Deploy Node.js to Cloud Run

Introduction

From the Cloud Run documentation:

> Cloud Run is a managed compute platform that enables you to run stateless containers that are invocable via web requests or Pub/Sub events. Cloud Run is serverless: it abstracts away all infrastructure management, so you can focus on what matters most — building great applications.
>
> — Cloud Run documentation intro

If that scratches your head, think of Cloud Run as Cloud Functions, but for containers. Instead of function instances in the case of Cloud Functions, Cloud Run deploys container instances in serverless fashion. It should be stateless, because like functions, the containers can come and go (don't expect them to stick around for long periods of time). Like functions also, the containers are invocable via HTTP requests or Pub/Sub events.

Cloud Run is built on Knative, which was originally created by Google to build and run serverless applications on Kubernetes.

> Knative offers features like scale-to-zero, autoscaling, in-cluster builds, and eventing framework for cloud-native applications on Kubernetes.
>
> — GCP Knative documentation

There are two flavors of Cloud Run:

- Cloud Run (fully managed)
- Cloud Run for Anthos

In this chapter we will concern ourselves with Cloud Run (fully managed).

Advantages of Cloud Run

- Have a container? Now you can deploy to production in seconds, from a single command.
- Serverless, pay only for the exact resources you use.
- Fast autoscaling. Unlike App Engine flexible environment, scaling with Cloud Run is much faster because it does not need to spin up a new VM to support a new container instance.
- With Cloud Run for Anthos (Read: Pricey enterprise version of Cloud Run / Kubernetes), you

can deploy containers to on-prem, GCP cloud, and *any* cloud provider out there (mix and match all you like in one interface).

- Run any language, any library, any binary. You supply the container, Cloud Run will just *run* it!
- Use your familiar Docker CLI and other container workflows and standards. When we dockerized our application, nothing about it was GCP specific.
- A Cloud Run container instance can handle up to 80 concurrent requests at a time. A Cloud Functions instance can only handle one request at a time.
- HTTPS URLs are for each deployed service. TLS termination is handled for you.
- Custom domain support is built in. All you need to do is map services to your own domains.
- Roll back to a previous version easily with just one click.
- Gradually roll out a new revision (no downtime).
- Traffic split between multiple revisions.
- Can use Cloud SQL as a database, Cloud SQL Proxy connections are natively supported.

Disadvantages of Cloud Run

- Cold start considerations still apply.
- Like any serverless service, if your application is being called *continuously*, it would likely be much cheaper to run this on App Engine or something else that is always on.
- At time of writing, Cloud Run cannot connect to VPC network, and cannot be used with Cloud Load Balancing. See here for more details. This means you cannot connect to a VM instance via private network in your VPC, it has to go through the public internet.
- Your app must be configured to not run anything in the background, especially not after the request has ended (background is fine if your request is still in progress). You should wait for background code to finish before returning.
- Your container must be stateless. If state is required, you need to store that state externally. Caching computed results is possible, but the first time you run it will be slightly more expensive.
- Can't run always on services like WebSockets.
- Each request must be returned within a timeout limit (5 minutes by default, configurable up to 15 minutes).

Use Cases for Cloud Run

- Host websites and applications (e.g. nginx, Express.js, django, Ruby on Rails), and have them talk to Cloud SQL, to render dynamic HTML pages.

- REST API for mobile backends, talking to Cloud SQL or Cloud Firestore (NoSQL).
- Custom app to integrate with G Suite (Docs, Sheets, etc.), only billed when used.
- Lightweight data transformation. Like Cloud Functions, Cloud Run containers can be the glue code that integrates between multiple GCP services, for instance between Pub/Sub and BigQuery. The keyword is *lightweight* because each Cloud Run request has a timeout limit of 15 minutes.
- With Cloud Scheduler, you can securely trigger a Cloud Run service on a schedule. Possible use cases in this context include:
  - Performing backups on a timed basis
  - Performing recurring admin tasks, such as batch sending emails, generating reports, and deleting old data
  - Generating documents, such as bills and invoices
- With third-party webhooks, you can configure Cloud Run service endpoints to respond and react to webhook events, and pass on the message to other GCP services.

Deploy to Cloud Run

We have already done the hard work of dockerizing the Bookshelf app in the previous chapter. Almost surprisingly, that's almost all the work that needs to be done.

You need to make sure that you have configured your application to use the port defined in the `PORT` environment variable. Cloud Run always sets `PORT` to 8080, but you should always try to read the port value from `PORT` anyways to ensure container portability, and not hardcode the value.

For more troubleshooting tips when deploying your container to Cloud Run, see the Troubleshooting page.

Deploy using the following command:

```
export PROJECT_ID=[PROJECT_ID]
gcloud run deploy --image gcr.io/$PROJECT_ID/bookshelf-node:v1 \
  --platform managed \
  --region us-central1 \
  --project $PROJECT_ID
```

```
Service name (bookshelf-node):
API [run.googleapis.com] not enabled on project [518599292794]. Would
you like to enable and retry (this will take a few minutes)? (y/N)?  y

Enabling service [run.googleapis.com] on project [518599292794]...
Operation "operations/acf.c7fdbcb8-184b-44bc-9a83-b4f32e8345ce" finished successfully.
Allow unauthenticated invocations to [bookshelf-node] (y/N)?  y

Deploying container to Cloud Run service [bookshelf-node] in project [gcp-demo-123456]
region [us-central1]
✓ Deploying new service... Done.
  ✓ Creating Revision...
  ✓ Routing traffic...
  ✓ Setting IAM Policy...
Done.
Service [bookshelf-node] revision [bookshelf-node-00001-zat] has been deployed and is
serving 100 percent of traffic at
https://bookshelf-node-bvr7gw4wza-uc.a.run.app
```

If you visit the `.run.app` URL, we see our familiar Moby Dick.

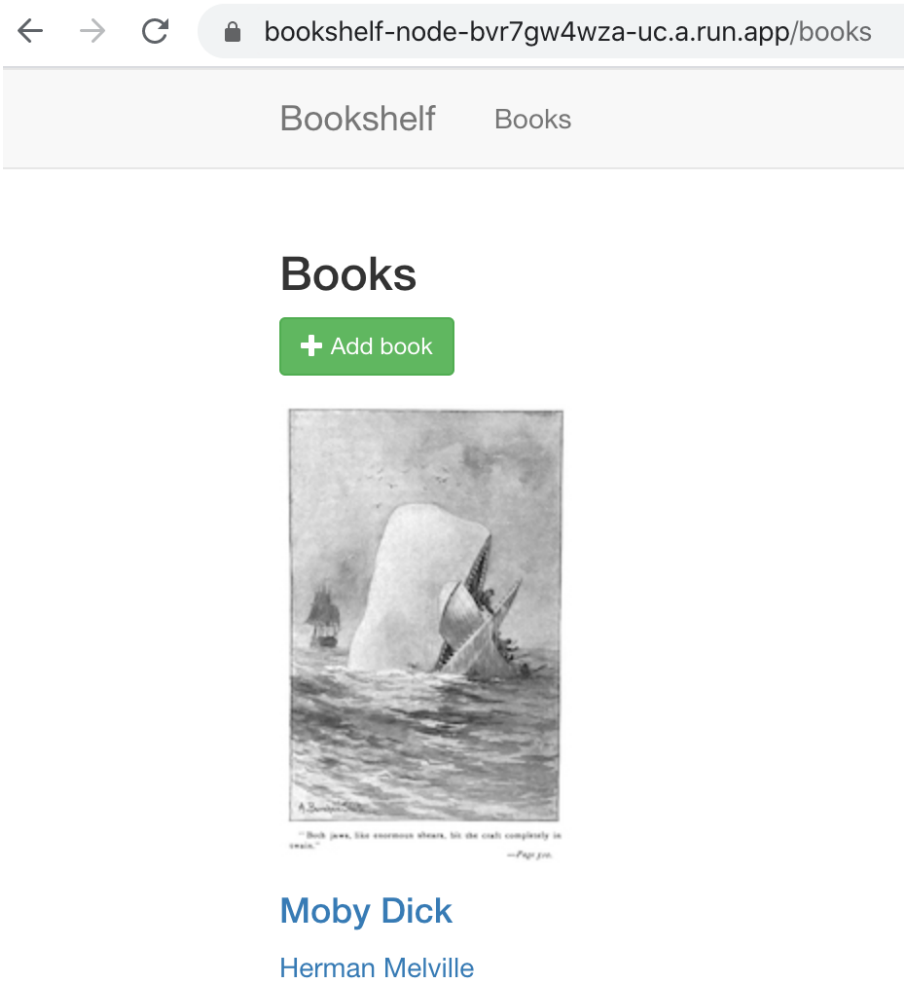

Figure 33. Moby Dick swimming via Cloud Run.

But if we try to upload a new image, we see a cryptic error:

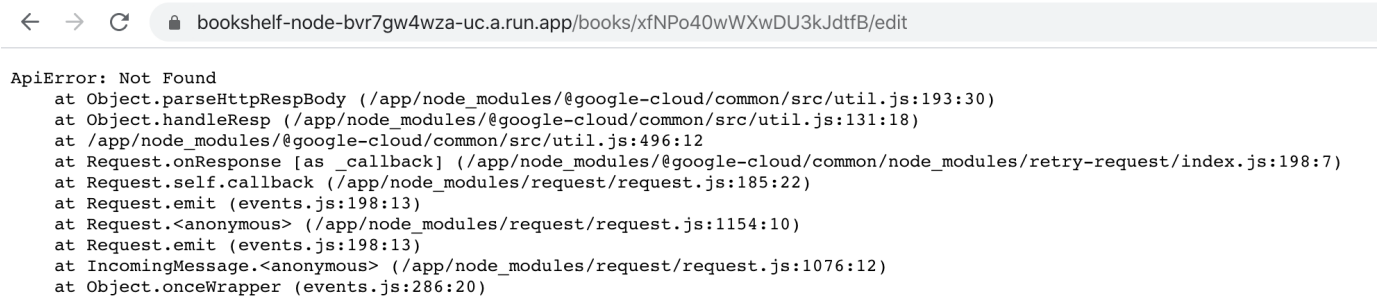

Figure 34. Upload image returns ApiError.

The app in Cloud Run could not find the Cloud Storage bucket because we did not configure the `GOOGLE_CLOUD_PROJECT` environment variable:

images.js

```
const GOOGLE_CLOUD_PROJECT = process.env['GOOGLE_CLOUD_PROJECT'];
const CLOUD_BUCKET = GOOGLE_CLOUD_PROJECT + '_bucket';
```

To fix this via the Cloud Console:

1. Go to `bookshelf-node` service in Cloud Run:

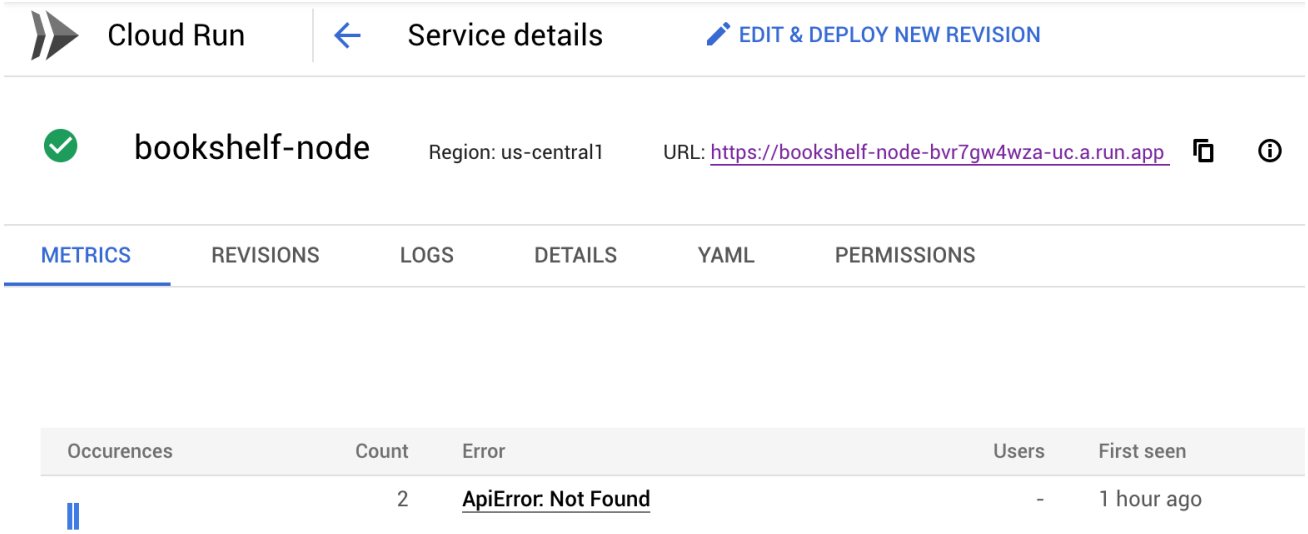

Figure 35. Cloud Run service page in Cloud Console.

2. Click on **Edit & Deploy New Revision**.
3. Click on **Variables & Secrets** tab.
4. Add the `GOOGLE_CLOUD_PROJECT` environment variable name and value.
5. Click **Deploy**.

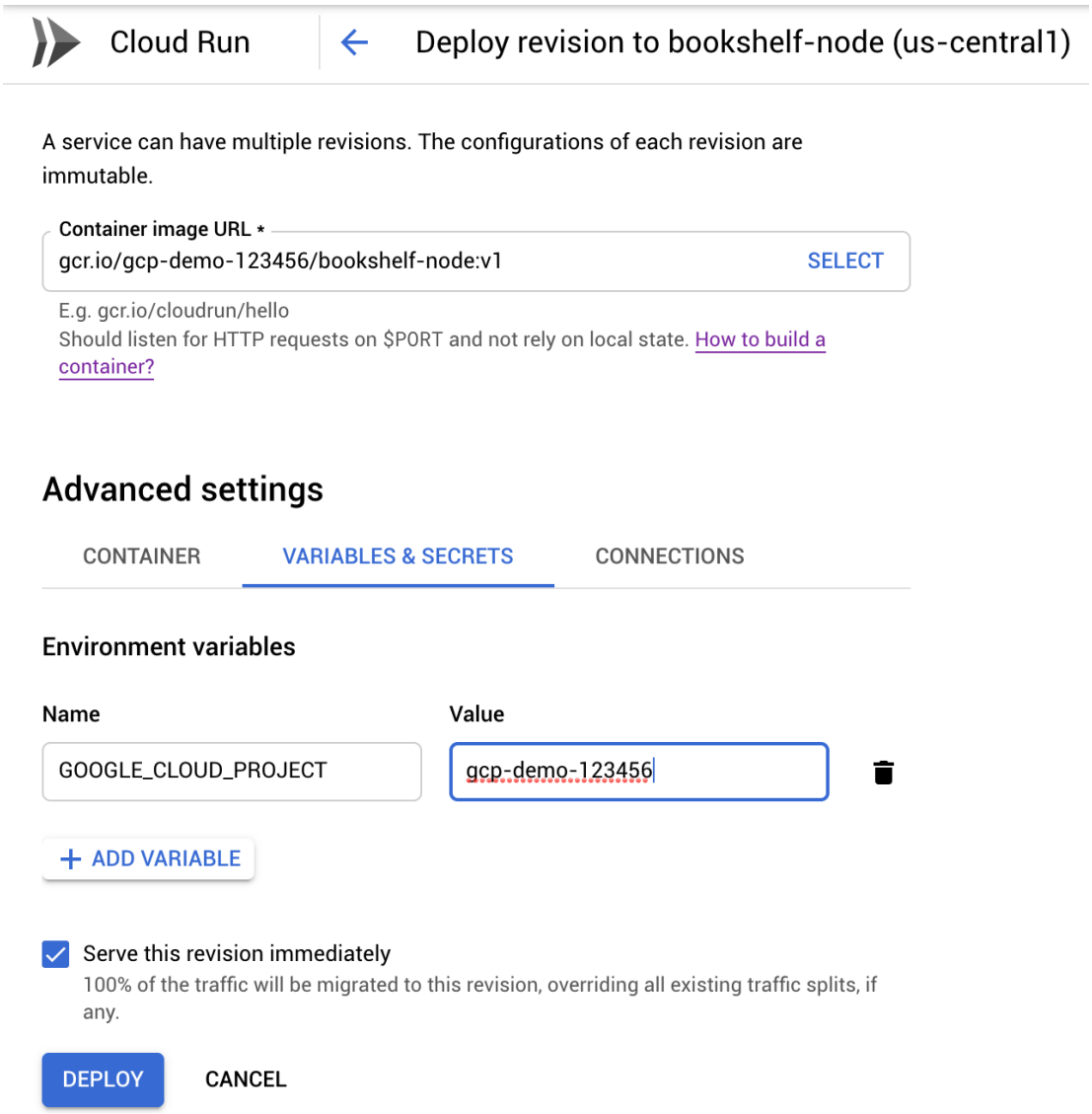

Figure 36. Setting the `GOOGLE_CLOUD_PROJECT` *environment variable in the Cloud Console.*

If you try uploading the image again, the image upload succeeds. You could also have redeployed using `gcloud` with the following command:

```
gcloud run deploy --image gcr.io/$PROJECT_ID/bookshelf-node:v1 \
  --set-env-vars=GOOGLE_CLOUD_PROJECT=gcp-demo-123456 \
  --platform managed \
  --region us-central1 \
  --project=$PROJECT_ID
```

Deploy with Cloud Build

Simply run the same deploy command in Cloud Build:

cloudbuild.yaml

```
steps:
- name: 'gcr.io/cloud-builders/gcloud'
  args:
    - 'run'
    - 'deploy'
    - '--image'
    - 'gcr.io/$PROJECT_ID/bookshelf-node:v1'
    - '--platform'
    - 'managed'
    - '--region'
    - 'us-central1'
  env:
    - 'GOOGLE_CLOUD_PROJECT=$PROJECT_ID'
    - 'ROOT_PATH=$_ROOT_PATH'
```

If this style of YAML configuration is frustrating, you can break out to `bash` like the following:

cloudbuild.yaml

```
steps:
- name: 'gcr.io/cloud-builders/gcloud'
  entrypoint: 'bash'
  args:
    - '-c'
    - |
      gcloud run deploy --image gcr.io/$PROJECT_ID/bookshelf-node:v1 \
        --platform managed --region us-central1
  env:
    - 'GOOGLE_CLOUD_PROJECT=$PROJECT_ID'
    - 'ROOT_PATH=$_ROOT_PATH'
```

Cloud Run as backend for Firebase Hosting

Cloud Run has custom domain with SSL support built-in, but if you use Firebase Hosting to host say a single-page application (SPA), and you want to use Cloud Run as an API in the `/api` path to avoid Cross-Origin Resource Sharing (CORS) overhead, well you can.

You will want to do something like the following:

```
"hosting": {
  // ...

  // Add the "rewrites" attribute within "hosting"
  "rewrites": [ {
    // Using "/books/**" only matches paths like "/books/xyz", but not "/books"
    "source": "/books{,/**}",
    "run": {
      "serviceId": "bookshelf-node",
      "region": "us-central1"
    }
  }]
}
```

After some trial and error I determined that using a `source` like `/api{,/**}` will route the request `/api/path` in Firebase Hosting domain to `/api/path` in your Cloud Run container. It does not actually rewrite the path from `/api/path` to `/path` in the Cloud Run container. So you will need to define an `/api` prefix for all your reachable endpoints in the Cloud Run container in order to achieve this, whether you like it or not.

Logging and Monitoring in Cloud Run

Viewing logs

Cloud Run has two types of logs, both of which are automatically sent to Cloud Logging:

- **Request logs**, i.e. logs of requests sent to Cloud Run services
- **Container logs**, i.e. logs emitted from container instances, usually from your own code

You can view logs from your Cloud Run service in the following ways:

- Cloud Run page in the Cloud Console. The **LOGS** tab in Cloud Run contains request and container logs for all revisions of this service. This is just a convenient wrapper view around Cloud Logging.
- Cloud Logging **Logs Viewer** page in the Cloud Console. Under the **Query builder** > **Resource dropdown**, look for Cloud Functions and filter by the name of your function.
- From the **command line**, run the following to read logs from your Cloud Run service:

```
gcloud logging read "resource.type=cloud_run_revision AND
resource.labels.service_name=SERVICE" --project PROJECT-ID --limit 10
```

Writing container logs

Logs written to any of the following locations will be picked up automatically by Cloud Run:

- Standard output (stdout) and standard error (stderr) streams
- Any files under the `/var/log` directory
- syslog (`/dev/log`)
- Logs written using Cloud Logging client libraries, such as `@google-cloud/logging` in npm

The documentation goes into much more detail about this topic.

Error Reporting

Error Reporting works out of the box in Cloud Run services.

All exceptions that include a stack trace in Error Reporting-supported languages, and that are sent to stdout, stderr, or to other logs, are automatically visible.

Handle errors and exceptions that occur during requests. Allowing errors to crash your application will result in a **cold start** (new container instance).

Optimizing Cloud Run

Memory limits

Cloud Run container instances that exceed their allowed memory limit are **terminated**. This is ruthless, so you need to ensure that you configure enough memory for your instance. The limit is 2GiB (gibibyte) per instance. Setting a higher memory limit will also use up more Gb-seconds in your free quota.

You can use the following formula to calculate the peak memory requirement for a service:

```
(Peak Memory) = (Standing Memory) + (Memory per Request) x (Service Concurrency)
```

Standing memory is memory usage when your application is idle. If your app's idle memory usage is 250MB, while each request potentially takes up to 1MB per request and your service concurrency is 80 (the default), then peak memory is about 330MB. Thus you should set the allocated memory to 512MB.

Concurrency

Each Cloud Run revision is automatically scaled to the number of container instances needed to handle all incoming requests. The number of container instance running is influenced by the **concurrency setting** (defaults to maximum of **80**).

This number is a maximum and Cloud Run might not send as many requests to a container

instance if the CPU of the instance is already highly utilized. In comparison, Functions-as-a-Service (FaaS) solutions like Cloud Functions have a fixed concurrency of 1.

You may want to limit concurrency to one request at a time if:

- Each request uses most of the available CPU or memory
- Your container image is not designed for handling multiple requests at the same time, e.g. container relies on global state that two requests cannot share

Note that limiting concurrency will affect scaling performance, because a spike in incoming requests will require many container instances to be started up cold.

You may also want to reduce concurrency of your container instance to avoid hitting the memory limit, instead of provisioning more memory to each instance.

Minimizing cold starts

Cloud Run may keep some instances idle even after they have handled all requests, in order to minimize cold starts. Note that despite this you are only billed when an instance is handling a request.

An idle container may persist resources such as open database connections. But CPU will not be available, at least not in Cloud Run (fully managed).

How long your instance will be kept idle is not disclosed. You should not count on it, and should optimize your app to start up fast.

Securing Cloud Run

Managing access with IAM

By default, Project Owners and Editors can create, update, delete, or invoke services. Only Project Owners and Cloud Run Admins can modify IAM policies in Cloud Run (e.g. make a service public).

Cloud Run IAM Roles:

Role	Description
Cloud Run Admin (`roles/run.admin`)	Full control over all Cloud Run resources.
Cloud Run Invoker (`roles/run.invoker`)	Can invoke a Cloud Run service.
Cloud Run Viewer (`roles/run.viewer`)	Can view the state of all Cloud Run resources, including IAM policies.

Service identity

When a Cloud Run service is invoked (either via HTTP or Pub/Sub triggers), the particular deployment of the service (called a **Cloud Run revision**) uses a service account as its identity. By default, this identity is the **Compute Engine default service account** (`PROJECT_NUMBER-compute@developer.gserviceaccount.com`), which by default has the Editor IAM role (you probably shouldn't change this).

This is very convenient to get started, and mostly doesn't get you into trouble (until it does), but the permissions are too wide and unacceptable from a security perspective. It is recommended for each service to use a dedicated service account with more restricted IAM roles instead. See this guide on how to get this going.

Under the hood, Cloud Run helps you set `GOOGLE_APPLICATION_CREDENTIALS` so that your client libraries can obtain and use this credential to authenticate with GCP.

Authentication

Cloud Run services are deployed privately by default, which means they cannot be accessed without providing authenticated credentials in the request. By default, services are only callable by Project Owners, Project Editors, Cloud Run Admins, and Cloud Run Invokers.

A few common use cases for authentication:

- Most of the time, you need to allow **public access** for your services. To do this, check the **Allow unauthenticated invocations** checkbox when creating a service. For an existing service, grant the Cloud Run Invoker role to the `allUsers` member type.
- If you need to give a **developer** access to a service (for development purposes), grant the developer the Cloud Run Invoker role (usually the G Suite or Gmail account email).
- If you need to give another **Cloud Run service** access to the service (service-to-service authentication), give the Cloud Run Invoker role to the service account representing the calling service. More details on setup here. The calling service will need to retrieve a Google-signed OAuth ID token and include it in the `Authorization` header when calling the receiving service.
- You can also grant access to specific end users that use **Google Sign-In**, or use either **Firebase Authentication** or **Identity Platform**, and manually validate user tokens using the provided SDK in your service code. This is a best practice to restrict access to only the allowed end users. See here for setup details.

Cloud Run pricing considerations

Cloud Run (fully managed) charges you only for the resources you use, rounded up to the nearest 100 milliseconds. Use the pricing calculator to estimate your costs.

A note on **vCPU-second** and **GB-second**. A vCPU-second means running a 1vCPU instance for 1 second. Similarly for GB-second.

Free quota (per billing account per month):

- 180,000 vCPU-seconds free
- 360,000 GB-seconds free
- 2 million requests free
- 1 GB free egress within North America

The *-seconds* part in the above units refer to the **billable time**. What's really interesting about this is how billable time is calculated. Instinctively I thought it would be the summation of all request times, e.g. request 1 takes 300ms, request 2 takes 400ms, total billable time would be 700ms. But it isn't like that.

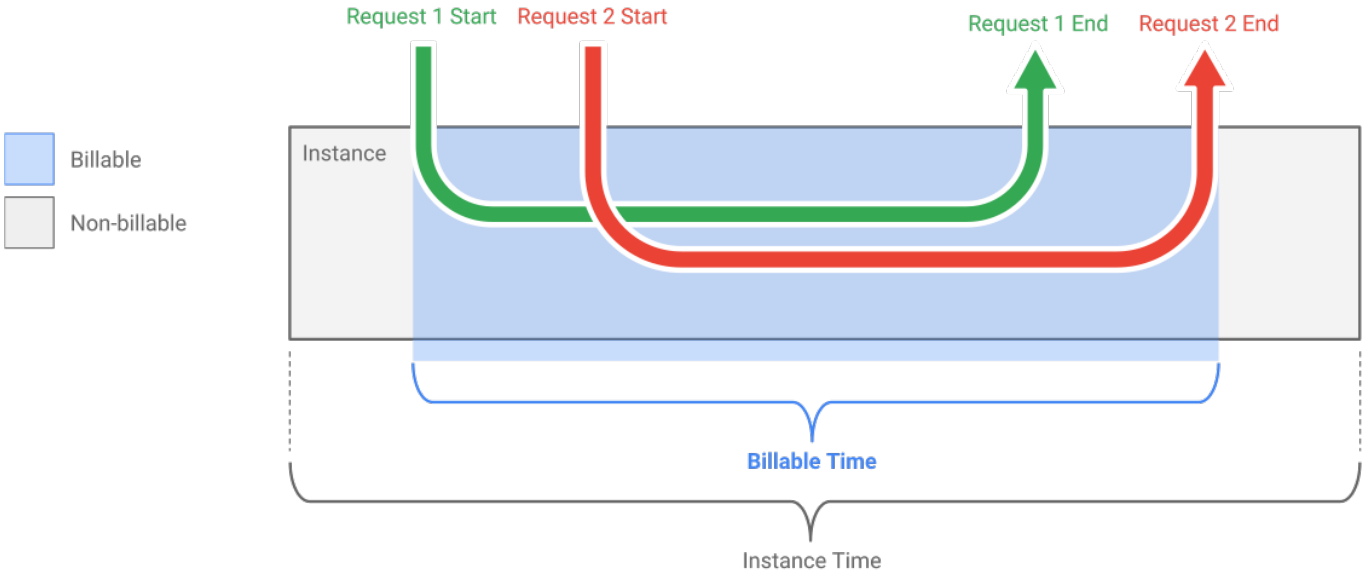

Figure 37. **Billable Time** *is the wallclock time that has elapsed between the first request and the final overlapping request. Source: Cloud Run Pricing*

As illustrated above, you are only billed for the CPU and memory allocated while a request is active on a container instance, rounded to the nearest 100 milliseconds. This makes sense, billable time is in the context of the container that uses the resources, not the request, because the same container handles multiple requests concurrency. This is in contrast to Cloud Functions, where each request is handled by a function instance that is the fully beneficiary of the allocated resources.

Hope that you start seeing why Cloud Run can be a very economical way to run your applications. Depending on your scenario, you can easily cut your cloud costs by a wide margin with Cloud Run compared to your previous deployment methods!

Cloud Run interoperability with other clouds

The container that you build for Cloud Run is portable to other clouds, but since your application may use Google client libraries, moving to another cloud will require you to generate a JSON private key associated with a service account, and pointing your application to it using `GOOGLE_APPLICATION_CREDENTIALS`.

Since Cloud Run is a managed version of Knative, if another cloud supports it, you're pretty much covered. Having said that, every cloud has its own nuances, but lock-in risk is very low since you are using a generic container, for the most part.

When to choose Cloud Run

1. You have a stateless container, and you want serverless.
2. You have code you can containerize, and you don't want it to know about how it is being deployed.
3. Your container doesn't really need to co-operate with other containers closely, like a single app that is run on its own.
4. You have an app that doesn't run continuously, it only gets requests from time to time.
5. You have code in a container that you want to trigger from a Pub/Sub topic.
6. You want to easily provide built-in authentication to developers, other services, specific end users, or to any public user.
7. Your website gets sporadic to moderate traffic, whereby there are lots of idle time in between.
8. You have a monolith app that you just want to run for cheap in GCP. You can even break up this monolith app into Cloud Run services, e.g. worker and frontend services, while maintaining a single codebase. Just load the part of the monolith that is relevant to the particular service.
9. You are willing to do some code rewrite to use Pub/Sub to handle app events (whereby previously you had code that was running in the background all the time).

When to not choose Cloud Run

1. Your code runs background jobs and runs outside the context of a HTTP request or a Pub/Sub event (and you don't want to rewrite it).
2. Your container or code is not stateless (but see above section on concurrency about this).
3. You are concerned about cold starts, and your application starts up too slowly.
4. You have a high traffic website that might as well just run continuously on App Engine or Compute Engine (what a nice problem to have!). Use the pricing calculator to verify.

5. You need to run WebSockets. Cloud Run does not support WebSockets.

Summary

Cloud Run is an underrated way to deploy your containers in serverless fashion. If you are on a heavier runtime unsupported by Cloud Functions (e.g. Ruby), Cloud Run is the ideal candidate to deploy such applications serverlessly, if some cold start latency is acceptable (many seem to like Heroku, so this isn't really an issue?). For "lighter" runtimes such as Node.js and Go, Cloud Run performs magically as if cold start isn't really there.

I see virtually little downside to using Cloud Run for both hobby and production projects. You pretty much get a "load balancer" for free, with TLS and custom domain to boot. Unless the lack of VPC connection (at time of writing) is a deal-breaker for you, Cloud Run is a strong serverless contender, especially when Cloud Functions is out of the question.

You can run literally anything in Cloud Run, because code in any runtime or dependencies in any form can be wrapped up and deployed in a container. That's what makes it so powerful.