

Be Wary of the Economics of "Serverless" Cloud Computing

Adam Eivy, Solutions Engineering Architect,
The Walt Disney Company

ONE OF THE LATEST DEVELOPMENTS IN CLOUD COMPUTING IS USUALLY CALLED "SERVERLESS" COMPUTING, EVEN THOUGH, OF COURSE, SERVERS ARE STILL WHERE PROCESSING TAKES PLACE. For example, Amazon Web Services (AWS) offers "Lambda Functions,"¹

Google has "Cloud Functions,"² and Microsoft has "Azure Functions".³ They differ in various ways; for example, Azure and IBM Bluemix "OpenWhisk"⁴ functions can run in a private or public cloud.

In standard cloud computing, dedicated hardware is replaced by dynamically allocated, pay-per-use resources, such as virtual servers. Although called "pay-per-use," these resources are typically billed based on allocation, not on actual use, potentially leading to a customer paying more than necessary. In "serverless," no resources are typically allocated or chargeable until a function is called. It's like the difference between a rental car and a taxi: you will be charged for the rental car even if you park it for a week, unlike a taxi. Moreover, some cloud providers are offering seemingly massive amounts of serverless computing at no charge. This holds the promise of the most efficient processing possible—for free or at least what seem to be attractive prices. Moreover, serverless fits with the modern approach to application construction—composing microservices rather than building hard-to-manage and scale monolithic applications.⁵

However, as with many things, the devil is in the details and the economic benefits of serverless computing heavily depend on the execution behavior and volumes of the application workloads. In the same way that pennies per day can add up to thousands of dollars eventually, low "per hit" prices can not only add up as transaction volumes increase, but can make serverless economics unattractive relative to what have now become more traditional approaches, such as virtual machines or even dedicated hardware.

Defining Serverless

To understand what serverless is, it may help to first understand what it isn't.

Serverless Isn't #NoOps


Serverless computing is sometimes conflated with "NoOps," the extreme evolution of the DevOps movement. However, even though serverless options potentially abstract away load balancing, autoscaling, high availability, and the security maintenance of compute infrastructure, it's worth pointing out that all of these underlying operational concerns are simply distractions from operational responsibility.

EDITOR:

JOE WEINMAN

joeweinman@gmail.com





ity. Sure, you don't need to manually spin up and manage compute, but you do need to ensure your services are tunable, testable, secure, performant, resilient, monitored, and KPI-instrumented. You also will need a deployment and versioning strategy that allows serverless tasks to play nicely with other services. The true operational concerns of your application are not generic enough to be a commodity—if they were, your service wouldn't be unique and probably wouldn't exist.

“Serverless” Isn't Really Serverless

This may seem obvious but this is where we get into what serverless really is at its core. In any supposed “serverless” compute offering, someone (or something) is managing compute servers. But what's on these servers? Serverless isn't specific about how it's implemented so different cloud providers have different components making up the stack—but there are common crucial elements shared between them. You'll need a way to run code in isolation (e.g. Docker), a way to autoscale underlying compute, security monitoring, metrics, some kind of input/output layer for logging and passing data to and from your serverless functions.

So What Is a Serverless Function?

Google has a great definition for what Cloud Functions offer that helps us build a model for serverless infrastructure: “Google Cloud Functions is a lightweight, event-based, asynchronous compute solution that allows you to create small, single-purpose functions that respond to cloud events without the need to manage a server or a runtime environment.”

“Lightweight, event-based, asynchronous” sounds a lot like Node.js. “Small, single-purpose functions... without the need to manage a server” sounds a lot like Docker containers.

Though commercial providers may not be using Docker under the hood, it's a simple way to visualize how serverless functions can be built as isolated code and executed ad hoc—and you could certainly build a serverless offering using Docker. For those who have missed the buzz, Docker is a tool for wrapping code in a containerized image. The code could be a robust web app or it could simply be a “Hello World” app “printing” out on standard output. The image can be run as a “container” that cleans itself

up after execution finishes. In this way, a serverless platform can be built to encapsulate code in a Docker image, then run it based on some input to the system, i.e., event. Combine that with something that monitors whatever triggers you want to advertise (HTTP, Email, S3 writes, etc.). Add to that an autoscaler to spin up and down more compute as needed (unless you are building this on Joyent's Triton data center and you've simply provisioned the whole data center as a big Docker host). And finally, add an automated service to manage security patching or changing the base image that the autoscaler uses. Now you've got a basic serverless compute infrastructure.

So to state this more simply, a serverless function is just a block of code that can be executed on demand by an orchestration layer or application programming interface (API) / function call, but doesn't use any run-time resources such as CPU or memory until then.

Using Serverless

AWS Lambda comes with ImageMagick 6, an image processing tool, ready to use. Doing image conversions on upload might be a great use of serverless, provided the images are not too large and your usage isn't massive. Keep in mind that Lambda has a 5-minute maximum execution time (and a good thing that is, lest you face unexpected compute cost for a rogue hit). Or what about workflow triggers? AWS Simple Workflow (SWF) is great for some things but at other times, deploying a function is much simpler—or running a serverless function is a step in a larger workflow managed by an orchestrator like AWS SWF. Serverless isn't just for HTTP API endpoints. Much of the power in serverless functions comes from the ability to trigger based on other types of events, such as files being uploaded to S3 (e.g. targets for image conversion, JSON files needing to be mapped/reduced, etc) or data going into DynamoDB (more candidates for extract, transform, load). Google Cloud Functions even has triggers for Gmail. Many of the potential uses for serverless echo offerings available in If This Then That (ifttt.com). These are great candidates for serverless functions (when something happens over there, do a tiny action over here).

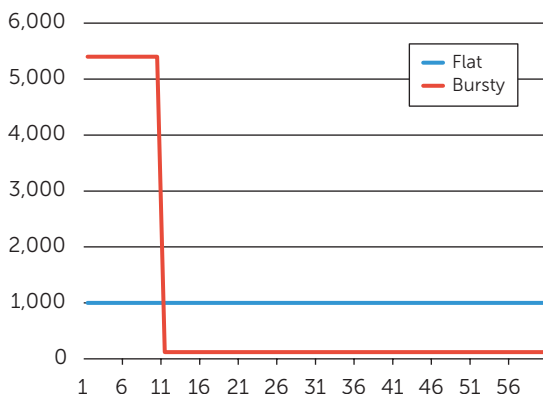


FIGURE 1. Two workloads, each with 60K transactions per minute

The Surprisingly Complicated Economics of Serverless

Amazon Web Services was the first cloud provider to introduce serverless with “Lambda functions” in 2014, but similar offerings from Google, Microsoft, IBM, and others have been introduced since then. Many smaller players are also planning offerings, eager to take part in the trend. There are many similarities and some differences between pricing models and actual prices, and cloud pricing is notoriously subject to frequent revision, but let’s take a look at one that is very representative—Amazon Web Services Lambda functions (at the time of this writing).

Peak vs. Average Transaction Rates, or, Scalability vs. Economics

The first thing you might notice about the AWS Lambda pricing documentation⁶ is that they offer the first 1 million hits and a 400,000 GB-seconds per month free! This might feel like a lot of free use, and for some services (a build trigger, an internal API that you call infrequently), it might be all you need. However, when we talk about public web services, thinking in terms of consumption per month is not the lingua franca for estimating service usage. When designing the scalability of an API, there is one key unit of measurement: Transactions Per Second (also referred as hits or requests; TPS, HPS, RPS, are just different acronyms for the same idea). Measuring by any other method puts your API at risk of unknown standard deviations

within the per-second timeframe that could render your service useless.

For example, if we measure an API by transactions per minute and expect an average of 60K hits per minute, we might be lulled into thinking that this averages to a maximum concurrent throughput of 1000 TPS.

However, what if sometimes 90% (54K) of those per-minute transactions come in bursts within a 10 second period? Rather than 1000 TPS, this API would need to scale to 5,400 TPS. As shown in Figure 1, this is a big difference that software architects and operational designers need to consider in order to properly design the scaling/capacity needs of an API.

But if we measure by hours, days, weeks, things get even fuzzier. What does 1 million hits per month really mean? It’s only about .38 hits per second, or 1 hit every 3 seconds ($1\text{M hits / month} = 12\text{M / year} = 12\text{M} / (365 \times 24 \times 60 \times 60)$ hits per second). If you are building an API with any reasonable amount of traffic (say 500 hits per second—the current average for one of my team’s smaller APIs in production today), the freebie of a million hits per month is only a 0.00076 discount on the per-hit charge. The meaningfulness of this discount gets much worse when we look at the projected scale of the API in the coming months and see that we are expecting thousands of requests per second. To put it another way, the first million hits per month works out to be the first .38 out of 500 hits per second free, not an especially large discount in environments of thousands of hits per second.

Amazon provides sample pricing calculations, but as your workload varies, so will the billing. The largest example is for a function that runs 30 million times per month. Well, that’s a huge, right? Not really. In more understandable API terms, this is about 11 hits per second. Cloud billing is confusing enough without converting the billable units—so customers need to be wary of how pricing is posted. One of the websites within my company gets 150,000 page loads per second at peak time, each load triggering dozens of APIs. Now, not everyone is supporting a Fortune 500 company or hyperscale ebusiness, so run your own numbers—but 79 hits per second is not a high-traffic API. If your API is only being hit 79 times per second, you probably don’t even need to autoscale!

Plan for Reality

Additionally, note that the demo apps are great learning examples, but their economics are unlikely to match yours. Amazon Web Services has a tested project that builds a chat app. This is a great example for getting your feet wet with Lambda, but imagine running this on the public internet! How many users could join this system and chat before your micro-billable hits per second makes this an untenable financial burden? Do you really want to measure the cost of each chat message? This could lead to strange behavioral policies on your team/users. “Hey everyone, we need to cut costs, so talk to each other less frequently and in larger bodies of text!”

The GB-second charge is much trickier to unravel and requires understanding what a GB-second actually is. Sadly, this is a case where you don’t just pay for what you use, you pay for what you allocate. When provisioning a Lambda function, you must specify how much compute memory it will be able to consume. That memory cap is what determines the GB-second cost of your function and ranges from 128MB-1.5GB. Each time the function runs, the execution time is rounded up to a coarse-grained time-based billing increment, then multiplied by the cost associated with the rounded-up memory allocation. This means that the cost to use one bit for a nanosecond is no different than the cost to use 128MB for 100 milliseconds.

Real Execution is the Only Valid Test

How long does your function take to execute? This will be different on each platform where you run it, and different based on real-time congestion levels, noisy neighbors, and varying generations of hardware infrastructure and thus performance. To get a true estimate for cost analysis, you really need to run it on Lambda and get the numbers from there. Running on your own machine or on static compute will produce different results. It’s important to get a handle on this number if you want a reliable estimate because increasing the compute time for each transaction can affect your billing significantly at high volume. Remember that each function call is a multiplier on your billable execution time.

The Hits Add Up

AWS provides a nice table on their pricing page to help explain: <https://aws.amazon.com/lambda/pricing/>

However, there’s a conflation of micro versus macro billing and second versus millisecond references that can confuse a casual reader. Let’s take a look at an example to illustrate it more clearly (Prices referenced in the following examples will use the amounts found on the Lambda pricing page as of February 25th, 2017—prices are subject to change).

Imagine we have a function that executes in 50ms, requires at most 128MB (the smallest allocation) of memory and will be hit 150 times per second. The billing breakdown is outlined in Figure 2 and explained below:

In Figure 2, we can see that the per hit charge is fairly easy to understand. At the time of this writing, it’s simply \$0.20 per 1 million monthly hits (minus 1 million free hits per month), which equates to \$0.0000002 per execution after the first 1 million free hits. Since our function is hit 150 times per second, that’s 394,200,000 hits per month (150 hits/second*60 sec/minute*60 minutes/hour*24 hours/day*365/12 days/ month) as shown in section A. We then subtract the free 1 million hits (B), which gives us 393,200,000 billable hits. Multiply that by 0.0000002 and we get \$78.64 per month (C).

The compute charge is more challenging to break down. First, Lambda will round up each of those function executions to 100ms, so fine-tuning your functions below 100ms won’t save any money! This is a huge consideration to performance engineers who are optimizing functions and services for scalability. I spent a great deal of time recently reducing an API runtime from 35ms to 2.7ms per hit, which was fantastic for scalability on reserved hardware but would have been a wasted effort on Lambda. Next, the compute charge is \$0.00001667 per GB-second. This is a difficult number to put into perspective until you enlarge it with the scale of your service, so let’s break it down:

Our function will be billed for 100ms of execution time (rounded up) each hit. The Lambda billing page did some math for us to show that 128MB is \$0.000000208 per 100ms (128MB/1024MB*\$0.00001667/10). So now we know that our function should cost \$0.000000208 per execution but we also get 400,000 GB-seconds

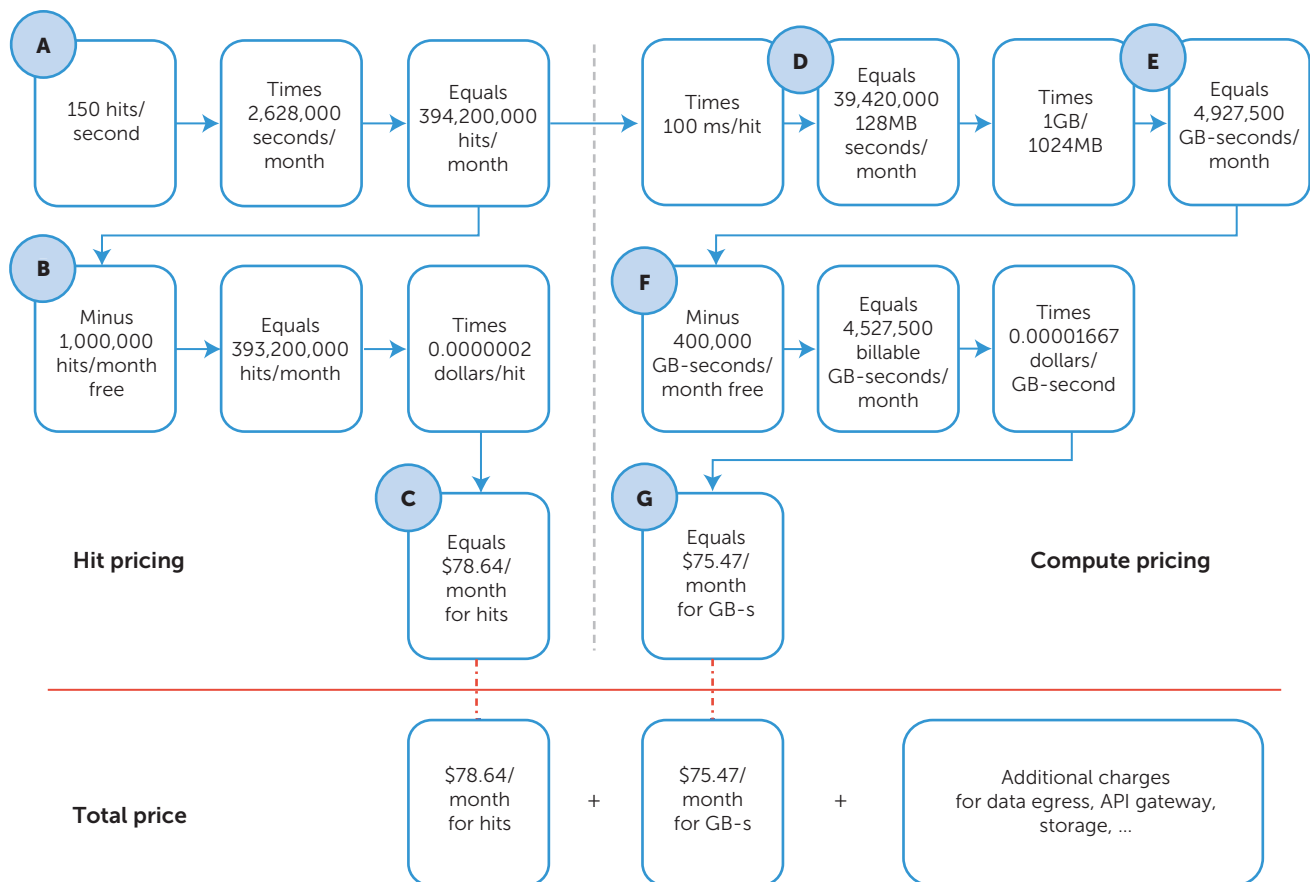


FIGURE 2: Calculating the total price of serverless functions based on transactions per second and execution time

per month free, so we need to expand that number to a monthly figure and expand our 128MB to 1GB in order to calculate it properly. The table shows free tier seconds at 128MB, but the billing is at GB-seconds, so this is extra confusing until we calculate our seconds of execution at GB memory level. We already know from calculating the per-hit charge that the function will run 394,200,000 times per month, so if we multiply that by 100ms (each execution), we see that our function will run for 39,420,000,000 milliseconds or 39,420,000 128MB compute seconds per month (D). Now, in (E), we can see that our function will use 4,927,500 GB-seconds of memory (39,420,000 seconds*128/1024MB). Subtracting the free tier gives us 4,527,500 (4,927,500-400,000) billable GB seconds (F). At \$0.00001667 per GB-second, this equates to \$75.47 per month (G).

So our function should cost about \$154.11 per month (as long as it maintains 150 hits per second). That may be a reasonable baseline cost for your function/API, but there's more to Lambda charges than the per-hit fee and the GB-seconds. You also pay for data egress, API Gateway (critical for rate-limiting public APIs), and anything else you use (S3, DynamoDB, etc). Serverless autoscales, and so does the cost. Is your traffic static or growing? You may need to consider what the cost will be if your traffic doubles, triples or grows by an order of magnitude.

Another option is Lambda@Edge, which has a higher per hit charge and no free tier, but it also rounds up to 50ms instead of 100ms, so when you run a cost analysis for your function, it's worth looking at both options to see which is more appropriate for your situation.

You Only Pay For What You Use—But You Pay For Everything You Use.

These extra charges add up, and add more complex variables to cost estimation. And when your service starts to succeed in growing usage, increasing that 300 hits per second to 10,000 hits per second, you may find yourself looking back at static hardware or reserved virtual-machines as a more cost effective solution. But are you now tied to an architecture you can't change? If the code isn't cloud-agnostic, you may have vendor lock-in, even in serverless. The interfaces are slightly different between all of the serverless providers—but of course, your development team wrote all the core logic in a module so you can swap between serverless and direct hosting, right? Vendor abstractions are sometimes useful, but if your DevOps team isn't empowered to cut and run at a moment's notice, you may regret it.

What about Microservices?

Frameworks are starting to appear that provide tooling for creating and managing an entire suite of microservices as serverless functions such as the Serverless Framework.⁷ Just as serverless functions aim to remove some of the complexity in operations by abstracting away everything under the hood, these frameworks abstract away the cloud provisioning and deployment with simple developer tools. When evaluating serverless for a set of functions, keep in mind that each function will receive the free tier, so it may be worth it to split a codebase into smaller functions to take more advantage of cost savings. However, because serverless functions are on-demand, there is a small startup lag time for running functions that may adversely affect your service performance. Times vary between language and memory allocation so run tests and see if it meets your needs.

What's in it for the Cloud Provider?

If you are a cloud provider who has already figured out autoscaling, security patching, load balancing and you've got a good handle on input/output queue management (e.g. Google, AWS, Azure), offering serverless is the next logical step. The 100ms micro-billing minimum creates re-usable billable compute time so you can sell 20ms of serverless to one customer, charge for 100ms and end up with 80ms of usable compute that you can keep selling. The func-

Table 1: Relative Costs Depend on Scale

Hits Per Second	VM Infrastructure Cost	Serverless Cost
150	\$200/month	\$167/month
30K	\$18K/month	\$55K/month

tions always round up to the nearest 100ms, which means some customers will run functions that take 10ms while others will run functions that take 101ms, billed at 100ms and 200ms respectively.

Additionally, offering serverless allows the cloud provider to have greater control over what underlying software exists within their datacenters. A savvy cloud provider could sell serverless as a way of unifying all operating system versions and security patches for customers with the same versions and tools they are using internally. Joyent's Triton, which only runs SmartOS across all deployed hardware is a great example of how this can be used to simplify the cloud provider's offering while maximizing the available compute to customers. When managed as a large Docker cluster, a cloud provider could be reselling compute time on top of already provisioned resources that they have allocated for large periodic jobs that run at intervals. Large cloud providers need to use a lot of compute in bursts to run builds, deployments, data extraction jobs, etc. So why not allow customers to pay for that compute while it's not being utilized at 100% capacity?

A Real-World Case Study

Several months ago, I was building a single API endpoint with Node.js. This sounded like a perfect candidate for Lambda, until I ran the cost analysis. My API was projected to be called about 150 times per second initially with an expected growth target of an average of 30K TPS in the next year. Initially, Lambda looked like a great option, but the cost analysis showed that this service would be triple the price on serverless compared to reserved compute instances. As the scale of the app grew, so did the enormity of the cost difference compared to classic on-demand compute instances, as shown in Table 1. These calculations are per month, so the cost difference per year would be nearing half a million dollars—for this one API. When looking at cost,

don't be fooled by a cheap cost per-hit. Looking at the planned and potential scale of your functions is critical for understanding the cost.

Doing this cost modeling is fairly complex as you have to account for everything mentioned above. Not every service will get the results shown in the table. If your function executes faster, you'll pay less per hit for serverless (up to the granularity of the billing increment) because the compute time charge will be lower, but if you get more hits per second, it adds up fast. Given the potential annual cost difference, it's smart to invest a few hours—or even months—modeling and testing execution time and understanding the financial implications.

Since this study was done for a public-facing API endpoint, it was also important to consider locking the API down to guard against denial-of-service attacks. When exposing your serverless function as a public API, don't forget to add rate limiting using API Gateway (additional charges may apply) or a simple API abuse attack could turn into an auto-scale billing nightmare.

SERVERLESS HAS THE POTENTIAL TO BE A GREAT ABSTRACTION OFFERING ECONOMIC ADVANTAGES FOR SIMPLE WORKFLOWS, BUT BEWARE OF THROWING EVERYTHING INTO IT TOO HASTILY. If your functions/APIs will use a whole compute instance, it's probably cheaper to get the instance—but you need to bear in mind the operational overhead of that path as well. Again, consider what you will be paying for. Even though serverless might be 3x the cost of on-demand compute, it might save DevOps cost in setting up autoscale, managing security patches and debugging issues with load balancers at scale. In short, it's wise to model the economic impact—now and in the future—of your architecture and operations choices. ●●●

References

1. "AWS Lambda," February 2017; <https://aws.amazon.com/lambda/>
2. "Cloud Functions," February 2017; <https://cloud.google.com/functions/>
3. "Azure Functions," February 2017; <https://azure.microsoft.com/en-us/services/functions/>
4. "IBM Bluemix OpenWhisk," February 2017; <https://www.ibm.com/cloud-computing/bluemix/openwhisk>
5. A. Singleton, "The Economics of Microservices," *IEEE Cloud Computing* (Volume: 3, Issue: 5, Sept.-Oct. 2016)
6. "AWS Lambda Pricing," February 2017; <https://aws.amazon.com/lambda/pricing>
7. "ServerLess Framework" February 2017; <https://serverless.com>

ADAM EIVY is a Solutions Architect for The Walt Disney Company in Seattle, where he investigates and experiments with the latest trends and techniques to set standards and bring learnings to the shared Engineering Services division. He also builds and supports developer tooling, build automation process, and production microservices. More about Adam on <http://adameivy.com>

JOE WEINMAN is a frequent global keynoter and author of *Clouconomics* and *Digital Disciplines*. He also serves on the advisory boards of several technology companies. Weinman has a BS in computer science from Cornell University and an MS in computer science from the University of Wisconsin-Madison. He has completed executive education at the International Institute for Management Development in Lausanne. Weinman has been awarded 22 patents. Contact him at joeweinman@gmail.com

myCS

Read your subscriptions through the myCS publications portal at <http://mycs.computer.org>.