

Serverless Computing: Design, Implementation, and Performance

Dr. Ranjan Kumar Mondal

Assistant Professor, Department of CSE, Swami
Vivekananda University, Kolkata, WB

ranjankm@svu.ac.in

Abstract

We introduce the architecture of a brand-new, performance-focused serverless computing platform built with .NET, hosted on Microsoft Azure, and using Windows containers for function execution. We detail implementation difficulties, including function scaling, container lifespan, reuse, and discovery. We describe metrics to assess serverless platform execution performance and run experiments on our prototype, AWS Lambda, Azure Functions, Google Cloud Functions, and IBM's Apache OpenWhisk deployment. We analyze the scaling and instance expiration trends in the implementations and find that; at most concurrent levels, the prototype outperforms competing platforms in terms of throughput. We also highlight future research to explore the shortcomings and limits of our existing design and offer potential solutions.

Introduction

Serverless computing is a cloud service that divides application logic into functions and runs in response to events. It was first introduced by AWS Lambda, and other services like Apache OpenWhisk, Azure Functions, Google Cloud Functions, Iron.io IronFunctions, and OpenLambda have followed suit. These events enable developers to quickly assemble applications that are dispersed over several services inside a cloud. Sources outside the cloud platform can trigger them, but they also frequently happen internally between the platform's service offerings.

An event-driven ideal, in which programs are defined by actions and the events that cause them, is partially realized by serverless computing. The event-driven literature has long postulated broad computing systems in which operations are handled reactively to event streams, and this terminology is suggestive of active database systems. These concepts are wholly embraced by serverless function platforms, which extend event-processing logic throughout their clouds and define actions using straightforward function abstractions. The OpenWhisk platform from IBM, now known as Apache OpenWhisk, reflects these ideas strongly by clearly defining functions in terms of event, trigger, and action.

Design conversations go from the event-driven base to software development techniques that utilize function-centric infrastructure and container management. When execution is needed, Iron.io pulls and runs function containers stored in private registries using Docker. An investigation of the scalability benefits of serverless computing and performance analysis of several container transitions are presented via peer work on the OpenLambda platform. Other performance assessments have tested AWS Lambda's potential for embarrassingly parallel high-performance scientific computing, and they have examined the effects of VPC and language runtime on AWS Lambda start times. Serverless computing is a suitable match for Internet of Things applications, which intersects with the edge/fog computing infrastructure discussion. To facilitate the anticipated growth of IoT devices, attempts are being made to include serverless computing in a "hierarchy of data centers." With the launch of their Lambda@Edge product, AWS has entered this market by enabling application developers to install restricted Lambda functions in edge nodes. AWS has also been working on various serverless computing extensions, such as Greengrass, which offers a standard programming paradigm for Lambda and IoT services. By breaking down massive programs into smaller functions, serverless computing enables application developers to scale individual components of the application. However, this creates a new challenge in the cohesive administration of various functions. AWS recently introduced Step Functions, which allows for more accessible organization and visualization of function interaction.

One area of ongoing development is the application of serverless computing. In our earlier research on serverless computing, we deployed monolithic applications on serverless platforms and examined serverless programming paradigms, including function cascades. The design of scalable chatbots on serverless systems has been discussed in other research. Several initiatives aim to increase the capabilities of current serverless technologies. Developers may now efficiently execute shell commands in AWS Lambda containers and explore the Lambda runtime environment thanks to Lambdash. Thanks to other initiatives like Apex and Sparta, users can now deploy functions to AWS Lambda in languages like Go that are not natively supported.

Many works highlight cost-saving opportunities in deploying microservices to serverless platforms rather than developing traditional applications, supporting serverless computing as a cost-saving solution. The points at which serverless or virtual machine deployments become more cost-effective have been attempted to be calculated by others. According to Gartner, “the value of [serverless computing] has been demonstrated, maps naturally to microservice software architecture, and is on a trajectory of increased growth and adoption”; serverless computing is becoming increasingly significant. According to Forrester, serverless computing results from “today’s PaaS investments” and is the next iteration of cloud service abstractions. Several mobile and Internet of Things apps are powered by serverless computing, rapidly spreading across various cloud providers. Ensuring the fundamental performance characteristics of serverless platforms remain solid is crucial as its scope and popularity grow. By describing the deployment of a new serverless platform with an emphasis on performance and contrasting its results with those of current products, we want to contribute to this endeavor.

Prototype Design

To research serverless implementation considerations and offer a standard against which to compare other platforms, we have created a performance-focused serverless computing platform 1. The platform features a modest feature set and a straightforward architecture. It was developed in .NET

and delivered on Microsoft Azure. The message layer and data durability of the prototype rely on Azure Storage. Our implementation comprises two parts besides Azure Storage services: a worker service that runs and maintains function containers and a web service that makes the platform's public REST API accessible. The web application uses the communications layer of many Azure Storage queues to find available workers. Azure Storage tables are used to store function information, while Azure Storage blobs are used to store function code.

Azure Storage was selected because it offers highly scalable, low-latency storage primitives via an easy-to-use API, which is in line with this implementation's objectives. For brevity, these storage entities will be referred to as queues, tables, and blobs, with the understanding that these terminologies relate to the corresponding Azure Storage services in the context of this research.

Function Metadata

- Across the platform, a function is linked to several things, such as its code, operating containers, metadata, and "warm queue." Four fields form function metadata, which is the source of truth for a function's existence:
 - **Function Identifier:** Function identifiers are randomly generated GUIDs issued during function formation to find and uniquely identify function resources.
 - **Language Runtime:** The language used in a function's code is specified by its language runtime. Just
 - Our language of choice, Node.js, is presently supported on all significant serverless computing platforms, so we selected it.

Memory Size:

The maximum amount of memory a function's container can use is determined by size. Currently, 1 GB is the maximum amount of function memory that may be used. A function's memory size determines how many CPU cores are allotted to its container.

Code Blob URI: When a function is created, a zip archive with the function's code is delivered. The URI of the blob that contains this code is added to the function's metadata. The blob is located within the platform's storage account.

we will discuss function containers and warm queues in more depth below. Warm queues are indexed by function identifier and include the running containers accessible for each function.

Function Execution

Our solution only allows manual invocations and offers a minimal style of function programming. Our study focuses on the execution processing of such systems, for which manual execution assistance is sufficient, even if the processing of event sources and the quality of programming constructs are significant factors in serverless offers.

Calling a `"/invoke"` route from function resources on the REST API is how functions are carried out. The functions get the call request bodies from the invocation as inputs, and the function outputs are contained in the response bodies. The web service initiates execution by receiving the invocation calls and obtaining function metadata from table storage. The function information and inputs are included in an execution request object produced by the web service. Next, the web service looks for an available container in the worker service to execute the execution request.

The shared messaging layer controls how the web and worker services interact. More specifically, every platform function has a global `"warm queue"` and a `"cold queue."` These queues carry messages about available containers, which are just URIs with the name of the accessible container and the address of the worker instance. Visible messages in a function's warm queue indicate that the function's containers are not currently handling execution requests. In contrast, messages in the cold queue suggest that a worker has unallocated memory in which it may start a container.

Initially, the web service attempts to dequeue a message from the warm queue of a function. The web service dequeues a

message from the cold queue, which, when forwarded to the worker service, will allocate a new container to the function if no messages are detected. The cold queue will be empty if every worker has a running container assigned to them. Consequently, the web service will return HTTP 503 Service if it cannot locate an available container in the warm or cold queue. Unavailable because the available resources cannot fulfill the execution request. The cold queue is an excellent target for auto-scaling because it represents the available space throughout the platform.

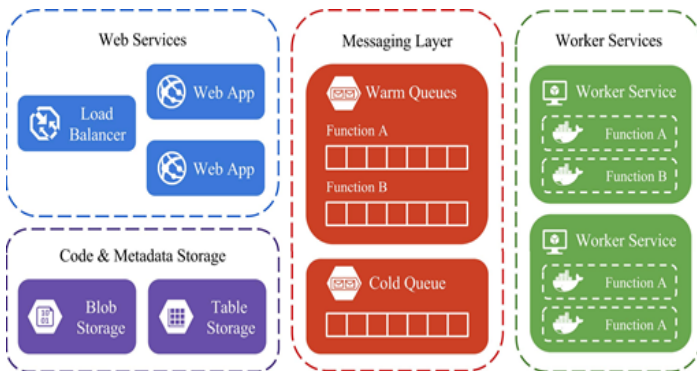


Fig. 1. Overview of prototype components, showing the organization of the web and worker services and code, metadata, and messaging entities in Azure Storage.

After discovering a container allocation message in a queue, the web service uses the message's URI to submit an HTTP request to a worker service. The worker then runs the function and sends its outputs back to the web service, which replies to the call for invocation.

Container Allocation

Every worker has a pool of memory that is not yet allocated, which it may use to allot function containers. A container name, produced when memory is reserved and included in the URI given in container allocation messages, uniquely identifies a container and its memory reservation. As a result, every message in the queues may be individually identified and linked to a particular memory reservation inside a worker

service instance. Worker services presume memory will be used for all functions and is allocated conservatively.

Allocations for containers are not yet allocated to a function when sent to the cold queue. The allocated function is believed to have the maximum function memory size to prevent employees from over-provisioning their memory pool. Then, if the assigned function uses less memory than the maximum size, a worker service reclaims memory upon receiving an execution request for an unassigned allocation. The container allocation message is added to the function's warm queue once the container has been constructed and its function has been used for the first time.

Container Removal

It is possible to delete a container in two ways. First, when a function is removed, the web service also removes its warm queue, which is checked for existence regularly by the worker service instances holding the function's containers. A worker service will eliminate the running containers for a deleted function and recover any memory reservations if it finds evidence of the deleted function queue. Second, our design allows for the removal of a container and the recovery of its memory if it remains idle for a certain amount of time—15 minutes, in our case. If worker services' unused memory surpasses the maximum function memory limit, they will send fresh container allocations to the cold queue whenever memory is reclaimed. Since an expired container might be dequeued from a function's warm queue, container expiration affects the web service. In this instance, the worker service will respond to HTTP 404 Not Found when the web service executes the request. After that, the web service will attempt again after removing the expired message from the queue.

Container Image

The platform runs Windows Nano Server containers with Docker and uses the Docker Engine API to interface with the service. The function runtime (currently limited to Node.js v6.9.5) and an execution handler are included in the container image by design. The lack of any function code in the graphic is noticeable. Instead of creating custom containers for every

platform function, we start the container with a read-only volume that contains the function code. Multiple factors led to the decision to go with a single-image design: managing a single image is more accessible, attaching volumes is a quick process, and Windows Nano Server container images are much bigger than those of lightweight Linux distributions like Alpine Linux, which impacts both startup times and storage expenses. The memory size and CPU percentage of the container are proportionally set based on the memory size of the function, in addition to the read-only volume.

The worker service provides function inputs to the straightforward Node.js server that serves as the container's execution handler. The handler executes the function with the given inputs and returns the function outputs to the worker service after receiving function inputs from the worker service in the request body of an HTTP request. Because containers are added to the default "nat" network—Windows's version of the Linux container "bridge" network—they are addressable on the worker service's local area network (LAN).

Performance Results

We created two tests to gauge the performance of our solution using AWS Lambda, Azure Functions, Google Cloud Functions, and Apache OpenWhisk. To carry out these tests, we created a performance tool that uses the Serverless Framework to distribute a Node.js test function to the various services [28]. We additionally developed a Serverless Framework plugin to allow compatibility with our platform.

This tool uses a simple test function that executes quickly and returns to measure the overhead provided by the platforms. This function is called synchronously via our platform's function invocation route and HTTP events/triggers supported by the different platforms. Since the other services are usually considered development and testing routes, we did not employ manual invocation calls; instead, we felt that a widely used production event or trigger, such as an HTTP endpoint, would more accurately represent current platform performance. Except for Microsoft Azure, which dynamically determines the memory requirements of functions, all systems utilized a 512MB function memory size.

The worker service in the prototype consisted of two DS2 v2 virtual machines running Windows Server 2016, while the web service was an API App in Azure App Service. The prototype was deployed in Microsoft Azure. A single Azure storage account held all platform tables, queues, and blobs.

We conducted our experiments from virtual machines within the same region as our target function, except OpenWhisk, where we measured from Azure's South Central US region and observed single-digit millisecond network latencies to our function endpoint in IBM's US South region. Although our tests did not consider network latencies, we minimized their effects. *Concurrency Test*

The concurrency test, intended to gauge how well serverless systems can efficiently call a function at scale, yields findings in Figure 2. By reissuing each request as soon as the preceding call's answer is received, our tool maintains invocation calls to the test function. In this manner, the test starts with a single invocation call and adds a new concurrent call every 10 seconds for up to 15 simultaneous calls to the test function. The tool counts the number of replies it receives in a second, and when concurrency increases, so should the count. On every platform, this test was run ten times.

The prototype shows almost linear scalability from level 1 to level 14; however, at 15 concurrent requests, performance significantly degrades. Increased latencies from the warm queue, which show that the load is getting close to the scalability limits of a single Azure Storage queue, cause this decline [29]. With 15 simultaneous queries, AWS Lambda has the highest throughput of all the commercial platforms and seems to increase linearly.

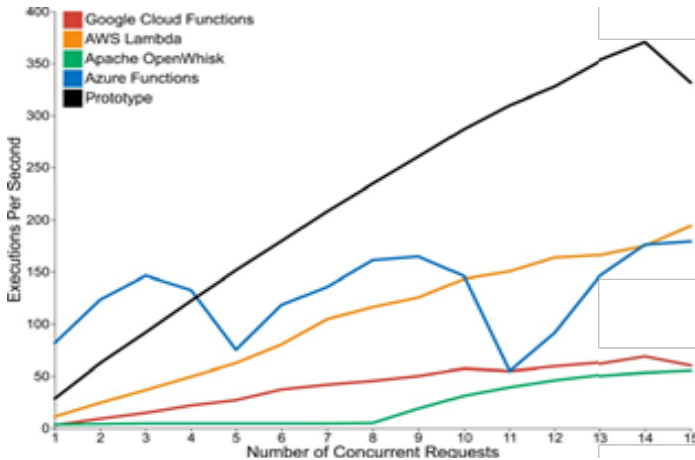


Fig. 2. Concurrency test results, plotting the average number of executions completed per second versus the number of concurrent execution requests to the function.

Sub-linear scaling is seen in Google Cloud Functions, and it seems to taper off when the number of concurrent requests approaches 15. Azure Functions’ performance is inconsistent, even if the recorded throughput is relatively good in some areas and outperforms other platforms at lower concurrent levels. Interestingly, this diversity holds throughout test runs. The peculiar behavior of OpenWhisk’s throughput is evident until eight simultaneous requests, beyond which the function starts to scale sub-linearly. This behavior relies on IBM’s deployment settings, although it might be caused by OpenWhisk’s container pool launching several containers before reusing them.

Figure 3 displays the outcomes of the backoff test, which is intended to investigate function instances’ cold start times and expiry behaviors across the different platforms. For one to thirty minutes, the backoff test sends the test function single execution requests at increasing intervals.

The function containers in the prototype design expire 15 minutes after not being used. This behavior is depicted in Figure 3, and the execution latencies demonstrate

our prototype's cold start performance after 15 minutes. Like our prototype, Azure Functions also has cold start timers and expires function resources after a few minutes. While our prototype and Azure Functions are Windows implementations, it's crucial to remember that their function execution contexts differ significantly since Azure Functions operate in Azure App Service while our prototype utilizes Windows containers. In addition, OpenWhisk offers far shorter cold start times than Azure Functions or our prototype, and it seems to deallocate containers after around ten minutes. Most remarkably, function idling doesn't seem to impact Google Cloud Functions or AWS Lambda. As discussed in the section on Windows containers below, rapid container start times or container preallocation are two possible causes for this phenomenon.

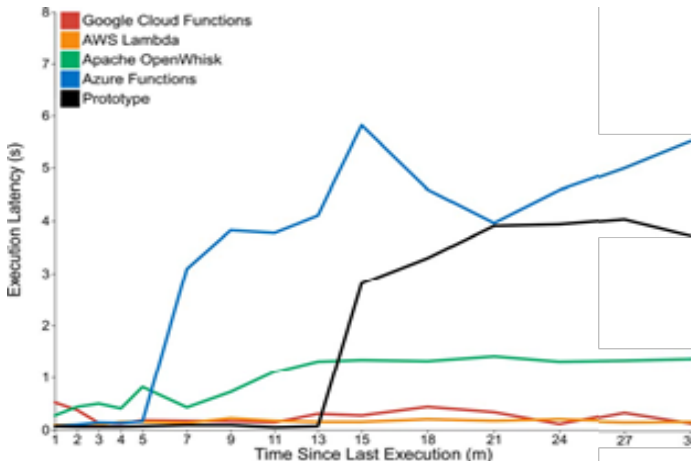


Fig. 3. Backoff test results, plotting the average execution latency of the function versus the time since the function's previous execution.

Limitations And Future Work

Warm Queues

The warm queue is a FIFO queue, so container expiry is an issue. Envision a scenario where a function experiencing high demand is assigned to ten containers for execution. Gradually,

the load is reduced to the point where a single container can manage every function execution. The extra nine containers should ideally expire soon. Still, due to the FIFO queue, all of the containers will be assigned to the function as long as there are ten executions of the function for each container expiration period. The workaround is, of course, to use “warm stacks” rather than “warm queues,” however, LIFO queues are not supported by Azure Storage at this time. This is perhaps the biggest problem with our present setup, although a consistent hashing solution or alternative warm stack storage choices show promise.

Asynchronous Executions

As of right now, the prototype can only handle synchronous calls. Put another way, a request to run a function will return the outcome of that execution rather than just starting and stopping it. Supporting asynchronous executions is straightforward; all that has to be done is for the web service to reply to the invocation call and proceed with processing the execution request as usual. Ensuring at least one execution, as opposed to best-effort execution, is the challenge with asynchronous execution. It’s critical to realize that execution, whether synchronous or asynchronous, is only assured when an invocation request receives a successful status code back. Since a successful status code is only provided after execution, no additional work is required for synchronous execution requests (like in our implementation). Nevertheless, extra logic is needed to guarantee the practical completion of asynchronous executions as they react to clients before executing functions. By keeping track of ongoing executions in a series of queues and adding a third service to oversee the condition of these queue messages, we think the prototype can fulfill this need. When a function is being executed, worker services would continuously update message visibility delays, and the monitoring service would look for visible messages to identify problems. Then, unsuccessful messages could be sent again. Remember that this is about handling platform execution problems rather than exceptions that the function throws while it is being executed, for which you would also want to retry.

Worker Utilization

The warm queue is a FIFO queue, so container expiry is an issue. Envision a scenario where a function experiencing high demand is assigned to ten containers for execution. Gradually, the load is reduced to the point where a single container can manage every function execution. The extra nine containers should ideally expire soon. Still, due to the FIFO queue, all of the containers will be assigned to the function as long as there are ten executions of the function for each container expiration period. The workaround is, of course, to use “warm stacks” rather than “warm queues,” however, LIFO queues are not supported by Azure Storage at this time. This is perhaps the biggest problem with our present setup, although a consistent hashing solution or alternative warm stack storage choices show promise.

Windows Containers

Compared to Linux containers, Windows containers are somewhat more limited. This is mainly because Linux containers were designed to work around Linux cgroups, which provide helpful functions that Windows does not allow. The ability to enable container resource updates and stopping is particularly noteworthy in serverless computing. Implementations of serverless platforms frequently pause containers while inactive to avoid consuming resources and unpauses them before execution restarts.

Updating container resources is an additional procedure that may be helpful. It would help cold start performance if we could start containers before they are allocated to a function and then resize the container after an execution request is received. This is because we reserve resources for containers before executions begin. Subsequent research endeavors may examine methods for facilitating these semantics within Windows containers, maybe through resource limitation or updating to the function process alone instead of the container whole. Alternatively, the prototype might compare start-up times and assess the feasibility of container resizing during cold starts by testing Linux containers.

Security

Serverless system security is another unanswered research topic. It is risky to host arbitrary user code in function containers on multi-tenant systems; hence, caution must be exercised while building and utilizing function containers to

Stop weak points. A major practical test of universal container security is this point where container security and remote procedure calls (RPC) meet. Therefore, further research is required to evaluate the attack surface inside function execution contexts, even though serverless platforms can carefully design the function containers and restrict function permissions arbitrarily, increasing the likelihood of secure execution.

Performance Measures

Defining performance metrics and tests for serverless platforms can improve our understanding of their performance. Although the platforms' overhead during single-function execution was the main focus of this work, better management of network latencies and timing considerations should enhance the accuracy of these evaluations. Further research is necessary to understand other elements of platform performance, such as latency changes across language runtimes and function code size, serverless platform system-wide performance, performance variations between event kinds, and CPU allocation scaling.

Conclusion

Powerful, event-driven integrations with several cloud services, straightforward deployment, and programming paradigms, and fine-grained pricing and scale control are all provided by serverless computing. These advantages have led to an increase in the use of serverless applications, which calls for assessing the quality of serverless platforms and creating new methods to realize the promise of the technology. The performance outcomes of our platform are promising, and our examination of the current setup offers several chances for further research and development. We anticipate seeing more academics interested in serverless computing and more

industry executives willing to discuss the broader benefits of serverless technology.

References

Amazon Web Services, "AWS Lambda," Available: <https://aws.amazon.com/lambda/>, 2017.

The Apache Software Foundation, "Apache OpenWhisk," Available: <http://openwhisk.org/>, 2017.

Microsoft, "Azure Functions," Available: <https://azure.microsoft.com/en-us/services/functions/>, 2017.

Google, "Google Cloud Functions," Available: <https://cloud.google.com/functions/>, 2017.

Iron.io, "Iron.io IronFunctions," Available: <http://open.iron.io/>, 2017.

OpenLambda, "OpenLambda," Available: <https://open-lambda.org/>, 2017.

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.

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.

I. Dwyer, "Serverless computing: Developer empowerment reaches new heights," Available: [http://cdn2.hubspot.net/hubfs/553779/PDFs/Whitepaper Serverless Screen Final V2.pdf](http://cdn2.hubspot.net/hubfs/553779/PDFs/Whitepaper%20Serverless%20Screen%20Final%20V2.pdf), 2016.

S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with open lambda," 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.

[R. Vojta, “AWS journey: API Gateway & Lambda & VPC performance,” Available: <https://robertvojta.com/aws-journey-API-gateway-lambda-vpc-performance-452c6932093b>, 2016.

E. Jonas, “Microservices and Teraflops,” Available: <http://ericjonas.com/pywren.html>, 2016.

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.

Amazon Web Services, “AWS Lambda@Edge,” Available: <http://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html>, 2017.

—, “AWS Greengrass,” Available: <https://aws.amazon.com/greengrass/>, 2017.

—, “AWS Step Functions,” Available: <https://aws.amazon.com/step-functions/>, 2017.

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.

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.

E. Hammond, “Lambdash: Run sh commands inside AWS Lambda environment,” Available: <https://github.com/alestic/lambdash>, 2017.

Apex, “Apex: Serverless Architecture,” Available: <http://apex.run/>, 2017.

Sparta, “Sparta: A Go framework for AWS Lambda microservices,” Available: <http://gosparta.io/>, 2017.

M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casillas, 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.

B. Wagner and A. Sood, “Economics of Resilient Cloud Services,” ArXiv e-prints, Jul. 2016.

A. Warzon, “AWS Lambda pricing in context: A comparison to EC2,”

Available: <https://www.trek10.com/blog/lambda-cost/>, 2016.

C. Lowery, “Emerging Technology Analysis: Serverless Computing and Function Platform as a Service,” Gartner, Tech. Rep., September 2016.

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.

B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, “Windows azure storage: A highly available cloud storage service with strong consistency,” in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ser. SOSP ’11. New York, NY, USA: ACM, 2011, pp. 143–157.

Serverless, Inc., “Serverless Framework,” Available: <https://serverless.com/>, 2017.

Microsoft, “Azure Storage Scalability and Performance Targets,” Available: <https://docs.microsoft.com/en-us/azure/storage/storage-scalability-targets>, March 2017.

Redis, “Redis,” Available: <https://redis.io/>, 2017.

I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for Internet applications,” in Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ser. SIGCOMM ’01. New York, NY, USA: ACM, 2001, pp. 149–160.

T. Wagner, "Understanding container reuse in AWS Lambda," Available:<https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/>,2014.