We've mentioned FaaS a lot already, but it's time to dig into what it really means. To do this let's look at the [opening description](https://aws.amazon.com/lambda/) for Amazon's FaaS product: Lambda. I've added some tokens to it, which I’ll expand on.

*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 backend code without managing your own server systems or your own long-lived server applications.** That second clause—long-lived 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, FaaS replaces 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.

1. 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, Go, any JVM language (Java, Clojure, Scala, etc.), or any .NET language. However your Lambda function can also 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 in this article).

FaaS functions have significant architectural restrictions though, especially when it comes to state and execution duration. We’ll get 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. The rest of the code (e.g., the code that writes to the database) is no different in a FaaS world.

1. Deployment is very different from traditional systems since we have no server applications to run ourselves. In a FaaS environment we upload the code for our function to the FaaS provider, and the provider does everything else necessary for provisioning resources, instantiating VMs, managing processes, etc.
2. 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 creating and destroying them purely driven by runtime need. Most importantly, with FaaS **the vendor handles all underlying resource provisioning and allocation**—no cluster or VM management is required by the user at all.

Let’s return to our click processor. Say that we were having a good day and customers were clicking on ten times as many ads as usual. For the traditional architecture, would our click-processing application be able to handle this? For example, did we develop our application to be able to handle multiple messages at a time? 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 autoscaling automatic or do we need to reconfigure that manually? With a FaaS approach all of these questions are already answered—you need to write the function ahead of time to assume horizontal-scaled parallelism, but from that point on the FaaS provider automatically handles all scaling needs.

1. Functions in FaaS are typically triggered by event types defined by the provider. With Amazon AWS such stimuli include S3 (file/object) updates, time (scheduled tasks), and messages added to a message bus (e.g., [Kinesis](https://aws.amazon.com/kinesis/)).
2. Most providers also allow functions to be triggered as a response to inbound HTTP requests; in AWS one typically enables this by way of using an API gateway. We used an API gateway in our Pet Store example for our “search” and “purchase” functions. Functions can also be invoked directly via a platform-provided API, either externally or from within the same cloud environment, but this is a comparatively uncommon use.

**State**

FaaS functions have significant restrictions when it comes to local (machine/instance-bound) state—i.e., data that you store in variables in memory, or data that you write to local disk. You do have such storage available, but you have no guarantee that such state is persisted across multiple invocations, and, more strongly, you should not assume that state from one invocation of a function will be available to another invocation of the same function. FaaS functions are therefore often described as stateless, but it’s more accurate to say that any state of a FaaS function that is required to be **persistent** needs to be **externalized** outside of the FaaS function instance.

For FaaS functions that are naturally stateless—i.e., those that provide a purely functional transformation of their input to their output—this is of no concern. But for others this can have a large impact on application architecture, albeit not a unique one—the “[Twelve-Factor app](http://12factor.net/)” concept has [precisely the same restriction](http://12factor.net/processes). Such state-oriented functions will typically make use of a database, a cross-application cache (like Redis), or network file/object store (like S3) to store state across requests, or to provide further input necessary to handle a request.

**Execution duration**

FaaS functions are typically limited in how long each invocation is allowed to run. At present the “timeout” for an AWS Lambda function to respond to an event is at most five minutes, before being terminated. Microsoft Azure and Google Cloud Functions have similar limits.

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

**Startup latency and “cold starts”**

It takes some time for a FaaS platform to initialize an instance of a function before each event. This startup latency can vary significantly, even for one specific function, depending on a large number of factors, and may range anywhere from a few milliseconds to several seconds. That sounds bad, but let’s get a little more specific, using AWS Lambda as an example.

Initialization of a Lambda function will either be a “warm start”—reusing an instance of a Lambda function and its host container from a previous event—or a “cold start” —creating a new container instance, starting the function host process, etc. Unsurprisingly, when considering startup latency, it’s these cold starts that bring the most concern.

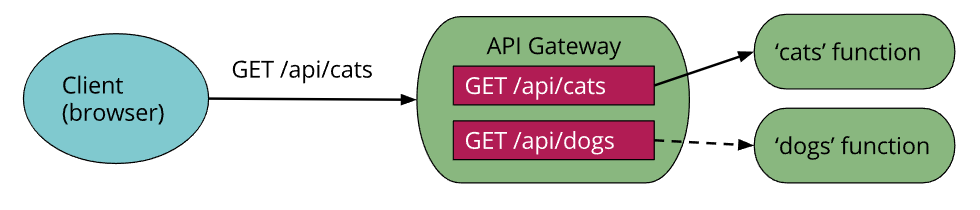
Cold-start latency depends on many variables: the language you use, how many libraries you’re using, how much code you have, the configuration of the Lambda function environment itself, whether you need to connect to [VPC](https://aws.amazon.com/vpc/) resources, etc. Many of these aspects are under a developer’s control, so it’s often possible to reduce the startup latency incurred as part of a cold start.

Equally as variable as cold-start duration is cold-start frequency. For instance, if a function is processing 10 events per second, with each event taking 50 ms to process, you’ll likely only see a cold start with Lambda every 100,000–200,000 events or so. If, on the other hand, you process an event once per hour, you’ll likely see a cold start for every event, since Amazon retires inactive Lambda instances after a few minutes. Knowing this will help you understand whether cold starts will impact you on aggregate, and whether you might want to perform “keep alives” of your function instances to avoid them being put out to pasture.

Are cold starts a concern? It depends on the style and traffic shape of your application. My former team at Intent Media has an asynchronous message-processing Lambda app implemented in Java (typically the language with the slowest startup time) which processes hundreds of millions of messages per day, and they have no concerns with startup latency for this component. That said, if you were writing a low-latency trading application you probably wouldn’t want to use cloud-hosted 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 performance with production-like load. If your use case doesn’t work now you may want to try again in a few months, since this is a major area of continual improvement by FaaS vendors.

#### API gateways



One aspect of Serverless that we brushed upon earlier is an “API gateway.” An API gateway is an HTTP server where routes and endpoints are defined in configuration, and each route is associated with a resource to handle that route. In a Serverless architecture such handlers are often FaaS functions.

When an API gateway receives a request, it finds the routing configuration matching the request, and, in the case of a FaaS-backed route, will call the relevant FaaS function with a representation of the original request. Typically the API gateway will allow mapping from HTTP request parameters to a more concise input for the FaaS function, or will allow the entire HTTP request to be passed through, typically as a JSON object. The FaaS function will execute its logic and return a result to the API gateway, which in turn will transform this result into an HTTP response that it passes back to the original caller.

Amazon Web Services have their own API gateway (slightly confusingly named “[API Gateway](https://aws.amazon.com/api-gateway/)”), and other vendors offer similar abilities. Amazon’s API Gateway is a BaaS (yes, BaaS!) service in its own right in that it’s an external service that you configure, but do not need to run or provision yourself.

Beyond purely routing requests, API gateways may also perform authentication, input validation, response code mapping, and more. (If your spidey senses are tingling as you consider whether this is actually such a good idea, hold that thought! We'll consider this further later.)

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

**Open source**

So far I’ve mostly discussed proprietary vendor products and tools. The majority of Serverless applications make use of such services, but there are open-source projects in this world, too.

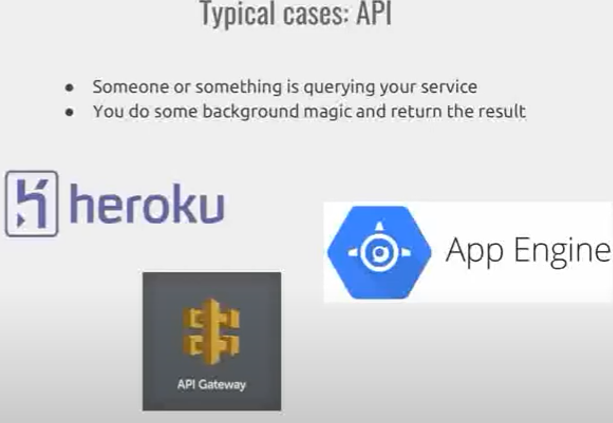
The most common uses of open source in Serverless are for FaaS tools and frameworks, especially the popular [Serverless Framework](https://github.com/serverless/serverless), which aims to make working with AWS API Gateway and Lambda easier than using the tools provided by AWS. It also provides an amount of cross-vendor tooling abstraction, which some users find valuable. Examples of similar tools include [Claudia](https://github.com/claudiajs/claudia) and [Zappa](https://github.com/Miserlou/Zappa). Another example is [Apex](https://github.com/apex/apex), which is particularly interesting since it allows you to develop Lambda functions in languages other than those directly supported by Amazon.

The big vendors themselves aren’t getting left behind in the open-source tool party though. AWS’s own deployment tool, SAM—the [Serverless Application Model](https://docs.aws.amazon.com/lambda/latest/dg/serverless_app.html)—is [also open source](https://github.com/awslabs/serverless-application-model).

One of the main benefits of proprietary FaaS is not having to be concerned about the underlying compute infrastructure (machines, VMs, even containers). But what if you *want* to be concerned about such things? Perhaps you have some security needs that can’t be satisfied by a cloud vendor, or maybe you have a few racks of servers that you’ve already bought and don’t want to throw away. Can open source help in these scenarios, allowing you to run your own “Serverful” FaaS platform?

Yes, and there’s been a good amount of activity in this area. One of the initial leaders in open-source FaaS was IBM (with [OpenWhisk](https://openwhisk.apache.org/), now an Apache project) and surprisingly—to me at least!—Microsoft, which open sourced much of its [Azure Functions](https://azure.microsoft.com/en-us/services/functions/) platform. Many other self-hosted FaaS implementations make use of an underlying container platform, frequently Kubernetes, which makes a lot of sense for many reasons. In this arena it’s worth exploring projects like [Galactic Fog](http://www.galacticfog.com/), [Fission](https://fission.io/), and [OpenFaaS](https://github.com/openfaas/faas). This is a large, fast-moving world, and I recommend looking at the work that the Cloud Native Computing Federation (CNCF) [Serverless Working Group](https://github.com/cncf/wg-serverless) have done to track it.

**Use Cases**











 Авторизация, поиск, …