Serverless Computation with OpenLambda

SCOTT HENDRICKSON, STEPHEN STURDEVANT, EDWARD OAKES,
TYLER HARTER, VENKATESHWARAN VENKATARAMANI,
ANDREA C. ARPACI-DUSSEAU, AND REMZI H. ARPACI-DUSSEAU

Scott Hendrickson is a Software Engineer at Google, working on the future of wireless Internet infrastructure. Formerly, he was a student at the University of Wisconsin-Madison where he participated in the OpenLambda project while working on his Bachelor of Science in computer engineering and computer science. research@shendrickson.com

Stephen Sturdevant is a student at the University of Wisconsin-Madison, where he has received his bachelor's degree and is currently pursuing a computer science PhD. Professors Andrea and Remzi Arpaci-Dusseau are his advisers. Stephen has recently participated in Google Summer of Code and is a contributor to the OpenLambda project. stephensturdevant@gmail.com

Edward Oakes is a student at the University of Wisconsin-Madison, where he is pursuing an undergraduate degree in computer science and mathematics. He is advised by Professor Remzi Arpaci-Dusseau. This fall, Edward will be continuing his work on OpenLambda at the Microsoft Gray Systems Lab in Madison, WI. oakes@cs.wisc.edu

Tyler Harter is a recent PhD graduate from the University of Wisconsin-Madison, where he was co-advised by Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau. He has published several storage papers at FAST and SOSP, including an SOSP '11 Best Paper. He is joining the Microsoft Gray Systems Lab soon, where he will continue contributing to OpenLambda (https://github.com/open-lambda). tylerharter@gmail.com

Rapid innovation in the datacenter is once again set to transform how we build, deploy, and manage Web applications. New applications are often built as a composition of microservices, but, as we will show, traditional containers are a poor fit for running these microservices. We argue that new serverless compute platforms, such as AWS Lambda, provide better elasticity. We also introduce OpenLambda, an open-source implementation of the Lambda model, and describe several new research challenges in the area of serverless computing.

In the preconsolidated datacenter, each application often ran on its own physical machine. The high costs of buying and maintaining large numbers of machines, and the fact that each was often underutilized, led to a great leap forward: virtualization. Virtualization enables tremendous consolidation of services onto servers, thus greatly reducing costs and improving manageability.

However, hardware-based virtualization is not a panacea, and lighter-weight technologies have arisen to address its fundamental issues. One leading solution in this space is *containers*, a server-oriented repackaging of UNIX-style processes with additional namespace virtualization. Combined with distribution tools such as Docker [8], containers enable developers to readily spin up new services without the slow provisioning and runtime overheads of virtual machines.

Common to both hardware-based and container-based virtualization is the central notion of a *server*. Servers have long been used to back online applications, but new cloud-computing platforms foreshadow the end of the traditional backend server. Servers are notoriously difficult to configure and manage [3], and server startup time severely limits elasticity.

As a result, a new model called *serverless computation* is poised to transform the construction of modern scalable applications. Instead of thinking of applications as collections of servers, developers define applications as a set of functions with access to a common datastore. An excellent example of this *microservice*-based platform is found in Amazon's Lambda [1]; we thus generically refer to this style of service construction as the Lambda model.

The Lambda model has many benefits as compared to traditional server-based approaches. Lambda handlers from different customers share common pools of servers managed by the cloud provider, so developers need not worry about server management. Handlers are typically written in languages such as JavaScript or Python; by sharing the runtime environment across functions, the code specific to a particular application will typically be small and easily deployable on any worker in a Lambda cluster. Finally, applications can scale up rapidly without needing to start new servers. The Lambda model represents the logical conclusion of the evolution of sharing between applications, from hardware to operating systems to (finally) the runtime environments themselves (Figure 1).

Venkat Venkataramani is the CEO and co-founder of Rockset, an infrastructure startup based in Menlo Park building a high performance cloud-first data service. Prior to Rockset, he was an Engineering Director in the Facebook infrastructure team responsible for all online data services that stored and served Facebook user data. Before Facebook, he worked at Oracle on the RDBMS for 5+ years after receiving his master's degree at UW-Madison. venkat@rockset.io

Andrea Arpaci-Dusseau is a Full Professor of Computer Sciences at the University of Wisconsin-Madison. She is an expert in file and storage systems, having published more than 80 papers in this area, co-advised 19 PhD students, and received nine Best Paper awards; for her research contributions, she was recognized as a UW-Madison Vilas Associate. She also created a service-learning course in which UW-Madison students teach CS to more than 200 elementary-school children each semester. dusseau@cs.wisc.edu

Remzi Arpaci-Dusseau is a Full Professor in the Computer Sciences Department at the University of Wisconsin-Madison. He co-leads a group with his wife, Professor Andrea Arpaci-Dusseau. They have graduated 19 PhD students in their time at Wisconsin, won nine Best Paper awards, and some of their innovations now ship in commercial systems and are used daily by millions of people. Remzi has won the SACM Student Choice Professor of the Year award four times, the Carolyn Rosner "Excellent Educator" award, and the UW-Madison Chancellor's Distinguished Teaching award. Chapters from a freely available OS book he and Andrea co-wrote, found at http://www.ostep.org, have been downloaded millions of times in the past few years. remzi@cs.wisc.edu

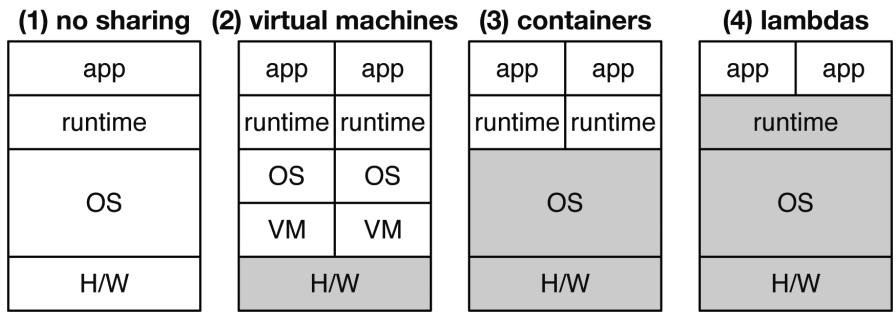

Figure 1: Evolution of sharing. Gray layers are shared.

There are many new research challenges in the context of serverless computing, with respect to efficient sandboxing, cluster scheduling, storage, package management, and many other areas. In order to explore these problems, we are currently building OpenLambda, a base upon which researchers can evaluate new approaches to serverless computing. More details can be found in Hendrickson *et al.* [5]. Furthermore, while research is a primary motivation for building OpenLambda, we plan to build a production-quality platform that could be reasonably deployed by cloud providers.

AWS Lambda Background

AWS Lambda allows developers to specify functions that run in response to various events. We focus on the case where the event is an RPC call from a Web application and the function is an RPC handler. A developer selects a runtime environment (for example, Python 2.7), uploads the handler code, and associates the handler with a URL endpoint. Clients can issue RPC calls by issuing requests to the URL.

Handlers can execute on any worker; in AWS, start-up time on a new worker is approximately one to two seconds. Upon a load burst, a load balancer can start a Lambda handler on a new worker to service a queued RPC call without incurring excessive latencies. However, calls to a particular Lambda are typically sent to the same worker(s) to avoid sandbox reinitialization costs [10]. Developers can specify resource limits on time and memory. In AWS, the cost of an invocation is proportional to the memory cap multiplied by the actual execution time, as rounded up to the nearest 100 ms.

Lambda functions are essentially stateless; if the same handler is invoked on the same worker, common state may be visible between invocations, but no guarantees are provided. Thus, Lambda applications are often used alongside a cloud database.

Motivation for Serverless Compute

A primary advantage of the Lambda model is its ability to quickly and automatically scale the number of workers when load suddenly increases. To demonstrate this, we compare AWS Lambda to a container-based server platform, AWS Elastic Beanstalk (hereafter Elastic BS). On both platforms we run the same benchmark for one minute: the workload maintains 100 outstanding RPC requests and each RPC handler spins for 200 ms.

Figure 2 shows the result: an RPC using AWS Lambda has a median response time of only 1.6 sec, whereas an RPC in Elastic BS often takes 20 sec. While AWS Lambda was able to start 100 unique worker instances within 1.6 sec to serve the requests, all Elastic BS requests were served by the same instance; as a result, each request in Elastic BS had to wait behind 99 other 200 ms requests.

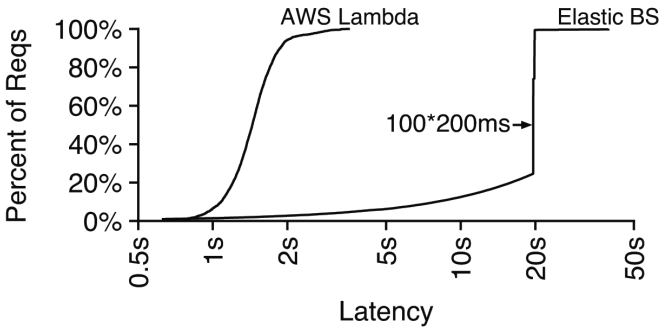

Figure 2: Response time. This CDF shows measured response times from a simulated load burst to an Elastic BS application and to an AWS Lambda application.

AWS Lambda also has the advantage of not requiring configuration for scaling. In contrast, Elastic BS configuration is complex, involving 20 different settings for scaling alone. Even though we tuned Elastic BS to scale as fast as possible (disregarding monetary cost), it still failed to spin up new workers for several minutes.

OpenLambda Overview

We now introduce OpenLambda, our open-source implementation of the Lambda model. Figure 3 illustrates how various servers and users interact in an OpenLambda cluster during the upload of a Lambda function F and a first call to that function. First, a *developer* uploads the Lambda code to the Lambda service, which stores it in a *code store*. Second, a *client* may issue an RPC to the service, via AJAX, gRPC, or some other protocol. A load balancer must decide which worker machine should service the request. In order to implement certain locality heuristics, the balancer may need to request the RPC schema from the code store in order to perform deep inspection on the RPC fields.

The *OpenLambda worker* that receives the request will then fetch the RPC handling code from the code store if it is not already cached locally. The worker will initialize a sandbox in which to run the handler. The handler may issue queries to a distributed database; if the balancer and database are integrated, this will hopefully involve I/O to a local shard.

There are different ways to implement worker sandboxes, but OpenLambda, like AWS Lambda, currently uses containers. Figure 4 shows how the Lambda model avoids common overheads faced by standard container use cases. Normally, each application runs inside a container, with its own server and runtime environment. Thus, application startup often involves deploying runtime engines to new machines and starting new servers. In contrast, servers run outside the containers with the Lambda model, so there is no server spinup overhead. Furthermore, many applications will share a small number of standard runtime engines. Although multiple instances of those runtime engines will run in each sandbox, the runtime engine code will already be on every worker, typically in memory.

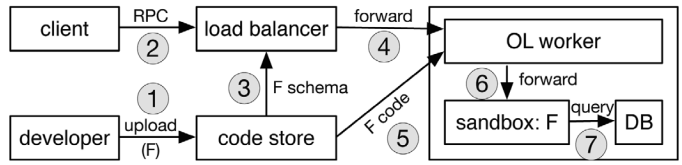

Figure 3: OpenLambda architecture. A new Lambda handler is uploaded, then called.

Tutorial: Running OpenLambda in development mode (with only a worker and no load balancer or code store) is relatively simple in our current pre-release:

```
# build and run standalone OL worker
curl -L -O https://github.com/open-lambda/open-lambda/archive
/v0.1.1.tar.gz
tar -xf v0.1.1.tar.gz
cd open-lambda-0.1.1
./quickstart/deps.sh
make
./bin/worker quickstart/quickstart.json

# from another shell, issue AJAX w/ curl
curl -X POST localhost:8080/runLambda/hello -d
'{"name":"alice"}'
```

Code for new handlers can be written in the `./quickstart/handlers` directory, but the worker must be restarted upon a handler update. RPC calls can be issued via AJAX `curl` POSTs, with the URL updated to reflect the handler name. Directions for running a full OpenLambda cluster are available online: https://www.open-lambda.org.

Research Agenda

We now explore a few of the new research problems in the serverless-computing space.

Lambda Workloads

Characterizing typical Lambda workloads will be key to the design of OpenLambda and other serverless compute platforms. Unfortunately, the Lambda model is relatively new, so there are not yet many applications to study. However, we can anticipate how future workloads may stress Lambda services by analyzing the RPC patterns of existing applications.

In this section, we take Google Gmail as an example and study its RPC calls during inbox load. Gmail uses AJAX calls (an RPC protocol based on HTTP requests that uses JSON to marshal arguments) to fetch dynamic content.

Figure 5 shows Gmail's network I/O over time, divided between GETs and POSTs. Gmail mostly uses POSTs for RPC calls and GETs for other requests; the RPC calls represent 32% of all requests and tend to take longer (92 ms median) than other requests (18 ms median).

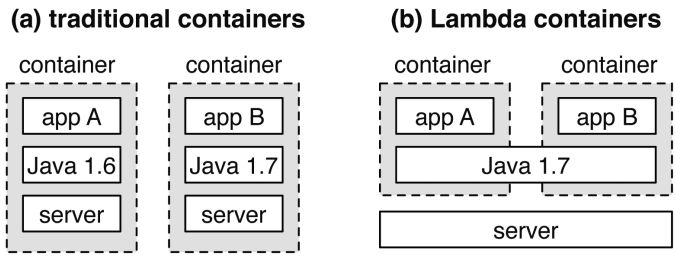

Figure 4: Container usage. The dashed lines represent container boundaries.

The average time for short RPCs (those under 100 ms) is only 27 ms. Since we only trace latency on the client side, we cannot know how long the requests were queued at various stages; thus, our measurements represent an upper bound on the actual time for the RPC handler. On AWS Lambda, charges are in increments of 100 ms, so these requests will cost at least 3.7x more than if charges were more fine-grained.

We also see a very long request that takes 231 seconds, corresponding to 93% of the cumulative time for all requests. Web applications often issue such long-lived RPC calls as a part of a *long polling* technique. When the server wishes to send a message to the client, it simply returns from the long RPC [2]. Unless Lambda services provide special support for these calls, idle handlers will easily dominate monetary costs.

Execution Engine

In the motivation section, we saw that Lambdas are far more elastic than containers. Unfortunately, under steady load, containers tend to be faster. In our experiments [5], Elastic BS request latencies are an order of magnitude shorter that AWS Lambda latencies. If Lambdas are to compete with VM and container platforms, base execution time must be improved.

Optimizing sandbox initialization and management is key to improving Lambda latencies. For example, AWS Lambda reuses the same sandbox for different calls when possible to amortize startup costs; between requests, containers are maintained in a paused state [10].

Unfortunately, there are difficult tradeoffs regarding when to garbage-collect paused containers. Resuming a paused container is over 100x faster than starting a new container, but keeping a container paused imposes the same memory overheads as an active container [5]. Reducing the time cost of fresh starts and reducing memory overheads of paused containers are both interesting challenges.

Interpreted Languages

Most Lambdas are written in interpreted languages. For performance, the runtimes corresponding to these languages typically have just-in-time compilers. JIT compilers have been built for Java, JavaScript, and Python that optimize compiled code based on dynamic profiling or tracing.

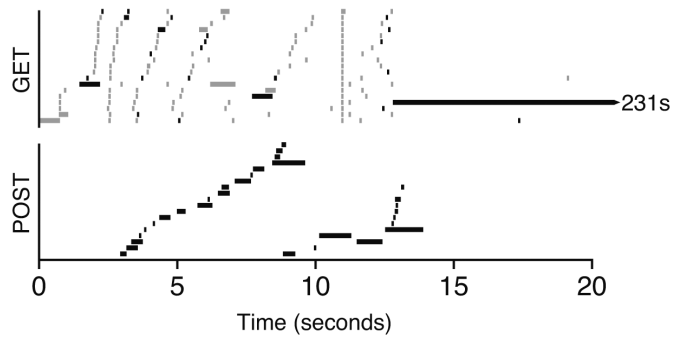

Figure 5: Google Gmail. Black bars represent RPC messages; gray bars represent other messages. The bar ends represent request and response times. The bars are grouped as POSTs and GETs; vertical positioning is otherwise arbitrary.

Applying these techniques with Lambdas is challenging because a single handler may run many times over a long period in a Lambda cluster, but it may not run long enough on any one machine to provide sufficient profiling feedback. Making dynamic optimization effective for Lambdas may require sharing profiling data between different Lambda workers.

Package Support

Lambdas can rapidly spin up because customers are encouraged to use one of a few runtime environments; runtime binaries will already be resident in memory before a handler starts. Of course, this benefit disappears if users bundle large third-party libraries inside their handlers, as the libraries need to be copied over the network upon a handler invocation on a new Lambda worker. Lazily copying packages could partially ameliorate this problem [4].

Alternatively, the Lambda platform could be package aware [7] and provide special support for certain popular package repositories, such as `npm` for Node.js or `pip` for Python. Of course, it would not be feasible to keep such large (and growing) repositories in memory on a single Lambda worker, so package awareness would entail new code locality challenges for scheduling.

Cookies and Sessions

Lambdas are inherently short-lived and stateless, but users typically expect to have many different but related interactions with a Web application. Thus, a Lambda platform should provide a shared view of cookie state across calls originating from a common user.

Furthermore, during a single session, there is often a two-way exchange of data between clients and servers; this exchange is typically facilitated by WebSockets or by long polls. These protocols are challenging for Lambdas because they are based on long-lived TCP connections. If the TCP connections are maintained within a Lambda handler, and a handler is idle between communication, charges to the customer should reflect the

fact that the handler incurs a memory overhead, but consumes no CPU. Alternatively, if the platform provides management of TCP connections outside of the handlers, care must be taken to provide a new Lambda invocation with the connections it needs that were initiated by past invocations.

Databases

There are many opportunities for integrating Lambdas with databases. Most databases support *user-defined functions* (UDFs) for providing a custom view of the data. Lambdas that transform data from a cloud database could be viewed as UDFs that are used by client-side code. Current integration with S3 and DynamoDB also allows Lambdas to act as *trigger* handlers upon inserts.

A new *change feed* abstraction is now supported by RethinkDB and CouchDB; when an iterator reaches the end of a feed, it blocks until there is more data, rather than returning. Supporting change feeds with Lambdas entails many of the same challenges that arise with long-lived sessions; a handler that is blocked waiting for a database update should probably not be charged the same as an active handler. Change-feed batching should also be integrated with Lambda state transitions; it makes sense to batch changes for longer when a Lambda is paused than when it is running.

Relaxed consistency models should also be re-evaluated in the context of RPC handlers. The Lambda compute model introduces new potential consistency boundaries, based not on what data is accessed, but on which actor accesses the data. For example, an application may require that all RPC calls from the same client have a read-after-write guarantee, but weaker guarantees may be acceptable between different clients, even when those clients read from the same entity group.

Data Aggregators

Many applications (search, news feeds, and analytics) involve search queries over large datasets. Parallelism over different data shards is key to efficiently supporting these applications. For example, with search, one may want to scan many inverted indexes in parallel and then gather and aggregate the results.

Building these search applications will likely require special Lambda support. In particular, in order to support the scatter/gather pattern, multiple Lambdas will need to coordinate in a tree structure. Each leaf Lambda will filter and process data locally, and a front-end Lambda will combine the results.

When Lambda leaves are filtering and transforming large shards, it will be important to co-locate the Lambdas with the data. One solution would be to build custom datastores that coordinate with Lambdas. However, the diversity of aggregator applications may drive developers to use variety of platforms for preprocessing the data (for example, MapReduce, Dryad, or Pregel). Thus, defining general locality APIs for coordination with a variety of backends may be necessary.

Load Balancers

Previous low-latency cluster schedulers (such as Sparrow [9]) target tasks in the 100 ms range. Lambda schedulers need to schedule work that is an order of magnitude shorter, while taking several types of locality into account. First, schedulers must consider *session locality*: if a Lambda invocation is part of a long-running session with open TCP connections, it will be beneficial to run the handler on the machine where the TCP connections are maintained so that traffic will not need to be diverted through a proxy.

Second, *code locality* becomes more difficult. A scheduler that is aware that two different handlers rely heavily on the same packages can make better placement decisions. Furthermore, a scheduler may wish to direct requests based on the varying degrees of dynamic optimization achieved on various workers.

Third, *data locality* will be important for running Lambdas alongside either databases or large datasets and indexes. The scheduler will need to anticipate what queries a particular Lambda invocation will issue, or what data it will read. Even once the scheduler knows what data a Lambda will access and where the replicas of the data reside, further communication with the database may be beneficial for choosing the best replica. Many new databases (such as Cassandra or MongoDB) store replicas as LSM trees. Read amplifications for range reads can range from 1x to 50x [6] on different replicas; an integrated scheduler could potentially coordinate with database shards to track these varying costs.

Cost Debugging

Prior platforms cannot provide a cost-per-request for any service. For example, applications that use virtual machine instances are often billed on an hourly basis, and it is not obvious how to divide that cost across the individual requests over an hour. In contrast, it is possible to tell exactly how much each individual RPC call to a Lambda handler costs the cloud customer. This knowledge will enable new types of debugging.

Currently, browser-based developer tools enable performance debugging: tools measure page latency and identify problems by breaking down time by resource. New Lambda-integrated tools could similarly help developers debug monetary cost: the exact cost of visiting a page could be reported, and breakdowns could be provided detailing the cost of each RPC issued by the page as well as the cost of each database operation performed by each Lambda handler.

Legacy Decomposition

Breaking systems and applications into small, manageable subcomponents is a common approach to building robust, parallel software. Decomposition has been applied to operating systems, Web browsers, Web servers, and other applications. In order to save developer effort, there have been many attempts to automate some or all of the modularization process.

Decomposing monolithic Web applications into Lambda-based microservices presents similar challenges and opportunities. There are, however, new opportunities for framework-aware tools to automate the modularization process. Many Web-application frameworks (for example, Flask and Django) use language annotations to associate URLs with handler functions. Such annotations would provide an excellent hint to automatic splitting tools that port legacy applications to the Lambda model.

Conclusion

We have seen that the Lambda model is far more elastic and scalable than previous platforms, including container-based services that autoscale. We have also seen that this new paradigm presents interesting challenges for execution engines, databases, schedulers, and other systems. We believe OpenLambda will create new opportunities for exploring these areas. Furthermore, we hope to make OpenLambda a platform that is suitable for actual cloud developers to deploy their serverless applications. The OpenLambda project is online at https://www.open-lambda.org.

Acknowledgments

Feedback from the anonymous reviewers has significantly improved this work. We also thank the members of the ADSL research group for their helpful suggestions and comments on this work at various stages.

This material was supported by funding from NSF grants CNS-1421033, CNS-1319405, CNS-1218405, CNS-1419199 as well as generous donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Seagate, Samsung, Veritas, and VMware. Tyler Harter is supported by an NSF Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of the NSF or other institutions.

References

[1] "AWS Lambda": https://aws.amazon.com/lambda/, May 2016.

[2] A. Russell, Infrequently Noted (blog), "Comet: Low Latency Data for the Browser" (blog entry), March 2006: https://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/.

[3] J. Gray, "Why Do Computers Stop and What Can We Do About It?" in *Proceedings of the 6th International Conference on Reliability and Distributed Databases,* June 1987: http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf.

[4] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast Distribution with Lazy Docker Containers," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16),* pp. 181–195: https://www.usenix.org/system/files/conference/fast16/fast16-papers-harter.pdf.

[5] 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 Workshop on Hot Topics in Cloud Computing (HotCloud '16):* https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_hendrickson.pdf.

[6] L. Lu, T. Sankaranarayana Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating Keys from Values in SSD-Conscious Storage," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16),* pp. 133–148: https://www.usenix.org/system/files/conference/fast16/fast16-papers-lu.pdf.

[7] M. Boyd, "Amazon Debuts Flourish, a Runtime Appliction Model for Serverless Computing": http://thenewstack.io/amazon-debuts-flourish-runtime-application-model-serverless-computing/, May 2016.

[8] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal,* 2014, no. 239: https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment.

[9] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, Low Latency Scheduling," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (ACM, 2013):* https://people.csail.mit.edu/matei/papers/2013/sosp_sparrow.pdf.

[10] T. Wagner, AWS Compute Blog, "Understanding Container Reuse in AWS Lambda," December 2014: https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.