Serverless Computing: Economic and Architectural Impact

Gojko Adzic
Neuri Consulting LLP
25 Southampton Buildings
London, United Kingdom WC2A 1AL
gojko@neuri.co.uk

Robert Chatley
Imperial College London
180 Queen's Gate
London, United Kingdom SW7 2AZ
rbc@imperial.ac.uk

ABSTRACT

Amazon Web Services unveiled their 'Lambda' platform in late 2014. Since then, each of the major cloud computing infrastructure providers has released services supporting a similar style of deployment and operation, where rather than deploying and running monolithic services, or dedicated virtual machines, users are able to deploy individual functions, and pay only for the time that their code is actually executing. These technologies are gathered together under the marketing term 'serverless' and the providers suggest that they have the potential to significantly change how client/server applications are designed, developed and operated.

This paper presents two case industrial studies of early adopters, showing how migrating an application to the Lambda deployment architecture reduced hosting costs – by between 66% and 95% – and discusses how further adoption of this trend might influence common software architecture design practices.

CCS CONCEPTS

• **Social and professional topics** → **Economic impact**; • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → *Software design tradeoffs*;

KEYWORDS

Serverless, Cloud Computing, Economics

ACM Reference format:
Gojko Adzic and Robert Chatley. 2017. Serverless Computing: Economic and Architectural Impact. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17),* 6 pages.
https://doi.org/10.1145/3106237.3117767

1 'SERVERLESS' COMPUTING

The marketing term 'serverless' refers to a new generation of platform-as-a-service offerings by major cloud providers. These new services were spearheaded by Amazon Web Services (AWS) Lambda[1], which was first announced at the end of 2014 [7], and which saw significant adoption in mid to late 2016. All the major cloud service providers now offer similar services, such as Google Cloud Functions[2], Azure Functions[3] and IBM OpenWhisk[4]. This paper primarily discusses AWS Lambda, as this was the first platform to launch and is the most fully-featured.

Historically, application developers would procure or lease dedicated machines, typically hosted in datacentres, to operate their systems. The initial capital expenditure required to purchase new machines, and the ongoing operational costs, were high. Lead times to increase capacity were long, and coping with peak computational loads in systems with varying demand required advance planning, and often provisioning (and paying for) many machines that were under utilised during periods of average load.

With the rise of cloud computing [2], developers switched from physical machines to virtual machines. Dramatic reductions in lead time led to the ability to scale an application's deployment footprint up and down in response to changes in demand, paying for machine usage at a per-hour resolution (as with AWS EC2). Together with auto-scaling policies [5], this allowed for significant reduction in operational costs, but still required development and operations staff to explicitly manage their virtual machines. So-called Platform-as-a-Service (PAAS) offerings such as Heroku[5] and Google App Engine[6], provided an abstraction layer on top of the cloud systems, to ease the operational burden, although at some cost in terms of flexibility and control.

'Serverless' refers to a new generation of platform-as-a-service offerings where the infrastructure provider takes responsibility for receiving client requests and responding to them, capacity planning, task scheduling and operational monitoring. Developers need to worry only about the logic for processing client requests. This is a significant change from the application hosting platform-as-a-service generation of providers. Rather than continuously-running servers, we deploy 'functions' that operate as event handlers, and only pay for CPU time *when these functions are executing*.

Traditional client/server architectures involve a server process, typically listening to a TCP socket, waiting for clients to connect and send requests. A classic example of this is the ubiquitous web server or a message queue listener. This server process plays the critical role of task dispatching, but is also traditionally assigned the role of a gate-keeper. With serverless deployments, the application developers are responsible for the logic of processing an event,

ESEC/FSE'17, September 4-8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3117767

[1] https://aws.amazon.com/lambda/
[2] https://cloud.google.com/functions/
[3] https://azure.microsoft.com/en-gb/services/functions/
[4] https://developer.ibm.com/openwhisk/
[5] https://www.heroku.com/platform
[6] https://cloud.google.com/appengine/

but the platform takes responsibility for receiving and responding to client requests, task dispatching and scheduling. Application developers are no longer in control of the 'server' process that listens to a TCP socket, hence the name 'serverless'.

2 ECONOMICS AND EFFECTS

Beyond the technical convenience of reducing boiler-plate code, the economics of AWS Lambda billing have a significant impact on the architecture and design of systems. Previous studies have shown reductions of costs in laboratory experiments [8] – here we examine the effects during industrial application. We discuss three main factors that we have observed affecting architectural decisions when serverless computing is available.

2.1 Actual Utilisation, not Reserved Capacity

With all deployment architectures from on-premise to application hosting, reserving processing capacity is a significant concern. Fail-over and load balancing are critical for handling usage loads exceeding the capacity of a single machine. Common practices include deploying redundant active services, or providing hot stand-by services that can take over if the primary service fails. Each such service would historically increase hosting costs proportionally, requiring careful capacity and disaster recovery planning.

With AWS Lambda, where application developers are no longer in control of the server process, usage is billed only when an application actively processes events, not when it is waiting. This, in effect, means that application idle time is free.

Application developers do not need to worry about reserving processing capacity. This is particularly important for auxiliary services that support high usage applications, and need to be highly available, but are not used continuously – rather they are used only in handling a subset of requests. For example, if a service task takes 200 milliseconds to execute, but needs to run only every five minutes, a traditional client/server architecture would require dedicating a service instance to it, and ensuring that a fail-over service instance is available just in case the primary crashes. This would result in paying for two dedicated server instances (or bundling with other services, which we will discuss more later, in Section 4.1). With AWS Lambda, the hosting would be billed only for 200 milliseconds out of every five minutes.

Table 1 gives a detailed comparison of how much such a task would cost to run on various hosting platforms. The smallest virtual machine on AWS EC2 service, with 512MB available memory, costs $0.0059 per hour. Running two such machines (primary and fail-over) would cost $0.0118. As a comparison, a 512MB Lambda instance executing for 100ms costs 0.000000834 USD, so running for 200ms every five minutes would cost $0.000020016 for one hour, a cost reduction of more than 99.8%. EC2 does not provide lower memory instances, but Lambda does, so if the task requires less memory, the reductions would be even greater. With a 128MB Lambda instance, the cost would be $0.000004992 per hour, resulting in a cost reduction of more than 99.95%. Note that with Lambda there is no need to reserve a separate fail-over instance, as the platform provides that implicitly.

Another current trend that aims at improving utilisation of reserved instances is containerisation using technologies such as Docker. We do not have space to explore this in detail in this paper, but serverless computing can be viewed as containerisation operated at a scale where the optimisation of resource usage can be done by the infrastructure provider, across all customers, rather than managed by a particular customer within their own deployment.

2.2 Distributed Request-Level Authorization

Applications based on serverless designs have to apply distributed, request-level authorization. A request to a Lambda function is equally untrusted whether it comes from a client application directly or from another Lambda function. As the platform scales up and down on demand, keeping a session in memory is pointless. Two sequential requests from the same client might connect to the same Lambda instance, or completely different ones. As the serverless platforms no longer have a gatekeeper server process, using the traditional model where back-end resources implicitly trust servers is not viable. Hence each request to traditional back-end resources, such as network storage or a database, needs to be separately authorised. In fact, with AWS services, a request directly from the client application to storage or a database is not trusted any more or less than a request coming from a Lambda function. They all need to be validated and authorised separately.

This means that it is perfectly acceptable, even expected, to allow client applications to directly access resources traditionally considered 'back-end'. AWS provides several distributed authentication and authorization mechanisms to support those connections. This is a major change from the traditional client/server model, where direct access from clients to back-end resources, such as storage, is a major security risk. With AWS IAM[7] or AWS Cognito[8], for example, clients can obtain security roles which enforce fine-grained control over resources and the types of actions they can perform.

A trivial example that illustrates this is collecting user analytics, a common task in most modern web or mobile applications. A typical client/server flow for such a scenario would be for the client application to post analytical events to a web server, and the web server would then write the events to a back-end data storage. With AWS Cognito, end-user devices can be allowed to directly connect to Amazon Mobile Analytics, but only authorised to write new events. Putting a Lambda function between the client device and the Amazon Mobile Analytics service would not improve security in any way, and would only introduce additional latency, and cost. The platform's requirement for distributed request-level authorization causes us to remove components from our design, which would traditionally be required to perform the role of a gatekeeper, but here only make our system more complex and costly to operate. We discuss below further optimizations in the same vein.

2.3 Different Services Billed According to Different Utilisation Metrics

Given the fact that different AWS services are billed according to different utilisation metrics, it is possible to significantly optimise costs by letting client applications directly connect to resources. For example, the Amazon Mobile Analytics service charges $1 per million recorded events (the first 100 million events are free each

[7] https://aws.amazon.com/iam/
[8] https://aws.amazon.com/cognito/

Table 1: Comparing hostings price for one hour of operation, assuming 200 ms of runtime, executing every five minutes.

Service instance	Billable unit	Unit cost (USD)	Fail-over costs (%)	Cost of 12 x 200ms exec'ns	% reference price
Lambda (128 MB)	100 ms	$0.000000208	included	$0.000004992	24.94%
Lambda (512 MB)	100 ms	$0.000000834	included	$0.000020016	100.00%
Heroku Hobby (512 MB)	1 month	$7.00	100%	$0.0097222222	48572.25%
AWS EC2 t2.nano (512 MB)	1 hour	$0.0059	100%	$0.0118	58952.84%
AppEngine B1 (128MB)	1 hour	$0.05	100%	$0.1	499600.32%
AppEngine B4 (512MB)	1 hour	$0.20	100%	$0.4	1998401.28%

month). In the typical client/server web workflow, where an event is submitted to a gateway service which then talks to the event store, the deployment architecture would require paying both for the processing and the storage. The serverless approach would be to use a Lambda function just to authorize write-only access to the analytics service, once per session, and then let clients connect directly to the event store. This would significantly reduce processing costs. Using AWS Cognito to authorise client requests directly could shift the authorisation cost from paying once per session to once for each monthly active user.

Lambda costs increase in proportion to maximum reserved memory and processing time, so any service that does not charge for processing time is a good candidate for such cost shifting. For example, the storage service S3 charges for data transfer only. Uploading a large file from a client application directly to Lambda would require setting a high memory limit so Lambda could receive large files. An alternative approach would be to use a Lambda function to authorize clients to write a specific file to S3, then upload the file directly from the client application to S3, followed by triggering a Lambda function to process the file. This would allow the second Lambda function to use streaming data APIs and work with a significantly lower memory limit. The latter approach can result in reducing hosting costs by an order of magnitude.

3 CASE STUDIES

Our first observations on the impact of adopting a serverless architecture came during the development of MindMup, a commercial online mind-mapping application, which one of the authors is involved in developing and operating. To investigate whether the effects observed in the MindMup case study transferred to other applications, we contacted colleagues at Yubl – a social networking application – who had been through a similar migration.

3.1 MindMup

With MindMup[9], a collaboration platform that moved from Heroku to AWS Lambda in 2016, the initial impact on application development was to significantly reduce the amount of boiler-plate code. For example, MindMup allows users to upload and edit mind map diagrams in its native format and produces a variety of read-only formats suitable for sharing or publishing, for example as PDF. When MindMup was originally designed and run on Heroku, each of the file conversion tasks connected to a message queue, waiting on a message to come in, and then executed the file conversion task for a particular format. Each such service dealt with disconnecting and reconnecting to queues, retrying failed tasks, logging and monitoring. Moving the file conversion to AWS Lambda made almost all the parts apart from actual file conversion obsolete. Allowing the developers to focus their work on application-specific logic rather than infrastructural plumbing reduced the amount of time and effort required to implement and launch each new feature.

MindMup allows users to convert documents into many different formats, including PDF, SVG, markdown text and Word documents. Some of those services are used frequently, some less frequently, so bundling them together on Heroku made it significantly cheaper than reserving separate primary and fail-over services for each format. The negative effect of bundling is that one service can then impact the others by exhausting all the resources available to the application. For example, due to a bug in one of the MindMup exporters, temporary file system space was not properly cleaned up after usage. Once the free temporary space was exhausted, all the exporters in that same virtual machine started to crash, increasing the load on other instances of the application, which also subsequently crashed in a domino effect.

Figure 1 shows the traditional web client/server flow for file processing. The web server process is a gatekeeper, and communicates on behalf of the client with other back-end processes. Although the workflow diagram is simple, the server process is busy during file transfers, and busy while waiting for back-end resources to finish. Figure 2 shows the current MindMup file conversion flow, since the application has been converted to AWS Lambda. There are two Lambda services, one that generates storage request signatures, and another that performs the actual file conversion. The request signature is a temporary authorisation for a client process to perform very specific operations on the storage directly, namely to upload a file of up to a certain size to a particular location, and to request the contents from the pre-defined result location. The client can then send the file directly to AWS S3 storage. The conversion Lambda is configured to start automatically whenever a file is uploaded to any of the source folders, and store the results into a result folder where the client expects to find it. The client application can meanwhile poll for the results at a well known location, and receive the converted file when available.

Although the flow is significantly more complex in the Lambda case than with a server, note that the Lambda functions are busy only when doing something useful – there is no waiting for external resources to complete. S3 storage is billed for transfer only, not for elapsed time, so pushing the workflow coordination to the client significantly reduces hosting costs, without compromising security.

It would also be possible for MindMup to make use of the AWS Cognito service, allowing clients themselves to assume predefined

[9] https://www.mindmup.com/

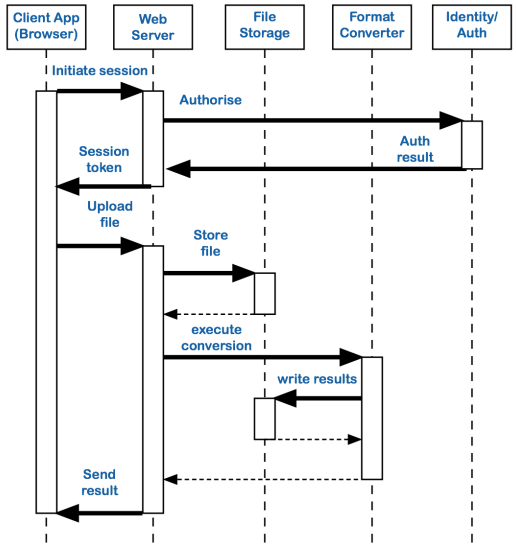

Figure 1: MindMup file conversion with a server

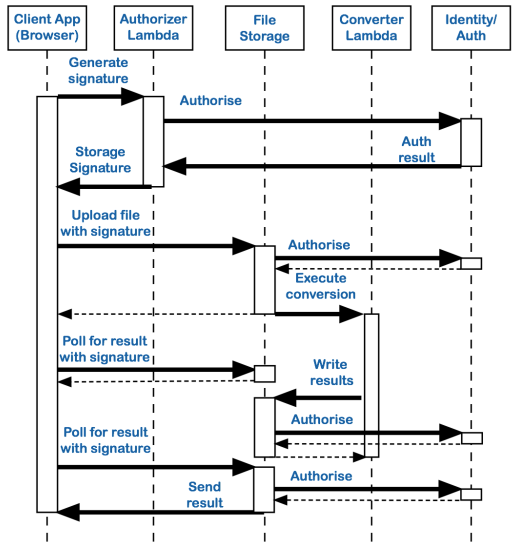

Figure 2: MindMup file conversion with Lambda

security roles. That would eliminate the need for the authorisation Lambda function. However, while Cognito allows setting limits on which operation types on which resources are available to a client application, it currently does not allow fine-grained control of file sizes. Pre-signed operations allow setting the maximum file size for uploads as well, which is why MindMup uses that approach.

Comparing the usage loads between February 2016 (before the transition to Lambda) and February 2017 (after the transition was fully completed) [1], the number of active users of MindMup increased by slightly more than 50%, but the hosting costs dropped slightly less than 50%, resulting in savings of about 66%, despite a number of new services being added during that year.

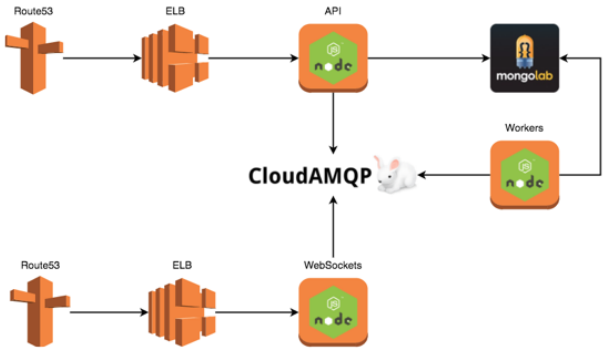

Figure 3: Yubl system architecture, before migration.

3.2 Yubl

Yubl is a London-based social networking company. In April 2016, Yubl had a monolithic system based around a small number of Node.js services providing a back-end to a smartphone app. Each service ran on a separate EC2 instance, and each was replicated across multiple Availability Zones for redundancy. The traffic pattern of the social network was subject to large spikes, particularly when influential users and advertisers ran campaigns on the network. These could bring traffic spikes up to 70x normal usage. To allow for this, Yubl ran with a good deal of headroom in their normal deployment footprint. Although they did employ AWS autoscaling, they found that scaling up an Autoscaling Group could take around 15 minutes on average, which was much too slow to allow them to deal with their aggressive traffic spikes. Therefore they set their autoscaling thresholds to trigger earlier, at around 50% CPU utilisation (more typical would be to trigger scaling when CPU utilisation goes above 75%). Therefore they were running with a lot of unused resources, and paying for them proportionally.

After a change of management and the appointment of some new technical staff, the engineering team set out some new objectives for the system, and began to rearchitect the system to work towards them. Notably they wanted to:

- be cost efficient
- minimise effort spent on keeping the infrastructure up (instead, spending that effort on making a better product)
- be able to deploy features quickly, frequently, independently, and without down time

Although Yubl closed down in Nov 2016 (as they were unable to secure another round of funding), by this time they had migrated large parts of their backend system to run on Lambda, as well as implementing many new features. At this point they had over 170 Lambda functions running in production, and yet the monthly bill for the services running on Lambda was less that $200 per month. That was compared to approximately $5000 per month for EC2 instances. To provide continuity during the migration, the team had to continue to run the EC2 instances until everything was moved over to Lambda, but this $5000 per month bill was for much smaller instances than they had been running previously, so their estimate is that moving to Lambda gave an operational cost reduction of greater than 95% for a comparable amount of compute.

Similarly to MindMup, the Yubl team found that the most natural parts of their system to migrate first were those that processed tasks from a work queue. Figure 3 shows the Yubl architecture before the migration. The existence of the work queue (implemented using CloudAMQP) decoupled the API servers from back end worker processes, and so these processes could be migrated to Lambda without having to change the client API. This was important at Yubl as mobile clients and server systems were developed by different teams, and they did not want to force their users to download new versions of their smartphone app just to support system refactoring. With more than 200,000 downloads already in the field, they did not want changes to be invasive for their existing users.

Yubl's other objectives were around decreasing time to market, and releasing more new features more quickly. In April 2016 Yubl were making 4-6 production releases per month. The product team saw this as a slow rate of delivery, and lost confidence in the engineering team's ability to deliver features. With a renewed team and the migration to Lambda, the rate of delivery went up to 60 releases in May, and by November they had been consistently averaging 80+ production releases per month with the same team size (6 engineers). The move to a service-oriented architecture, breaking the monolith into functions that could be independently deployed, meant that they were better able to split the team up to work on more things in parallel, and to deploy each feature separately (compared with the monolithic system, where everyone worked on the same codebase, resulting in more conflicts and more coordination required). Delegating responsibility for a lot of infrastructural concerns to the platform meant that they could spend more of their time developing new business features. Where previously deployments were risky and required careful planning, after moving to Lambda and working with smaller units they were fast and low effort, with the option to instantly roll back if things went wrong. This was combined with greater confidence gained by running each change through an automated deployment pipeline [3] providing more rigorous and automated testing for each unit. As a result the team was able to deliver much more quickly and reliably. The more the team was able to deliver, the more trust they found they got from the rest of the business, and the more ambitious and confident the product team became.

4 OPPORTUNITIES CREATED

By changing the economics of hosting, in particular paying only for actual utilisation instead of reserved capacity, serverless platforms offer interesting opportunities for future cloud-hosted software, particularly in the following two areas:

4.1 Removing Incentives for Bundling

Paying for reserved capacity often leads to bundling related services into a single application package. Continuing the example from Table 1, it would be very hard to economically justify having a dedicated service instance for an infrequent but important task, let alone two service instances (primary and fail-over). A single reserved instance could probably perform many such tasks. Modern client/server applications can consist of hundreds of such tasks, particularly given the current trend for microservices [6], and using dedicated instances for each would be financially unreasonable. With virtual cloud machines (EC2) or application hosting (Google App Engine or Heroku), such tasks typically get bundled into a single application so they can be provisioned, secured, scaled and monitored cheaply. But the downside is that those tasks can then influence and interfere with each other.

In section 3.1 we discussed how when deployed on Heroku, bundling together MindMup's various file conversion services made operation significantly cheaper than reserving separate primary and fail-over services for each format. However, a bug in one exporter (even one that was infrequently used) could easily impact the performance of the others.

For serverless architectures, billing is proportional to usage, not reserved capacity. This removes the economic benefit of creating a single service package so different tasks can share the same reserved capacity. The platform itself provides security, fail-over and load balancing, so all benefits of bundling are obsolete. If each exporter were a separate Lambda function, bugs in one exporter would not have a negative impact on other exporters, as they are all separately instantiated, scaled and destroyed.

Without strong economic and operational incentives for bundling, serverless platforms open up an opportunity for application developers to create smaller, better isolated modules, that can more easily be maintained and replaced. The two case studies covered by this paper offer some anecdotal evidence for a significant impact on application maintenance, but more research on a wider sample is needed to draw a statistically significant conclusion. As this is a new way of deploying applications, there is not currently much data available on production usage, so this would be a good topic for follow-up study in the future.

4.2 Removing Barriers to Versioning

The AWS Lambda pricing model creates significant opportunities for application developers to utilise end-to-end service versioning. Each deployment of a Lambda function is assigned a unique numeric identifier, and multiple versions of the same function can be available concurrently. With on-premise, virtual cloud hosting or even application hosting, deploying multiple versions at the same time increases the cost of operations proportionally. Deploying two versions of a payment service would cost twice as much for hosting as just running a single version. Because of that, most small and medium-sized companies apply A/B testing [4] and similar research techniques mostly to front-end layers of their software, where it is cheaper to manage multiple versions.

With AWS Lambda, there is no additional cost to operating multiple versions. For example, the cost for 10,000 requests going to a single version is the same as the cost of two groups of 5,000 requests each going to two different versions. This removes the financial downside of deploying and operating experimental versions. Application developers can use techniques such as A/B testing or gradual releases of features cheaply.

5 WEAKNESSES AND LIMITATIONS

As all the serverless platforms are still in their infancy, there are still significant limitations that restrict their applicability for many use cases. At the time of writing, AWS Lambda was by far the most

production-ready and advanced platform in the market, so other platforms may have even bigger limitations than outlined here.

No strong service-level agreements AWS Lambda is a relatively new service, so Amazon is not yet offering any uptime or operational service level agreements. Occasionally, Lambda users experience increased error rates and latencies. According to the AWS Lambda status feed[10] for the US-East-1 region, such events occurred seven times[11] during the calendar year before this paper was written, for a total of 36 hours and 52 minutes, giving Lambda an approximate uptime of 99.6%. (This calculation is illustrative rather than completely fair because the problems were not total outages, and due to the nature of the service, likely to affect only a portion of customers or requests. AWS does not publish official uptime numbers for Lambda.) This makes Lambda unsuitable for mission critical tasks requiring higher availability.

Potentially high latency Lambda automatically decides when to scale the number of active instances up or down, so a new request might end up creating a completely fresh instance. Application developers have no control over this process. Anecdotally, creating a new Lambda instance for JavaScript or Python functions takes about one second, and between three and ten seconds for other environments, such as Java. An inactive instance is likely to be reused within three minutes, but discarded after a longer period of inactivity. (Amazon does not publish these numbers officially and does not offer any strict service-level agreements, so these are just our observations). This means that very infrequently used services might experience constant high latency. Even for frequently used services, clients might experience some additional latency during usage spikes when many new instances get created. At least at the time of writing, it is not possible to guarantee low latency for each request to a service deployed on Lambda.

No compliance Some AWS services are compliant with various government and industry security standards, allowing application providers to deploy sensitive applications to the cloud, process and transmit certain types of records. (For example, the Amazon EC2 service is compliant with SOC, PCI, HIPAA BAA and FedRAMP standards.) At the time of writing, Lambda was officially not included on any of the Amazon Compliance Services in Scope data sheets.[12] This severely restricts the type of applications that can execute directly within Lambda instances.

Relatively short life-span The maximum configurable time for a Lambda function execution is currently five minutes. There is no way for application developers to extend that limit, so each individual task needs to complete within that time frame. This limits the applicability of Lambda services. For example, it is not possible to create a task that keeps an open HTTP connection and receives a stream of information during a longer period of time. AWS does offer several workflow solutions to chain and connect Lambda functions into longer executions, but this does not fully remove the time limit for a single task.

[10] https://status.aws.amazon.com/rss/lambda-us-east-1.rss

[11] between 06:08 PM and 07:20 PM PDT on 29 Jul 2016, between 5:18 PM and 7:14 PM PDT on 10 Aug 2016, between 11:47 AM and 1:13 PM PDT on 16 Aug 2016, between 5:25 PM and 6:02 PM on 22 Aug 2016, between November 3 1:15 PM and November 4 8:13 AM PDT, between 9:37 AM and 6:45 PM PST 28 Feb 2017, and between 5:25 PM and 9:00 PM PDT 10 Apr 2017.

[12] https://aws.amazon.com/compliance/services-in-scope/

No local execution environment Currently there is no way for an application developer to run a fully simulated Lambda environment on a local machine, for development or testing purposes. Lambda does provide easy versioning, so test or staging instances can be deployed easily, but this changes the usual development workflow for many cases, and makes it impossible to run integration tests locally.

Vendor lock-in Although very little executable code depends on the Lambda environment itself (the platform just triggers the execution of a custom event handler), working in a serverless environment can make application code highly dependent on the entire platform. The platform provides everything from authentication and configuration management to scaling and monitoring. Serverless architectures provide incentives to let client applications connect directly to storage resources and queues, so client code becomes more tightly coupled to other platform services. In practice, this means that moving a serverless application from one cloud provider to another would require a significant rewrite.

6 CONCLUSION

In conclusion, serverless platforms today are useful for important (but not five-nines mission critical) tasks, where high-throughput is key, rather than very low latency, and where individual requests can be completed in a relatively short time window. The economics of hosting such tasks in a serverless environment make it a compelling way to reduce hosting costs significantly, and to speed up time to market for delivery of new features.

ACKNOWLEDGMENTS

We would like to thank Yan Cui and colleagues at Yubl for sharing and discussing their system designs and operating costs.

REFERENCES

[1] Gojko Adzic. 2017. The key lesson from our serverless migration. https://gojko.net/2017/02/23/serverless-migration-lesson.html. (2017). Accessed: 2017-04-20.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (April 2010), 50–58. DOI: http://dx.doi.org/10.1145/1721654.1721672

[3] Jez Humble, Chris Read, and Dan North. 2006. The Deployment Production Line. In *AGILE 2006 Conference (AGILE 2006), 23-28 July 2006, Minneapolis, Minnesota, USA*. 113–118. DOI: http://dx.doi.org/10.1109/AGILE.2006.53

[4] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. 2009. Controlled Experiments on the Web: Survey and Practical Guide. *Data Min. Knowl. Discov.* 18, 1 (Feb. 2009), 140–181. DOI: http://dx.doi.org/10.1007/s10618-008-0114-1

[5] Ming Mao and Marty Humphrey. 2011. Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 49, 12 pages. DOI: http://dx.doi.org/10.1145/2063384.2063449

[6] Sam Newman. 2015. *Building Microservices* (1st ed.). O'Reilly Media, Inc.

[7] Amazon Web Services. 2014. Release: AWS Lambda on 2014-11-13. https://aws.amazon.com/releasenotes/AWS-Lambda/8269001345899110. (2014). Accessed: 2017-04-20.

[8] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. 2017. Cost Comparison of Running Web Applications in the Cloud Using Monolithic, Microservice, and AWS Lambda Architectures. *Service Oriented Computing and Applications* (2017), 1–15. DOI: http://dx.doi.org/10.1007/s11761-017-0208-y