Development tips

This guide provides best practices for designing, implementing, testing, and deploying a Cloud Run service. For more tips, see Migrating an Existing Service (https://cloud.google.com/run/docs/migrating).

Writing effective services

This section describes general best practices for designing and implementing a Cloud Run service.

Avoiding background activities

When an application running on Cloud Run finishes handling a request, the container instance's access to CPU will be disabled or severely limited. Therefore, you should not start background threads or routines that run outside the scope of the request handlers.

Running background threads can result in unexpected behavior because any subsequent request to the same container instance resumes any suspended background activity.

Background activity is anything that happens after your HTTP response has been delivered. Review your code to make sure all asynchronous operations finish before you deliver your response.

If you suspect there may be background activity in your service that is not readily apparent you can check your logs: look for anything that is logged after the entry for the HTTP request.

Deleting temporary files

In the Cloud Run (fully managed) environment disk storage is an in-memory filesystem. Files written to disk consume memory otherwise available to your service, and can persist between invocations. Failing to delete these files can eventually lead to an out-of-memory error and a subsequent cold start.

Reporting errors

Handle all exceptions and do not let your service crash on errors. A crash leads to a cold start while traffic is queued for a replacement container instance.

See the Error reporting guide (https://cloud.google.com/run/docs/error-reporting) for information on how to properly report errors.

Optimizing performance

This section describes best practices for optimizing performance.

Starting services quickly

Because container instances are scaled as needed, a typical method is to initialize the execution environment completely. This kind of initialization is called "cold start". If a client request triggers a cold start, the container instance startup results in additional latency.

The startup routine consists of:

- Starting the service
    - Retrieving the container image
    - Starting the container
    - Running the entrypoint (https://docs.docker.com/engine/reference/builder/#entrypoint) command to start your server.
- Checking for the open service port.

Optimizing for service startup speed minimizes the latency that delays a container instance from serving requests.

Using dependencies wisely

If you use a dynamic language with dependent libraries, such as importing modules in Node.js, the load time for those modules adds latency during a cold start. Reduce startup latency in these ways:

- Minimize the number and size of dependencies to build a lean service.
- Lazily load code that is infrequently used, if your language supports it.
- Use code-loading optimizations such as PHP's composer autoloader optimization (https://getcomposer.org/doc/articles/autoloader-optimization.md).

Using global variables

In Cloud Run, you cannot assume that service state is preserved between requests. However, Cloud Run does reuse individual container instances to serve ongoing traffic, so you can declare a variable in global scope to allow its value to be reused in subsequent invocations. Whether any individual request receives the benefit of this reuse cannot be known ahead of time.

You can also cache objects in memory if they are expensive to recreate on each service request. Moving this from the request logic to global scope results in better performance.

NODE.JS PYTHON GO JAVA

functions/tips/index.js (https://github.com/GoogleCloudPlatform/nodejs-docs-samples/blob/master/functions/tips/index.js)

:OM/GOOGLECLOUDPLATFORM/NODEJS-DOCS-SAMPLES/BLOB/MASTER/FUNCTIONS/TIPS/INDEX.JS)

FEEDBACK (#)

```
// Global (instance-wide) scope
// This computation runs at instance cold-start
const instanceVar = heavyComputation();

/**
 * HTTP function that declares a variable.
 *
 * @param {Object} req request context.
 * @param {Object} res response context.
 */
exports.scopeDemo = (req, res) => {
  // Per-function scope
  // This computation runs every time this function is called
  const functionVar = lightComputation();
```

```
  res.send(`Per instance: ${instanceVar}, per function: ${functionVar}`);
};
```

Performing lazy initialization of global variables

The initialization of global variables always occurs during startup, which increases cold start time. Use lazy initialization for infrequently used objects to defer the time cost and decrease cold start times.

NODE.JS PYTHON GO JAVA

functions/tips/index.js (https://github.com/GoogleCloudPlatform/nodejs-docs-samples/blob/master/functions/tips/index.js)

COM/GOOGLECLOUDPLATFORM/NODEJS-DOCS-SAMPLES/BLOB/MASTER/FUNCTIONS/TIPS/INDEX.JS)

FEEDBACK (#)

```
// Always initialized (at cold-start)
const nonLazyGlobal = fileWideComputation();

// Declared at cold-start, but only initialized if/when the function executes
let lazyGlobal;

/**
 * HTTP function that uses lazy-initialized globals
 *
 * @param {Object} req request context.
 * @param {Object} res response context.
 */
exports.lazyGlobals = (req, res) => {
  // This value is initialized only if (and when) the function is called
  lazyGlobal = lazyGlobal || functionSpecificComputation();

  res.send(`Lazy global: ${lazyGlobal}, non-lazy global: ${nonLazyGlobal}`);
};
```

Minimizing container image size

A larger container image size has several effects:

- Increased security vulnerability because more code is a larger attack surface.
- Slower build time for your container image while many files are downloaded.
- Slower deployment time for your service as the container image is prepared for use in a new revision (https://cloud.google.com/run/docs/resource-model).
- Increased network egress costs (https://cloud.google.com/container-registry/pricing) with Container Registry if your container storage bucket is geographically distant from your service region.

On Cloud Run, the size of your container image *does not affect* cold start or request processing time and does not count towards the available memory (https://cloud.google.com/run/docs/configuring/memory-limits) of your container.

See below for more on container security (#container-security).

To build a minimal container, consider working off a lean base image such as:

- alpine (https://hub.docker.com/_/alpine)
- distroless (https://github.com/GoogleContainerTools/distroless)
- scratch (https://hub.docker.com/_/scratch)

Ubuntu (https://hub.docker.com/_/ubuntu) is larger in size, but is a commonly used base image with a more complete out-of-box server environment.

If your service has a tool-heavy build process consider using multi-stage builds (https://docs.docker.com/develop/develop-images/multistage-build/#use-multi-stage-builds) to keep your container light at run time.

These resources provide further information on creating lean container images:

- Best Practices for Building Containers (https://cloud.google.com/solutions/best-practices-for-building-containers#build-the-smallest-image-possible)
- Kubernetes best practices: How and why to build small container images (https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-how-and-why-to-build-small-container-images)
- 7 best practices for building containers (https://cloud.google.com/blog/products/gcp/7-best-practices-for-building-containers)

Using concurrency

Cloud Run supports configurable concurrency (https://cloud.google.com/run/docs/about-concurrency) which has special considerations for your service. This is different from Cloud Functions, which does not support concurrency.

Tuning concurrency for your service

You can enable a service's container instances to serve multiple requests simultaneously, that is, "concurrently". The number of concurrent requests that each container instance can serve is limited by the technology stack and the use of shared resources like global variables and database connections.

To optimize your service for maximum stable concurrency:

1. Optimize your service performance.
2. Set your expected level of concurrency support in any code-level concurrency configuration. Not all technology stacks require such a setting.
3. Deploy your service.
4. Set Cloud Run concurrency for your service equal or less than any code-level configuration. If there is no code-level configuration, use your expected concurrency.
5. Use load testing (https://en.wikipedia.org/wiki/Load_testing) tools that support a configurable concurrency. You need to confirm that your service remains stable under expected load and concurrency.
6. If the service does poorly, go to step 1 to improve the service or step 2 to reduce the concurrency. If the service does well, go back to step 2 and increase the concurrency.

Continue iterating until you find the maximum stable concurrency.

Matching memory to concurrency

Each request your service handles requires some amount of additional memory. So, when you adjust concurrency up or down, make sure you adjust your memory limit as well.

Avoiding mutable global state

If you want to leverage mutable global state in a concurrent context, take extra steps in your code to ensure this is done safely. Minimize contention by limiting global variables to one-time initialization and reuse as described above under Performance (#performance).

If you use mutable global variables in a service that serves multiple requests at the same time, make sure to use locks or mutexes to prevent race conditions.

Container security

Many general purpose software security practices apply to containerized applications. There are some practices that are either specific to containers or that align with the philosophy and architecture of containers.

To improve container security:

- Use actively maintained and secure base images such as Container Registry's managed base images (https://cloud.google.com/container-registry/docs/managed-base-images). or Docker Hub's official images (https://hub.docker.com/search?q=&type=image&image_filter=official)
- Apply security updates to your services by regularly rebuilding container images and redeploying your services.
- Include in the container only what is necessary to run your service. Extra code, packages, and tools are potential security vulnerabilities. See above for the related performance impact (#mimimize-container).
- Implement a deterministic build process (https://en.wikipedia.org/wiki/Reproducible_builds) that includes specific software and library versions. This prevents unverified code from being included in your container.
- Set your container to run as a user other than `root` with the Dockerfile `USER` statement (https://docs.docker.com/engine/reference/builder/#user). Some container images may already have a specific user configured.

Automated Security Scanning

Enable the Container Registry image vulnerability scanner (https://cloud.google.com/container-registry/docs/get-image-vulnerabilities) for security scanning of container images stored in the Container Registry (https://cloud.google.com/container-registry).

If using Cloud Run for Anthos on Google Cloud, you may further use Binary Authorization (https://cloud.google.com/binary-authorization/docs/quickstart) to ensure only secure container images are deployed.

Last updated November 14, 2019.