name = request_json['name']
    elif request_args and 'name' in request_args:
        name = request_args['name']
    else:
        name = 'World'
    return 'Hello {}!'.format(escape(name))
```

Do not start background activities

Background activity is anything that happens after your function has terminated. A function invocation finishes once the function returns or otherwise signals completion, such as by calling the `callback` argument in Node.js background functions. Any code run after graceful termination cannot access the CPU and will not make any progress.

Note: If a Node.js background function returns a Promise, Cloud Functions ensures that the Promise is settled before terminating.
In addition, when a subsequent invocation is executed in the same environment, your background activity resumes, interfering with the new invocation. This may lead to unexpected behavior and errors that are hard to diagnose. Accessing the network after a function terminates usually leads to connections being reset (`ECONNRESET` error code).

Background activity can often be detected in logs from individual invocations, by finding anything that is logged after the line saying that the invocation finished. Background activity can sometimes be buried deeper in the code, especially when asynchronous operations such as callbacks or timers are present. Review your code to make sure all asynchronous operations finish before you terminate the function.

Always delete temporary files

Local disk storage in the temporary directory is an in-memory filesystem. Files that you write consume memory available to your function, and sometimes persist between invocations. Failing to explicitly delete these files may eventually lead to an out-of-memory error and a subsequent cold start.

You can see the memory used by an individual function by selecting it in the list of functions in the Cloud Console and choosing the *Memory usage* plot.

Do not attempt to write outside of the temporary directory, and be sure to use platform/OS-independent methods to construct file paths.

You can reduce memory requirements when processing larger files using pipelining. For example, you can process a file on Cloud Storage by creating a read stream, passing it through a stream-based process, and writing the output stream directly to Cloud Storage.

Note: For an example of this process in Node.js, see image resizing using Node.js Stream and Sharp.

Tools

This section provides guidelines on how to use tools to implement, test, and interact with Cloud Functions.

Local development

Function deployment takes a bit of time, so it is often faster to test the code of your function locally.

Developers can use the Cloud Functions Framework for local testing.

Error reporting

In languages that use exception handling, do not throw uncaught exceptions, because they force cold starts in future invocations. See the Error Reporting guide for information on how to properly report errors.

Do not manually exit

Manually exiting can cause unexpected behavior. Please use the following language-specific idioms instead:

Node.js Python Go Java C# Ruby PHP
Do not use `sys.exit()`. HTTP functions should explicitly return a response as a string, and event-driven functions will exit once they return a value (either implicitly or explicitly).

Use Sendgrid to send emails

Cloud Functions does not allow outbound connections on port 25, so you cannot make non-secure connections to an SMTP server. The recommended way to send emails is to use SendGrid. You can find other options for sending email in the Sending Email from an Instance tutorial for Google Compute Engine.

Performance

This section describes best practices for optimizing performance.

Use dependencies wisely

Because functions are stateless, the execution environment is often initialized from scratch (during what is known as a *cold start*). When a cold start occurs, the global context of the function is evaluated.

If your functions import modules, the load time for those modules can add to the invocation latency during a cold start. You can reduce this latency, as well as the time needed to deploy your function, by loading dependencies correctly and not loading dependencies your function doesn't use.

Use global variables to reuse objects in future invocations

There is no guarantee that the state of a Cloud Function will be preserved for future invocations. However, Cloud Functions often recycles the execution environment of a previous invocation. If you declare a variable in global scope, its value can be reused in subsequent invocations without having to be recomputed.

This way you can cache objects that may be expensive to recreate on each function invocation. Moving such objects from the function body to global scope may result in significant performance improvements. The following example creates a heavy object only once per function instance, and shares it across all function invocations reaching the given instance:

```
# Global (instance-wide) scope
# This computation runs at instance cold-start
instance_var = heavy_computation()
def scope_demo(request):
    """
    HTTP Cloud Function that declares a variable.
    Args:
        request (flask.Request): The request object.
        <http://flask.pocoo.org/docs/1.0/api/#flask.Request>
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`
        <http://flask.pocoo.org/docs/1.0/api/#flask.Flask.make_response>.
    """
    # Per-function scope
    # This computation runs every time this function is called
    function_var = light_computation()
    return 'Instance: {}; function: {}'.format(instance_var, function_var)
```

It is particularly important to cache network connections, library references, and API client objects in global scope. See Optimizing Networking for examples.

Note: Background tasks should not be performed outside of the duration of a request. If you need to initialize a global variable with the result from an expensive background task, perform the task during your function's execution and store its result before sending a response.

Do lazy initialization of global variables

If you initialize variables in global scope, the initialization code will always be executed via a cold start invocation, increasing your function's latency. In certain cases, this causes intermittent timeouts to the services being called if they are not handled appropriately in a `try` / `catch` block. If some objects are not used in all code paths, consider initializing them lazily on demand:

```
# Always initialized (at cold-start)
non_lazy_global = file_wide_computation()
# Declared at cold-start, but only initialized if/when the function executes
lazy_global = None
def lazy_globals(request):
    """
    HTTP Cloud Function that uses lazily-initialized globals.
    Args:
        request (flask.Request): The request object.
        <http://flask.pocoo.org/docs/1.0/api/#flask.Request>
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`
        <http://flask.pocoo.org/docs/1.0/api/#flask.Flask.make_response>.
    """
    global lazy_global,

 non_lazy_global


    # This value is initialized only if (and when) the function is called
    if not lazy_global:
        lazy_global = function_specific_computation()
    return 'Lazy: {}, non-lazy: {}.'.format(lazy_global, non_lazy_global)
```

This is particularly important if you define several functions in a single file, and different functions use different variables. Unless you use lazy initialization, you may waste resources on variables that are initialized but never used.

Note: this technique can also be used when importing dependencies in Node.js and Python, albeit at the expense of code readability.

Additional resources

Find out more about optimizing performance in the "Google Cloud Performance Atlas" video Cloud Functions Cold Boot Time.