Serverless Architectures

https://martinfowler.com/articles/serverless.html

Serverless

- **Serverless computing**, or more simply *Serverless*, is a hot topic in the software architecture world.
- The “Big Three” cloud vendors—Amazon, Google, and Microsoft—are heavily invested in Serverless

Serverless Architectures

- *Serverless architectures are application designs that incorporate third-party "Backend as a Service" (BaaS) services, and/or that include custom code run in managed, ephemeral containers on a "Functions as a Service" (FaaS) platform.*
- *By using these ideas, and related ones like single-page applications, such architectures remove much of the need for a traditional always-on server component.*
- *Serverless architectures may benefit from significantly reduced operational cost, complexity, and engineering lead time, at a cost of increased reliance on vendor dependencies and comparatively immature supporting services.*

What is Serverless?

- there's no one clear view of what Serverless is.
- For starters, it encompasses two different but overlapping areas:

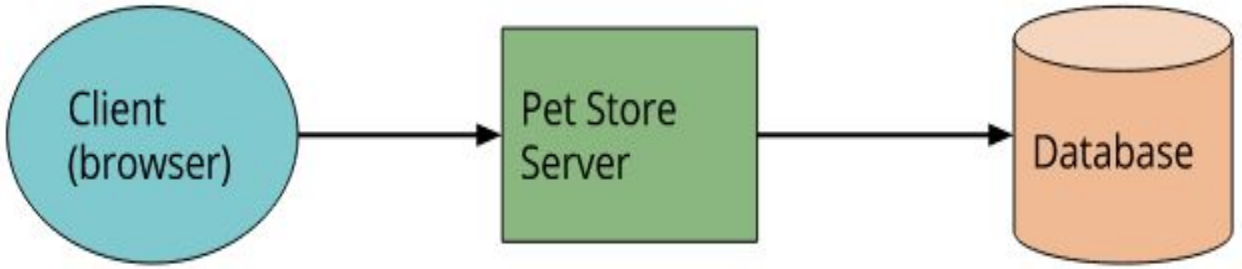
1. Serverless was first used to describe applications that significantly or fully incorporate third-party, cloud-hosted applications and services, to manage server-side logic and state.
   - "rich client" applications—think single-page web apps, or mobile apps—that use the vast ecosystem of cloud-accessible databases (e.g., Firebase), authentication services (e.g., Auth0, AWS Cognito)
   - These types of services have been previously described as "(Mobile) Backend as a Service", and I use "BaaS"

What is Serverless?

2. Serverless can also mean applications where server-side logic is still written by the application developer, but, unlike traditional architectures, it's run in stateless compute containers that are event-triggered, ephemeral (may only last for one invocation), and fully managed by a third party.
   - One way to think of this is "Functions as a Service" or **"FaaS"**.
   - AWS Lambda is one of the most popular implementations of a Functions-as-a-Service platform at present. Besides Azure Functions, Google Cloud Functions etc.

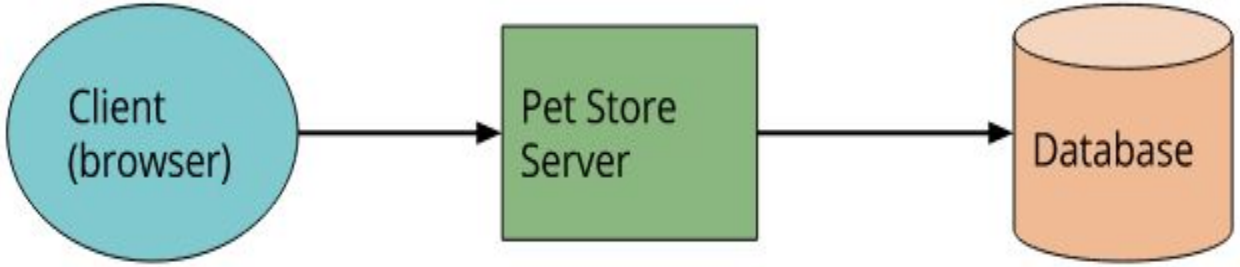
Example

- **UI-driven applications**
- Let's think about a traditional three-tier client-oriented system with server-side logic. A good example is a typical ecommerce- (pet store).
- Traditionally, the architecture will look something like the diagram below. Let's say it's implemented in Java or Javascript on the server side, with an HTML + Javascript component as the client:



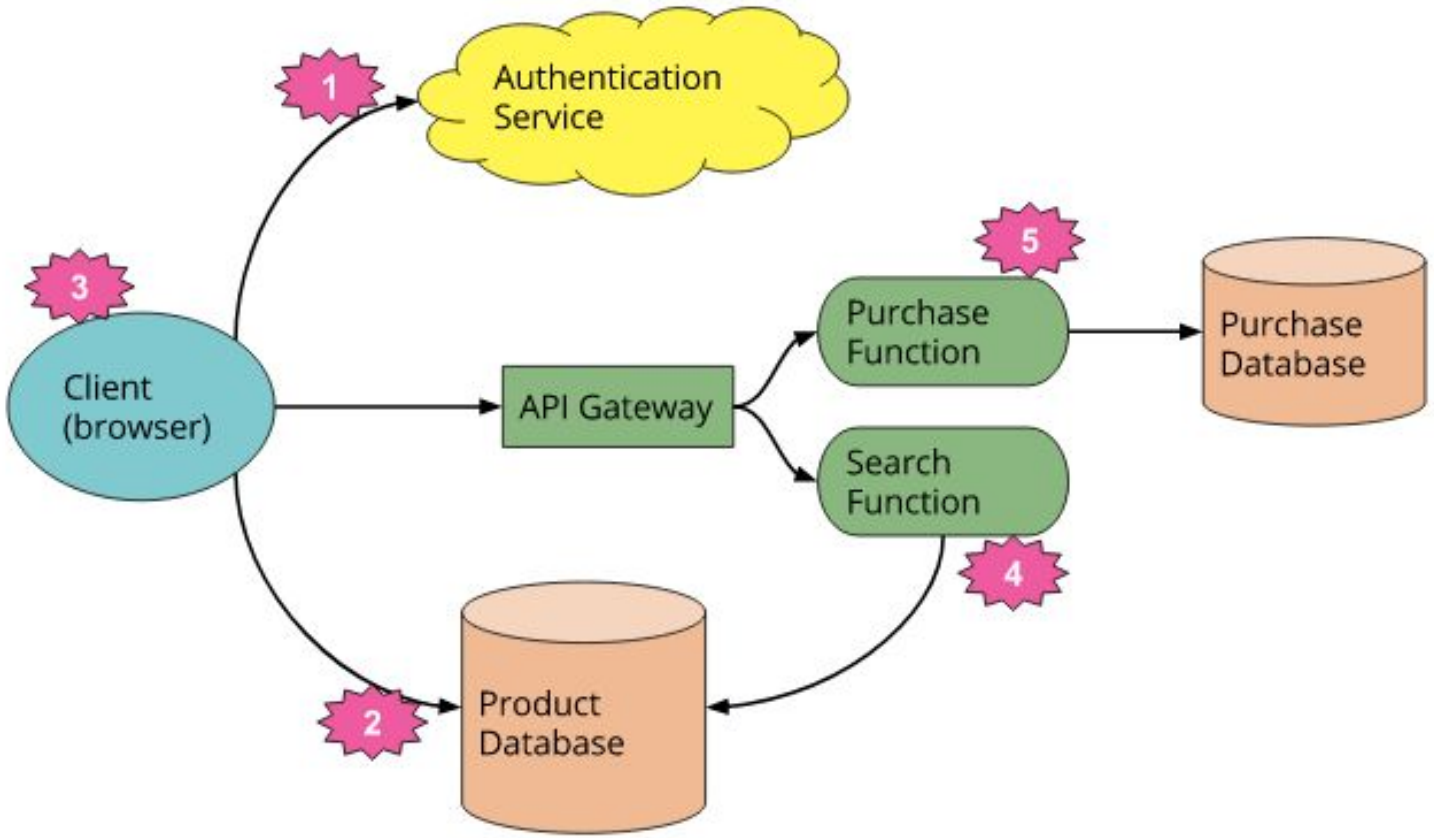
Example

- 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.



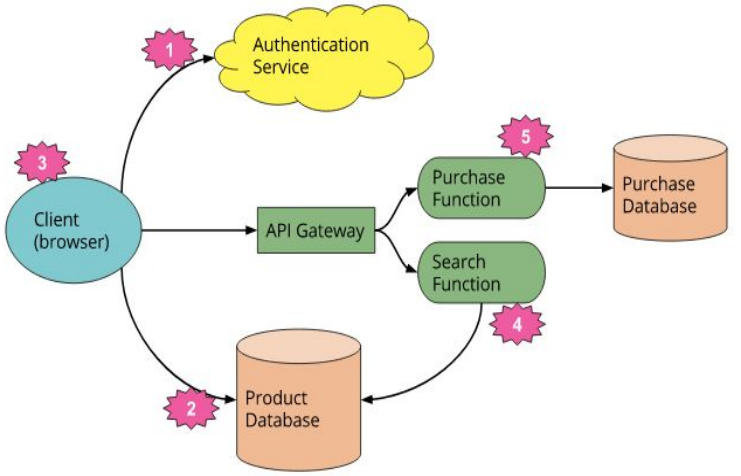
Example

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

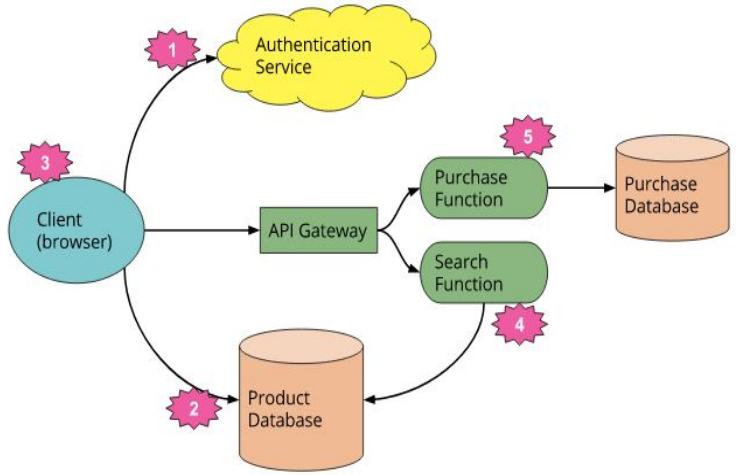


Example

- here we see a number of significant changes:
  a. We've deleted the authentication logic in the original application and have replaced it with a third-party BaaS service (e.g., Auth0.)
  b. 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 hosted by a third party (e.g., Google Firebase.). We likely have a different security profile for the client accessing the database in this way than for server resources that access the database.



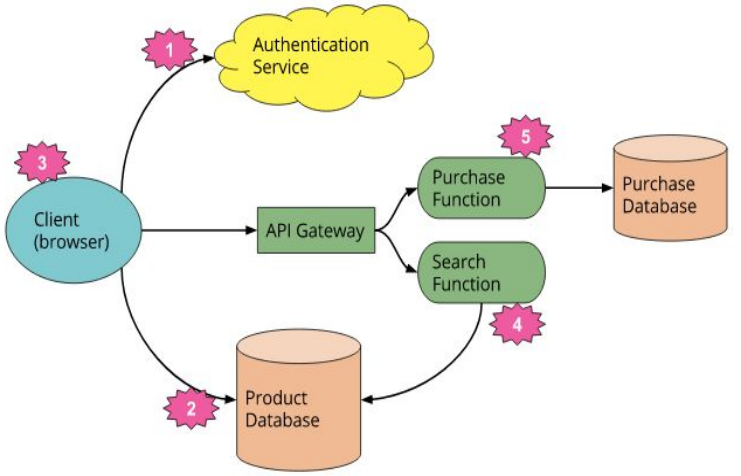
Example



- here we see a number of significant changes:
  - 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, reading from a database and translating that into a usable view, etc. The client is well on its way to becoming a Single Page Application.
    - We may want to keep some UX-related functionality in the server, if, for example, it's compute intensive or requires access to significant amounts of data. In our pet store, an example is "search." Instead of having an always-running server, as existed in the original architecture, we can instead implement a FaaS function that responds to HTTP requests via an API gateway (described later). Both the client and the server "search" function read from the same database for product data.
    - If we choose to use AWS Lambda as our FaaS platform we can port the search code from the original Pet Store server to the new Pet Store Search function without a complete rewrite, since Lambda supports Java and Javascript—our original implementation languages

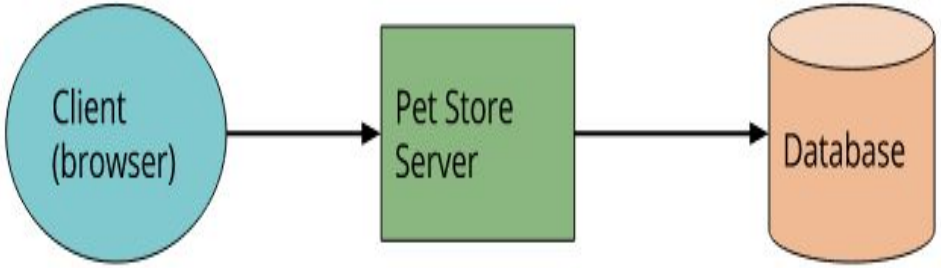
Example

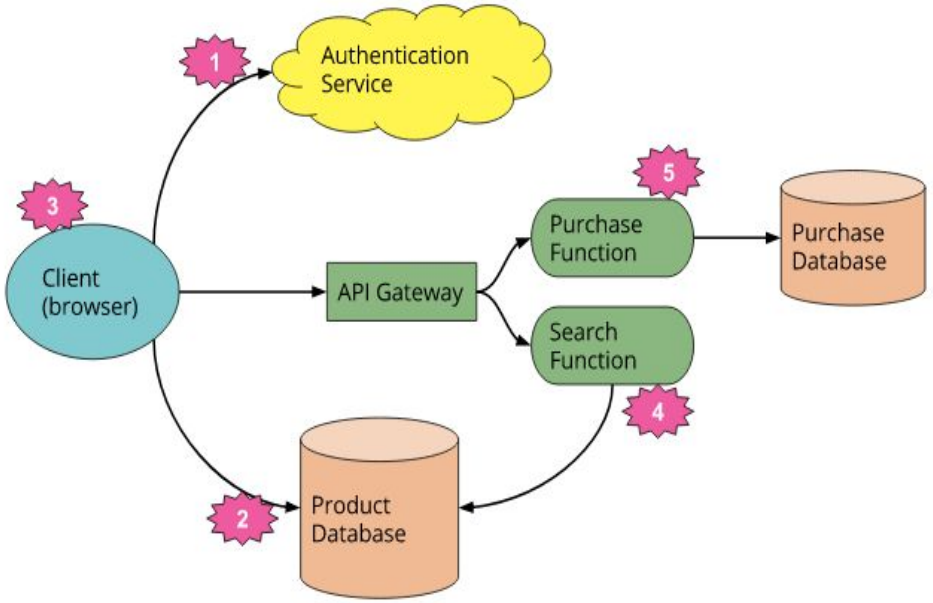
- here we see a number of significant changes:
  - Finally, we may replace our “purchase” functionality with another separate FaaS function, choosing to keep it on the server side for security reasons, rather than reimplement it in the client. It too is fronted by an API gateway. Breaking up different logical requirements into separately deployed components is a very common approach when using FaaS.



Example

- In the original version, all flow, control, and security was managed by the central server application. In the Serverless version there is no central arbiter of these concerns. Instead we see a preference for **choreography over orchestration**, with each component playing a more architecturally aware role—an idea also common in a microservices approach.





Example

- Benefits:
  - more flexible and amenable to change.
  - both as a whole and through independent updates to components;
  - there is better division of concerns

- trade-off:
  - it requires better distributed monitoring, and
  - we rely more significantly on the security capabilities of the underlying platform.
  - More fundamentally, there are a greater number of moving pieces to get our heads around than there are with the monolithic application we had originally.
- Whether the benefits of flexibility and cost are worth the added complexity of multiple backend components is very context dependent

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, and, secondarily, it needs to capture all the different types of user activity that are occurring, for subsequent processing.
- Think about an online advertisement system: when a user clicks on an ad you want to very quickly redirect them to the target of that ad. At the same time, you need to collect the fact that the click has happened so that you can charge the advertiser.

Message-driven applications

- The “Ad Server” synchronously responds to the user (not shown) and also posts a “click message” to a channel. This message is then 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 as follows:

Message-driven applications

- Can you see the difference?

- asynchronous message processing is a very popular use case for Serverless technologies.
- We've replaced a long-lived message-consumer *application* with a FaaS *function*.
- This function runs within the event-driven context the vendor provides.
- The FaaS environment may also process several messages in parallel by instantiating multiple copies of the function code.
- Note that the cloud platform vendor supplies both the message broker *and* the FaaS environment—the two systems are closely tied to each other.

Unpacking "Function as a Service"

- 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)**.

Unpacking "Function as a Service"

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.

Unpacking "Function as a Service"

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, 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.

Unpacking "Function as a Service"

3. 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.
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 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.

Unpacking "Function as a Service"

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.

Unpacking "Function as a Service"

5. 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).
6. 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.

Unpacking "Function as a Service"

- **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 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.
- **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.

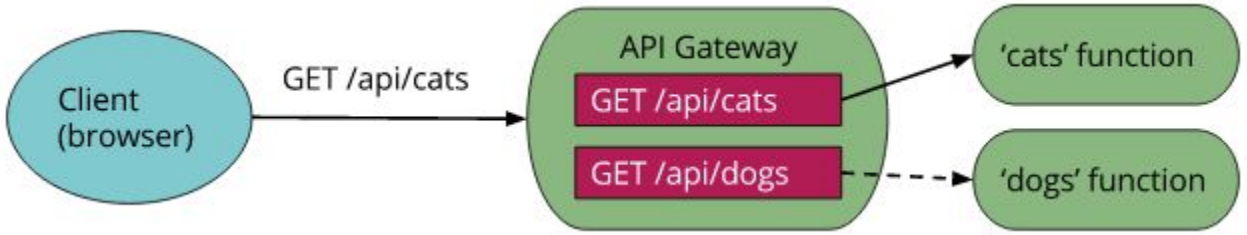
Unpacking "Function as a Service"

- **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.
- 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
- “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.

Unpacking "Function as a Service"

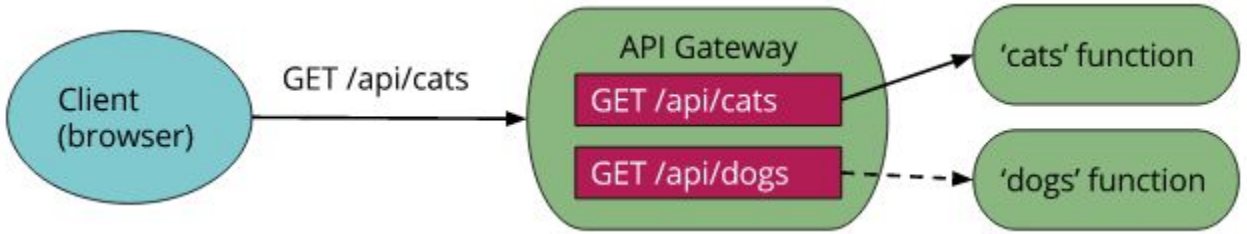
- Cold-start latency depends on many variables: the language you use, how many libraries you're using, how much code you have, the configuration of the Lambda function environment itself, whether you need to connect to other resources, etc
  - Many of these aspects are under a developer's control, so it's often possible to reduce the startup latency incurred as part of a cold start.

Unpacking "Function as a Service"



- **API gateways**
  - 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. in the case of a FaaS-backed route, will call the relevant FaaS function with a representation of the original request.
- 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'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

Unpacking "Function as a Service"



- API gateways may also perform authentication, input validation, response code mapping, and more.
- 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.

Benefits

- **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.
  - The reduced costs appear to you as the total of two aspects.
    - The first are infrastructure cost gains that come purely from sharing infrastructure (e.g., hardware, networking) with other people
    - The second are labor cost gains: you'll be able to spend less of your own time on an outsourced Serverless system than on an equivalent developed and hosted by yourself

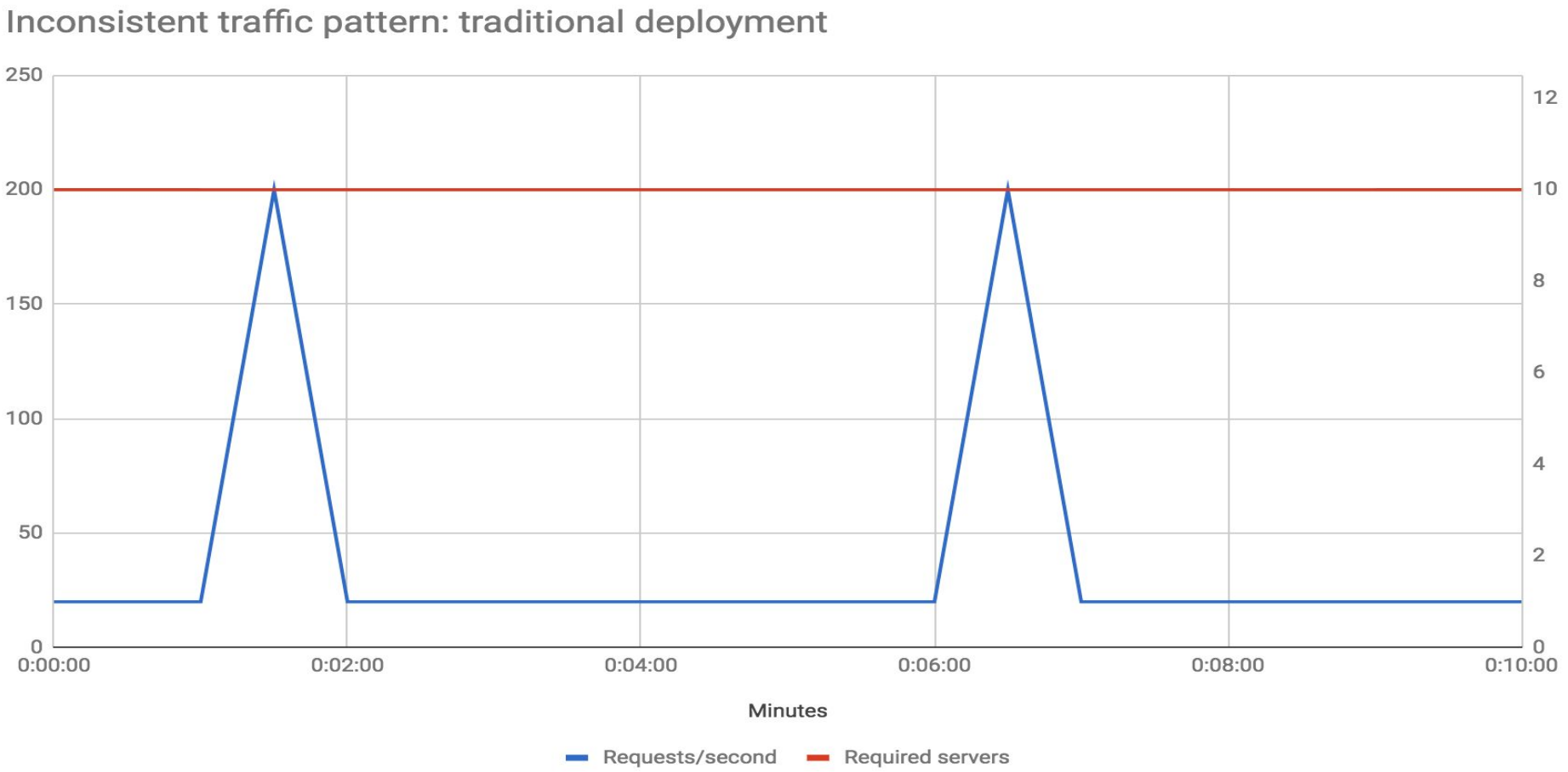
Benefits

- **BaaS: reduced development cost**
  - A BaaS database removes much of the database administration overhead, and typically provides mechanisms to perform appropriate authorization for different types of users, in the patterns expected of a Serverless app.

- **FaaS: scaling costs**
  - “horizontal scaling is completely automatic, elastic, and managed by the provider.”
  - Depending on your traffic scale and shape, this can be a huge economic win for you.

Benefits

Example: inconsistent traffic



Benefits

Optimization is the root of some cost savings

- any performance optimizations you make to your code will not only increase the speed of your app, but they'll have a direct and immediate link to reduction in operational costs.
- For example, say an application initially takes one second to process an event. If, through code optimization, this is reduced to 200 ms, it will (on AWS Lambda) immediately see an 80 percent savings in compute costs without making any infrastructural changes.

Benefits

Easier operational management

- **Scaling benefits of FaaS beyond infrastructure costs**
  - scaling functionality of FaaS reduce compute cost, it also reduces operational management because the scaling is automatic.
  - since scaling is performed by the provider on every request/event, **you no longer need to 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.
- **Reduced packaging and deployment complexity**
  - Packaging and deploying a FaaS function is simple compared to deploying an entire server.
  - All you're doing is packaging all your code into a zip file, and uploading it. No Puppet/Chef, no start/stop shell scripts, no decisions about whether to deploy one or many containers on a machine.

Benefits

- **Time to market and continuous experimentation**
  - The new-idea-to-initial-deployment story for FaaS is often excellent, especially for simple functions triggered by a maturely defined event in the vendor’s ecosystem.
- **"Greener" computing?**

Inherent drawbacks

- **Vendor control**
  - With any outsourcing strategy you are giving up control of some of your system to a third-party vendor. Such lack of control may manifest as system downtime, unexpected limits, cost changes, loss of functionality, forced API upgrade
- **Multitenancy problems**
  - Multitenancy refers to the situation where multiple instances of software for several different customers (or tenants) are run on the same machine, and possibly within the same hosting application.
  - 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).

Inherent drawbacks

- **Vendor lock-in**
  - It's very likely that whatever Serverless features you're using from one vendor will be implemented differently by another vendor.
  - If you want to switch vendors you'll almost certainly need to update your operational tools (deployment, monitoring, etc.)
- **Security concerns**
  - Embracing a Serverless approach opens you up to a large number of security questions.
    - increases your surface area for malicious intent and ups the likelihood of 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.

Inherent drawbacks

- **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.
  - all your client apps (perhaps web, native iOS, and native Android) now 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.
- **Loss of server optimizations**
  - With a full BaaS architecture there is no opportunity to optimize your server design for client performance.
- **No in-server state for Serverless FaaS**
  - FaaS functions have significant restrictions when it comes to local .. state. .. You should not assume that state from one invocation of a function will be available to another invocation of the same function.

Implementation drawbacks

- **Configuration**
  - very little in the way of configuration for Lambda functions. worth checking if you use a less mature platform.
- **DoS yourself**
  - AWS Lambda limits how many concurrent executions of your Lambda functions you can be running at a given time.
  - Some organizations use the same AWS account for both production and testing. That means if someone, somewhere, in your organization performs a new type of load test and starts trying to execute one thousand concurrent Lambda functions, you'll accidentally DoS your production applications.
- **Execution duration**
  - AWS Lambda functions are aborted if they run for longer than five minutes.
- **Startup latency**

Implementation drawbacks

Testing

- Unit testing Serverless apps is fairly simple: any code that you write is “just code,” and for the most part there aren’t 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.

- **Debugging**
  - Debugging with FaaS is an interesting area. There’s been progress here, mostly related to running FaaS functions locally, Different vendors may not have adequate support for debugging.

Consideration before Migrating

- Tooling
- State management
- Platform improvements
- Education