Google Cloud

Serverless at scale: From design to production

Table of Contents

Forward
Serverless gets real

For as long as there have been developers, they've been saying "I just want to write code to solve my business problems, and let someone else take care of all the provisioning, deployment, scaling, and updates." And for the past decade or so, that desire, which we refer to as "serverless," has been a reality—at least for developers writing applications in programming languages like Go, Java and Node.js.

Recently, though, there's been a big bump in the capabilities of serverless systems. What's changed? For one, many serverless systems now support event-driven application architectures, where you write code that executes in response to an external trigger, like a file upload, or an authentication request. Among Google Cloud's serverless compute offerings, both Cloud Run and Cloud Functions support event triggers, for instance.

Speaking of Cloud Functions, the rise of functions as an abstraction layer represents another big change in serverless systems, where instead of writing big monolithic apps you write small code snippets that perform a single, granular task.

Third, serverless systems have embraced containers. In addition to being able to leverage application- or function-level abstraction, platforms like Cloud Run now serverlessly manage arbitrary code running in a Docker container, as long as it communicates on an HTTP or HTTPS port.

What kinds of applications are developers building on these new serverless systems? Increasingly sophisticated and complex apps, that's what. Google Cloud customers use serverless for everything from running ecommerce sites to processing satellite images to extending legacy ERP systems. Needless to say, serverless has moved beyond the early adopter phase and is rapidly becoming the preferred way to build the next-generation of business apps.

But together, the way modern serverless combines events, functions and containers represents a novel way of looking at system design. To help, Google Cloud Developer Advocates and Solutions Architects have sample code and best practices to help you jumpstart your serverless development practice. In the pages that follow, you'll find blog posts from serverless subject matter experts to help you decide between different serverless offerings, how serverless fits into modern application architecture, and how to build a serverless system that can survive viral success! We hope these articles whet your appetite for serverless, so you can focus on writing code to solve your business problem. Don't worry, we'll take care of the rest.

- ***Oren Teich***, *Product Management Director*

Chapter 1
GKE or Cloud Run: Which should you use?

By Christoph Bussler and Amina Mansour, Solutions Architects, Google Cloud

When it comes to managed Kubernetes services, Google Kubernetes Engine (GKE) is a great choice if you are looking for a container orchestration platform that offers advanced scalability and configuration flexibility. GKE gives you complete control over every aspect of container orchestration, from networking, to storage, to how you set up observability—in addition to supporting stateful application use cases. However, if your application does not need that level of cluster configuration and monitoring, then fully managed Cloud Run might be the right solution for you.

Fully managed Cloud Run is an ideal serverless platform for stateless containerized microservices that don't require Kubernetes features like namespaces, co-location of containers in pods (sidecars) or node allocation and management.

Why Cloud Run?

The managed serverless compute platform Cloud Run provides a number of features and benefits:

- **Easy deployment of microservices.** A containerized microservice can be deployed with a single command without requiring any additional service-specific configuration.
- **Simple and unified developer experience.** Each microservice is implemented as a Docker image, Cloud Run's unit of deployment.
- **Scalable serverless execution.** A microservice deployed into managed Cloud Run scales automatically based on the number of incoming requests, without having to configure or manage a full-fledged Kubernetes cluster. Managed Cloud Run scales to zero if there are no requests, i.e., uses no resources.
- **Support for code written in any language.** Cloud Run is based on containers, so you can write code in any language, using any binary and framework.

Cloud Run is available in two configurations: as a fully managed Google Cloud service, and as Cloud Run for Anthos (this option deploys Cloud Run into an Anthos GKE cluster). If you're already using Anthos, Cloud Run for Anthos can deploy containers into your cluster, allowing access to custom machine types, additional networking support, and GPUs to enhance your Cloud Run services. Both managed Cloud Run services and GKE clusters can be created and managed completely from the console as well as from the command line.

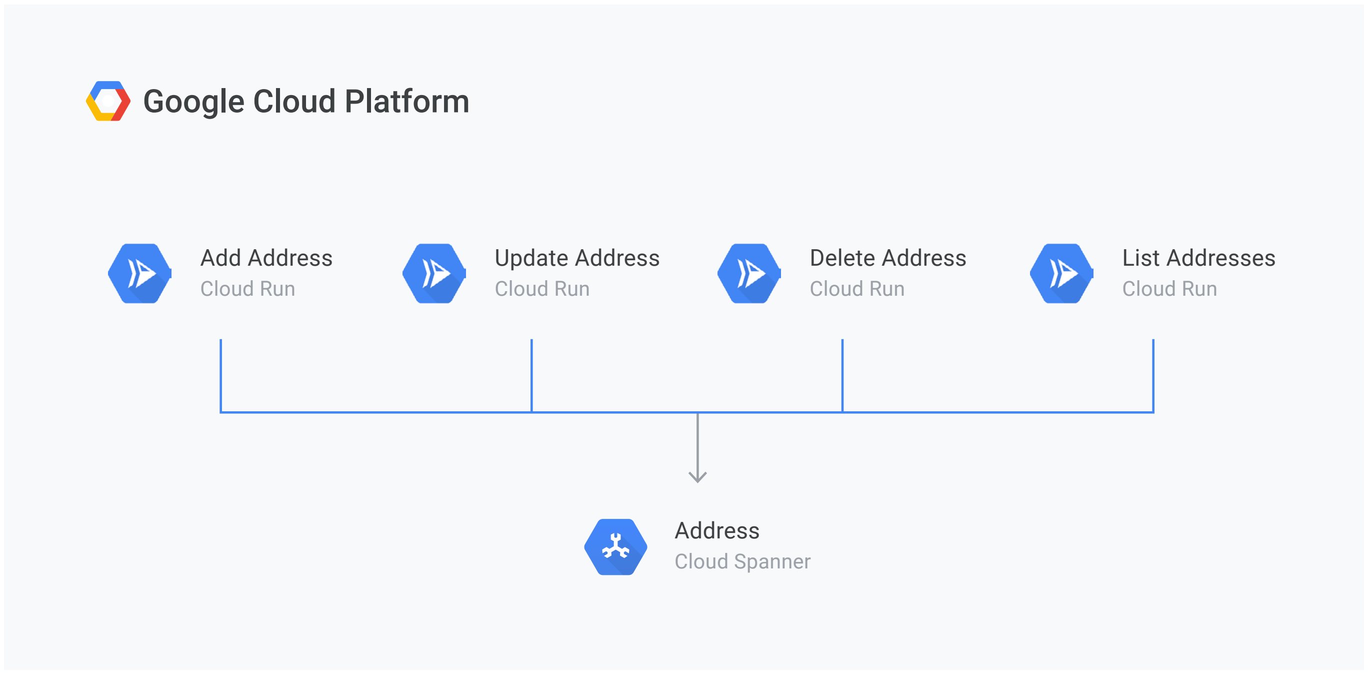
The best part is you can easily change your mind later, switching from managed Cloud Run to Cloud Run for Anthos or vice versa without having to reimplement your service.

A Cloud Run use case

To illustrate these points, let's take a look at an example use case, a service that adds, updates, deletes and lists addresses.

You can implement this address management service by creating one containerized microservice for each operation. Then, once the images have been created and registered in a container registry, you can deploy them to managed Cloud Run with a single command. After executing four commands (one deployment for each microservice), the service is up and running on a completely serverless platform. The following figure shows the deployment using Cloud Spanner as the underlying database.

Serverless scaling patterns



For use cases such as this one, managed Cloud Run is a great choice as the address management service does not require complex configurations as supported by Kubernetes. Nor does this address management service need 24/7 cluster management and operational supervision. Running this address management service as containers in managed Cloud Run is the better production workload strategy.

As a managed compute platform, managed Cloud Run supports essential configuration settings: the maximum concurrent requests a single container receives, the memory size to be allocated to the container as well as request timeout can be configured. No additional configurations or management operations are required.

The right tool for the job

Both managed Cloud Run and GKE are powerful offerings for different use cases. Make sure to understand your functional and non-functional service requirements like ability to scale to zero or ability to control detailed configuration before choosing one over the other.

In fact, you might want to use both at the same time. An enterprise might have complex microservice-based applications that require advanced configuration features of GKE, and some that do not, but that still want to take advantage of Cloud Run's ease of use and scalability.

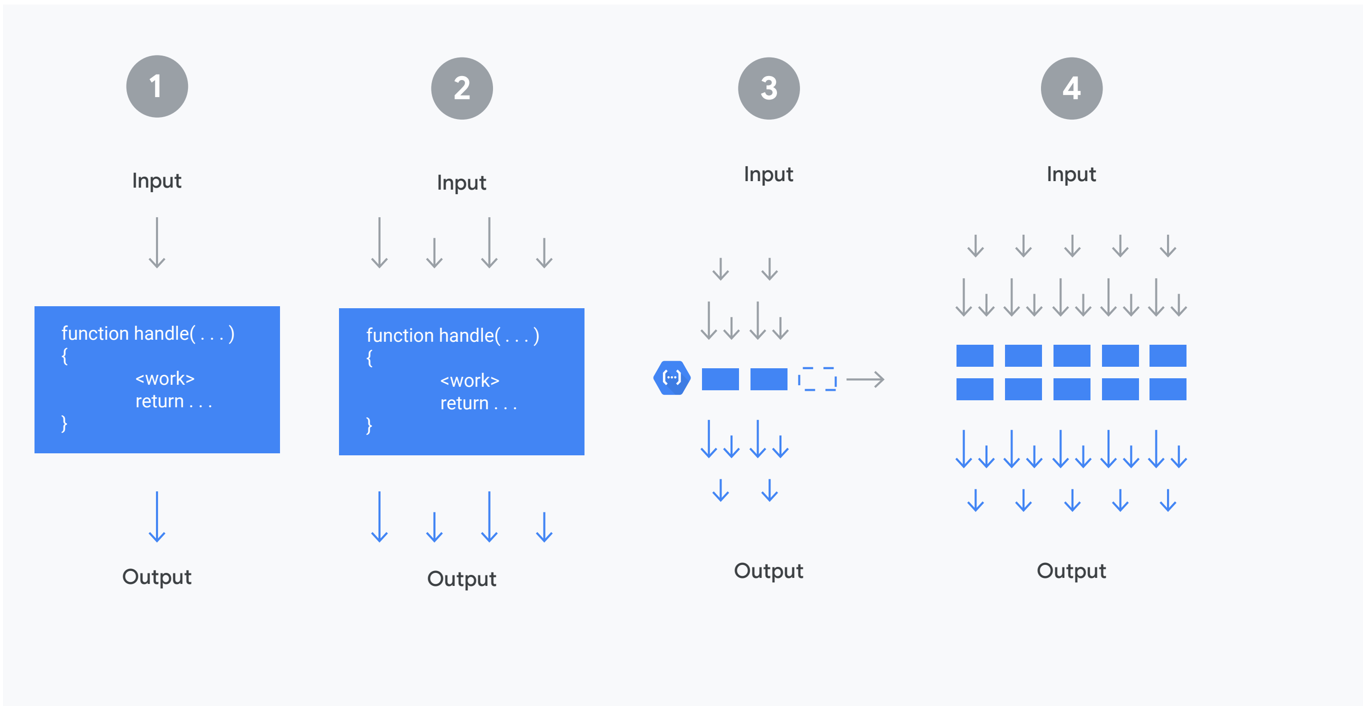
Chapter 2
Scaling your serverless applications

By Preston Holmes, Cloud Solution Architect

A core promise of a serverless compute platform like Cloud Functions is that you don't need to worry about infrastructure: write your code, deploy it and watch your service scale automatically. It’s a beautiful thing.

That works great when your whole stack auto-scales. But what if your service depends on an APIs or databases with rate or connection limits? A spike of traffic might cause your service to scale (yay!) and quickly overrun those limits (ouch!). In this post, we'll show you the features of Cloud Functions, Google Cloud’s event-driven serverless compute service, and products like Cloud Tasks that can help serverless services play nice with the rest of your stack.

Serverless scaling patterns



Lets review the basic way in which serverless functions scale as you take a function from your laptop to the cloud.

1. At a basic level, a function takes input, and provides an output response.

2. That function can be repeated with many inputs, providing many outputs.

3. A serverless platform like Cloud Functions manages elastic, horizontal scaling of function instances.

4. Because Google Cloud can provide near-infinite scale, that can have consequences for other systems with which your serverless function interacts.

Most scale-related problems are the result of limits on infrastructure resources and time. Not all things scale the same way, and not all serverless workloads have the same expected behaviors in terms of how they get work done. For example, whether or not the result of a function is returned to the caller or is only directed elsewhere, can change how you handle increasing scale in your function. Different situations may call for one or more different strategies to manage challenges scale can introduce.

Luckily, you have lots of different tools and techniques at your disposal to help ensure that your serverless applications scale effectively. Let's take a look.

1. Use Max Instances to manage connection limits

Because serverless compute products like Cloud Functions and Cloud Run are stateless, many functions use a database like Cloud SQL for stateful data. But this database might only be able to handle 100 concurrent connections. Under modest load (e.g., fewer than 100 queries per second), this works fine. But a sudden spike can result in hundreds of concurrent connections from your functions, leading to degraded performance or outages.

One way to mitigate this is to configure instance scaling limits on your functions. Cloud Functions offers the max instances setting. This feature limits how many concurrent instances of your function are running and attempting to establish database connections. So if your database can only handle 100 concurrent connections, you might set max instances to a lower value, say 75. Since each instance of a function can only handle a single request at a time, this effectively means that you can only handle 75 concurrent requests at any given time.

2. Use Cloud Tasks to limit the rate of work done

Sometimes the limit you are worried about isn't the number of concurrent connections, but the rate at which work is performed. For example, imagine you need to call an external API for which you have a limited per-minute quota. Cloud Tasks gives you options in managing the way in which work gets done. It allows you to perform the work outside of the serverless handler in one or more work queues. Cloud Tasks supports rate and concurrency limits, making sure that regardless of the rate work arrives, it is performed with rates applied.

3. Use stateful storage to defer results from long-running operations

Sometimes you want your function to be capable of deferring the requested work until after you provide an initial response. But you still want to make the result of the work available to the caller eventually. For example, it may not make sense to try to encode a large video file inside a serverless instance. You could use Cloud Tasks if the caller of your workload only needs to know that the request was submitted. But if you want the caller to be able to retrieve some status or eventual result, you need an additional stateful system to track the job. In Google APIs this pattern is referred to as a long-running operation. There are several ways you can achieve this with serverless infrastructure on Google Cloud, such as using a combination of Cloud Functions, Cloud Pub/Sub, and Firestore.

4. Use Redis to rate limit usage

Sometimes you need to perform rate-limiting in the context of the HTTP request. This may be because you are performing per-user rate limits, or need to provide a back-pressure signal to the caller of your serverless workload. Because each serverless instance is stateless and has no knowledge of how many other instances may also be serving requests, you need a high-performance shared counter mechanism. Redis is a common choice for rate-limiting implementations. Read more about rate limiting and GCP, and see this tutorial for how to use serverless VPC access to reach a private Redis instance and perform rate limiting for serverless instances.

5. Use Cloud Pub/Sub to process work in batches

When dealing with a large number of messages, you may not want to process every message individually. A common pattern is to wait until a sufficient number of messages have accumulated before handling all of them in one batch. Cloud Functions integrates seamlessly with Cloud Pub/Sub as a trigger source, but serverless workloads can also use Cloud Pub/Sub as a place to accumulate batches of work, as the service will store messages for up to seven days.

Then, you can use Cloud Scheduler to handle these accumulated items on a regular schedule, triggering a function that processes all the accumulated messages in one batch run.

You can also trigger the batch process more dynamically based on the number and age of accumulated messages. Check out this tutorial, which uses Cloud Pub/Sub, Stackdriver Alerting and Cloud Functions to process a batch of messages.

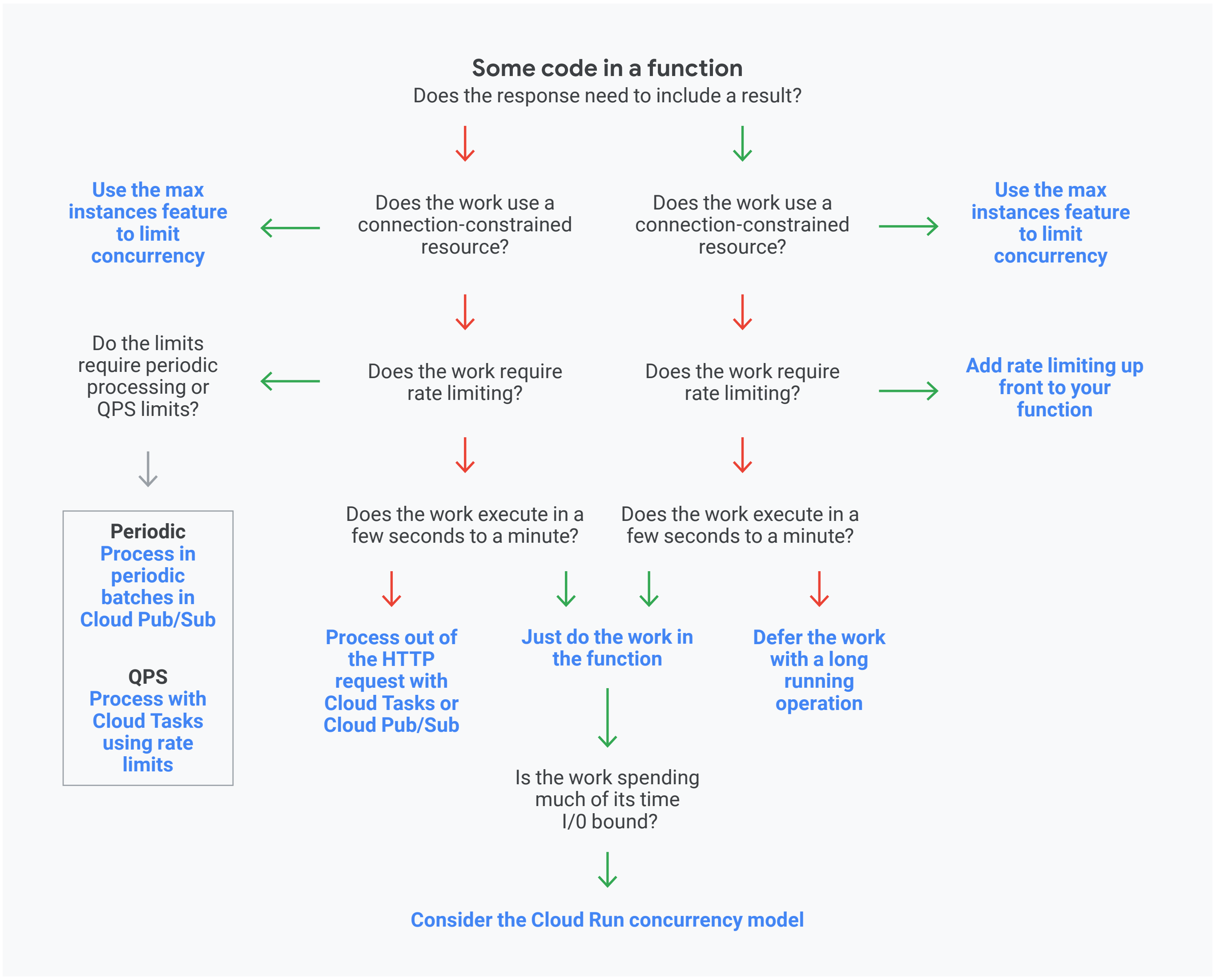
6. Use Cloud Run for heavily I/O-bound work

One of the more "expensive" components of many infrastructure products is compute cycles. This is reflected in the pricing of many managed services which include how many time-units of CPU you use. When your serverless workload is just waiting around for a remote API call it may make to return, or waiting for a file to read, these are moments where you are not using the CPU, but are still "occupying it" so will be billed. Cloud Run, which lets your run fully managed serverless containers, allows your workload to specify how many concurrent requests it can handle. This can lead to significant increases in efficiency for I/O bound workloads.

For example, if the work being done spends most of its time waiting for replies from slow remote API calls, Cloud Run supports up to 80 requests concurrently on the same serverless instance which shares the use of the same CPU allocation. Learn more about tuning this capability for your service.

When to use which strategy

After reading the above, it may be clear which strategy might help your current project. But if you are looking a little more guidance, here’s a handy flow-chart (next page).


Some code in a function
Does the response need to include a result?
Does the work use a connection-constrained resource?
Does the work use a connection-constrained resource?
Use the max instances feature to limit concurrency
Use the max instances feature to limit concurrency
Do the limits require periodic processing or QPS limits?
Does the work require rate limiting?
Does the work require rate limiting?
Add rate limiting up front to your function
Periodic
Process in periodic batches in Cloud Pub/Sub
QPS
Process with Cloud Tasks using rate limits
Does the work execute in a few seconds to a minute?
Does the work execute in a few seconds to a minute?
Process out of the HTTP request with Cloud Tasks or Cloud Pub/Sub
Just do the work in the function
Defer the work with a long running operation
Is the work spending much of its time I/0 bound?
Consider the Cloud Run concurrency model

Of course you might choose to use more than one strategy together if you are facing multiple challenges.

Just let it scale

Even if you don't have any scaling problems with your serverless workload, you may still be uneasy, especially if this is your first time building software in a serverless environment—what if you're about to hit some limit, for example? Rest easy, the default limits for Google Cloud serverless infrastructure are high enough to accomodate most workloads without having to do anything. And if you do find yourself approaching those limits, we are happy to work with you to keep things running at any scale. When your serverless workload is doing something useful, more instances is a good thing!

Chapter 3
Cloud Run and the Twelve-Factor App methodology

By Katie McLaughlin, Developer Advocate, Serverless

Cloud Run makes it easier than ever to deploy serverless applications on Google Cloud Platform (GCP) that are automatically provisioned, scaled up, and scaled down. But in a serverless world, being able to ensure your service meets the twelve factors is paramount.

The Twelve-Factor App denotes a paradigm that, when followed, should make it frictionless for you to scale, port, and maintain web-based software as a service. The more factors your environment has, the better.

So, on a scale of 1 to 12, just how twelve-factor compatible is Cloud Run? Let's take the factors, one by one.

The Twelve Factors

1. Codebase

One codebase tracked in revision control, many deploys

Each service you intend to deploy on Cloud Run should live in its own repository, whatever your choice of source control software. When you want to deploy your service, you need to build the container image, then deploy it. For building your container image, you can use Docker, or Cloud Build, GCP's own build system.

You can even supercharge your deployment story by integrating Build Triggers, so any time you, say, merge to master, your service builds, pushes, and deploys to production.

You can also deploy an existing container image as long as it listens on a PORT, or use one of the many images that sport a shiny Deploy on Cloud Run button.

2. Dependencies

Explicitly declare and isolate dependencies

Since Cloud Run is a bring-your-own container environment, you can declare whatever you want in this container, and the container encapsulates the entire environment. Nothing escapes, so two containers won't conflict with each other.

When you need to declare dependencies, these can be captured using environment variables, keeping your service stateless.

It is important to note that there are some limitations to what you can put into a Cloud Run container due to the environment sandboxing, and what ports can be used (which we'll cover later in Section VII).

3. Config

Store config in the environment

Yes, Cloud Run stores configuration in the environment by default.

Your code goes in your container, and configuration is captured in the specification of your service. Configuration includes for example the amount of memory, CPU or environment variables. These can be declared when you create the service, in the Optional Settings.

Don't worry if you miss setting environment variables when you create your service. You can always edit them again by clicking "+ Deploy New Revision" when viewing your service, or by using the `--update-env-vars flag in gcloud beta run deploy`.

Each revision you deploy is not editable, which means revisions are reproducible, as the configuration is frozen. To make changes you must deploy a new revision.

For bonus points, consider using berglas, which leverages Cloud KMS and Cloud Storage to secure your environment variables. It works out of the box with Cloud Run (and the repo even comes with multiple language examples).

4. Backing services

Treat backing services as attached resources

Much like you would connect to any external database in a containerized environment, you can connect to a plethora of different hosts in the GCP universe.

And since your service cannot have any internal state, to have *any* state you must use a backing service.

5. Build, release, run

Strictly separate build and run stages

Having separate build and run stages is *how* you deploy in Cloud Run land! If you set up your continuous deployment back in Section I, then you've already automated that step.

If you haven't, building a new version of your Cloud Run service is as easy as building your container image:

```
gcloud builds submit --tag gcr.io/YOUR_PROJECT/YOUR_IMAGE
```

to take advantage of Cloud Build, and deploying the built container image:

```
gcloud beta run deploy --image gcr.io/YOUR_PROJECT/YOUR_IMAGE YOUR SERVICE
```

Cloud Run creates a new revision of the service, ensures the container starts, and then re-routes traffic to this new revision for you. If for any reason your container image encounters an error, the service is still active with the old version, and no downtime occurs.

You can also create continuous deployment by configuring Cloud Run automations using Cloud Build triggers, further streamlining your build, release, and run process.

6. Processes

Execute the app as one or more stateless processes

Each Cloud Run service runs its own container, and each container should have one process. If you need multiple concurrent processes, separate those out into different services, and use a stateful backing service (Section 4) to communicate between them.

7. Port binding

Export services via port binding

Cloud Run follows modern architecture best practices and each service must expose itself on a port number specified by the PORT environment variable.

This is the fundamental design of Cloud Run: *any container you want, as long as it listens on port 8080.*

Cloud Run does support outbound gRPC and WebSockets, but does not currently work with these protocols inbound.

8. Concurrency

Scale out via the process model

Concurrency is a first-class factor in Cloud Run. You declare the maximum number of concurrent requests that your container can receive. Using this maximum and other factors, Cloud Run will automatically scale by adding more container instances to handle all incoming requests.

9. Disposability

Maximize robustness with fast startup and graceful shutdown

Since Cloud Run handles scaling for you, it's in your best interest to ensure your services are as efficient as they can be. The faster they are to start up, the more seamless scaling can be.

There are a number of tips around how to write effective services, so be sure to consider the size of your containers, the time they take to start up, and how gracefully they handle errors without terminating.

10. Dev/prod parity

Keep development, staging, and production as similar as possible

A container-based development workflow means that your local machine can be the development environment, and Cloud Run can be your production environment! Even if you're running on a non-Linux environment, a local Docker container should behave in the same way as the same container running elsewhere. It's always a good idea to test your container locally when developing. Testing locally helps you achieve a more efficient iterative development strategy, allowing you to work more effectively. To ensure that you get the same port-binding behaviour as Cloud Run in production, make sure you run with a port flag:

```
PORT=8080 && docker run -p 8080:${PORT} -e PORT=${PORT} gcr.io/[PROJECT_ID]/[IMAGE]
```

When testing locally, consider if you're using any GCP external services, and ensure you point Docker to the authentication credentials.

Once you've confirmed your service is sound, you can deploy the same container to a staging environment, and after confirming it's working as intended there, to a production environment.

A GCP Project can host many services, so it's recommended that your staging and production (or green and blue, or however you wish to call your isolated environments) are separate projects. This also ensures isolation between databases across environments.

11. Logs

Treat logs as event streams

Cloud Run uses Stackdriver Logging out of the box. The "Logs" tab on your Cloud Run service view will show you what's going on under the covers, including log aggregation across all dynamically created instances. Stackdriver Logging automatically captures stdout and stderr, and there may also be a native client for Logging in your preferred programming language.

In addition, since logs are captured in Stackdriver Logging, you can then use the tools available for Stackdriver Logging to further work with your logs; for example, exporting to BigQuery.

12. Admin processes

Run admin/management tasks as one-off processes

Administration tasks are outside the scope of Cloud Run. If you need to do any project configuration, database administration, or other management changes, you can perform these tasks using the GCP Console, gcloud CLI, or Cloud Shell.

A near-perfect score, as a matter of fact(or)

With the exception of one factor being outside of scope, Cloud Run maps near perfectly with Twelve-Factor, which means it will map well to scalable, manageable infrastructure for your next serverless deployment.

Resources

Google Cloud Serverless

Webpage

Cloud Run

Webpage

Quickstart

Cloud Functions

Webpage

Quickstarts