



Developer's cheat sheet

The essential guide to serverless technologies and architectures

Microservices architectures may be more flexible and scalable than their monolithic predecessors, but what happens when your application, [built using microservices](#), gets lashed to a legacy, distributed systems infrastructure that can't keep up? The physical and software components of a traditional, three-tier infrastructure are expensive to set up, maintain, and manage. And for the developer, what should be a simple modification to an application can turn into an extensive and time-consuming project to ensure

[OUR CONTRIBUTORS](#)[ABOUT](#)[SUBSCRIBE](#)[Agile](#) [App Dev](#) [DevOps](#) [IT Ops](#) [Mobile](#) [Quality](#) [Security](#)  [Free Downloads](#)

performance, and let you pay only for the resources your application actually consumes. Suddenly, there's a direct, linear relationship between the efficiency of your code and what it costs to run it. The faster it runs, the less you pay.

While it's not a fit for every use case, [serverless is well positioned to play a central role](#) in the enterprise. In this essential guide, I tell you everything you need to know about the essentials of serverless computing, including the basic principles behind it, the pros and cons, the best use cases, and how the technology is evolving. I've included a list of resources at the end for further reading, including my book, "[Serverless Architectures on AWS](#)", on which this essential guide is based.

Note that in this article I primarily use Lambda as an example of a serverless compute service, as I consider it to be the most mature solution at the moment. However, there are other solutions worth a looking into, including Azure functions, Google Cloud Functions, IBM Open Whisk, and Auth0 WebTask.

But before diving into the details of serverless, let's start by defining the problem.

Application Migration to Cloud: Best Practices Guide

[Download Now](#)



[writes Tom Killalea](#), have fueled interest in and adoption of microservices, and even helped to popularize important practices, such as DevOps.

Developers and solution architects think of microservices as small, autonomous services built around a particular business capability or purpose. Fans of microservices point out numerous flaws with regard to monolithic systems, such as slow development cycles, complex deployments, high levels of coupling, and shared state.

Microservices address these issues, in theory, by allowing teams to work in parallel, build resilient and robust distributed architectures, and create “less coupled systems that can be changed faster and scaled more effectively,” [writes Sam Gibson](#).

That last point about building systems that can be changed faster and scale more effectively is an interesting one. A well-built distributed microservices architecture may scale better than a tightly coupled monolith. However, distributed systems have their own set of challenges, including more complex error handling, and the need to make remote, rather than in-process, calls.

Another challenge—and this goes for many monolithic systems as well—is high infrastructure provisioning and management overhead. Servers need to be provisioned, containers or VMs prepared and deployed, software patched, and the system stress-tested to make sure that it can scale under a heavy load. There is a constant need to maintain infrastructure and system software that comes at a price.



that may not be commensurate with what's needed to handle a sudden spike in traffic.

Compute capacity is usually overprovisioned, and for good reason. See that server getting to 70% capacity? It's time to provision a new one. All of these issues raise an obvious question: can we do better? Can we go one step further and build scalable, efficient, high-performance systems without the overhead of infrastructure management? Can we scale and pay exactly for what we need to run our software, and not more?

Aside from infrastructure concerns, there is also a question of software design and architecture, particularly with respect to web and mobile applications. Traditionally, three-tier systems (such as most web applications) have been written with thick middle tiers composed of multiple layers. These layers are designed to separate concerns such as the API, services, domain logic, business entities and models, data access and persistence, and so on.

This type of layering can lead to a maddening level of complexity and overhead. In the age of single-page apps (SPA) this problem has worsened due to the fact that many of these layers are replicated in the front-end tier of the application as well (see Figure 1). All of this makes changes to the system a difficult and time-consuming exercise. Introducing a simple feature often requires that every layer be modified, and the entire system tested and redeployed.

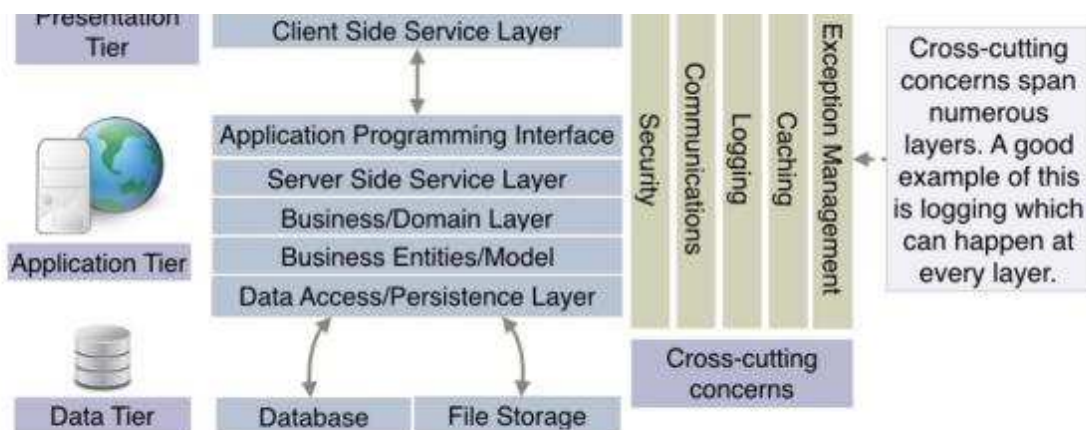


Figure 1: Tiers and layers in a typical web application (Sbarski, *Serverless Architectures on AWS*, 2016)

We want to move fast, iterate quickly, and scale our software. However, our current practices—even with microservices—do not allow us to do it very well. There is an overhead of infrastructure- and systems-level work that must be carried out. And the way we structure software, especially in case of web and mobile applications, can be inefficient, thanks to our multi-tier, multi-layer, multi-abstraction approach to design.

What is serverless?

That's where serverless computing comes into play. Serverless is an approach that aims to address infrastructure and software architecture issues I've outlined above by:

- Using a managed compute service such as AWS Lambda, Azure Functions, Google Cloud Functions, IBM OpenWhisk, or Auth0 WebTask to execute code



of code they need to write. By taking a serverless compute service and making use of various powerful single-purpose APIs and web services, developers can build loosely coupled, scalable, and efficient architectures quickly. In this way, developers can move away from servers and infrastructure concerns, and focus primarily on code, which is the [ultimate goal behind serverless](#).

Let's unpack what this really means. A serverless compute service, such as AWS Lambda, can execute code in response to events in a massively parallel way. It can respond to HTTP requests (using the AWS API Gateway), events raised by other AWS services, or it can be invoked directly using an API. As with S3, there is no need to provision capacity for Lambda, [writes Sam Kroonenburg](#). There are no servers to monitor or scale, there is no risk to cost by over-provisioning, and there's [no risk to performance by under-provisioning](#) (PDF document). Users are only charged for the time that their code executes, which is measured in seconds.

A user never pays for any idling servers or unused capacity. This is fundamentally different from what developers have been used to doing. The unit of scale in serverless is an ephemeral function that runs only when needed. This leads to an interesting and noticeable outcome: the performance of code visibly and measurably affects its cost. The quicker the function stops executing, the cheaper it is to run. This can, in turn, influence the design of functions, approach to caching, and how many and which dependencies the function relies on to run much more than in traditional server-based systems.

Serverless compute functions are typically considered to be stateless. That is, the state isn't kept between function instantiation. Developers must not



that, the Lambda runtime does reuse its lightweight Linux containers, which run functions under the hood. Developers who copied files over to the container's filesystem (/tmp directory) or ran processes, might find them again on subsequent instantiations.

This can be useful, but it can lead to additional work for the function if it has to clean up at the start or the end of each invocation. Nevertheless, Lambda does reserve the right to recreate containers at any time, so developers cannot rely on those files and processes being there.

It's important to note that the term "serverless" has received a lot of pushback, as some people feel that it is misleading. Serverless refers to the fact that we, as users of these technologies, do not have access to the actual servers. Naturally, there are still servers running in the background, but we do not have any ability to access or manage them. Rather, it is the vendor, such as Amazon, Google, or Microsoft, that is in charge of the machines. The vendor is responsible for providing a highly-available compute infrastructure—including capacity provisioning and automated scaling.

Some people argue that serverless should be referred to as function as a service (FaaS), and while it's not a bad name, serverless is a much broader concept than just an ephemeral function running in the cloud. But FaaS is a good term to use when referring specifically to services such as AWS Lambda, Google Cloud Functions, or Azure Functions.

Serverless functions, or FaaS, are not just a platform as a service



When a new serverless function needs to be created, it happens on the order of a few seconds for a cold function, and on the order of milliseconds for a warm function. The actual time it takes to spin up a cold function depends on several factors, such as the language runtime (Lambda supports JavaScript, Python and Java) and the number of dependencies you need to load.

Serverless cloud technologies such as Lambda are built on containers, but the advantage for us is that we don't need to manage them. We can spend most of our time thinking about code and software architecture instead. It's the vendor who has to work out the most efficient way to allocate resources and computing capacity, not us.

Tim Wagner, GM of Lambda, referenced this [in the keynote at the first conference](#) on serverless technologies in New York, when he said that Lambda attempts to solve the world's largest bin packing problem. It's just as well that they have to do it and not us, he said.

Serverless: Understanding third-party services and APIs

Another way to look at serverless technologies is from the perspective of third-party services and APIs. Using third-party services to outsource most of the work that isn't core or unique to the overall goal, can make a difference. If developers don't need to build yet another user registration and authentication system (using Auth0), or to manage complexities such as payments (using Stripe or Braintree), they can focus on problems that are



Serverless cloud functions and third-party services often make for a good fit. Cloud functions allow developers to add a bit of glue between third-party services, orchestrate invocations, marshal data, and coordinate workflows. Where before you needed a server to do this, FaaS is a perfect tool to be able to manage and use third-party services without having to pay for and look after servers.

Applying serverless patterns and architectures can further help developers pick up speed. One approach you can take is to allow the front end to communicate directly with services (including the database) where it's feasible and secure to do so. You can accomplish this by using technologies such as JSON Web Tokens and delegation tokens.

Allowing the front end to directly communicate and orchestrate interactions can reduce latency and make the overall application architecture simpler. A basic example of this is a web application that allows a user to upload files straight to an S3 bucket from the browser (entirely bypassing the need for a server). Figure 2 shows a sample architecture where the front end communicates directly with services, a database, and invokes Lambda functions via the API Gateway.

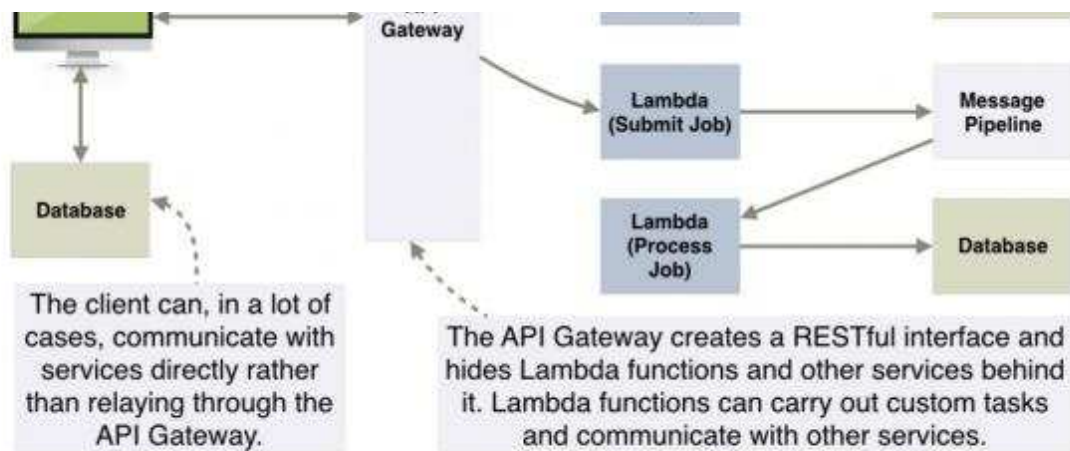


Figure 2: The front end can communicate with services directly and invoke Lambda functions through the API Gateway (Sbarski, *Serverless Architectures on AWS*, 2016).

Systems that require a lot of back end processing can adapt a microservices-like approach using serverless architectures. Each serverless function can be written as a microservice: It can be self-contained and autonomous, and it can address a business capability. However, serverless doesn't force you to adopt microservices. It gives you the flexibility to work within your own set of requirements. Instead of building microservices, developers can [create collections of small functions \(nanoservices\)](#) to carry out simple operations system-wide.

Developers often adopt a nanoservice-like style of development by designing functions that have very few responsibilities. An example is a back end for a web or a mobile application where the traditional middle tier is no longer present. Most of the logic that used to reside in that tier gets moved to the front end.

Developers use FaaS to create functions that execute protected actions that cannot be run on the front end due to privacy or security concerns. These



can update the database and if the operation is successful, submit the record to a search service for indexation). So this is not a microservices approach, but a way to coalesce most of the logic in the front end and use FaaS to perform certain operations in a way that's secure and outside of the user's sphere of influence. This pattern helps to reduce the size of the overall codebase, as most of the code is now concentrated in the front end.

The 5 principles of serverless design

There are several principles that can provide guidance with respect to building an entirely serverless system. These define what a serverless system looks like, and what properties it exhibits (for more information see [Serverless Architectures on AWS](#).)

These principles generally apply if you decide to create an entire system (back end and front end) using a serverless approach. If you are building other types of systems, such as a pipeline for transforming a file, the principle relevant to the front end will not apply.

1. Use a compute service to execute code on demand

A serverless compute service such as Lambda, Azure Functions, Auth0 WebTask, or Google Cloud Functions must be used to execute code. Do not run or manage any servers, VMs, or containers of your own. Your custom code should be entirely run out of FaaS to gain the most benefit.

2. Write single-purpose, stateless functions



3. Design push-based, event-driven pipelines

Create push-based, event-driven pipelines to carry out complex computations and tasks. Use a serverless compute service to orchestrate actions between different services, and try to build in a way that creates event-driven pipelines. Avoid polling or manual intervention where possible (see Figure 3).

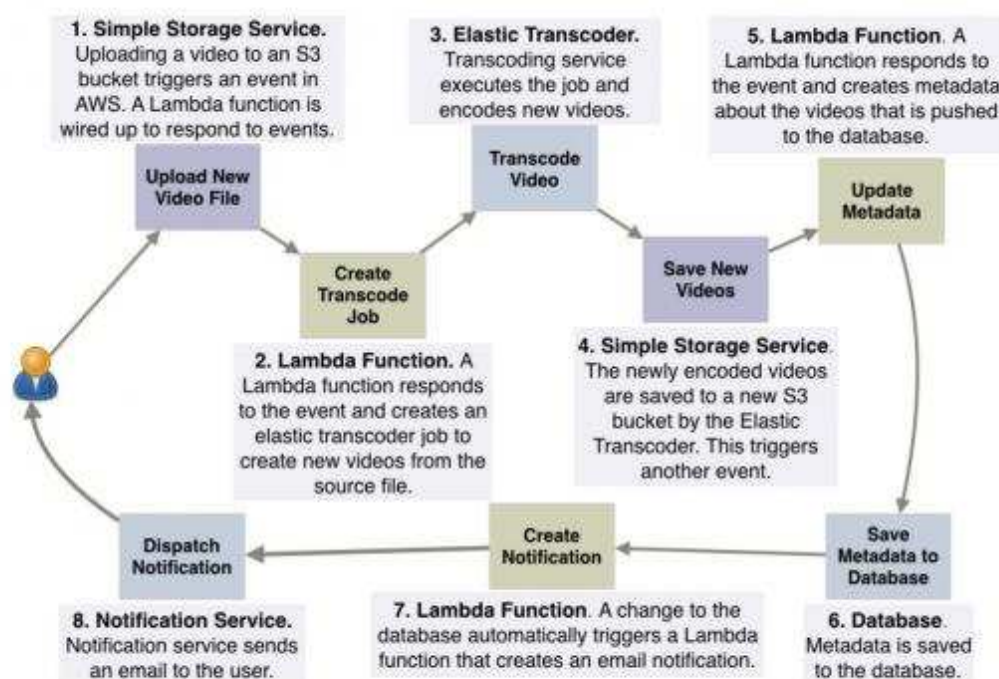


Figure 3: An event-driven, push-based pipeline for transcoding video files. This pipeline requires the user to upload a new file. From there everything happens without any additional user input as services invoke Lambda functions and Lambda functions invoke other service (Sbarski, *Serverless Architectures on AWS*, 2016).

4. Create thicker, more powerful front ends

Move as much logic as possible to the front end and make it smarter. Your front end should be able to interact with services directly in order to limit the number of serverless functions. Naturally, there will be instances where the client cannot or must not communicate with services directly, possibly for



SERVICES BUILT BY OTHERS, WITH ONE CAVEAT. ALWAYS MAKE AN ASSESSMENT AND think about risks. If you apply this principle, you trade control for speed. But, as with most things, you need to decide whether it is the right trade-off for you.

The best use cases for serverless

A serverless approach can work for everything from large applications that need a scalable back end all the way down to small tasks that only need to run occasionally. Creating a RESTful interface and putting functions behind it is a common use case in the serverless world (see Figure 2). Note that AWS is the only platform that uses a separate API Gateway; other vendors integrate HTTP listeners into their functions.

We have also seen developers combine serverless functions and technologies such as [GraphQL](#) to create back-end systems. GraphQL can run from a single function, it can talk to multiple data sources, and it can [coalesce a response in the shape of the original request](#). It negates the need to develop a full RESTful interface, which makes GraphQL an attractive choice for some systems. (Figure 4 shows what a basic serverless GraphQL architecture might look like.)

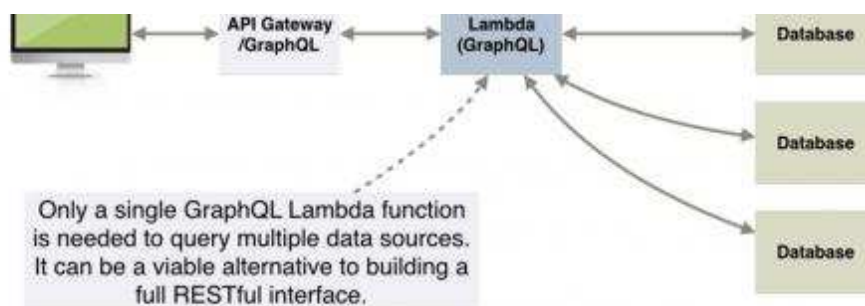


Figure 4: GraphQL can provide an alternative way of building back ends without having to deal with issues that REST systems tend to experience (for example, poor versioning or overly chatty communications) (Sbarski, *Serverless Architectures on AWS*, 2016).

Other popular use cases for serverless technologies include data processing, format conversion, encoding, aggregation of data, and image resizing (See Figure 5). You can schedule a compute service such as AWS Lambda to run at certain times or to automatically respond to new files, such as S3, as they are added to storage.

Serverless compute functions are useful for heavy and irregular work loads, as they scale rapidly and automatically. That makes them suitable for real-time analytics and processing of events, logs, transactions, click-data, and so forth, when coupled with services such as Amazon Kinesis Streams. And, thanks to the fact that some Kinesis Streams. And, thanks to the fact that some serverless compute services can run on a schedule, it's straightforward to build all kinds of useful utilities and helpers, such as daily backup routines, that need to execute at specific times.

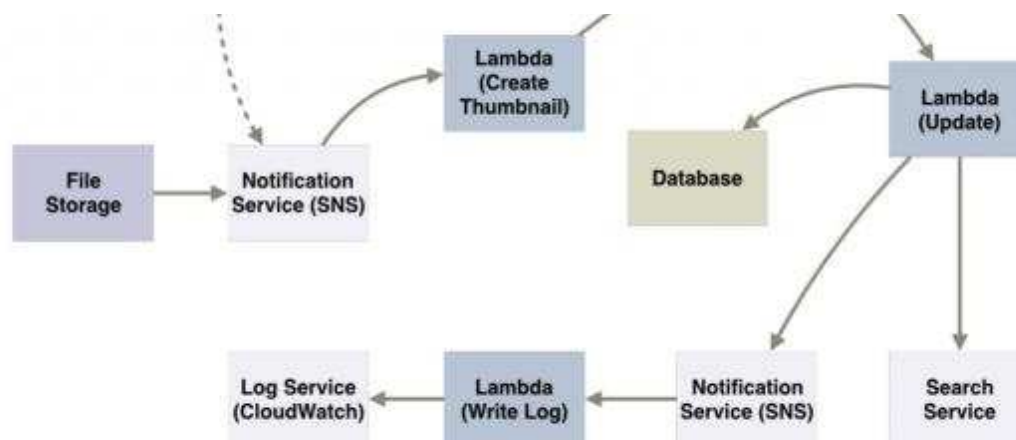


Figure 5: Use FaaS to bring different services together. You can create powerful transformation and processing systems using serverless technologies (Sbarski, *Serverless Architectures on AWS*, 2016).

Creating a wrapper for legacy APIs is another interesting use case for serverless technologies. Old services and APIs can be difficult to deal with, especially if they require multiple consumers. Instead of modifying each consumer to support a dated protocol, it's often easier to create a wrapper around the service by putting a function in front of it. A serverless function can marshal data, process requests from consumers, and create new requests that the legacy service can work with. Likewise, it can read responses and transform them to modern representations (See Figure 6).

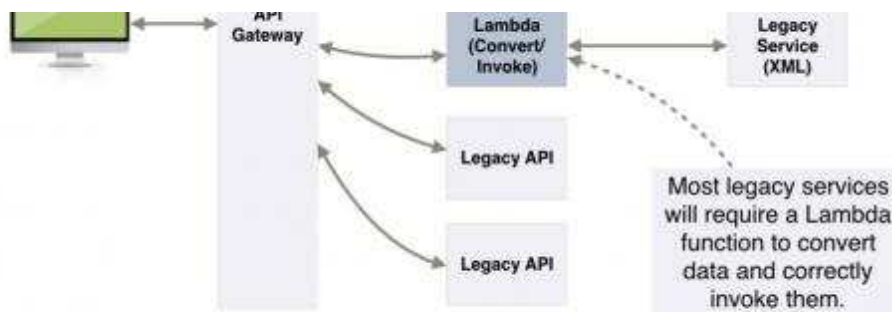


Figure 6: Serverless technologies can be used to create a wrapper between modern consumers and legacy APIs thus giving them an extended lease on life (Sbarski, *Serverless Architectures on AWS*, 2016).

Serverless is not an all-or-nothing proposition. Existing server-based legacy systems can be extended with FaaS and third-party services. In fact, by using FaaS developers can gradually refactor legacy systems to give them a longer lease on life. They can make parts of their system more scalable and at a cheaper cost (See Figure 7).

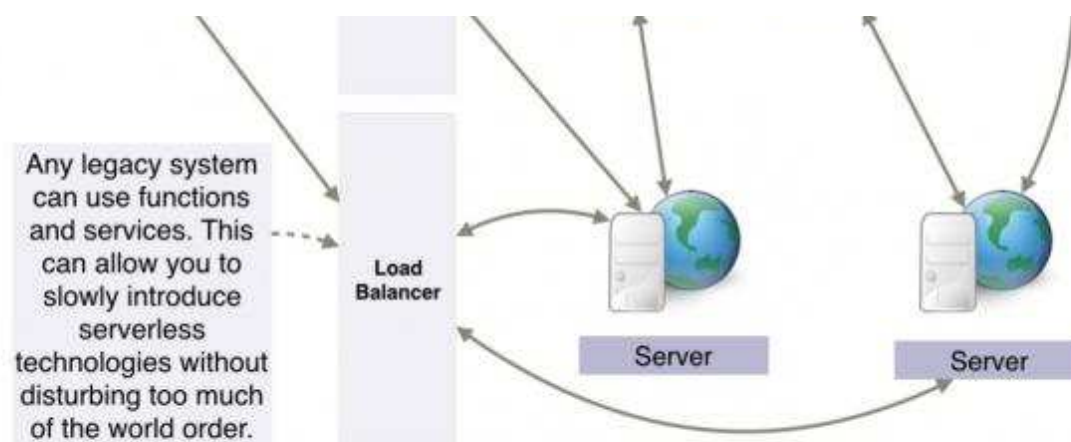


Figure 7: Existing legacy systems can be extended with FaaS and third-party services (Sbarski, *Serverless Architectures on AWS*, 2016).

Still another popular use case for serverless technologies is to create online bots and back ends for IoT devices. For example, there are serverless bots built for slack, telegram, and Facebook's messaging platform. Given how easy it is to program and deploy a serverless function, you will undoubtedly see a lot of other interesting and useful applications for serverless technologies.

Operations: A powerful serverless advantage

Another area where serverless computing conveys a powerful advantage is in operations. DevOps brings a different focus in the serverless world, with more attention paid to the deployment of functions and organization of environments, rather than infrastructure management. From a purely serverless perspective, there is no longer need to worry about infrastructure, operating systems, or network configuration, since all of those tasks are outsourced to a vendor.



automation. Deploying functions by hand and organizing environments manually is untenable. Thankfully, frameworks such as the [Serverless Framework](#), and [Apex](#) are available to assist with the organization and deployment of functions.

Decision factors

Like any technology, serverless is not a silver bullet. I've talked about best use cases, but there are others that are simply not suitable for serverless right now.

Public cloud

Running any service from a public cloud—including FaaS—leaves you at the whimsy of the vendor. Performance and reliability need to be considered and the architecture set up so that if an availability region goes down, the application continues to function. The scalability offered by public cloud providers is what makes serverless cloud functions powerful, but if you are building mission critical systems, you might have to consider additional ways to guarantee service. Interestingly, two vendors — IBM and Microsoft — provide a way to run their FaaS implementations, Azure Functions and IBM OpenWhisk, on your own machines.

Vendor lock-in

Vendor lock-in, data sovereignty, cost, support, documentation, and company viability are all issues to think about. Migrating serverless functions from one vendor to another may not actually be the main issue. A much bigger problem

[OUR CONTRIBUTORS](#)[ABOUT](#)[SUBSCRIBE](#)[Agile](#) [App Dev](#) [DevOps](#) [IT Ops](#) [Mobile](#) [Quality](#) [Security](#)  [Free Downloads](#)

SERVERLESS tooling needs improvement. Software such as [the Serverless Framework](#) has made things easier, but there is still more maturation that needs to happen. Logging, introspection, and debugging can be difficult, but these are not insurmountable problems. More work around serverless architectures and patterns still needs to occur.

Distributed computing

Working with distributed computing architectures can be complex and take time to get right. This applies to microservices and serverless architectures in equal measure. Even decisions about how granular a function should be, could take time to assess, implement, test, and refactor. Make something too granular and you will have too many functions to manage. But ignore granularity and you'll end up dealing with mini monoliths.

Natural limitations

Serverless functions may have natural limitations. With serverless, there's a limit to the amount of RAM you can have, and there's no way to change out the CPU. Some vendors, such as Amazon, impose timeouts on functions, so long running tasks may be impossible. There will be limits on the number of functions that can run in parallel, and so on.

And, what are the advantages of taking a serverless approach?

No infrastructure management

Infrastructure management is now someone else's concern. You are left to deal with functions and third-party services, instead of servers, system patching, and configuration. This is a major advantage of a serverless



automated, allowing for fast releases. The packaging and deployment of serverless functions is definitely easier than having to roll your own containers, or write Puppet and Chef scripts. Deployment with Lambda, for example, requires that you zip up your function, along with any dependencies, and upload it to AWS.

Cost

From a FaaS perspective, you only incur the cost when a function runs. This means that you are only paying when something useful is happening — that is, when the user performs an action or a system event takes place.

Every vendor has a different pricing system. Amazon, for example, has a generous free tier for Lambda. A 128MB function running 5 million times a month for 5 seconds each time costs \$45.43/month, while a 512MB function running 200,000 times each month for 10 seconds each time prices out to \$10.00/month (I included the free tier provided by Amazon in these calculations).

Go to Serverlesscalc.com to model your expected cost.

Scalability and availability

Out-of-the-box scalability is a big reason why you might use serverless functions. Functions scale effortlessly in response to events, without the need for the developer to do additional work. There is still a need to think through architecture, but the scalability and availability aspects are, to a great extent, handled by the vendor (a lot of architectural recommendations with regards to cloud still apply—e.g. having functions replicated across multiple regions).



Moving most of the code base to the front end and not having a large middle tier can result in more speed. FaaS also allows developers to begin trying things straight away because it has a low barrier to entry. It's easy to create a function and begin trying things immediately.

What's next for serverless

All of the major public cloud vendors are now on board with serverless, including Amazon, Google, Microsoft, and IBM. Each company is building its own FaaS offerings and working on services (authentication, databases, storage, notifications, messaging, queuing) that developers can use. And there are plenty of other, smaller companies that have fantastic, reliable services that developers can leverage.

It is likely that many developers and organizations will try out serverless technologies and architectures due to the benefits they provide. Developers will use them to build web, mobile, game, and IoT back ends; to process data, and to create powerful pipelines that perform complex operations.

We will see a marketplace for serverless functions, and widespread adoption of serverless architectures, particularly among those who have already adopted cloud technologies. Companies that value competitiveness and innovation will study serverless technologies closely, and move quickly to adopt serverless wherever possible.


Do you have questions about [serverless](#) that I didn't answer? Post them below and

OUR CONTRIBUTORS

ABOUT

SUBSCRIBE



Agile App Dev DevOps IT Ops Mobile Quality Security 

 Free Downloads

Download now

Topics: [IT Ops](#)

Share this article



Peter Sbarski
VP of Engineering
A Cloud Guru
1 article

Follow @sbarski

Article Tags


Cloud Computing

OUR CONTRIBUTORS

ABOUT

SUBSCRIBE



Agile App Dev DevOps IT Ops Mobile Quality Security 

 Free Downloads

Get fresh whitepapers, reports,
case studies, and articles weekly.

Enter your email address

 SUBSCRIBE

[OUR CONTRIBUTORS](#)[ABOUT](#)[SUBSCRIBE](#)[Agile](#) [App Dev](#) [DevOps](#) [IT Ops](#) [Mobile](#) [Quality](#) [Security](#) [Free Downloads](#)**Frank Wang** • 2 months ago

Hi Peter- Great post! I'm a big believer of this approach, and hacked on a couple of projects with the Serverless framework on AWS. I felt like there is a lack of comprehensive and detailed resources on the complete serverless stack like the one you covered here.

I decided (along with a friend) to put together a really comprehensive 60 chapter tutorial on how to build a serverless CRUD app using the Serverless Framework on AWS. It's called Serverless Stack - <http://serverless-stack.com>. It's a completely free resource for the community. I'm hoping you could take a look at it, give us some feedback, and possibly use your help spreading the word for it.

Cheers,

Frank

1 • [Reply](#) • [Share](#) ›

Related Articles

OUR CONTRIBUTORS

ABOUT

SUBSCRIBE



Agile App Dev DevOps IT Ops Mobile Quality Security 

 Free Downloads



Google goes AI-first at I/O: What dev and ops need to know

All in on AI



25 IT Ops pros and experts to follow on Twitter

Follow the leaders



An essential guide to the serverless ecosystem

Today's ocean of options



The state of IT Ops visual management: Is the big picture in reach?

Frames of reference

OUR CONTRIBUTORS

ABOUT

SUBSCRIBE



Agile App Dev DevOps IT Ops Mobile Quality Security 

 Free Downloads



How to take an architectural approach to IT automation

Do more with less



5 best practices for container orchestration in IT production

Plan your voyage



How do you recover when your cloud is the disaster?

Weatherproof your apps



Note to CIOs in the age of continuous IT: Prepare for existential change

IT on the move

OUR CONTRIBUTORS

ABOUT

SUBSCRIBE



[Agile](#) [App Dev](#) [DevOps](#) [IT Ops](#) [Mobile](#) [Quality](#) [Security](#) 

 Free Downloads



How IT Ops can boost your user experience

Design thinking



The business case for IT4IT: Why you need it. How to build it.

Core arguments



5 GDPR compliance tips for your IT Ops team

Privacy, please



With immutable infrastructure, your systems can rise from the dead

ZombieOps with Kubernetes

Show More




Brought to you by

OUR CONTRIBUTORS

ABOUT

SUBSCRIBE



Agile App Dev DevOps IT Ops Mobile Quality Security 

 Free Downloads

Quality

Security

© Copyright 2017 Hewlett Packard Enterprise Development LP