A Multivocal Literature Review of Function-as-a-Service (FaaS) Infrastructures and Implications for Software Developers

This is the pre-print for a paper published in: the European Conference on Systems, Software and Services Process Improvement (EuroSPI) 2020
DOI: https://doi.org/10.1007/978-3-030-56441-4_5

An analysis of Function-as-a-Service (FaaS): vendors, challenges and implications for software developers

Jake Grogan[1], Connor Mulready[1], James McDermott[1], Martynas Urbanavicius[1], Murat Yilmaz[1,2], Yalemisew Abgaz[1], Andrew McCarren[1], Silvana MacMahon[1,2], Vahid Garousi[3], Pooyan Jamshidi[4], Paul Clarke[1,2]

[1] School of Computing, Dublin City University, Ireland
[2] Lero – the Science Foundation Ireland Research Centre for Software
[3] School of Electronics, Electrical Engineering And & Computer Science, Queens University Belfast, UK
[4] Computer Science and Engineering Department, University of South Carolina, USA

jake.grogan8@mail.dcu.ie, james.mcdermott7@mail.dcu.ie, connor.mulready2@mail.dcu.ie, martynas.urbanavicius2@mail.dcu.ie murat.yilmaz@dcu.ie, yalemisewm.abgaz@dcu.ie amccarren@computing.dcu.ie, silvana.macmahon@dcu.ie, v.garousi@qub.ac.uk, pjamshid@cse.sc.edu, paul.m.clarke@dcu.ie

Abstract. In this paper, we provide an analysis of Function as a Service (FaaS) infrastructures. FaaS is an important, emerging category of cloud computing, which requires that software applications are designed and deployed using distributed, highly-decoupled service-based architectures, one example of which is the microservices architecture paradigm. FaaS is associated with on-demand functionality and allows developers to build applications without the overhead associated with server management. As such, FaaS is a type of serverless provisioning model wherein a provider dynamically manages and allocates machine resources, with the developers deploying source code into a production environment. This research provides an analysis of scalability, cost, execution times, integration support, and the constraints associated with FaaS services provided by several vendors: AWS Lambda, Google Cloud Functions, and Azure Functions. We discuss the implications of the findings for software developers.

Keywords: Functions-as-a-Service, Infrastructures, Serverless, Cloud Computing, Scalability, Constraints, AWS Lambda, Microsoft Azure, Google Cloud Functions.

1. Introduction

Software engineerig is a complex undertaking [44], the process for which must take account of various situational factors [45, 46]. One such factor is software architecture which comes in many forms but which is presently challenged to produce software systems that can be deployed more quickly and with less disruption to existing operational code, such as is exemplified in continuous software engineering for microservices architectures [47]. A further benefit, and a key consideration in commercial software engineering settings, is the fact that microservices architectures can take advantage of emerging lower cost cloud hardware provisioning models, one example of which is Function-as-a-Service (FaaS), defined by IBM as "a type of cloud-computing service that allows you to execute code in response to events without the complex infrastructure typically associated with building and launching microservice applications." [29]. FaaS, perhaps the most central technology in the serverless model, is focused on the event-driven computing paradigm wherein application code, or containers, only run in response to events or requests [29]. Fundamentally,

FaaS is about running backend code without managing your own server systems or your own long-lived server applications [18].

Many cloud-computing vendors, such as Google, AWS, and Azure, among others, offer FaaS services. AWS Lambda, Google Cloud Functions, and Azure Functions are among the most commonly used FaaS services in industry today [30]. Each vendor offers a different set of capabilities with their FaaS infrastructure implementations, from language runtime support and memory usage to the ability to execute functions at regional edge cache locations in response to events generated by content delivery networks [31]. When considering FaaS as part of a systems architecture, it is vital to choose the solution that works best for the system under consideration. For this reason, it is vital that factors surrounding FaaS infrastructures which influence this decision are discussed and investigated.

A multivocal literature review (MLR) approach was adopted when researching this topic. For this reason, both peer reviewed sources and non-peer reviewed, grey sources were included. This is important as perspectives from both academia and industry professionals are considered. The area of "FaaS infrastructures" is broad and fragmented and for this reason it was of utmost importance to identify relevant search terms and to include a broad range of literature in this study.

The set of FaaS infrastructure factors under analysis in each of our primary sources varies from source to source, therefore, we suggest, that to analyse various FaaS infrastructures, as many of these factors as possible must be taken into account. The role of software engineering is changing [48] this paper therefore outlines some of the major implications for software developers as required by the FaaS innovation.

In this paper, factors which will be analysed include execution times, memory configurations, abilities to scale, pricing and cost of FaaS services, the constraints of FaaS infrastructures, and how well integrated vendor FaaS infrastructures are, not only with their own platforms, but how they can be integrated with third party services. This is an important factor as vendor lock-in is a major barrier to the adoption of cloud computing due to the lack of standardisation [32].

2. Related Literature

2.1. Methodology

This research has been conducted in the context of a multivocal literature review [49] using both grey and white literature. The primary research was conducted by a team of four students who were allocated the research paper title: "An Analysis of Functions-As-A-Service Infrastructures". The topic was examined under four key sub-topics: "How do FaaS providers compare in performance and scalability", "Constraints comparison of AWS Lambda, GC Functions and Azure Functions", "How do FaaS providers differ in pricing", and "How well integrated are FaaS infrastructures with vendor platforms".

The team identified both academic and non-academic sources from searches on google.com and scholar.google.com, as well as IEEE, ACM and Wiley, among others. From these sources, we identified keywords relevant to the research being conducted and snowballing was incorporated to identify important sections of current sources and to identify other possible sources for inclusion in this research. These identified sources were examined and marked according to an inclusion/exclusion criteria outlined in section 2.2, leading to 43 sources being included in this analysis.

Given that the topic under study is broad, sub-topics were agreed upon by all members of the team as they emerged as the most pressing questions raised in other literature when comparing key elements of various FaaS providers. This research was conducted in the period February-April 2020, the primary focus of this research was to gain the best possible understanding of the topic area, dive deep on the identified research questions and present our findings in a coherent manner within this paper.

Each member of the team was allocated one of the four research questions identified in order to conduct a review of said research question. This led to each member gaining a deep understanding of their allocated topic. Each member spent four weeks researching their topic and combining their findings in a single, summarized and coherent document in the final two weeks.

On a weekly basis throughout this period, the student team liaised with the research supervisor, for guidance on methodology and research direction.

2.2. Inclusion/Exclusion Criteria

Following the completion of the search and source selection phase, the identified sources underwent further evaluation in order to grade their relevance to our study. In our criteria, high impact factor journals were given preference over lower impact factor publications, and when evaluating white literature for inclusion in our sources, we graded these sources based on the number of citations they have received elsewhere and their relevance to the research questions. When including non peer-reviewed grey literature such as blogs or videos, the team approached with caution, only including those sources which are considered reliable and relevant based on the level of administration or moderation of the source and the level of bias introduced by the source, though clearly even well moderate online blogs fall some way short of the rigor, validity and generalization expected from academic peer reviewed research articles.

4. Research Questions and Findings

In this section, we will present the research questions and the associated findings from our literature review. The primary focus in this analysis is to examine three of the established major FaaS providers and to compare and contrast their various infrastructures: Microsoft, Google and AWS. Key concerns for software developers targeting FaaS platforms are expressed in the form of research questions []. For example, the ability of FaaS platforms to scale under higher system load is an important consideration, as it the basic runtime performance of the platform. The amount of memory available to runtime functions is also a key consideration in software design, as is the maximum allowable number of concurrent running functions. The FaaS pricing model is furthermore a key operational concern. These and other considerations are examined in each of the three providers under study.

4.1. - RQ 1: How do FaaS providers compare in performance and scalability?

In this section, we compare and contrast AWS Lambda, Google Cloud Functions and Azure Functions in terms of their relative performances and their abilities to scale. For the purposes of this research and due to limitations outlined in Section 5, we limit the definition of performance to hot-start execution times (Section 4.1.1.) and cold-start execution times (Section 4.1.2.).

From our primary studies [1-7, 22], hot-start execution times are dependent on a number of factors. In section 4.1.1, we discuss the factors of language runtimes and memory configurations. Each vendor, AWS, Google and Microsoft Azure, offer different language runtimes which will be outlined in Section 4.2.1. Our primary studies focus on NodeJS, Python and C# .NET. Each vendor also offers different memory configurations, each of which will be outlined in section 4.2.2, however, we will discuss the impact of varying memory configurations on performance in Section 4.1.2.

4.1.1. Hot-start execution times

In this section we compare hot-start execution times between FaaS vendors. From our primary studies, hot-start times vary dramatically from runtime to runtime. In tests carried out by D. Jackson and G. Clynch [3] on AWS Lambda, the average execution times (ms) across all warm start tests show Python has the best execution time of 6.13ms followed closely by C# .NET with an average execution time of 6.39ms. Go had an average execution time of 19.21ms making it the poorest performer [3]. It may be unexpected for C# .NET to come second in the test as the Just-In-Time nature of the compiler would be expected to be slower [3, 7].

Comparing to Azure, of the two runtimes (NodeJS and C# .NET) tested, C# far outperforms NodeJS, showing an average sub-millisecond performance of just 0.93ms compared to 4.91ms for NodeJS.

For hot-start tests, C# .NET significantly out-performed on Azure compared to AWS. This is expected however, as C# is a Microsoft technology and Azure Functions would be expected to have a solid support for .NET. This suggests that vendor specific implementations of their FaaS infrastructures impact execution time by language runtime as Azure Functions are run on windows containers compared to AWS Lambda which uses the open-source .NET Common Language Runtime on Linux containers for C# [3].

4.1.2. Cold-start execution times

From our primary studies it is evident that cold-start times are of great importance when comparing execution times as very little literature focuses on hot-start executions. In this section we compare the cold-start times of AWS, Google and Azure using the NodeJS language runtime.

AWS Lambda with a memory configuration of 128 MB had a median cold-start execution time of 265.2ms compared to Google which had a median execution-time of 493ms with the same memory configuration [8]. Azure assigns their VM instances 1.5GB of memory [5, 8]. The cold start execution time of Azure Functions was 3640ms [8], a stark increase compared to AWS and Google. From the findings of the tests carried out by L. Wang *et al.* [8] an interesting result is that when the memory assigned to functions is increased (AWS at 1536MB and Google at 2048MB), the median cold-start execution of AWS is 250ms, only a 15ms decrease in execution time, however, the median cold start for Google Cloud Functions was 110ms, a decrease of 140ms [8]. This suggests that memory size has a far greater impact on cold-start execution times with Google Cloud Functions compared to AWS Lambda.

Further tests were carried out by L. Wang *et al.* [8], whereby cold-start tests were carried out over a 168 hour period and the median execution times were calculated each hour across the three vendors. Results from these tests show AWS has the most stable cold-start execution times of ~200ms. Google Cloud Functions also had relatively stable cold-start times of ~400ms (except for a few spikes). Azure had the highest variation over time, ranging from 1500ms to 16000ms [8].

From other primary studies, it also suggests again that a vendor's infrastructure implementation can impact cold start times. One study carried out [3] compares cold-start times with AWS Lambda and Azure Functions on the C# .NET runtime. The primary study shows AWS had an average cold-start execution time of 2500ms, compared to Azure which had an average cold-start time of 276.4ms [3]. We refer our reasoning back to Section 4.1.1 where Azure Functions are run on windows containers compared to AWS Lambda functions which are run on Linux containers.

4.1.3. Scalability

Scalability is an extremely important factor for anyone considering building or moving parts of their infrastructure to a serverless infrastructure. In this section we analyse findings from our primary studies on the ability of FaaS infrastructures to scale.

In tests performed by G. McGrath *et al.* [2], a framework was developed to test the ability of serverless FaaS platforms to performantly invoke functions at scale. Starting with a single invocation call, every 10 seconds an additional concurrent call, up to a maximum of 15 was added. The findings of their tests show AWS Lambda is able to scale linearly and exhibits the highest throughput of the three platforms being examined in this review. Google Cloud Functions scales sub-linearly but begins to taper off as the number of concurrent requests approaches 15. Again, similar to the cold-start execution times discussed in Section 4.1.2, Azure experiences a high degree of variance in the number of concurrent requests it can handle. Initially it out-performs AWS and Google but that number drops as the number of concurrent requests increases, then decreases, and continues to fluctuate in this manner [2].

4.2 - RQ 2: Constraints comparison of FaaS providers

In this section we compare FaaS vendors and highlight constraints associated with each. From our primary studies, the main constraints associated with FaaS vendors lie in the area of supported language runtimes, memory configurations and integration capabilities. On the topic of integration

capabilities, we analyse the vendor lock-in problem and the extent to which each vendor suffers from this. The main constraints are outlined in Table 1.

	AWS	Azure	Google Cloud
Supported runtimes	Nodejs, Python, Ruby, Java, GO, C#, Powershell, additional languages via runtime API	Nodejs, Python, Java, C#, F#, Powershell	Nodejs, Python, GO
Maximum Concurrent Executions	1000 *upgradable (Section 4.2.1)	200	1000
Minimum Function Memory	128 MB	128 MB	128 MB
Maximum Function Memory	3008 MB	1536 MB	2048 MB
Maximum Function Timeout	900 seconds	600 seconds	540 seconds
Maximum Deployment package size	50 MB (compressed) for sources 250 MB (uncompressed) for sources, modules	No limit	100 MB (compressed) for sources. 500 MB (uncompressed) for sources, modules.
Maximum HTTP request/response payload size	6 MB	100 MB	10 MB
Maximum number of functions	No limit	No limit	1000 functions per project

Table 1 - Constraints Comparison

4.2.1 Performance and Scalability Constraints

AWS offers support for the greatest number of language runtimes. AWS supports NodeJS, Python, Ruby, Java, GO, C#, and Powershell language runtimes [3, 6] as well as support for third party runtimes via the AWS Lambda Runtime Interface [40]. Azure offers support for C#, F#, NodeJS, Java, Powershell, Python, and Typescript [3, 5]. Google Cloud has the least extensive list of supported language runtimes with support only for NodeJS, Python and GO [4].

AWS and Google Cloud have a default limit of 1000 simultaneous executions for a function, but AWS allows this limit to be increased on request. Azure in comparison has a limit of only 200 simultaneous executions [2, 4-6].

4.2.2 - Memory Constraints

Each vendor has different limits on the amount of memory available to the function during execution. All three providers offer the same minimum memory configuration for functions ,but have different maximum limits, Azure has the lowest maximum memory limit of 1536MB [5] followed by Google Cloud with a limit of 2048MB [1, 4] and AWS with the largest limit of 3008MB [1, 6], therefore being able to support functions that require more memory resources than other vendor FaaS services can provide. For AWS lambda the allocated memory linearly translates to the CPU power available for the function [9].

FaaS providers limit the amount of time that a function is allowed to run before it times out. AWS has a timeout of 900 seconds, Azure 600 seconds and Google Cloud 540 seconds [4-6]. Functions are shipped as packages and some providers have certain package size limits. Out of the three providers Azure is the only FaaS vendor not to have any package size limit. AWS has a size limit of 50 MB for compressed sources and a 250 MB uncompressed limit for the whole package which includes dependencies [36]. Google Cloud doubles those limits at 100 MB and 500 MB

respectively [4]. Functions can be triggered by HTTP request events which can contain data to be passed to the function for processing. AWS limits this data up to 6 MB, Google Cloud at 10 MB and Azure at 100 MB [36, 4, 5].

4.2.3 - Vendor Lock-in

Vendor lock-in refers to a problem in cloud-computing wherein customers become dependent on a single cloud provider technology implementation. It may become difficult for a customer to move to another vendor without substantial costs or possible legal constraints [32].

In the case of FaaS, vendor migration can be very expensive if a language runtime does not have cross vendor support. As covered in section 4.2.1 Google Cloud supports only a fraction of language runtimes in comparison to other providers. This can pose a migration challenge if the existing codebase languages are not supported by the platform.

As FaaS infrastructures are generally well integrated with the rest of a vendors serverless platform (as outlined in Section 4.4), they may not integrate as well into third party services or other vendor platforms. Interoperability and portability are essential qualities that affect FaaS infrastructures and pose as a major barrier to entry into cloud-computing. [32].

As FaaS are event driven services, attempts have been made to standardise how event publishers describe events in order to support interoperability and portability. An example of one such attempt is CloudEvents [37]. Recently, Azure have announced first class support for CloudEvents [38]. AWS took a different direction when they launched EventBridge [39]. The format to describe events in AWS EventBridge differs from that of CloudEvents, which has not helped in solving the issue of cross-vender interoperability. Therefore there is still a constraint of integrability between vendors and their FaaS infrastructures.

4. 3 - RQ 3: How do FaaS providers differ in pricing?

In this section, we conduct a cost analysis comparison of FaaS solutions such as AWS Lambda, Microsoft Azure, and Google Cloud Functions. This section looks at how the pricing of a function model is influenced by performance and invocation of that function. For the purposes of this research, we will look in particular at the factors that inform the difference in pricing by limiting the research to each FaaS vendor's free tier, the tier which is most widely accessible.

From our initial studies, we can discern that each vendor, AWS, Microsoft Azure and Google Cloud, offer broken down examples of how each billing is computed. There are three main factors in a function's execution cost - execution time, fixed invocation cost per individual function execution and memory allocated to the function [3]. Thus, we explore how request invocations, the duration of execution, memory provisioning and compute time influence the pricing model, while also taking time to delve into the double billing of functions, as it violates a principle of the serverless model [28]. This primary study does not focus on a particular set of languages or runtimes as pricing is language-agnostic and follows a flat charge rate for each factor.

4.3.1 Request Invocations

Our primary studies indicate the prioritisation of request invocations as an important contributing factor in the pricing of the FaaS solution. In this section we compare the number of free invocations granted per month as well as the charge rate after passing that threshold.

In accordance with the Google Cloud Functions free tier pricing [25] invocations are charged at a flat rate, independent of the source of the invocation, including functions invoked from a HTTP request (HTTP functions), background and invocations resulting from the call API. These invocations are charged at a per-unit rate of $0.0000004 per invocation, and this excludes the first 2 million free invocations per month, though these free invocations are charged regardless of the outcome of the function or its duration.

In the case of AWS Lambda pricing [23], we note that the first million requests per month are free. After this point, the invocation of a function is not charged at a per-unit rate but rather at a per-million rate, with the next one million requests charged at $0.20.

When looking at Microsoft's Azure Functions pricing [24], invocations are counted each time a function is executed in response to an event, triggered by a binding. The first million executions are included free each month. Further executions are charged like AWS Lambda, at a price-per-million of $0.20.

In regards to contrasting the factor of pricing request invocation, we can see from the primary studies that Google Cloud Functions grants the most free requests on a per monthly basis, boasting one million more free invocations per month than both Azure Functions and AWS Lambda. In terms of comparing the charge rate thereafter, we must look at price-per-million, as both Azure and AWS offer the price in this format already. Taking Google Cloud Functions price-per unit rate of $0.0000004 and multiplying by one million, we are left with the product of $0.40. We can now say that in regards to charges after passing the free request invocation threshold, Azure Functions and AWS Lambda both offer the cheapest access based on the number of requests invoked.

4.3.2 Duration

From our primary studies of all three FaaS vendor's we can see the prioritisation of duration, or time until the completion of execution, as an important contributing factor in the pricing of the FaaS solution. In this section we compare the amount of compute time freely granted by each provider, along with how it was calculated, and the associated charging after passing this granted threshold. Compute time is measured from the time your function receives a request to the time it completes, either through your signaling of completion, or through a timeout, other failure or any other termination [25].

Taking a look at granted free tier resources we can note that Azure Functions pricing includes a monthly fee grant of 400,000 GB-seconds of resource consumption per month per subscription [24]. With AWS Lambda pricing, free tier provides 400,000 GB-seconds of resource consumption per month per subscription [23]. Finally looking at Google Cloud Functions pricing, the free tier also provides 400,000 GB-seconds [25].

Beyond this point we can see that AWS Lambda bills at $0.0000166667 per GB-second [3, 23, 25, 26], Azure Functions are billed at a flat rate of $0.000016 per GB-second [3, 24] and Google Cloud Functions at a rate of $0.0000025 per GB-second [25]. Comparing across vendor's we see each free tier capping the allowance of free resource consumption at the 400,000 GB-seconds mark.

4.3.3 Memory Provisioning

Looking beyond the granted allowance, in Azure Functions, memory used by a function is measured by rounding up to the nearest 128 MB, up to the max. memory size of 1,536 MB [24]. With AWS Lambda the memory allocated to a function can be between 128MB and 3008MB, in 64MB increments [23, 26]. With Google Cloud Functions the memory allocated to a function can be between 128MB and 2048MB, in doubling increments [25]. With provisioning we see the widest range of options offered by AWS, offering up to 3008MB, while Azure and Google Cloud respectively offer 1536MB and 2048MB max. [23, 24-26].

4.3.4 Compute Time

The cost of serverless functions are directly related to their execution times. This is due to the prevalent billing model across the major serverless platforms of cost per milliseconds of execution. [3]. Both AWS and Azure solutions round up to the nearest 1ms [23, 24] while the Google Cloud solution rounds to the nearest 100ms [25]. Across each provider except Azure, compute time is measured in 100ms increments, with AWS Lambda offering a significantly cheaper solution (at $0.0000002083 per 100ms) than Google Cloud ($0.0000025 per 100ms) [3, 23-25]. The execution time of Azure Functions is calculated by rounding up to the nearest 1 ms, with a price of $0.0000000034167 per 1ms.

4.3.5 Double Billing

Double billing refers to charging for the same product twice. In the case of FaaS vendor's double billing refers to the synchronous invocation of other functions. [27]

When a function makes a call to another service, you pay for the waiting time — even if the *code* is using async I/O. This means you wait for the result of each function called, and hence must pay for the wait time. This wait time is directly correlated to section 4.3.2. This additional billing violates one of the serverless principles: pay for what you use, not idle time. [28]
Without intrinsic support from the FaaS vendor, it is impossible to avoid double billing while still maintaining the core principle of the serverless computing model. [28]

4.4. - RQ 4: How well integrated are FaaS infrastructures with vendor's other services?

In this section we will cover what the serverless model is and the possibilities of integrating a pre-existing application and other services with it through the use of FaaS. In order to achieve this, it is necessary to cover the different architectural services attached to the serverless model for AWS, Google Cloud and Azure (Section 4.4.1). Depending on certain workflows different sections of the serverless model will be more prominent than others and understanding the entry criteria for these services for each provider will also be necessary (Section 4.4.2). By the end of this section the core differences between each provider's serverless model should be apparent, along with the integration capabilities of the provided services and FaaS.

4.4.1. - What is the serverless model?

Serverless models abstract the underlying computer infrastructure.. It eliminates infrastructure management tasks such as server or cluster provisioning, patching, operating system maintenance, and capacity provisioning. It also works as a pay-what-you-use model [33-35].

In this section we will cover the different architectural related serverless services provided by each provider. AWS, Google Cloud and Azure offers traditional and unique services which fall under the compute, storage, data stores and integration categories. Noting that even though these categories are used in both serverless and traditional cloud computing, services retaining to traditional cloud computing are outside the scope of this paper.

4.4.2 What architectural services does the serverless model provide?

FaaS and serverless are generally synonymous with one another due to its ability to listen and act upon the events of other serverless services (Section 4.4.3). However FaaS is a subset of serverless [20], which resides within the compute category. By understanding the variety of categories and the services that lie within we can better ascertain the best provider for the users architectural needs. The core categories that relate to architecture that are consistent across all three providers are the following, compute, storage, data stores and integration. The main purpose of this research is to see how FaaS integrates with other serverless services. For the sake of brevity, other compute services are omitted as the core focus of this research is FaaS.

4.4.2.1 - Storage

Storage provides consistent object-level data storage across all three providers [35, 34, 43]. However AWS also provides EFS; an NFS file system service that can be easily integrated with on-premises or cloud resources [35].

4.4.2.2 - Data Store

In regards to data stores AWS provides the most unique services, those being DynamoDB, AWS DocumentDB, AWS MCS, and Amazon Aurora Serverless [35, 41, 42]. However, in regards to variety Azure Cosmos DB provides a multi-model with wire protocol compatible API endpoints for Cassandra, MongoDB, SQL, Etcd and Table [43]. Finally Google Cloud provides their Cloud Firestore a NoSQL database [34].

4.4.2.3 - Integration

As the topic of this section, integration provides the services required in order for pre-existing resources to interact with the cloud's serverless environment. Whilst integration has a plethora of services such as Simple Notification Service for AWS, Pub/Sub for Google Cloud and Messaging for Azure, the core services that will be of most use for integration in relation to FaaS are API related creation & management services such as API Gateway for AWS and Api Management for Azure [35, 43]. Google Cloud has no direct service for this however their Cloud functions can be directly invoked by using traditional HTTP means (Section 4.4.3.1).

During our studies we also found that a large portion of the integration services based on serverless orchestration make heavy use of FaaS and act as a layer of abstraction to the service. An example of such a service would be AWS Step Functions [11, 21].

4.4.3. - FaaS entry criteria

In this section, we will look at FaaS and its ability to execute functions based on events that occur within the serverless model. This functionality allows for the design of workflows and promotes automation within the serverless model. As services may differ for each provider, so do the events, triggers and bindings associated with the relative services as discussed in the sections below. The primary study here is not to list all unique triggers but establish the possibilities of executing FaaS using external resources and in turn, highlight its transitive nature by covering the possibilities of invoking additional calls to services.

4.4.3.1 - On-Premises to FaaS

Executing lambda, cloud functions and Azure functions from outside their respective provider is possible, however the process of doing so differs from provider to provider but the underlying approach remains consistent.

Google Cloud Functions is the most straightforward with each function being accessible through a RESTful approach [4]. Accessing Lambda from outside the AWS platform requires the use of API Gateway. By using API gateway it is possible to create a public-facing API and attach a respective lambda function to an endpoint. With this additional step, the function can be triggered via a RESTful API call [9]. Azure Functions can be triggered over HTTP also without any additional set up. However in order to receive a HTTP response from the function a corresponding binding (Section 4.4.3.2) must also be created for that particular function [10].

4.4.3.2 - FaaS and Serverless Integration

As stated in section 4.4.2, FaaS is a subset of the serverless architecture. As it lies within the domain of serverless, it has the ability to react to other serverless service actions. In this section we will cover the different criteria in which other services can execute serverless functions and the overhead involved in doing so.

AWS Lambda can be viewed as a reactionary service. Depending on specific events that occur within the cloud platforms environment, it may or may not trigger a Lambda function. Whilst the list of possible triggers is vast [9], the general consensus is that any event within the services defined in section 4.4.2 contain events that can trigger a function. Additional services may also trigger

Lambda for example, AWS Cloud Watch events or responses to HTTP as mentioned in section 4.4.3.1. [9, 12, 13]

Google cloud defines events as “things that happen within your cloud environment that you might want to take action on” [14]. It also defines a trigger as “a declaration that you are interested in a certain event or set of events”. In regards to integration, the list of events provided by google may seem inherently limited [14], however a transitive approach may be taken with services to allow for additional integration with other resources within or outside of the platform [15].

Whilst Azure has the same functionality as the other platforms, it uses different terminology to a certain degree. Whilst events do exist on Azure as made apparent by the Event Hub service; a service which allows the user to orchestrate and consolidate particular events in a single easy to manage space [16]. Azure also provides the additional optionality of manually creating bindings between Azure functions and serverless resources [17]. Whilst Event Hub adds a layer of abstraction over the operation, bindings provides a hands on approach for the user. Allowing them to further customise the level of granularity needed for integration. As the only difference between bindings and Event Hub is abstraction, any service available in Event Hub will also be available to listen to through bindings [17].

5. Limitations of research

FaaS is a new and emerging technology. Whilst it first began to be discussed around 2010 [19], it is only in recent years that the paradigm itself has witnessed sustained interest and is now being offered by the largest cloud providers (AWS, Azure, Google Cloud) as a service. As the team conducting this basic review did not have much experience with research, there was some difficulty in finding informative, trusted sources on the subject and a degree of uncertainty when judging the reliability of some grey sources included in this review. However, advice and training was given each week throughout the course of the six week research period on how to correctly conduct research, and in particular, multivocal literature reviews.

Although AWS, Google Cloud and Microsoft Azure provide documentation for their FaaS services, we found it to be largely is oriented towards developers learning to use the service (API documentation) rather than the underlying infrastructure. Whilst more topics could have been included and certain topics which were included could have been expanded on, due to the six week time constraint given, the team had to limit themselves in terms of scope. Had more time been available to the researchers, the topics analysed in this paper could have undergone further testing and validation through broader and deeper research outlined in Section 2.

With these limitations having been considered, the research team feels the relevance of the work carried out has not been diminished and that this work is still a useful contribution to the study of serverless computing and, in particular, FaaS infrastructures.

6. Discussion

While FaaS infrastructures might not immediately appear to have a major impact on software developers and software development processes, upon deeper examination, we find that there are several important considerations that should not be overlooked.

To take advantage of the potential cost savings and elastic scalability that FaaS infrastructure provides, it is necessary to design and architect a software system that is suited to FaaS, and as the analysis in section 4 has shown, there are many details involved and they vary not insignificantly amongst the FaaS vendors. Traditional so-called monolith-based architectures will simply not be in a position to take advantage of the reduced operational risk and costs that FaaS can deliver, and so there is an emerging large job for the broader software industry to modernize their architectures to take advantage of FaaS. In the authors’ opinion, this is an aspect of software development work which will be large and strategically important in many software companies in the coming years.

For individual software developers, the early parts of their careers may have been characterized by a largely distant relationship with operational infrastructure that was provisioned and managed

by another department in their business or perhaps externally. However, with FaaS, individual developers may now be required to change their process of work towards actually issuing their own code in the form of functions to the FaaS infrastructure. The performance and cost of individual functions will be monitored and so the individual developer in a FaaS environment can expect to be drawn closer to the real operational system than might have been the case heretofore. Indeed, in some instances, function/service developer(s) might be entirely responsible for the creation, operation and maintenance of their code. Such a move would reduce costs related to communication and training in software companies, and could be disruptive to existing development processes, pushing DevOps to the limit and perhaps even into an entirely new space, where developers become responsible for vertical software services from conception through to operation and maintenance.

There are many other changes that will result from the adoption of FaaS, the role and process adopted by software testers will change, in previous work we have already noted that this has changed significantly in recent years [50]. The space that testers have traditionally occupied between developers and operations becomes less distinct and may fundamentally reshape the role of software test. For example, the need to monitor and log FaaS systems is high and testers would at least have to re-train to develop skill in these areas. Various other changes will undoubtedly arise and will be the subject of much future research and practice.

7. Conclusion

Function-as-a-Service (FaaS) is an emerging serverless cloud computing hardware provisioning model that allows developers to essentially be completely abstracted from hardware concerns. Three separate FaaS vendors were analyzed (AWS Lambda, Google Cloud Functions and Microsoft Azure Functions) and we found that there is a complex cocktail of factors that developers should consider when selecting a FaaS provider, we recommend that careful consideration be given to the various factors, many of which are outlined in this paper. To adopt FaaS, software developers will need to ensure that their software follows a distributed high-decoupled service-based architecture such as microservices, and to take full advantage of the potentially significant cost savings FaaS can deliver, there are certain new software design issues that must be addressed. For example, the traditional thinking associated with chains of functions/methods executing in a stack and awaiting responses for potentially long periods will need to curtailed so as to avoid the double billing problem (where one function is executing in a waiting state, awaiting the output from a second with is running concurrently). These are not insignificant mindset changes for software developers, indeed the shift away from monolith architectures and towards distributed microservices itself poses a major challenge for many established successful firms. FaaS has started to rise in popularity, and the conclusion from our research is that this is a trend that is likely to be sustained or even accelerate. Why? Because the speed of delivery of new software can be very rapid, concerns in relation to hardware scaling are removed, and the cost model is one where only the resources used are billed (e.g. software vendors pay for hardware only when their software is actually executing on it). These are large and disruptive forces, but caution needs to be adopted to avoid many pitfalls and FaaS is not a model that will work for all software developers.

There are various FaaS constraints to be considered, including the supported language runtimes, with significant differences to be observed in the vendors analyzed. The idea of custom runtimes means the scope of usage for AWS Lambdas is much wider than that of the other providers, and therefore is not a significant constraining factor. A further significant constraint discovered during our research was the idea of vendor lock in. Generally, migrating application functions from a non-cross vendor supported runtime can be very expensive due to requiring a complete rewrite of function logic to another language (Section 4.2.3). In our cost analysis our research found that the concept of double billing in the serverless model breaks a core principle of the serverless model and that without intrinsic support from the FaaS vendor, it may be impossible to avoid double billing [28]. We suggest that vendors consider the provision of a method to overcome this so they may truly achieve the serverless model. Beyond this, the cost analysis conducted by the research team concluded that Microsoft Azure and AWS Lambda offers the cheapest access to FaaS based on request invocations alone and also offers the best flat rate charge

for duration of a function's execution - at $0.000016 per GB-second (Section 4.3). The first real divergence outside of flat charges is found at memory provisioning with AWS Lambda giving the widest range of options [23]. With compute time, both AWS and Google Cloud round up to the nearest 100ms, whilst Azure only rounds to the nearest 1ms (Section 4.3.4).

In regards to the integration capabilities of each solution, Google Cloud provides a more streamlined approach due to the absence of integrating the function with an API, whereas both AWS and Azure require the user to either deploy or have a pre-existing API (Section 4.4.3.1). Generally across FaaS infrastructures, functions can be called via a HTTP request. This makes integrating functions into a pre-existing service a very simple process. These are all concerns that software developers need to be acquainted with.

Overall, our analysis of FaaS has yielded insights into the varying costs, performance, integration capabilities and constraints of each examined solution. Adopting FaaS requires significant changes to the way software is designed and deployed, and the process in these areas in particular will need to be adapted if FaaS is to be successfully and fruitfully implemented. Despite the difficult challenges to be managed in adopting FaaS, the economic imperatives driving its rise are compelling: delivery of software at a faster pace, with less impact on operational systems, and at a reduced hardware provisioning cost. FaaS in combination with well-disciplined microservices architectures may be the closest we have yet come to realizing Better and Faster and Cheaper software.

Abstract. This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (www.lero.ie)

References

1. M. Pawlik, *et al.*, 'Performance considerations on execution of large scale workflow applications on cloud functions', *arXiv:1909.03555 [cs]*, Sep. 2019.
2. G. McGrath and P. R. Brenner, 'Serverless Computing: Design, Implementation, and Performance', in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, pp. 405–410, doi: 10.1109/ICDCSW.2017.36.
3. D. Jackson and G. Clynch, 'An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions', in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 154–160, doi: 10.1109/UCC-Companion.2018.00050.
4. "Cloud Functions," *Google Cloud*. [Online]. Available: https://cloud.google.com/functions. [Accessed: 04-Mar-2020].
5. "Azure Functions Serverless Compute | Microsoft Azure." [Online]. Available: https://azure.microsoft.com/en-us/services/functions/. [Accessed: 04-Mar-2020].
6. "AWS Lambda – Serverless Compute - Amazon Web Services," *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/lambda/. [Accessed: 04-Mar-2020].
7. S. Hendrickson, *et al.* "Serverless computation with openLambda," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, Denver, CO, 2016, pp. 33–39.
8. L. Wang, *et al.*, "Peeking Behind the Curtains of Serverless Platforms", in *2018 USENIX Annual Technical Conference*, 2018.
9. "AWS Lambda - Developer Guide," , p. 184. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf [Accessed: 02-Mar-2020].
10. "Azure Functions HTTP triggers and bindings." [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook. [Accessed: 06-Mar-2020].

11. “AWS Step Functions - Developer Guide,” [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/step-functions-dg.pdf [Accessed: 01-Mar-2020]
12. “Using AWS Lambda with Other Services - AWS Lambda.” [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html. [Accessed: 05-Mar-2020].
13. “AWS Lambda Event Source Mapping - AWS Lambda.” [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/invocation-eventsourcemapping.html. [Accessed: 05-Mar-2020].
14. “Events and Triggers | Cloud Functions Documentation,” *Google Cloud*. [Online]. Available: https://cloud.google.com/functions/docs/concepts/events-triggers. [Accessed: 05-Mar-2020].
15. “Calling Cloud Functions | Cloud Functions Documentation | Google Cloud.” [Online]. Available: https://cloud.google.com/functions/docs/calling. [Accessed: 28-Feb-2020].
16. “Event Hubs—Real-Time Data Ingestion | Microsoft Azure.” [Online]. Available: https://azure.microsoft.com/en-us/services/event-hubs/. [Accessed: 02-Mar-2020].
17. “Triggers and bindings in Azure Functions.” [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings. [Accessed: 03-Mar-2020].
18. Martin Fowler, “Serverless Architectures,”. [Online]. Available: https://martinfowler.com/articles/serverless.html. [Accessed: 03-Mar-2020].
19. G. C. Fox, *et al.*, “Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research,” *arXiv:1708.08028 [cs]*, 2017, doi: 10.13140/RG.2.2.15007.87206.
20. M. Sewak and S. Singh, “Winning in the Era of Serverless Computing and Function as a Service,” in 2018 3rd International Conference for Convergence in Technology (I2CT), 2018, pp. 1–5, doi: 10.1109/I2CT.2018.8529465.
21. P. García López, *et al*, "Comparison of FaaS Orchestration Systems," 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), Zurich, 2018, pp. 148-153.
22. K. Figiela, *et al.*, “Performance evaluation of heterogeneous cloud functions,” *Concurrency and Computation: Practice and Experience*, vol. 30, no. 23, p. e4792, 2018, doi: 10.1002/cpe.4792.
23. “AWS Lambda Pricing.” [Online]. Available: https://aws.amazon.com/lambda/pricing/. [Accessed: 06-Mar-2020].
24. Pricing - Functions | Microsoft Azure” [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/functions/. [Accessed: 06-Mar-2020].
25. “Pricing | Cloud Functions Documentation | Google Cloud” [Online]. Available: https://cloud.google.com/functions/pricing. [Accessed: 06-Mar-2020].
26. “AWS Lambda Cost Guide.” [Online]. Available: https://lumigo.io/aws-lambda-cost-guide/. [Accessed: 06-Mar-2020].
27. “The need for asynchronous FaaS call chains in serverless systems” [Online]. Available: https://read.acloud.guru/the-need-for-asynchronous-rpc-architecture-in-serverless-systems-ff168f1c8785 [Accessed: 06-Mar-2020].
28. “Composing Functions into Applications - Apache OpenWhisk - Medium” [Online]. Available: https://medium.com/openwhisk/composing-functions-into-applications-70d3200d0fac [Accessed: 06-Mar-2020].
29. “An introduction to FaaS—a cloud computing service that makes it easier for cloud application developers to run and manage microservices applications.”, 2020. [Online]. Available:

https://www.ibm.com/cloud/learn/faas. [Accessed: 11-Mar-2020].

30. Nemanja Novkovic, "Top Function As A Service (Faas) Providers," *Dashbird*, 14-May-2018. [Online]. Available: https://dashbird.io/blog/top-function-as-a-service-faas-providers/. [Accessed: 11-Mar-2020].
31. "Edge Computing | CDN, Global Serverless Code, Distribution | AWS Lambda@Edge," *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/lambda/edge/. [Accessed: 11-Mar-2020].
32. J. Opara-Martins, *et al.*, "Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective," *Journal of Cloud Computing*, vol. 5, no. 1, p. 4, Apr. 2016, doi: 10.1186/s13677-016-0054-z.
33. "Overview of serverless applications in Azure." 2019. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/serverless/. [Accessed: 25-Feb-2020]
34. "Serverless Architecture | Google Cloud" [Online]. Available: https://cloud.google.com/serverless/whitepaper/. [Accessed: 26-Feb-2020]
35. "Serverless Computing - Amazon Web Services" [Online]. Available: https://aws.amazon.com/serverless/ [Accessed: 28-Feb-2020]
36. "AWS Lambda Limits - AWS Lambda." [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. [Accessed: 12-Mar-2020].
37. "CloudEvents." [Online]. Available: https://cloudevents.io/. [Accessed: 12-Mar-2020].
38. "Announcing first-class support for CloudEvents on Azure" [Online]. Available: https://azure.microsoft.com/es-es/blog/announcing-first-class-support-for-cloudevents-on-azure/. [Accessed: 12-Mar-2020].
39. "Amazon EventBridge - Amazon Web Services," *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/eventbridge/. [Accessed: 12-Mar-2020].
40. "Custom AWS Lambda Runtimes - AWS Lambda." [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html. [Accessed: 12-Mar-2020].
41. "Amazon Managed Apache Cassandra Service - Developer Guide." [Online]. Available: https://docs.aws.amazon.com/mcs/latest/devguide/ManagedCassandraService.pdf#what-is-mcs [Accessed: 03-Mar-2020].
42. "Amazon DocumentDB - Developer Guide." [Online]. Available: https://docs.aws.amazon.com/documentdb/latest/developerguide/developerguide.pdf#what-is [Accessed: 04-Mar-2020].
43. "Azure Serverless | Microsoft Azure" [Online]. Available: https://azure.microsoft.com/en-us/solutions/serverless/#solutions [Accessed: 28-Feb-2020]
44. Clarke, P., O'Connor, R.V., Leavy, B.: A Complexity Theory viewpoint on the Software Development Process and Situational Context. In: proceedings of the International Conference on Software and Systems Process (ICSSP), Co-Located with the International Conference on Software Engineering (ICSE), pp. 86-90, DOI:10.1145/2904354.2904369 (2016)
45. Clarke, P., O'Connor, R.V.: The situational factors that affect the software development process: Towards a comprehensive reference framework, Information and Software Technology, Vol. 54(5), May 2012, pp.433-447.
46. Clarke, P., O'Connor, R.V., Solan, D., Elger, P., Yilmaz, M., Ennis, A., Gerrity, M., McGrath, S., Treanor, R.: Exploring Software Process Variation Arising from Differences in Situational

Context. In: Proceedings of the 24th European and Asian Conference on Systems, Software and Services Process Improvement (EuroSPI 2017), pp.29-42, 5-8 September 2017, Ostrava, Czech Republic.

47. O'Connor, R.V., Elger, P., Clarke, P.: Continuous Software Engineering - A Microservices Architecture Perspective. Journal of Software: Evolution and Process, 29(11), 2017, pp.1-12.
48. Meade E. et al.: The Changing Role of the Software Engineer. In: Proceedings of the 26th European and Asian Conference on Systems, Software and Services Process Improvement (EuroSPI 2019), Springer CCIS Vol. 1060, pp.682-694, 18-20 September 2019, Edinburgh, Scotland.
49. Garousi V, Felderer M, Mäntylä MV (2016) The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In: Proceedings of the 20th international conference on evaluation and assessment in software engineering. ACM.
50. Cunningham S. et al.: Software Testing: A Changing Career. In: Proceedings of the 26th European and Asian Conference on Systems, Software and Services Process Improvement (EuroSPI 2019), Springer CCIS Vol. 1060, pp.731-742, 18-20 September 2019, Edinburgh, Scotland.