Serverless Architectures

Serverless architectures refer to applications that significantly depend on third-party services (knows as Backend as a Service or "BaaS") or on custom code that's run in ephemeral containers (Function as a Service or "FaaS"), the best known vendor host of which currently is AWS Lambda. By using these ideas, and by moving much behavior to the front end, such architectures remove the need for the traditional 'always on' server system sitting behind an application. Depending on the circumstances, such systems can significantly reduce operational cost and complexity at a cost of vendor dependencies and (at the moment) immaturity of supporting services.

04 August 2016

Mike Roberts

Mike is an engineering leader living in New York City. While spending much of his time these days managing people and teams he also still gets to code occasionally, especially in Clojure, and has Opinions about software architecture. He is cautiously optimistic that Serverless architectures may be worth some of the hype that they are currently receiving.

Find **similar articles** at the tag:
application architecture

Contents expand

Serverless is a hot topic in the software architecture world. We’re already seeing books, open source frameworks, plenty of vendor products, and even a conference dedicated to the subject. But what is Serverless and why is (or isn’t) it worth considering? Through this evolving publication I hope to enlighten you a little on these questions.

To start we'll look at the ‘what’ of Serverless where I try to remain as neutral as I can about the benefits and drawbacks of the approach - we'll look at those topics later.

This article provides an in-depth look at serverless architecture and as a result is a long read. If you need a concise summary of what serverless is and its trade-offs - take a look at the bliki entry on serverless

What is Serverless?

Like many trends in software there’s no one clear view of what ‘Serverless’ is, and that

isn't helped by it really coming to mean two different but overlapping areas:

1. Serverless was first used to describe applications that significantly or fully depend on 3rd party applications / services ('in the cloud') to manage server-side logic and state. These are typically 'rich client' applications (think single page web apps, or mobile apps) that use the vast ecosystem of cloud accessible databases (like Parse, Firebase), authentication services (Auth0, AWS Cognito), etc. These types of services have been previously described as '(Mobile) Backend as a Service', and I'll be using **'BaaS'** as a shorthand in the rest of this article.

2. Serverless can also mean applications where some amount of server-side logic is still written by the application developer but unlike traditional architectures is run in stateless compute containers that are event-triggered, ephemeral (may only last for one invocation), and fully managed by a 3rd party. (Thanks to ThoughtWorks for their definition in their most recent Tech Radar.) One way to think of this is 'Functions as a service / FaaS' . AWS Lambda is one of the most popular implementations of FaaS at present, but there are others. I'll be using **'FaaS'** as a shorthand for this meaning of Serverless throughout the rest of this article.

Mostly I'm going to talk about the second of these areas because it is the one that is newer, has significant differences to how we typically think about technical architecture, and has been driving a lot of the hype around Serverless.

However these concepts are related and, in fact, converging. A good example is Auth0 - they started initially with BaaS 'Authentication as a Service', but with Auth0 Webtask they are entering the FaaS space.

Furthermore in many cases when developing a 'BaaS shaped' application, especially when developing a 'rich' web-based app as opposed to a mobile app, you'll likely still need some amount of custom server side functionality. FaaS functions may be a good solution for this, especially if they are integrated to some extent with the BaaS services you're using. Examples of such functionality include data validation (protecting against imposter clients) and compute-intensive processing (e.g. image or video manipulation.)

Origin of 'Serverless'

The term 'Serverless' is confusing since with such applications there are both server hardware and server processes running somewhere, but the difference to normal approaches is that the organization building and supporting a 'Serverless' application is not looking after the hardware or the processes - they are outsourcing this to a vendor.

First usages of the term seem to have appeared around 2012, including this article by Ken Fromm. Badri Janakiraman says that he also heard usage of the term around this time in regard to continuous integration and source control systems being hosted as a service, rather than on a company's own servers. However this usage was about development infrastructure rather than incorporation into products.

We start to see the term used more frequently in 2015, after AWS Lambda's launch in 2014 and even more so after Amazon's API Gateway launched in July 2015. Here's an example where Ant Stanley writes about Serverless following the API Gateway announcement. In October 2015 there was a talk at Amazon's re:Invent conference titled "The Serverless Company using AWS Lambda", referring to PlayOn! Sports. Towards the end of 2015 the 'Javascript Amazon Web Services (JAWS)' open source project renamed themselves to the Serverless Framework, continuing the trend.

Fast forward to today (mid 2016) and one sees examples such as the recent Serverless Conference, plus the various Serverless vendors are embracing the term from product descriptions to job descriptions. Serverless as a term, for better or for worse, is here to stay.

A couple of examples

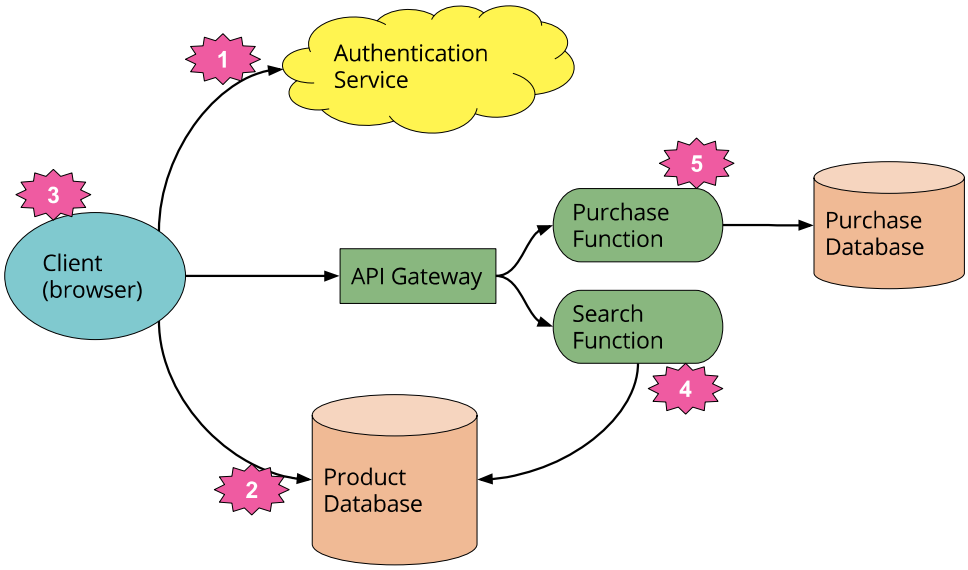
UI-driven applications

Let's think about a traditional 3-tier client-oriented system with server-side logic. A good example is a typical ecommerce app (dare I say an online pet store?)

Traditionally the architecture will look something like this, and let's say it's implemented in Java on the server side, with a HTML / Javascript component as the client:

With this architecture the client can be relatively unintelligent, with much of the logic in the system - authentication, page navigation, searching, transactions - implemented by the server application.

With a Serverless architecture this may end up looking more like this:



This is a massively simplified view, but even with this there are a number of significant changes that have happened here. Please note this is not a recommendation of an architectural migration, I'm merely using this as a tool to expose some Serverless concepts!

1. We've deleted the authentication logic in the original application and have replaced it with a third party BaaS service.

2. Using another example of BaaS, we've allowed the client direct access to a subset of our database (for product listings), which itself is fully 3rd party hosted (e.g. AWS Dynamo.) We likely have a different security profile for the client accessing the database in this way from any server resources that may access the database.

3. These previous two points imply a very important third - some logic that was in the Pet Store server is now within the client, e.g. keeping track of a user session, understanding the UX structure of the application (e.g. page navigation), reading from a database and translating that into a usable view, etc. The client is in fact well on its way to becoming a Single Page Application.

4. Some UX related functionality we may want to keep in the server, e.g. if it's compute

intensive or requires access to significant amounts of data. An example here is 'search'. For the search feature instead of having an always-running server we can implement a FaaS function that responds to http requests via an API Gateway (described later.) We can have both the client, and the server function, read from the same database for product data.

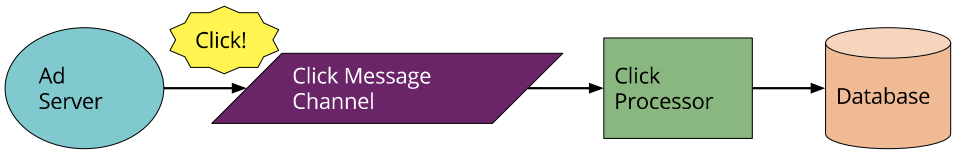
Since the original server was implemented in Java, and AWS Lambda (our FaaS vendor of choice in this instance) supports functions implemented in Java, we can port the search code from the Pet Store server to the Pet Store Search function without a complete re-write.

5. Finally we may replace our 'purchase' functionality with another FaaS function, choosing to keep it on the the server-side for security reasons, rather than re-implement it in the client. It too is fronted by API Gateway.

Message-driven applications

A different example is a backend data-processing service. Say you're writing a user-centric application that needs to quickly respond to UI requests, but secondarily you want to capture all the different types of activity that are occurring. Let's think about an online ad system - when a user clicks on an advertisement you want to very quickly redirect them to the target of the ad, but at the same time you need to collect the fact that the click has happened so that you can charge the advertiser. (This example is not hypothetical - my former team at Intent Media recently went through this exact redesign.)

Traditionally, the architecture may look like this. The 'Ad Server' synchronously responds to the user - we don't care about that interaction for the sake of this example - but it also posts a message to a channel that can be asynchronously processed by a 'click processor' *application* that updates a database, e.g. to decrement the advertiser's budget.



In the Serverless world this looks like:

There's a much smaller difference to the architecture here compared to our first example. We've replaced a long lived consumer *application* with a FaaS *function* that runs within the event driven context the vendor provides us. Note that the vendor supplies both the Message Broker and the FaaS environment - the two systems are closely tied to each other.

The FaaS environment may also process several clicks in parallel by instantiating

multiple copies of the function code - depending on how we'd written the original process this may be a new concept we need to consider.

Unpacking 'Function as a Service'

We've mentioned the FaaS idea a lot already but it's time to dig into what it really means. To do this let's look at the opening description for Amazon's Lambda product. I've added some tokens to it, which I then expand upon.

> *AWS Lambda lets you run code without provisioning or managing servers. **(1)** ... With Lambda, you can run code for virtually any type of application or backend service **(2)** - all with zero administration. Just upload your code and Lambda takes care of everything required to run **(3)** and scale **(4)** your code with high availability. You can set up your code to automatically trigger from other AWS services **(5)** or call it directly from any web or mobile app **(6)**.*

1. **Fundamentally FaaS is about running back end code without managing your own server systems or your own server applications.** That second clause - **server applications** - is a key difference when comparing with other modern architectural trends like containers and PaaS (Platform as a Service.)

   If we go back to our click processing example from earlier what FaaS does is replace the click processing server (possibly a physical machine, but definitely a specific application) with something that doesn't need a provisioned server, nor an application that is running all the time.

2. FaaS offerings do not require coding to a specific framework or library. FaaS functions are regular applications when it comes to language and environment. For instance AWS Lambda functions can be implemented 'first class' in Javascript, Python and any JVM language (Java, Clojure, Scala, etc.). However your Lambda function can execute another process that is bundled with its deployment artifact, so you can actually use any language that can compile down to a Unix process (see Apex later on.) FaaS functions do have significant architectural restrictions, especially when it comes to state and execution duration, and we'll come to that soon.

   Let's consider our click processing example again - the only code that needs to change when moving to FaaS is the 'main method / startup' code, in that it is deleted, and likely the specific code that is the top-level message handler (the 'message listener interface' implementation), but this might only be a change in method signature. All of the rest of the code (e.g. the code that writes to the database) is no different in a FaaS world.

3. Since we have no server applications to run deployment is very different to traditional systems - we upload the code to the FaaS provider and it does everything else. Right now that typically means uploading a new definition of the code (e.g. in a zip or JAR file), and then calling a proprietary API to initiate the update.

4. Horizontal scaling is completely automatic, elastic, and managed by the provider. If your system needs to be processing 100 requests in parallel the provider will handle that without any extra configuration on your part. The 'compute containers' executing your functions are ephemeral with the FaaS provider provisioning and destroying them purely driven by runtime need.

   Let's return to our click processor. Say that we were having a good day and customers were clicking on 10 times as many ads as usual. Would our click

processing application be able to handle this? For example did we code to be able to handle multiple messages at a time? Even if we did would one running instance of the application be enough to process the load? If we are able to run multiple processes is auto-scaling automatic or do we need to reconfigure that manually? With FaaS you need to write the function ahead of time to assume parallelism, but from that point on the FaaS provider automatically handles all scaling needs.

5. Functions in FaaS are triggered by event types defined by the provider. With Amazon AWS such stimuli include S3 (file) updates, time (scheduled tasks) and messages added to a message bus (e.g. Kinesis). Your function will typically have to provide parameters specific to the event source it is tied to. With the click processor we made an assumption that we were already using a FaaS-supported message broker. If not we would have needed to switch to one, and that would have required making changes to the message producer too.

6. Most providers also allow functions to be triggered as a response to inbound http requests, typically in some kind of API gateway. (e.g. AWS API Gateway, Webtask) . We used this in our Pet Store example for our 'search' and 'purchase' functions.

State

FaaS functions have significant restrictions when it comes to local (machine / instance bound) state. In short you should assume that for any given invocation of a function none of the in-process or host state that you create will be available to *any* subsequent invocation. This includes state in RAM and state you may write to local disk. In other words from a deployment-unit point of view *FaaS functions are stateless*.

This has a huge impact on application architecture, albeit not a unique one - the 'Twelve-Factor App' concept has precisely the same restriction.

Given this restriction what are alternatives? Typically it means that FaaS functions are either naturally stateless - i.e. they provide pure functional transformations of their input - or that they make use of a database, a cross-application cache (e.g. Redis), or network file store (e.g. S3) to store state across requests or for further input to handle a request.

Execution Duration

FaaS functions are typically limited in how long each invocation is allowed to run. At present AWS Lambda functions are not allowed to run for longer than 5 minutes and if they do they will be terminated.

This means that certain classes of long lived task are not suited to FaaS functions without re-architecture, e.g. you may need to create several different coordinated FaaS functions where in a traditional environment you may have one long duration task performing both coordination and execution.

Startup Latency

At present how long it takes your FaaS function to respond to a request depends on a large number of factors, and may be anywhere from 10ms to 2 minutes. That sounds bad, but let's get a little more specific, using AWS Lambda as an example.

If your function is implemented in Javascript or Python and isn't huge (i.e. less than a thousand lines of code) then the overhead of running it in should never be more than 10 - 100 ms. Bigger functions may occasionally see longer times.

If your Lambda function is implemented on the JVM you may occasionally see long

response times (e.g. > 10 seconds) while the JVM is spun up. However this only notably happens with either of the following scenarios:

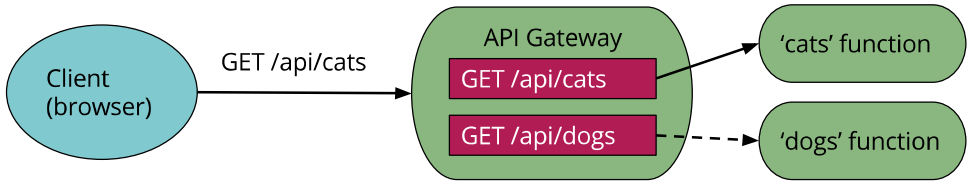
- Your function processes events infrequently, on the order of longer than 10 minutes between invocations.
- You have very sudden spikes in traffic, for instance you typically process 10 requests per second but this ramps up to 100 requests per second in less than 10 seconds.

The former of these may be avoided in certain situations by the ugly hack of pinging your function every 5 minutes to keep it alive.

Are these issues a concern? It depends on the style and traffic shape of your application. My former team has an asynchronous message-processing Lambda app implemented in Java which processes hundreds of millions of messages / day, and they have no concerns with startup latencey. That said if you were writing a low-latency trading application you probably wouldn't want to use FaaS systems at this time, no matter the language you were using for implementation.

Whether or not you think your app may have problems like this you should test with production-like load to see what performance you see. If your use case doesn't work now you may want to try again in a few months time since this is a major area of development by FaaS vendors.

API Gateway



One aspect of FaaS that we brushed upon earlier is an 'API Gateway'. An API Gateway is an HTTP server where routes / endpoints are defined in configuration and each route is associated with a FaaS function. When an API Gateway receives a request it finds the routing configuration matching the request and then calls the relevant FaaS function. Typically the API Gateway will allow mapping from http request parameters to inputs arguments for the FaaS function. The API Gateway transforms the result of the FaaS function call to an http response, and returns this to the original caller.

Amazon Web Services have their own API Gateway and other vendors offer similar abilities.

Beyond purely routing requests API Gateways may also perform authentication, input validation, response code mapping, etc. Your spidey-sense may be buzzing about whether this is actually such a good idea, if so hold that thought - we'll consider this further later.

One use case for API Gateway + FaaS is for creating http-fronted microservices in a Serverless way with all the scaling, management and other benefits that come from FaaS functions.

At present tooling for API gateways is achingly immature and so while defining applications with API gateways is possible it's most definitely not for the faint-hearted.

Tooling

The comment above about API Gateway tooling being immature actually applies, on the whole, to Serverless FaaS in general. There are exceptions however - one example is Auth0 Webtask which places significant priority on Developer UX in its tooling. Tomasz Janczuk gave a very good demonstration of this at the recent Serverless Conference.

Debugging and monitoring are tricky in general in Serverless apps - we'll get into this further in subsequent installments of this article.

Open Source

One of the main benefits of Serverless FaaS applications is transparent production runtime provisioning, and so open source is not currently as relevant in this world as it is for, say, Docker and containers. In future we may see a popular FaaS / API Gateway platform implementation that will run 'on premise' or on a developer workstation. IBM's OpenWhisk is an example of such an implementation and it will be interesting to see whether this, or an alternative implementation, picks up adoption.

Apart from runtime implementation though there are already open source tools and frameworks to help with definition, deployment and runtime assistance. For instance the Serverless Framework makes working with API Gateway + Lambda significantly easier than using the first principles provided by AWS. It's Javascript heavy but if you're writing JS API Gateway apps it's definitely worth a look.

Another example is Apex - a project to 'Build, deploy, and manage AWS Lambda functions with ease.' One particularly interesting aspect of Apex is that it allows you to develop Lambda functions in languages other than those directly supported by Amazon, e.g. Go.

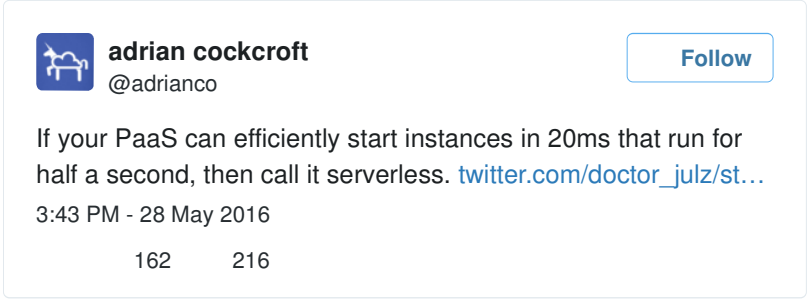
What isn't Serverless?

So far in this article I've defined 'Serverless' to mean the union of a couple of other ideas - 'Backend as a Service' and 'Functions as a Service'. I've also dug into the capabilities of the second of these.

Before we start looking at the very important area of benefits and drawbacks I'd like to spend one more moment on definition, or at least defining what Serverless isn't. I've seen some people (including me in the recent past) get confused about these things and I think it's worth discussing them for clarity's sake.

Comparison with PaaS

Given that Serverless FaaS functions are very similar to 12-Factor applications, are they in fact just another form of 'Platform as a Service' (PaaS) like Heroku? For a brief answer I refer to Adrian Cockcroft



In other words most PaaS applications are not geared towards bringing entire applications up and down for every request, whereas FaaS platforms do exactly this.

OK, but so what, if I'm being a good 12-Factor App developer there's still no difference to how I code? That's true, but there is a big difference to how you *operate* your app. Since we're all good DevOps-savvy engineers we're thinking about operations as much as we are about development, right?

The key operational difference between FaaS and PaaS is *scaling*. With most PaaS's you still need to think about scale, e.g. with Heroku how many Dynos you want to run. With a FaaS application this is completely transparent. Even if you setup your PaaS application to auto-scale you won't be doing this to the level of individual requests (unless you have a very specifically shaped traffic profile), and so a FaaS application is much more efficient when it comes to costs.

Given this benenfit, why would you still use a PaaS? There are several reasons but tooling, and maturity of API gateways, are probably the biggest. Furthermore 12-Factor Apps implemented in a PaaS may use an in-app readonly cache for optimization, which isn't an option for FaaS functions.

Comparison with containers

One of the reasons for Serverless FaaS is to avoid having to manage computational processes at the operating system level or lower. Platforms-as-a-Service (like Heroku) are another, and I've described above how PaaS's are different to Serverless FaaS. Another popular abstraction of processes are containers, with Docker being the most visible example of such a technology. We also see increasing popularity of container hosting systems, such as Mesos and Kubernetes, which abstract individual applications from OS-level deployment. And even further there are cloud-hosting container platforms like Amazon ECS and Google Container Engine which, like Serverless FaaS, let teams avoid having to manage their own server systems at all. So given all the momentum around containers is it still worth considering Serverless FaaS?

Principally the argument I made for PaaS still holds with containers - for Serverless FaaS **scaling** is automatically managed, transparent, and fine grained. Container platforms do not yet offer such a solution.

Furthermore I'd argue that container technology while having seen massive popularity in the last couple of years is still not mature. That's not to say that Serverless FaaS is mature either, but picking which rough edges you'd like is still the order of the day.

I'll admit, however, that both of these arguments may start to wear thin over time. While true no-management auto-scaling in container platforms isn't at the level of Serverless FaaS yet, we see areas like Kubernetes 'Horizontal Pod Autoscaling' as tending towards it. I can imagine some very smart traffic pattern analysis being introduced to such features, as well as more load-implying metrics. Furthermore the rapid evolution of Kubernetes may give a wonderfully simple, stable, platform before too long.

If we see the gap of management and scaling between Serverless FaaS and hosted containers narrow the choice between them may just come down to style, and type of application. For example it may be that FaaS is seen as a better choice for event driven style with few event types per application component, and containers are seen as a better choice for synchronous-request driven components with many entry points. I expect in 5 years time that many applications and teams will use both architectural approaches, and it will be fascinating to see patterns of such use emerge.

#NoOps

Serverless doesn't mean 'No Ops'. It *might* mean 'No internal Sys Admin' depending on how far down the serverless rabbit hole you go. There are 2 important things to consider here.

Firstly 'Ops' means a lot more than server administration. It also means at least monitoring, deployment, security, networking and often also means some amount of production debugging and system scaling. These problems all still exist with Serverless apps and you're still going to need a strategy to deal with them. In some ways Ops is *harder* in a Serverless world because a lot of this is so new.

Second even the Sys Admin is still happening - you're just outsourcing it with Serverless. That's not necessarily a bad thing - we outsource a lot. But depending on what precisely you're trying to do this might be a good or a bad thing, and either way at some point the abstraction will likely leak and you'll need to know that human sys admins somewhere are supporting your application.

Charity Majors gave a great talk on this subject at the recent Serverless Conference and I recommend checking it out once it's available online. Until then you can read her write-up here and here.

Stored Procedures as a Service

Another theme I've seen is that Serverless FaaS is 'Stored Procedures as a Service'. I think that's come from the fact that many examples of FaaS functions (including some I've used in this article) are small pieces of code that wrap access to a database. If that's *all* we could use FaaS for I think the name would be useful but because it is really just a subset of FaaS's capability then thinking of FaaS in such a way is an invalid constraint.

Camille Fournier
@skamille

I wonder if serverless services will become a thing like stored procedures, a good idea that quickly turns into massive technical debt

7:49 PM - 11 Apr 2016

272 236

That being said it *is* worth considering whether FaaS comes with some of the same problems of stored procedures, including the technical debt concern Camille mentions in the referenced tweet. There are many lessons that come from using stored procs that are worth reviewing in the context of FaaS and seeing whether they apply. Some of these are that stored procedures:

1. Often require vendor-specific language, or at least vendor-specific frameworks / extensions to a language
2. Are hard to test since they need to be executed in the context of a database
3. Are tricky to version control / treat as a first class application

Note that not all of these may apply to all implementations of stored procs, but they're certainly problems that I've come across in my time. Let's see if they might apply to FaaS:

(1) is definitely not a concern for the FaaS implementations I've seen so far, so we can scrub that one off the list right away.

For (2) since we're dealing with 'just code' unit testing is definitely just as easy as any other code. Integration testing is a different (and legitimate) question though which we'll discuss later.

For (3), again since FaaS functions are 'just code' version control is OK. But as to

application packaging there are no mature patterns on this yet. The Serverless framework which I mentioned earlier does provide its own form of this, and AWS announced at the recent Serverless Conference in May 2016 that they are working on something for packaging also ('Flourish'), but for now this is another legitimate concern.

Benefits

So far I've mostly tried to stick to just defining and explaining what Serverless architectures have come to mean. Now I'm going to discuss some of the benefits and drawbacks to such a way of designing and deploying applications.

It's important to note right off the bat that some of this technology is very new. AWS Lambda - a leading FaaS implementation - isn't even 2 years old at time of writing. As such some of the benefits we perceive may end up being just hype when we look back in another 2 years, on the other hand some of the drawbacks will hopefully be resolved.

Since this is an unproven concept at large scale you should definitely not take any decision to use Serverless without significant consideration. I hope this list of pros and cons helps you get to such a choice.

We're going to start off in the land of rainbows and unicorns and look at the benefits of Serverless.

Reduced operational cost

Serverless is at its most simple an outsourcing solution. It allows you to pay someone to manage servers, databases and even application logic that you might otherwise manage yourself. Since you're using a defined service that many other people will also be using we see an Economy of Scale effect - you pay less for your managed database because one vendor is running thousands of very similar databases.

The reduced costs appear to you as the total of two aspects - infrastructure costs and people (operations / development) costs. While some of the cost gains *may* come purely from sharing infrastructure (hardware, networking) with other users, the expectation is that most of all you'll need to spend less of your own time (and therefore reduced operations costs) on an outsourced serverless system than on an equivalent developed and hosted by yourself.

This benefit, however, isn't too different than what you'll get from Infrastructure as a Service (IaaS) or Platform as a Service (PaaS). But we can extend this benefit in 2 key ways, one for each of Serverless BaaS and FaaS.

BaaS - reduced development cost

IaaS and PaaS are based on the premise that server and operating system management can be commoditized. Serverless Backend as a Service on the other hand is a result of **entire application components** being commoditized.

Authentication is a good example. Many applications code their own authentication functionality which often includes features such as sign-up, login, password management, integration with other authentication providers, etc. On the whole this logic is very similar across most applications, and so services like Auth0 have been created to allow us to integrate ready-built authentication functionality into our application without us

having to develop it ourselves.

On the same thread are BaaS databases, like Firebase's database service. Some mobile applications teams have found it makes sense to have the client communicate directly with a server-side database. A BaaS database removes much of the database administration overhead, plus it will typically provide mechanisms to provide appropriate authorization for different types of users, in the patterns expected of a Serverless app.

Depending on your background you may squirm at both of these ideas (for reasons that we'll get into in the drawbacks section - don't worry!) but there is no denying the number of successful companies who have been able to produce compelling products with barely any of their own server-side code. Joe Emison gave a couple of examples of this at the recent Serverless Conference.

FaaS - scaling costs

One of the joys of serverless FaaS is, as I put it earlier in this article, that *'horizontal scaling is completely automatic, elastic, and managed by the provider'.* There are several benefits to this but on the basic infrastructural side the biggest benefit is that you **only pay for the compute that you need**, down to a 100ms boundary in the case of AWS Lambda. Depending on your traffic scale and shape this may be a huge economic win for you.

Example - occasional requests

For instance say you're running a server application that only processes 1 request every minute, that it takes 50 ms to process each request, and that your mean CPU usage over an hour is 0.1%. From one point of view this is wildly inefficient - if 1000 other applications could share your CPU you'd all be able to do your work on the same machine.

Serverless FaaS captures this inefficiency, handing the benefit to you in reduced cost. In this scenario you'd be paying for just 100ms of compute every minute, which is 0.15% of the time overall.

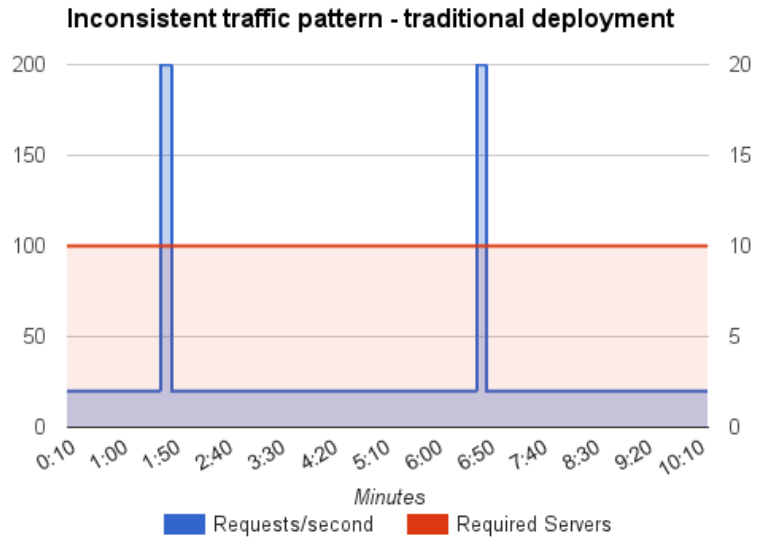
This has the following knock-on benefits:

- For would-be microservices that have very small load requirements it gives support to breaking down components by logic / domain even if the operational costs of such fine granularity might have been otherwise prohibitive.
- Such cost beneftis are a great democratizer. As companies or teams want to try out something new they have extremely small operational costs associated with ‘dipping their toe in the water’ when they use FaaS for their compute needs. In fact if your total workload is relatively small (but not entirely insignificant) you may not need to pay for any compute at all due to the 'free tier' provided by some FaaS vendors.

Example - inconsistent traffic

Let's look at another example. Say your traffic profile is very 'spikey' - perhaps your baseline traffic is 20 requests / second but that every 5 minutes you receive 200 requests / second (10 times the usual number) for 10 seconds. Let's also assume for the sake of example that your baseline performance maxes out your preferred server, and that you don't want to reduce your response time during the traffic spike phase. How do you solve for this?

In a traditional environment you may need to increase your total hardware capability by a factor of 10 to handle the spikes, even though they only account for less than 4% of total machine uptime. Auto-scaling is likely not a good option here due to how long new instances of servers will take to come up - by the time your new instances have booted the spike phase will be over.



With Serverless FaaS however this becomes a non-issue. You literally do nothing differently than if your traffic profile was uniform and only pay for the extra compute capacity during the spike phases.

Obviously I've deliberately picked examples here for which Serverless FaaS gives huge cost savings, but the point is to show that unless you have a very steady traffic shape that consistently uses a whole number's worth of server systems that you may save money using FaaS purely from a scaling viewpoint.

One caveat about the above - if your traffic **is** uniform **and** would consistently make good utilization of a running server you may not see this cost benefit and may actually spend more using FaaS. You should do some math with current provider costs vs the equivalents of running full-time servers to check to see whether costs are acceptable.

Optimization is the root of some cost savings

There is one more interesting aspect to FaaS costs - any performance optimizations you make to your code will not only increase the speed of your app but will also have a direct and immediate link to reduction in operational costs, subject to the granularity of your vendor’s charging scheme. For example if each of your operations currently take 1 second to run and you reduce that to 200ms you’ll immediately see 80% savings in compute costs without making any infrastructural changes.

Easier Operational Management

This section comes with a giant asterisk - some aspects of operations are still tough for Serverless, but for now we’re sticking with our new unicorn and rainbow friends…

On the Serverless BaaS side of the fence it’s fairly obvious why operational management is more simple than other architectures: less components that you support equals less

work.

On the FaaS side there are a number of aspects at play though and I'm going to dig into a couple of them.

Scaling benefits of FaaS beyond costs

While scaling is fresh in our minds from the previous section it's worth noting that not only does the scaling functionality of FaaS reduce compute cost it also reduces operational management because the scaling is automatic.

In the best case if your scaling process was manual, e.g. a human being needs to explicitly add and remove instances to an array of servers, with FaaS you can happily forget about that and let your FaaS vendor scale your application for you.

Even in the case that you've got to the point of using 'auto-scaling' in a non FaaS architecture then that still requires setup and maintenance - this work is no longer necessary with FaaS.

Similarly since scaling is performed by the provider on every request / event, you **no longer need to even think about the question of how many concurrent requests you can handle** before running out of memory or seeing too much of a performance hit, at least not within your FaaS hosted components. Downstream databases and non FaaS components will have to be reconsidered in light of a possibly signifcant increase in their load.

Reduced packaging and deployment complexity

While API gateways are not simple yet, the act of packaging and deploying a FaaS function is really pretty simple compared with deploying an entire server. All you're doing is compiling and zip'ing / jar'ing your code, and then uploading it. No puppet / chef, no start / stop shell scripts, no decisions about whether to deploy one or many containers on a machine. If you're just getting started you don't need to even package anything - you may be able to write your code right in the vendor console itself (this, obviously, is not recommended for production code!)

This doesn't take long to describe but in some teams this benefit may be absolutely huge - a fully Serverless solution requires **zero system administration**.

Platform-as-a-Service (PaaS) solutions have similar deployment benefits but as we saw earlier when comparing PaaS with FaaS the scaling advantages are unique to FaaS.

Time to market / experimentation

'Easier operational management' is a benefit which us as engineers understand, but what does that mean to our businesses?

The obvious case is cost: less time on operations = less people needed for operations. But by far the more important case in my mind is 'time to market'. As our teams and products become increasingly geared around lean and agile processes we want to continually try new things and rapidly update our existing systems. While simple re-deployment allows rapid iteration of stable projects, having a good *new-idea-to-initial-deployment* capability allows us to try new experiments with low friction and minimal cost.

The *new-idea-to-initial-deployment* story for FaaS is in some cases excellent, especially for simple functions triggered by a maturely-defined event in the vendor's ecosystem.

For instance say your organization is using AWS Kinesis, a Kafka-like messaging system, for broadcasting various types of real-time events through your infrastructure. With AWS Lambda you can develop and deploy a new production event listener against that Kinesis stream in minutes - you could try several different experiments all in one day!

For web-based APIs the same cannot quite yet be said in the bulk of cases but various open source projects and smaller scale implementations are leading the way. We'll discuss this further later.

'Greener' computing?

We've seen an explosion over the last couple of decades in the numbers and sizes of data centers in the world, and the associated energy usage that goes with them along with all the other physical resources required to build so many servers, network switches, etc. Apple, Google and the like talk about hosting some of their data centers near sources of renewable energy to reduce the fossil-fuel burning impact of such sites.

Part of the reason for this massive growth is the number of servers that are idle and yet powered up.

> *Typical servers in business and enterprise data centers deliver between 5 and 15 percent of their maximum computing output on average over the course of the year.*
>
> *-- Forbes*

That's extraordinarily inefficient and a huge environmental impact.

On one hand it's likely that cloud infrastructure has probably helped since companies can 'buy' more servers on-demand, rather than provision all possibly necessary servers a long-time in advance. However one could also argue that the ease of provisioning servers may have made the situation worse if a lot of those servers are getting left around without adequate capacity management.

Whether we use a self-hosted, IaaS or PaaS infrastructure solution we're still making capacity decisions about our applications that will often last months or years. Typically we are cautious, and rightly so, about managing capacity and over-provision, leading to the inefficiencies just described. With a Serverless approach we **no longer make such capacity decisions ourselves** - we let the Serverless vendor provision just enough compute capacity for our needs in real time. The vendor can then make their own capacity decisions in aggregate across their customers.

This difference should lead to far more efficient use of resources across data centers and therefore to a reduced environmental impact compared with traditional capacity management approaches.

Drawbacks

So, dear reader, I hope you enjoyed your time in the land of rainbows, unicorns and all things shiny, because it's about to get ugly as we get slapped around the face by the wet fish of reality.

There's a lot to like about Serverless architectures and I wouldn't have spent time writing about them if I didn't think there were a lot of promise in them, but they come with

significant trade-offs. Some of these are inherent with the concepts - they can't be entirely fixed by progress and are always going to need to be considered. Others are down to the current implementations and with time we could expect to see those resolved.

Inherent Drawbacks

Vendor control

With any outsourcing strategy you are giving up control of some of your system to a 3rd-party vendor. Such lack of control may manifest as system downtime, unexpected limits, cost changes, loss of functionality, forced API upgrades, and more. Charity Majors, who I referenced earlier, explains this problem in much more detail in the Tradeoffs section of this article:

> *[The Vendor service] if it is smart, will put strong constraints on how you are able to use it, so they are more likely to deliver on their reliability goals. When users have flexibility and options it creates chaos and unreliability. If the platform has to choose between your happiness vs thousands of other customers' happiness, they will choose the many over the one every time — as they should.*
>
> *-- Charity Majors*

Multitenancy Problems

Multitenancy refers to the situation where multiple running instances of software for several different customers (or tenants) are run on the same machine, and possibly within the same hosting application. It's a strategy to achieve the economy of scale benefits we mentioned earlier. Service vendors try their darndest to make it feel as a customer that we are the only people using their system and typically good service vendors do a great job at that. But no-one's perfect and sometimes multitenant solutions can have problems with security (one customer being able to see another's data), robustness (an error in one customer's software causing a failure in a different customer's software) and performance (a high load customer causing another to slow down.)

These problems are not unique to Serverless systems - they exist in many other service offerings that use multitenancy - but since many Serverless systems are new we may expect to see more problems of this type now than we will once these systems have matured.

Vendor lock-in

Here's the 3rd problem related to Serverless vendors - lock in. It's very likely that whatever Serverless features you're using from a vendor that they'll be differently implemented by another vendor. If you want to switch vendors you'll almost certainly need to update your operational tools (deployment, monitoring, etc.), you'll probably need to change your code (e.g. to satisfy a different FaaS interface), and you may even need to change your design or architecture if there are differences to how competing vendor implementations behave.

Even if you manage to be able to do this for one part of your ecosystem you may be locked in by another architectural component. For instance say you're using AWS Lambda to respond to events on an AWS Kinesis message bus. The differences between AWS Lambda, Google Cloud Functions and Microsoft Azure Functions may be

relatively small, but you're still not going to be able to directly hook up the latter 2 vendor implementations directly to your AWS Kinesis stream. This means that moving, or **porting, your code from one solution to another isn't going to be possible without also moving other chunks of your infrastructure too.**

And finally even if you figure out a way to reimplement your system with a different vendor's capabilities you're still going to have a migration process dependent on what your vendor offers you. For example if you're switching from 1 BaaS database to another do the export and import features of the original and target vendors do what you want? And even if they do, at what amount of cost and effort?

One possible mitigation to some of this could be an emerging general abstraction of multiple Serverless vendors, and we'll discuss that further later.

Security concerns

This really deserves an article in and of itself but embracing a Serverless approach opens you up to a large number of security questions. Two of these are as follows, but there are many others that you should consider.

- Each Serverless vendor that you use increases the number of different security implementations embraced by your ecosystem. This increases your surface area for malicious intent and the likelihood for a successful attack.
- If using a BaaS Database directly from your mobile platforms you are losing the protective barrier a server-side application provides in a traditional application. While this is not a dealbreaker it does require significant care in designing and developing your application.

Repetition of logic across client platforms

With a 'full BaaS' architecture no custom logic is written on the server-side - it's all in the client. This may be fine for your first client platform but as soon as you need your next platform you're going to need to repeat the implementation of a subset of that logic that you wouldn't have done in a more traditional architecture. For instance if using a BaaS database in this kind of system all your client apps (perhaps Web, native iOS and native Android) are now going to need to be able to communicate with your vendor database, and will need to understand how to map from your database schema to application logic.

Furthermore if you want to migrate to a new database at any point you're going to need to replicate that coding / coordination change across all your different clients too.

Loss of Server optimizations

Again with a 'full BaaS' architecture there is no opportunity to optimize your server-design for client performance. The 'Backend For Frontend' pattern exists to abstract certain underlying aspects of your whole system within the server, partly so that the client can perform operations more quickly and use less battery power in the case of mobile applications. Such a pattern is not available for 'full BaaS'.

I've made it clear that both this and previous drawback exist for 'full BaaS' architectures where all custom logic is in the client and the only backend services are vendor supplied. A mitigation of both of these is actually to embrace FaaS or some other kind of lightweight server-side pattern to move certain logic to the server.

No in-server state for Serverless FaaS

After a couple of BaaS-specific drawbacks let's talk about FaaS for a moment. I said earlier:

> *FaaS functions have significant restrictions when it comes to local .. state. .. You should assume that for any given invocation of a function none of the in-process or host state that you create will be available to any subsequent invocation.*

I also said that the alternative to this was to follow factor number 6 of the 'Twelve Factor App' which is to embrace this very constraint:

> *Twelve-factor processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database.*
>
> *-- The Twelve-Factor App*

Heroku recommends this way of thinking but you can bend the rules when running on their PaaS. With FaaS there's no bending the rules.

So where does your state go with FaaS if you can't keep it in memory? The quote above refers to using a database and in many cases a fast NoSQL Database, out-of-process cache (e.g. Redis) or an external file store (e.g. S3) will be some of your options. But these are all a lot slower than in-memory or on-machine persistence. You'll need to consider whether your application is a good fit for this.

Another concern in this regard is in-memory caches. Many apps that are reading from a large data set stored externally will keep an in-memory cache of part of that data set. You may be reading from 'reference data' tables in a database and use something like Ehcache. Alternatively you may be reading from an http service that specifies cache headers, in which case your in-memory http client can provide a local cache. With a FaaS implementation you can have this code in your app but your cache is rarely, if ever, going to be of much benefit. As soon as your cache is 'warmed up' on the first usage it is likely to be thrown away as the FaaS instance is torn down.

A mitigation to this is to no longer assume in-process cache, and to use a low-latency external cache like Redis or Memcached, but this (a) requires extra work and (b) may be prohibitively slow depending on your use case.

Implementation Drawbacks

The previously described drawbacks are likely always going to exist with Serverless. We'll see improvements in mitigating solutions, but they're always going to be there.

The remaining drawbacks, however, are down purely to the current state of the art. With inclination and investment on the part of vendors and/or a heroic community these can all be wiped out. But for right now there are some doozies...

Configuration

AWS Lambda functions offer no configuration. None. Not even an environment variable. How do you have the same deployment artifact run with different characteristics according to the specific nature of the environment? You can't. You have to redefine the deployment artifact, perhaps with a different embedded config file. This is an ugly hack. The Serverless framework can abstract this hack for you, but it's still a hack.

I have reason to believe that Amazon are fixing this (and probably pretty soon) and I don't know whether other vendors have the same problem, but I mention it right at the

top as an example of why a lot of this stuff is on the bleeding edge right now.

DoS yourself

Here's another fun example of why Caveat Emptor is a key phrase whenever you're dealing with FaaS at the moment. AWS Lambda, for now, limits you to how many concurrent executions you can be running of all your lambdas. Say that this limit is 1000, that means that at any one time you are allowed to be executing 1000 functions. If something causes you to need to go above that you may start getting exceptions, queueing, and/or general slow down.

The problem here is that this limit is across your whole AWS account. Some organizations use the same AWS account for both production and testing. That means if someone, somewhere, in your organization does a new type of load test and starts trying to execute 1000 concurrent Lambda functions you'll accidentally DoS your production applications. Oops.

Even if you use different AWS accounts for production and development one overloaded production lambda (e.g. processing a batch upload from a customer) could cause your separate real-time lambda-backed production API to become unresponsive.

Other types of AWS resources can be separated by context of environment and application area through various security and firewalling concepts. Lambda needs the same thing, and I've no doubt it will before too long. But for now, again, be careful.

Execution Duration

Earlier on in the article I mentioned that AWS Lambda functions are aborted if they run for longer than 5 minutes. That's a limitation which I would expect could be removed later, but it will interesting to see how AWS approach that.

Startup Latency

Another concern I mentioned before was how long it may take a FaaS function to respond, which is especially a concern of occasionally used JVM-implemented functions on AWS. If you have such a Lambda function it may take in the order of 10s of seconds to startup.

I expect AWS will implement various mitigations to improve this over time, but for now it may be a deal-breaker for using JVM Lambdas under certain use cases.

OK, that's enough picking on AWS Lambda specifically. I'm sure the other vendors also have some pretty ugly skeletons barely in their closets.

Testing

Unit testing Serverless Apps is fairly simple for reasons I've talked about earlier - any code that you write is 'just code' and there aren't for the most part a whole bunch of custom libraries you have to use or interfaces that you have to implement.

Integration testing Serverless Apps on the other hand is hard. In the BaaS world you're deliberately relying on externally provided systems rather than (for instance) your own database. So should your integration tests use the external systems too? If yes how amenable are those systems to testing scenarios? Can you easily tear-up / tear-down state? Can your vendor give you a different billing strategy for load testing?

If you want to stub those external systems for integration testing does the vendor provide

a local stub simulation? If so how good is the fidelity of the stub? If the vendor doesn't supply a stub how will you implement one yourself?

The same kinds of problems exist in FaaS-land. At present most of the vendors do not provide a local implementation that you can use so you're forced to use the regular production implementation. But this means deploying remotely and testing using remote systems for all your integration / acceptance tests. Even worse the kinds of problems I just described (no configuration, cross-account execution limits) are going to have an impact on how you do testing.

Part of the reason that this is a big deal is that our units of integration with Serverless FaaS (i.e. each function) are a lot smaller than with other architectures and therefore we rely on integration testing a lot more than we may do with other architectural styles.

Tim Wagner (general manager of AWS Lambda) made a brief reference at the recent Serverless Conference that they were tackling testing, but it sounded like it was going to rely heavily on testing in the cloud. This is probably just a brave new world, but I'll miss being able to fully test my system from my laptop, offline.

Deployment / packaging / versioning

This is a FaaS specific problem. Right now we're missing good patterns of bundling up a set of functions into an application. This is a problem for a few reasons:

- You may need to deploy a FaaS artifact separately for every function in your entire logical application. If (say) your application is implemented on the JVM and you have 20 FaaS functions that means deploying your JAR 20 times.
- It also means you can't atomically deploy a group of functions. You may need to turn off whatever event source is triggering the functions, deploy the whole group, and then turn the event source back on. This is a problem for zero-downtime applications.
- And finally it means there's no concept of versioned applications so atomic rollback isn't an option.

Again there are open source workarounds to help with some of this, however it can only be properly resolved with vendor support. AWS announced a new initiative named 'Flourish' to address some of these concerns at the recent Serverless Conference, but have released no significant details as of yet.

Discovery

Similarly to the configuration and packaging points there are no well-defined patterns for discovery across FaaS functions. While some of this is by no means FaaS specific the problem is exacerbated by the granular nature of FaaS functions and the lack of application / versioning definition.

Monitoring / Debugging

At present you are stuck on the monitoring and debugging side with whatever the vendor gives you. This may be fine in some cases but for AWS Lambda at least it is very basic. What we really need in this area are open APIs and the ability for third party services to help out.

API Gateway definition, and over-ambitious API Gateways

A recent ThoughtWorks Technology Radar discussed over-ambitious API Gateways.

While the link refers to API Gateways in general it can definitely apply to FaaS API Gateways specifically, as I mentioned earlier. The problem is that API Gateways offer the opportunity to perform much application specific-logic within their own configuration / definition domain. This logic is typically hard to test, version control, and even often times define. Far better is for such logic to remain in program code like the rest of the application.

With Amazon's API Gateway at present you are forced into using many Gateway-specific concepts and configuration areas even for the most simple of applications. This is partly why open source projects like the Serverless framework and Claudia.js exist, to abstract the developer from implementation-specific concepts and allow them to use regular code.

While it is likely that there will always be the opportunity to over-complicate your API gateway, in time we should expect to see tooling to avoid you having to do so and recommended patterns to steer you away from such pitfalls.

Deferring of operations

I mentioned earlier that Serverless is not 'No Ops' - there's still plenty to do from a monitoring, architectural scaling, security, networking, etc. point of view. But the fact that some people (ahem, possibly me, mea culpa) have described Serverless as 'No Ops' comes from the fact that it is so easy to ignore operations when you're getting started - "Look ma - no operating system!" The danger here is getting lulled into a false sense of security. Maybe you have your app up and running but it unexpectedly appears on Hacker News, and suddenly you have 10 times the amount of traffic to deal with and oops - you're accidentally DoS'ed and have no idea how to deal with it.

The fix here, like part of the API Gateway point above, is education. Teams using Serverless systems need to be considering operational activities early and it is on vendors and the community to provide the teaching to help them understand what this means.

The Future of Serverless

We're coming to the end of this journey into the world of Serverless architectures. To close out I'm going to discuss a few areas where I think the Serverless world may develop in the coming months and years.

Mitigating the Drawbacks

Serverless, as I've mentioned several times already, is new. And as such the previous section on Drawbacks was extensive and I didn't even cover everything I could have. The most important developments of Serverless are going to be to mitigate the inherent drawbacks and remove, or at least improve, the implementation drawbacks.

Tooling

The biggest problem in my mind with Serverless FaaS right now is tooling. Deployment / application bundling, configuration, monitoring / logging, and debugging all need serious work.

Amazon's announced but as-yet unspecified Flourish project could help with some of

these. Another positive part of the announcement was that it would be open source, allowing the opportunity for portability of applications across vendors. I expect that we'll see something like this evolve over the next year or two in the open source world, even if it isn't Flourish.

Monitoring, logging and debugging are all part of the vendor implementation and we'll see improvements across both BaaS and FaaS here. Logging on AWS Lambda at least is painful right now in comparison with traditional apps using ELK and the like. We are seeing the early days of a few 3rd party commercial and open source efforts in this area (e.g. IOPipe and lltrace-aws-sdk) but we're a long way away from something of the scope of New Relic. My hope is that apart from AWS providing a better logging solution for FaaS that they also make it very easy to plug into 3rd party logging services in much the same way that Heroku and others do.

API Gateway tooling also needs massive improvements, and again some of that may come from Flourish or advances in the Serverless Framework, and the like.

State Management

The lack of in-server state for FaaS is fine for a good number of applications but is going to be a deal breaker for many others. For example many microservice applications will use some amount of in-process cached state to improve latency. Similarly connection pools (either to databases or via persistent http connection to other services) are another form of state.

One workaround for high throughput applications will likely be for vendors to keep function instances alive for longer, and let regular in-process caching approaches do their job. This won't work 100% of the time since the cache won't be warm for every request, but this is the same concern that already exists for traditionally deployed apps using auto-scaling.

A better solution could be very low-latency access to out-of-process data, like being able to query a Redis database with very low network overhead. This doesn't seem too much of a stretch given the fact that Amazon already offer a hosted Redis solution in their Elasticache product, and that they already allow relative co-location of EC2 (server) instances using Placement Groups.

More likely though what I think we're going to see are different kinds of application architecture embraced to take account of the no-in-process-state constraint. For instance for low latency applications you may see a regular server handling an initial request, gathering all the context necessary to process that request from it's local and external state, then handing a fully-contextualized request off to a farm of FaaS functions that themselves don't need to look up data externally.

Platform Improvements

Certain drawbacks to Serverless FaaS right now are down to the way the platforms are implemented. Execution Duration, Startup Latency, non-separation of execution limits are 3 obvious ones. These will likely be either fixed by new solutions, or given workarounds with possible extra costs. For instance I could imagine that Startup Latency could be mitigated by allowing a customer to request 2 instances of a FaaS function are always available at low latency, with the customer paying for this availability.

We'll of course see platform improvements beyond just fixing current deficiencies, and these will be exciting to see.

Education

Many vendor-specific inherent drawbacks with Serverless are going to be mitigated through education. Everyone using such platforms needs to think actively about what it means to have so much of their ecosystems hosted by one or many application vendors. Questions that need to be considered are ones like 'do we want to consider parallel solutions from different vendors in case one becomes unavailable? How do applications gracefully degrade in the case of a partial outage?'

Another area for education is technical operations. Many teams these days have fewer 'Sys Admins' than they used to, and Serverless is going to accelerate this change. But Sys Admins do more than just configure Unix boxes and chef scripts - they're also often the people on the front line of support, networking, security and the like.

A true DevOps culture becomes *even more* important in a Serverless world since those other non Sys Admin activities still need to get done, and often times it's developers who'll be responsible for them. These activities may not come naturally to many developers and technical leads, so education and close collaboration with operations folk will be of utmost importance.

Increased transparency / clearer expectations from Vendors

And finally on the subject of mitigation Vendors are going to have to be even more clear in the expectations we can have of their platforms as we continue to rely on them for more of our hosting capabilities. While migrating platforms is hard, it's not impossible, and untrustworthy vendors will see their customers taking their business elsewhere.

The emergence of patterns

Apart from the rawness of the underlying platforms our understanding of how and when to use Serverless architectures is still very much in its infancy. Right now teams are throwing all kinds of ideas at a Serverless platform and seeing what sticks. Thank goodness for pioneers!

But at some point soon we're going to start seeing patterns of recommended practice emerge.

Some of these patterns will be in application architecture. For instance how big can FaaS functions get before they get unwieldy? Assuming we can atomically deploy a group of FaaS functions what are good ways of creating such groupings - do they map closely to how we'd currently clump logic into microservices or does the difference in architecture push us in a different direction?

Extending this further what are good ways of creating hybrid architectures between FaaS and traditional 'always on' persistent server components? What are good ways of introducing BaaS into an existing ecosystem? And, for the reverse, what are the warning signs that a fully- or mostly-BaaS system needs to start embracing or using more custom server-side code?

We'll also see more usage-patterns emerge. One of the standard examples for FaaS is media conversion: "whenever a large media file is stored to an S3 bucket then automatically run processes to create smaller versions in another bucket." But we need more usage-patterns to be catalogued in order to see if our particular use cases might be a good fit for a Serverless approach.

Beyond application architecture we'll start seeing recommended operational patterns

once tooling improves. How do we logically aggregate logging for a hybrid architecture of FaaS, BaaS and traditional servers? What are good ideas for discovery? How do we do canary releases for API-gateway fronted FaaS web applications? How do we most effectively debug FaaS functions?

Beyond 'FaaSification'

Most usages of FaaS that I've seen so far are mostly about taking existing code / design ideas and 'FaaSifying' them - converting them to a set of stateless functions. This is powerful, but I also expect that we'll start to see more abstractions and possibly languages using FaaS as an underlying implementation that give developers the benefits of FaaS without actually thinking about their application as a set of discrete functions.

As an example I don't know whether Google use a FaaS implementation for their Dataflow product, but I could imagine someone creating a product or open source project that did something similar, and used FaaS as an implementation. A comparison here is something like Apache Spark . Spark is a tool for large-scale data processing offering very high level abstractions which can use Amazon EMR / Hadoop as its underlying platform.

Testing

As I discussed in the 'Drawbacks' section there is a lot of work to do in the area of integration and acceptance testing for Serverless systems. We'll see vendors come out with their suggestions, which will likely be cloud based, and then I suspect we'll see alternatives from old crusties like me who want to be able to test everything from their development machine. I suspect we'll end up seeing decent solutions for both on- and offline testing, but it could take a few years.

Portable implementations

At present all popular Serverless implementations assume deployment to a 3rd-party Vendor system in the cloud. That's one of the benefits to Serverless - less technology for us to maintain. But there's a problem here if a company wants to take such technology and run it on their own systems and offer it as an **internal service**.

Similarly all the implementations have their own specific flavor of integration points - deployment, configuration, function interface, etc. This leads to the **vendor lock-in** drawbacks I mentioned earlier.

I expect that we'll see various **portable implementations** created to mitigate these concerns. I'm going to discuss the 2nd one first.

Abstractions over Vendor implementations

We're already starting to see something like this in open source projects like the Serverless Framework and the Lambada Framework. The idea here is that it would be nice to be able to code and operate our Serverless apps with a development-stage neutrality of where and how they are deployed. It would be great to be able to easily switch, even right now, between AWS API Gateway + Lambda, and Auth0 webtask, depending on the operational capabilities of each of the platforms.

I don't expect such an abstraction to be complete until we see much more of a commoditization of products, but the good thing is that this idea can be implemented incrementally. We may start with something like a cross-vendor deployment tool - which

may even be AWS's Flourish project that I mentioned earlier - and build in more features from there.

One tricky part to this will be modeling abstracted FaaS coding interfaces without some idea of standardization, but I'd expect that progress could be made on non proprietary FaaS triggers first. For instance I expect we'll see an abstraction of web-request and scheduled ('cron') Lambdas before we start seeing things like abstraction of AWS S3 or Kinesis Lambdas.

Deployable implementations

It may sound odd to suggest that we use Serverless techniques without using 3rd-party providers, but consider these thoughts:

- Maybe we're a large technical organization and we want to start offering a Firebase-like database experience to all of our mobile application development teams, but we want to use our existing database architecture as the back end.
- We'd like to use FaaS style architecture for some of our projects, but for compliance / legal / etc. reasons we need to run our applications 'on premise'.

In either of these cases there are still many benefits of using a Serverless approach without those that come from vendor-hosting. There's a precedent here - consider Platform-as-a-Service (PaaS). The initial popular PaaS's were all cloud based (e.g. Heroku), but fairly quickly people saw the benefits of running a PaaS environment on their own systems - a so-called 'Private PaaS' (e.g. Cloud Foundry).

I can imagine, like private PaaS implementations, seeing both open source and commercial implementations of both BaaS and FaaS concepts becoming popular. Galactic Fog is an example of an early-days open source project in this area, which includes its own FaaS implementation. Similarly to my point above about vendor abstractions we may see an incremental approach. For example the Kong project is an open source API Gateway implementation. At the time of writing it doesn't currently integrate with AWS Lambda (although this is being worked on), but if it did this would be an interesting hybrid approach.

Community

I fully expect to see a huge growth of the Serverless community. Right now there's been one conference and there are a handful of meetups around the world. I expect that we'll start seeing something more like the massive communities that exist around Docker and Spring - many conferences, many communities, and more online fora than you can possibly keep track of.

Conclusion

Serverless, despite the confusing name, is a style of architecture where we rely to a smaller extent than usual on running our own server side systems as part of our applications. We do this through two techniques - Backend as a Service (BaaS), where we tightly integrate third party remote application services directly into the front-end of our apps, and Functions as a Service (FaaS), which moves server side code from long running components to ephemeral function instances.

Serverless is very unlikely to be the correct approach for every problem, so be wary of anyone who says it will replace all of our existing architectures. And be even more careful if you take the plunge into Serverless systems now, especially in the FaaS realm. While there are riches (of scaling and saved deployment effort) to be plundered, there also be dragons (of debugging, monitoring) lurking right around the next corner.

Those benefits shouldn't be quickly dismissed however since there are significant positive aspects to Serverless Architecture, including reduced operational and development costs, easier operational management, and reduced environmental impact. The most important benefit to me though is the reduced feedback loop of creating new application components - I'm a huge fan of 'lean' approaches, largely because I think there is a lot of value in getting technology in front of an end user as soon as possible to get early feedback, and the reduced time-to-market that comes with Serverless fits right in with this philosophy.

Serverless systems are still in their infancy. There will be many advances in the field over the coming years and it will be fascinating to see how they fit into our architectural toolkit.

Acknowledgements

Thanks to the following for their input into this article: Obie Fernandez, Martin Fowler, Paul Hammant, Badri Janakiraman, Kief Morris, Nat Pryce, Ben Rady, Carlos Nunez, John Chapin, Robert Bagge, Karel Sague Alfonso, Premanand Chandrasekaran, Augusto Marietti, Roberto Sarrionandia

Thanks to Badri Janakiraman and Ant Stanley who provided input for the sidebar on origins of the term.

Thanks to members of my former team at Intent Media for tackling this new technology with appropriately sceptical enthusiasm: John Chapin, Pete Gieser, Sebastián Rojas and Philippe René.

Finally thanks to all the people who've put thoughts out there already on this subject, especially those whose content I link to.

Significant Revisions

04 August 2016: Added "Future" and "Conclusion"

25 July 2016: Added origins sidebar and section "Comparison with containers"

18 July 2016: Added "Drawbacks"

13 July 2016: Added "Benefits"

17 June 2016: Added "What isn't Serverless'"

16 June 2016: Added "Unpacking 'Function as a Service'"

15 June 2016: Published first installment - A couple of examples

ThoughtWorks®