Serverless Computing: Design, Implementation, and Performance

Garrett McGrath
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, Indiana
Email: mmcgrat4@nd.edu

Paul R. Brenner
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, Indiana
Email: paul.r.brenner@nd.edu

***Abstract*—We present the design of a novel performance-oriented serverless computing platform implemented in .NET, deployed in Microsoft Azure, and utilizing Windows containers as function execution environments. Implementation challenges such as function scaling and container discovery, lifecycle, and reuse are discussed in detail. We propose metrics to evaluate the execution performance of serverless platforms and conduct tests on our prototype as well as AWS Lambda, Azure Functions and IBM's deployment of Apache OpenWhisk. Our measurements show the prototype achieving greater throughput than other platforms at most concurrency levels, and we examine the scaling and instance expiration trends in the implementations. Additionally, we discuss the gaps and limitations in our current design, propose possible solutions, and highlight future research.**

I. Introduction

Following the lead of AWS Lambda [1], services such as Apache OpenWhisk [2], Azure Functions [3], Google Cloud Functions [4], Iron.io IronFunctions [5], and OpenLambda [6] have emerged and introduced serverless computing, a cloud offering where application logic is split into functions and executed in response to events. These events can be triggered from sources external to the cloud platform but also commonly occur internally between the cloud platform's service offerings, allowing developers to easily compose applications distributed across many services within a cloud.

Serverless computing is a partial realization of an event-driven ideal, in which applications are defined by actions and the events that trigger them. This language is reminiscent of active database systems, and the event-driven literature has theorized for some time about general computing systems in which actions are processed reactively to event streams [7]. Serverless function platforms fully embrace these ideas, defining actions through simple function abstractions and building out event processing logic across their clouds. IBM strongly echoes these concepts in their OpenWhisk platform (now Apache OpenWhisk), in which functions are explicitly defined in terms of event, trigger, and action [8].

Beyond the event-driven foundation, design discussions shift toward container management and software development strategies used to leverage function centric infrastructure. Iron.io uses Docker to store function containers in private registries, pulling and running the containers when execution is required [9]. Peer work on the OpenLambda platform presents an analysis of the scaling advantages of serverless computing, as well as a performance analysis of various container transitions [10]. Other performance analyses have studied the effect of language runtime and VPC impact on AWS Lambda start times [11], and measured the potential of AWS Lambda for embarrassingly parallel high performance scientific computing [12].

Serverless computing has proved a good fit for IoT applications, intersecting with the edge/fog computing infrastructure conversation. There are ongoing efforts to integrate serverless computing into a "hierarchy of datacenters" to empower the foreseen proliferation of IoT devices [13]. AWS has recently joined this field with their Lambda@Edge [14] product, which allows application developers to place limited Lambda functions in edge nodes. AWS has been pursuing other expansions of serverless computing as well, including Greengrass [15], which provides a single programming model across IoT and Lambda functions. Serverless computing allows application developers to decompose large applications into small functions, allowing application components to scale individually, but this presents a new problem in the coherent management of a large array of functions. AWS recently introduced Step Functions [16], which allows for easier organization and visualization of function interaction.

The application of serverless computing is an active area of development. Our previous work on serverless computing studied serverless programming paradigms such as function cascades, and experimented with deploying monolithic applications on serverless platforms [17]. Other work has studied the architecture of scalable chatbots in serverless platforms [18]. There are multiple projects aimed at extending the functionality of existing serverless platforms. Lambdash [19] is a shim allowing the easy execution of shell commands in AWS Lambda containers, allowing developers to explore the Lambda runtime environment. Other efforts such as Apex [20] and Sparta [21] allow users to deploy functions to AWS Lambda in languages not supported natively, such as Go.

Serverless computing is often championed as a cost-saving tool, and there are multiple works which report cost saving op-

portunities in deploying microservices to serverless platforms rather than building out traditional applications [22] [23]. Others have tried to calculate the points at which serverless or virtual machine deployments become more cost effective [24].

Serverless computing is becoming increasingly relevant, with Gartner reporting that "the value of [serverless computing] has been clearly demonstrated, maps naturally to microservice software architecture, and is on a trajectory of increased growth and adoption" [25]. Forrester argues that "today's PaaS investments lead to serverless computing," viewing serverless computing as the next-generation of cloud service abstractions [26]. Serverless computing is quickly proliferating across many cloud providers, and is powering an increasing number of mobile and IoT applications. As its scope and popularity expands, it is important to ensure the fundamental performance characteristics of serverless platforms are sound. In this work we hope to aid in this effort by detailing the implementation of a new performance-focused serverless platform, and comparing its performance to existing offerings.

II. Prototype Design

We have developed a performance-oriented serverless computing platform[1] to study serverless implementation considerations and provide a baseline for existing platform comparison. The prototype has a small feature-set and a simple implementation. We approach performance in our design with the guiding assumption that of all the supported operations and their possible outcomes, successful warm function executions are by far the most common. Therefore, other operations such as function creation and cold executions are designed to avoid reducing the performance of warm executions.

Our platform is implemented in .NET and deployed to Microsoft Azure. It depends upon Azure Storage for data persistence and for its messaging layer. Besides Azure Storage services, our implementation is comprised of two components: a web service which exposes the platform's public REST API, and a worker service which manages and executes function containers. The web service discovers available workers through a messaging layer consisting of various Azure Storage queues. Function metadata is stored in Azure Storage tables, and function code is stored in Azure Storage blobs.

Figure 1 shows an overview of the platform's components. Azure Storage was chosen because it provides highly scalable and low-latency storage primitives through a simple API, aligning well with the goals of this implementation. For the sake of brevity all further references to these storage entities will simply use the terms queues, tables, and blobs, with the understanding that in the context of this paper these terms refer to the respective Azure Storage services.

A. Function Metadata

A function is associated with a number of entities across the platform, including its metadata, code, running containers, and "warm queue". Function metadata is the source of truth for function existence and is defined by four fields:

1) Function Identifier - Function identifiers are randomly generated GUIDs assigned during function creation and used to uniquely identify and locate function resources.
2) Language Runtime - A function's language runtime specifies the language of the function's code. Only Node.js functions are currently supported, which is our chosen language because of its availability in all major serverless computing platforms.
3) Memory Size - A function's memory size determines the maximum memory a function's container can consume. The maximum function memory size is currently set at 1GB. The CPU percentage of a function's container is also set proportionally to its memory size.
4) Code Blob URI - A function is created by specifying a memory size and providing a zip archive containing the function's code. This code is copied to a blob inside the platform's storage account, and the URI of that blob is placed in the function's metadata.

Function containers will be discussed in detail below, as will warm queues, which are queues indexed by function identifier which hold the available running containers of each function.

B. Function Execution

Our implementation provides a very basic function programming model and only supports manual invocations. While the processing of event sources and quality of programming constructs are important considerations in serverless offerings, our work focuses on the execution processing of such systems, for which manual execution support is sufficient.

Functions are executed by calling an "/invoke" route off of function resources on the REST API. The request body of these invocation calls is provided to the functions as inputs, and the response body comprises the outputs returned by the function. Execution begins in the web service which receives the invocation call and subsequently retrieves function metadata from table storage. An execution request object is created containing the function metadata and inputs, and then the web service attempts to locate an available container in the worker service to process the execution request.

Interaction between the web and worker services is controlled through a shared messaging layer. Specifically, there is a global "cold queue", as well as a "warm queue" for each function in the platform. These queues hold available container messages, which simply consist of a URI containing the address of the worker instance and the name of the available container. Messages in the cold queue indicate a worker has unallocated memory in which it could start a container, and messages in a function's warm queue mean a worker has a container running that function that is not currently handling an execution request.

The web service first tries to dequeue a message from a function's warm queue. If no messages are found, the web service dequeues a message from the cold queue, which will assign a new container to the function when sent to the worker service. If all workers are fully allocated with running containers, the cold queue will be empty. Therefore, if the

[1]Available: https://github.com/mgarrettm/serverless-prototype

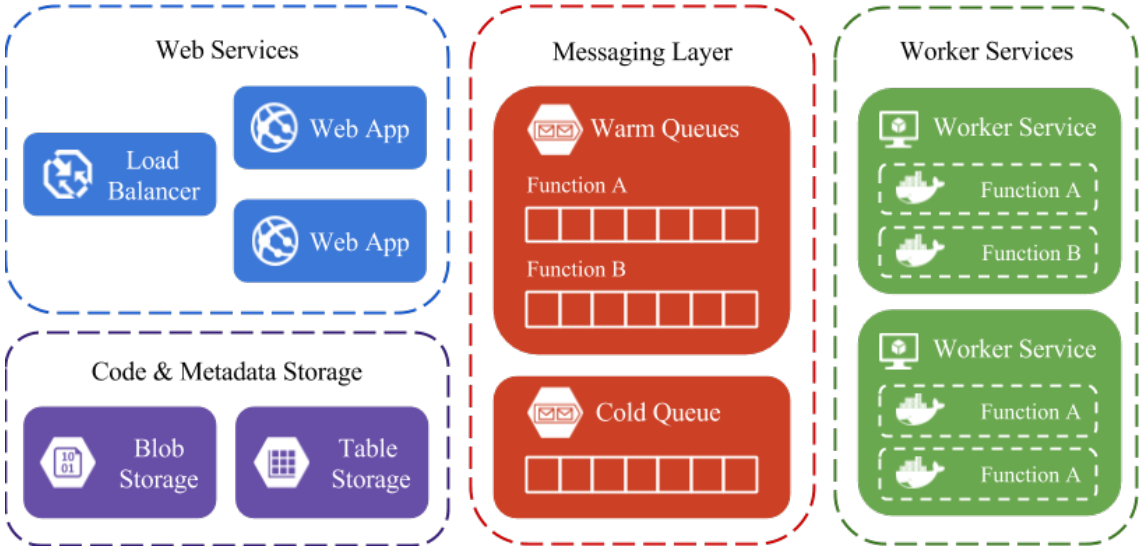

Fig. 1. Overview of prototype components, showing the organization of the web and worker services, as well as code, metadata, and messaging entities in Azure Storage.

web service is unable to find an available container in both the warm queue and cold queue, it will return HTTP 503 Service Unavailable because there are no resources to fulfill the execution request. For this reason, the cold queue is an excellent target for auto-scaling, as it reflects the available space across the platform.

Once a container allocation message is found in a queue, the web service sends an HTTP request to a worker service using the URI contained in the message. The worker then executes the function and returns the function outputs to the web service, which in turn responds to the invocation call.

C. Container Allocation

Each worker manages a count of unallocated memory which it can assign to function containers. When memory is reserved, a container name is generated, which uniquely identifies a container and its memory reservation, and is embedded in the URI sent in container allocation messages. Therefore, each message in the queues is uniquely identifiable and can be associated with a specific memory reservation within a worker service. Memory is allocated conservatively, and worker services assume all functions will consume their memory size.

When container allocations are sent to the cold queue, they have not yet been assigned to a function. Therefore, before this assignment occurs, it is assumed the assigned function will have the maximum function memory size. Then, when a worker service receives an execution request on one of these unassigned allocations, it reclaims memory if the assigned function requires less than the maximum. After the container is created and its function executed for the first time, the container allocation is placed in that function's warm queue.

D. Container Removal

There are two ways a container can be removed. Firstly, when a function is deleted the web service deletes the function's warm queue, which any workers with running instances of that function periodically monitor for existence. If a worker service detects that a function's queue has been deleted, it removes all of that function's running containers and reclaims their memory sizes. Secondly, a container can be removed if it is idle for a period of time. In our implementation, that period has been arbitrarily set at 15 minutes, after which time idle containers are removed, and their memory reclaimed. Whenever memory is reclaimed, worker services send new container allocations to the cold queue if they have enough unallocated memory.

Container expiration has some implications for the web service because it is possible to dequeue an expired container from a function's warm queue. In this case, when the web service sends the execution request the worker service will return HTTP 410 Gone, indicating the container has been removed. The web service will then delete the expired message from the queue and retry.

E. Container Image

Our platform uses Docker to run Windows Nano Server containers and communicates with the Docker service through the Docker Engine API. Our container image is built to include the function runtime (currently only Node.js v6.9.5), and an execution handler. Notably absent from the image is any function code. We do not build containers for each function in the platform, and instead attach a read-only volume containing function code when starting the container. In addition to the read-only volume, the memory size and CPU percentage of the container are set based upon the function's metadata.

The container's execution handler is a simple Node.js file which starts a web server to receive function inputs from the worker service. The worker service sends function inputs to the handler in the request body of an HTTP request, the handler calls the function with the specified inputs, and responds to the worker service with the function outputs.

III. Performance Results

We designed two tests to measure the execution performance of our platform, and to compare its performance to AWS Lambda, Apache OpenWhisk, and Azure Functions. To conduct these experiments, we developed a performance tool[2] used to measure latency and throughput of serverless platforms. This tool uses the Serverless Framework [27] to deploy Node.js functions to the different services, and we built a Serverless Plugin[3] to provide support for our prototype platform as well. This tool deploys a simple test function which immediately completes execution and returns.

This tool sends synchronous invocation calls by executing the function with HTTP events/triggers as supported by the various platforms. For the prototype we simply call the function's invocation route. Manual invocation calls were not used on the other services as they are typically used in the context of development and testing, and we felt a popular production event/trigger such as an HTTP endpoint would better reflect the performance of the other platforms. A 512MB function memory size was used in all platforms except Microsoft Azure, which recently switched to auto-discovering the memory requirements of functions.

For these experiments, the prototype was deployed to Microsoft Azure. The web service was deployed as an API App in Azure App Service, and the worker service was deployed to two DS2_v2 virtual machines running Windows Server 2016. All platform tables, queues, and blobs resided in a single Azure storage account.

Network latencies were not accounted for in our tests, but to mitigate their effects we performed our experiments from virtual machines inside the same datacenter as our target function, except in the case of OpenWhisk, which we measured from Azure's US South-Central region, and from which we observed single-digit ms network latencies to our function endpoint in IBM's US South datacenter.

A. Concurrency Test

Figure 2 shows the results of the concurrency test, which is designed to measure the ability of serverless platforms to performantly scale and execute a function. Our tool maintains invocation calls to the test function by reissuing each request immediately after receiving the response from the previous call. The test begins by maintaining a single invocation call in this way, and every 10 seconds adds an additional concurrent call, up to a maximum of 15 concurrent requests to the test function. This tool measures the number of responses received per second, and should ideally see a linear increase in this rate as the level of concurrency increases. This test was repeated 10 times in each of the platforms.

The prototype demonstrates near-linear scaling between concurrency levels 1 and 14, but sees a significant performance drop at 15 concurrent requests. We are investigating the reasons for this behavior, but believe it to be a resource or implementation issue, rather than a design issue. AWS Lambda appears to scale linearly and exhibits the highest throughput of the commercial platforms at 15 concurrent requests. The performance of Azure Functions is extremely variable, although the throughput reported is quite high in places, outperforming the other platforms at lower concurrency levels. This variability is intriguing, especially because it persists across test iterations. Even more curious is OpenWhisk's performance, which shows low throughput and no scalability until eight concurrent requests, at which point the function suddenly begins to sub-linearly scale. This suggests that OpenWhisk has multiple modes of execution, but more study is needed to understand its scaling strategies.

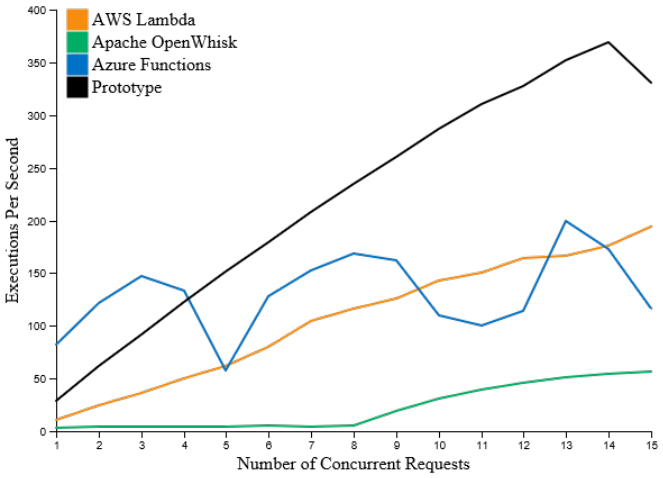

Fig. 2. Concurrency test results, plotting the average number of executions completed per second versus the number of concurrent execution requests to the function.

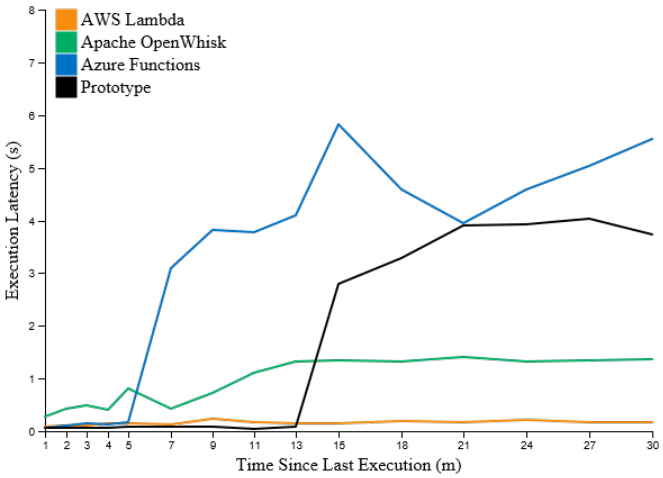

Fig. 3. Backoff test results, plotting the average execution latency of the function versus the time since the function's previous execution.

B. Backoff Test

Figure 3 shows the results of the backoff test, which is designed to study the cold start times and expiration behaviors of function instances in the various platforms. The backoff

[2]Available: https://github.com/mgarrettm/serverless-performance
[3]Available: https://github.com/mgarrettm/serverless-prototype-plugin

test sends single execution requests to the test function at increasing intervals, ranging from one to thirty minutes.

As described in the prototype design, function containers expire after 15 minutes of unuse. Figure 3 shows this behavior, and the execution latencies after 15 minutes show the cold start performance of our prototype. It appears Azure Functions also expires function resources after a few minutes, and exhibits similar cold start times as our prototype. It is important to note, however, that although both our prototype and Azure Functions are Windows implementations, their function execution environments are very different, as our prototype uses Windows containers and Azure Functions runs in Azure App Service. OpenWhisk also appears to deallocate containers after about 10 minutes, and has much lower cold start times than Azure Functions or our prototype. Most notable in this experiment is AWS Lambda, which appears largely unaffected by function idling. Possible explanations for this behavior could be extremely fast container start times or preallocation of containers as considered below in the discussion of Windows container limitations.

IV. Limitations and Future Work

A. Function Immutability

Functions are immutable. In other words, only creation, deletion, and read operations are supported on function resources in the REST API. Immutability is a desirable attribute for the execution logic, but in reality it is useful to have many versions of a function and/or to update a function's existing code. This behavior could be supported by treating function versions as the objects being executed instead of functions as a whole, where functions could then be represented as a current version and a list of past version identifiers, and function versions could store the information currently contained in function metadata. Execution requests would either target a specific version identifier or the latest version by default, and because the latest version is included in the top-level function object, metadata fetching would still only require one table read. Therefore, we believe function updating would be a useful addition to the platform and that supporting multiple function versions would not adversely affect execution performance.

B. Warm Queues vs. Warm Stacks

The fact that the warm queue is a FIFO queue is problematic for container expiration. For example, imagine that a function is under heavy load and has 10 containers allocated for execution. Then, load drops to the point that a single container would be able to handle all of the function's executions. Ideally, the extra 9 containers would expire after a short time, but because of the FIFO nature of the queue, so long as there are 10 executions of the function per container expiration period, all 10 containers will remain allocated to the function. The solution to this issue is of course to use "warm stacks" instead of "warm queues", but Azure Storage does not currently have support for a LIFO queue. This is perhaps the largest issue with our current implementation, however, it may be possible to create a LIFO queue with the same semantics as Azure Storage queues from Azure Storage tables, and is a promising opportunity to improve the design of the prototype. However, potential solutions would need to be evaluated carefully for performance, as the warm queues have a large effect on function execution latency and throughput.

C. Asynchronous Executions

Currently the prototype only supports synchronous invocations. In other words, a request to execute a function will return the result of that function execution, it will not simply start the function and return. However, asynchronous execution is an important feature in a serverless platform and it deserves some treatment here. Asynchronous executions by themselves are simple to support, the web service can simply respond to the invocation call and then process the execution request normally. The difficulty in asynchronous execution is in guaranteeing at-least-once execution rather than best effort execution. It is important to understand that execution is only guaranteed once an invocation request returns with a successful status code. Therefore, with synchronous execution requests (as in our implementation), no further work is needed because a successful status code is only returned once execution has completed. However, asynchronous executions respond to clients before function execution, and it is therefore necessary to have additional logic to ensure these executions complete successfully.

We believe the prototype can support this requirement by tracking active asynchronous executions in a table and introducing a third service responsible for monitoring the status of these table records. Worker services would periodically update active execution records during function execution, and the monitoring service would continually scan the active asynchronous executions table to detect when workers fail by inspecting the last updated times of its records. We aim to explore these solutions in future work.

D. Worker Utilization

A large area for improvement in our implementation is worker utilization. Realistic utilization designs would require an over-allocation of worker resources, with the assumption or observation that not all functions on a worker are constantly executing, or using all of their memory reservation. Studying utilization in a serverless setting is a promising area of research, and presents many interesting tradeoffs between execution performance and operating costs. However, the evaluation of utilization strategies is difficult without representative datasets of function execution in serverless platforms. It would be beneficial for future research if existing platforms were more open about their execution patterns. Alternatively, methods of synthesizing serverless computing loads would also aid research efforts, and defining categories of serverless load is an interesting problem.

E. Windows Containers

Windows containers have some limitations compared to Linux containers, largely because Linux containers were de-

signed around Linux cgroups which support useful operations not available on Windows. Most notably in the context of serverless computing is the support of container resource updating and container pausing. A common pattern in serverless platforms is to pause containers when they are not executing to avoid resource consumption, and then unpause them before executing that container's function again [28].

Another potentially useful operation is container resource updating. Because we allocate space for containers before any executions are required on them, it would be very beneficial for cold start performance if we were able to start containers when space is first reserved for them on a worker service, and then populate container volumes and resize container CPU and memory reservations once an execution request is received. Future work can study how to support these semantics in Windows containers, perhaps by limiting or updating the resources to the function process itself rather than the container as a whole.

F. Security

Security of serverless systems is also an open research question. Hosting arbitrary user code in containers on multi-tenant systems is a dangerous proposition, and care must be taken when constructing and running function containers to prevent security vulnerabilities. This is an intersection of issues related to remote procedure calls (RPC) and container security, and a significant real-world test of general container security. Therefore, although serverless platforms are able to carefully craft the function containers and restrict function permissions arbitrarily, increasing the chances of secure execution, further study is needed to assess the attack vectors within function execution environments.

G. Performance Measures

There are significant opportunities to expand understanding of serverless platform performance, and to define performance measures and tests thereof. This work focused on the latency and throughput of single-function execution, but the quality of these measurements can be improved and other aspects of platform performance such as latency variations between language runtimes or function code sizes, system-wide performance of serverless platforms, and performance differences between event types warrant study.

V. Conclusion

Serverless computing offers powerful, event-driven integrations with numerous cloud services, simple programming and deployment models, and fine-grained scaling and cost management. Driven by these benefits, the growing adoption of serverless applications warrants the evaluation of serverless platform quality, and the development of new techniques to maximize the technology's potential. The performance results of our platform are encouraging, and our analysis of the current implementation presents many opportunities for continued development and study. We hope to see increased interest in serverless computing by academia, and increased openness by the industry leaders for the wider benefit of serverless technologies.

References

[1] "AWS Lambda," https://aws.amazon.com/lambda/, 2017.
[2] "Apache OpenWhisk," http://openwhisk.org/, 2017.
[3] "Azure Functions," https://azure.microsoft.com/en-us/services/functions/, 2017.
[4] "Google Cloud Functions," https://cloud.google.com/functions/, 2017.
[5] "Iron.io IronFunctions," http://open.iron.io/, 2017.
[6] "OpenLambda," https://open-lambda.org/, 2017.
[7] K. uwe Schmidt, D. Anicic, and R. Sthmer, "Event-driven reactivity: A survey and requirements analysis," in *In 3rd International Workshop on Semantic Business Process Management*, 2008, pp. 72–86.
[8] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and P. Suter, "Cloud-native, event-based programming for mobile applications," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA: ACM, 2016, pp. 287–288.
[9] I. Dwyer, "Serverless computing: Developer empowerment reaches new heights," http://cdn2.hubspot.net/hubfs/553779/PDFs/Whitepaper_Serverless_Screen_Final_V2.pdf, 2016.
[10] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 33–39.
[11] R. Vojta, "AWS journey: API Gateway & Lambda & VPC performance," https://robertvojta.com/aws-journey-api-gateway-lambda-vpc-performance-452c6932093b.
[12] E. Jonas, "Microservices and Teraflops," http://ericjonas.com/pywren.html, 2016.
[13] E. d. Lara, C. S. Gomes, S. Langridge, S. H. Mortazavi, and M. Roodi, "Poster abstract: Hierarchical serverless computing for the mobile edge," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, Oct 2016, pp. 109–110.
[14] "AWS Lambda@Edge," http://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html, 2017.
[15] "AWS Greengrass," https://aws.amazon.com/greengrass/, 2017.
[16] "AWS Step Functions," https://aws.amazon.com/step-functions/, 2017.
[17] G. McGrath, J. Short, S. Ennis, B. Judson, and P. Brenner, "Cloud event programming paradigms: Applications and analysis," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 400–406.
[18] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, ser. MOTA '16. New York, NY, USA: ACM, 2016, pp. 5:1–5:4.
[19] "Lambdash: Run sh commands inside AWS Lambda environment," https://github.com/alestic/lambdash, 2017.
[20] "Apex: Serverless Architecture," http://apex.run/, 2017.
[21] "Sparta: A Go framework for AWS Lambda microservices," http://gosparta.io/, 2017.
[22] M. Villamizar, O. Garcs, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 179–182.
[23] B. Wagner and A. Sood, "Economics of Resilient Cloud Services," *ArXiv e-prints*, Jul. 2016.
[24] A. Warzon, "AWS Lambda pricing in context: A comparison to EC2," https://www.trek10.com/blog/lambda-cost/, 2016.
[25] C. Lowery, "Emerging Technology Analysis: Serverless Computing and Function Platform as a Service," Gartner, Tech. Rep., September 2016.
[26] J. S. Hammond, J. R. Rymer, C. Mines, R. Heffner, D. Bartoletti, C. Tajima, and R. Birrell, "How To Capture The Benefits Of Microservice Design," Forrester Research, Tech. Rep., May 2016.
[27] "Serverless Framework," https://serverless.com/, 2017.
[28] T. Wagner, "Understanding container reuse in AWS Lambda," https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, 2014.