Serverless Computation with OpenLambda

Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani[†], Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin, Madison [†] Unaffiliated

Abstract

We present *OpenLambda*, a new, open-source platform for building next-generation web services and applications in the burgeoning model of *serverless computation*. We describe the key aspects of serverless computation, and present numerous research challenges that must be addressed in the design and implementation of such systems. We also include a brief study of current web applications, so as to better motivate some aspects of serverless application construction.

1 Introduction

The rapid pace of innovation in datacenters [18] and the software platforms within them is once again set to transform how we build, deploy, and manage online applications and services. In early settings, every application ran on its own physical machine [15, 24]. 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 [20]. 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 [17, Ch. 4] with additional namespace virtualization [39, 40]. Combined with distribution tools such as Docker [39], 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 [27, 46, 47], and server startup time severely limits an application's ability to quickly scale up and down.

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 instead define applications with a set of functions with access to a common data store. An excellent example of this *microservice-*based platform is found in Amazon's Lambda [3]; we thus generically refer to this style of service construction as the Lambda model.

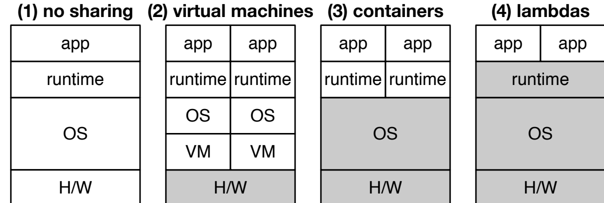

Figure 1: **Evolution of Sharing.** *Gray layers are shared.*

The Lambda model has many benefits as compared to more 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 hence it is inexpensive to send the handler code to any worker in a cluster. Finally, applications can scale up rapidly without needing to start new servers. In this manner, 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).

In this paper, we present the Lambda model and discuss pertinent research challenges. A Lambda execution engine must safely and efficiently isolate handlers (§4.1). Handlers are inherently stateless, so there are many opportunities for integration between Lambda and database services (§4.5). Lambda load balancers must make low-latency decisions while considering session, code, and data locality (§4.7). We further explore new challenges for just-in-time compilation (§4.2), package management (§4.3), web sessions (§4.4), data aggregation, (§4.6), monetary cost (§4.8), and portability (§4.9).

Unfortunately, most existing implementations [3, 6, 8] (except OpenWhisk [7]), are closed and proprietary. In order to facilitate research on Lambda architectures (including our own, and hopefully others), we are currently building OpenLambda, a base upon which researchers can evaluate new approaches to serverless computing (§5). This paper is a first step towards realizing the OpenLambda platform.

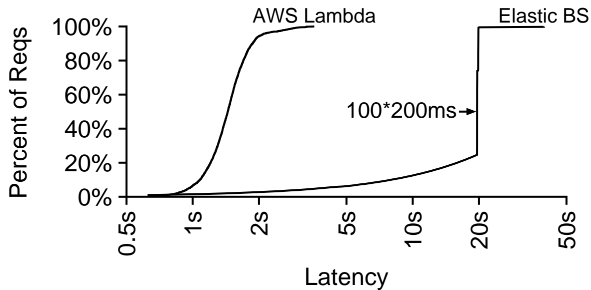

Figure 2: **Response Time.** *This CDF shows measured response times from a simulated load burst to an Elastic BS application and to an AWS Lambda application.*

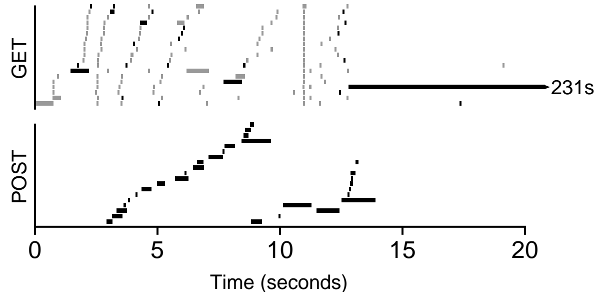

Figure 3: **Google Gmail.** *Black bars represent RPC messages; gray bars represent other messages. The bar ends represent request and response times. The bars are grouped as POSTs and GETs; vertical positioning is otherwise arbitrary.*

2 Lambda Background

To focus our attention on one specific implementation of a Lambda environment, we consider the AWS Lambda cloud platform. We describe the AWS Lambda programming model (§2.1) and demonstrate some of its advantages over server-based models (§2.2).

2.1 Programming Model

The Lambda model 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 uploaded function is an RPC handler. A developer selects a runtime environment (*e.g.*, Python27), uploads the relevant code, and specifies the name of the function that should handle events. The developer can then associate the Lambda with a URL using the separate AWS gateway service [2]. Client-side code can then issue RPC calls by issuing requests to the URL (*e.g.*, JavaScript may POST to the URL via AJAX).

Handlers can execute on any worker; in AWS, startup time for a new worker is approximately 1-2 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 [45].

The developer can bound the resources that may be utilized by a handler (*e.g.*, by setting memory and time caps). In AWS, the cost of an invocation is proportional to the memory cap (not the actual memory consumed) multiplied by the actual execution time, as rounded up to the nearest 100ms.

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.

2.2 Lambda Advantages

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 [4] (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 200ms.

Figure 2 shows the result: an RPC using AWS Lambda has a median response time of only 1.6s, whereas an RPC in Elastic BS often takes 20s. Investigating the cause for this difference, we found that while AWS Lambda was able to start 100 unique worker instances within 1.6s 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 200ms requests.

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.

3 Lambda Workloads

Unfortunately, we do not yet have access to Lambda workloads, as major web services (such as Gmail or Facebook) were built before the serverless paradigm arose. However, we can understand how future workloads may stress Lambda environments by analyzing these existing services. Specifically, we analyze the client-to-server patterns in an existing RPC-based application: Google Gmail. Gmail uses RPCs from client-side JavaScript to fetch dynamic content. JavaScript RPC libraries (*e.g.*, AJAX) are based on the XHR interface [16], which sends a POST or GET request over HTTP to a backend server; arguments and return values are encoded in URLs or message bodies (*e.g.*, with JSON). We trace these RPC calls using a Chrome extension that injects wrappers; we correlate our RPC traces with Chrome's network trace. Our workload consists of refreshing the inbox page (browser caches should be warm).

Figure 3 shows Gmail's network I/O over time, divided between GETs and POSTs. Gmail mostly uses

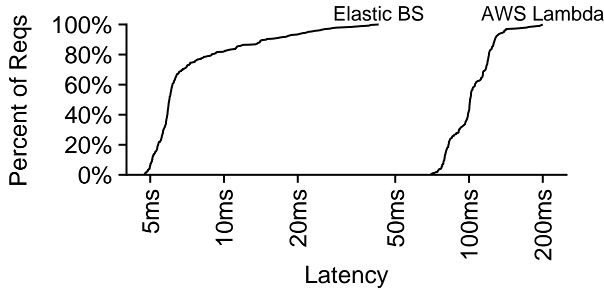

Figure 4: **Containers vs. Lambdas: Latency.** *The lines represent a latency CDF for AWS Lambdas, and containers running in Elastic BS, with both services under low load.*

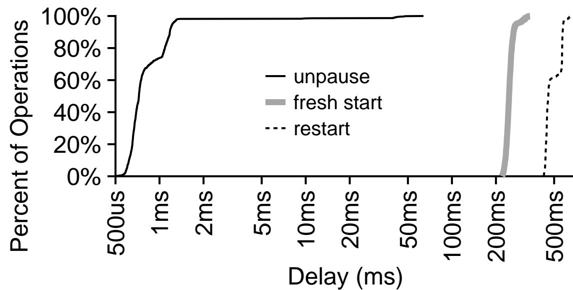

Figure 5: **Readiness Latency.** *The lines represent readiness latency CDFs for three startup transitions.*

POSTs for RPC calls and GETs for other requests; the RPC calls represent 32% of all requests and tend to take longer (92ms median) than other requests (18ms median). We see that there are two categories of RPC requests: very short and very long requests.

The average time for short RPCs (those under 100ms) is only 27ms. 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.

In our measurements, 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 [13].

Design Implications: Many RPCs are shorter than 100ms. On AWS Lambda, charges are in increments of 100ms, so these requests will cost at least 3.7× more than if charges were more fine-grained. One solution would be to design applications to do fewer, longer RPC calls [44]; alternatively, reducing Lambda initialization costs may enable fine-grained accounting. Applications also use long-lived RPCs to support server-side pushes; these calls are presumably blocked, waiting for updates. Unless Lambda environments provide special support for these Lambdas, idle handlers will easily dominate the cost of an application.

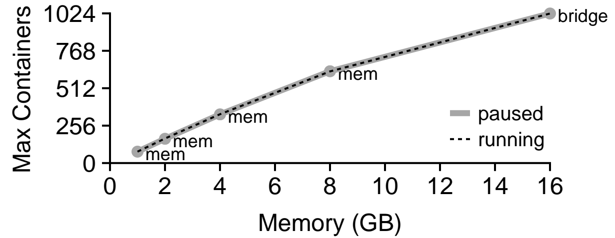

Figure 6: **Container Density.** *The maximum number of containers a machine can run is shown vs. the amount of memory available. Labels identify bottlenecks.*

4 Research Agenda

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

4.1 Execution Engine

A sandbox for executing handlers is at the heart of the Lambda architecture. AWS Lambda uses containers to sandbox handlers [45], but avoids the overheads of Elastic BS and other container-based services by sharing servers and runtimes between different instances.

To amortize container startup costs, AWS Lambda reuses the same container to execute multiple handlers when possible. Unfortunately, even with this optimization, Lambdas are significantly slower than containers at low request volumes. Figure 4 shows results for the same setup as in (§2.2), except for a steady light load instead of a heavy burst. When load is light, latencies with AWS Lambda are ten times worse than with Elastic BS. If Lambdas are to compete with VM and container platforms, base execution time must be improved.

In this section, we explore some of the basic tradeoffs that arise when running Lambdas in containers. In particular, a container must be in a `running` state to handle requests. When there are no requests, a container is either `paused` or `stopped`.

Figure 5 compares the latencies of unpausing (switching from `paused` to `running`) and restarting (switching from `stopped` to `running`) with the latency of a fresh start. Restarting and fresh starting both takes hundreds of milliseconds. In contrast, unpausing takes about 1ms.

Unfortunately, keeping containers `paused` entails a high memory cost. Figure 6 shows how the number of `running` or `paused` containers we can pack on a machine corresponds to available memory. Each data point shows the resource that prevents us from starting new nodes. Memory is the main bottleneck (we believe the network bridge bottleneck could easily be eliminated), and `paused` containers impose the same overhead as `running` containers. Thus, there is a difficult tradeoff between putting non-`running` containers in the `paused` or `stopped` states. Reducing the memory costs in `paused` and reducing the restart costs from `stopped` are both interesting research challenges.

4.2 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 [12], JavaScript [25], and Python [19] that optimize compiled code based on dynamic profiling or tracing of the code as it executes.

Of course, the aggressiveness of these optimizations presents a tradeoff. Expensive profiling may not be worth the cost if the code only runs a short time, so the HotSpot JVM [12] can be tuned to assume short-running or long-running programs. 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.

4.3 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. Such bundling can increase startup latency by an order of magnitude [1]. Lazily copying packages could partially ameliorate this problem [29].

Alternatively, the Lambda platform could be package aware and provide special support for certain popular package repositories, such as `npm` for Node.js [9] or `pip` for Python [10]. 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 (§4.7).

4.4 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 [36].

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 [31], or by long polls, as with Gmail (§3). 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.

4.5 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 allow Lambdas to act as *trigger* handlers upon inserts.

A new *change feed* abstraction is now supported by RethinkDB [11] and CouchDB [14]; 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 (§4.4); 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.

4.6 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 [26].

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 data stores that coordinate with Lambdas. However, the diversity of aggregator applications may drive developers to use variety of platforms for preprocessing the data (*e.g.*, MapReduce [22], Dryad [33], or Pregel [38]). Thus, defining general locality APIs for coordination with a variety of backends may be necessary.

4.7 Load Balancers

Previous low-latency cluster schedulers (*e.g.*, Sparrow [41]) target tasks in the 100ms 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 (§4.4).

Second, *code locality* [42] becomes more difficult. A scheduler that is aware that two different handlers rely heavily on the same packages (§4.3) 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 (§4.2).

Third, *data locality* will be important for running Lambdas alongside either databases (§4.5) or large datasets and indexes (§4.6). 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 (*e.g.*, Cassandra [35] and MongoDB [21, 32]) store replicas as LSM trees. Read amplifications for range reads can range from $1\times$ to $50\times$ [37] on different replicas; an integrated scheduler could potentially coordinate with database shards to track these varying costs.

4.8 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.

4.9 Legacy Decomposition

Breaking systems and applications into small, manageable sub-components is a common approach to building robust, parallel software. Decomposition has been applied to operating systems, web browsers, web servers, and other applications [23, 34, 43]. In order to save developer effort, there have been many attempts to automate some or all of the modularization process [30, 43].

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 (*e.g.*, Flask [28] and Django [5]) 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.

5 Towards OpenLambda

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. In order to facilitate research in these areas, we are building OpenLambda, an open-source implementation of the Lambda platform.

OpenLambda will consist of a number of subsystems that will coordinate to run Lambda handlers: a Lambda store to host and distribute handler code, a local-execution engine that sandboxes handlers, a load balancer to spread requests across workers, and a Lambda-aware distributed database. We further plan to build LambdaBench, a new benchmark suite based on ports of various applications to the Lambda programming model. Our hope is that providing a complete set of all the components making up the Lambda infrastructure will enable researchers to evaluate novel designs and implementations of various subsystems within the serverless computation platform. The OpenLambda project is online at `http://www.open-lambda.org`.

6 Acknowledgements

Feedback from the anonymous reviewers have 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.

7 References

[1] AWS Developer Forums: Java Lambda Inappropriate for Quick Calls? https://forums.aws.amazon.com/thread.jspa?messageID=679050, July 2015.

[2] Amazon API Gateway Client Documentation. http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/apigateway/AmazonApiGatewayClient.html, March 2016.

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

[4] Deploying Elastic Beanstalk Applications from Docker Containers. http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_docker.html, May 2016.

[5] Django. https://www.djangoproject.com/, March 2016.

[6] Google Cloud Functions. https://cloud.google.com/functions/docs/, May 2016.

[7] IBM OpenWhisk. https://developer.ibm.com/openwhisk/, May 2016.

[8] Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions/, May 2016.

[9] Nuclear Powered Macros. https://www.npmjs.com/, May 2016.

[10] Pip Installs Packages. https://pip.pypa.io/en/stable/, May 2016.

[11] ReQL Changes Command Documentation. https://www.rethinkdb.com/api/javascript/changes/, May 2016.

[12] The Java HotSpot Performance Engine Architecture. http://www.oracle.com/technetwork/java/whitepaper-135217.html, May 2016.

[13] Alex Russell. Comet: Low Latency Data for the Browser. https://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/, March 2006.

[14] J. Chris Anderson, Jan Lehnardt, and Noah Slater. CouchDB: The Definitive Guide. http://guide.couchdb.org/draft/notifications.html, May 2016.

[15] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[16] Anne van Kesteren. XMLHttpRequest. https://xhr.spec.whatwg.org/, April 2016.

[17] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[18] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The Datacenter as a Computer: an Introduction to the Design of Warehouse-Scale Machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[19] Carl Friedrich Bolz. Applying a Tracing JIT to an Interpreter. http://morepypy.blogspot.com/2009/03/applying-tracing-jit-to-interpreter.html, March 2009.

[20] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[21] Charity Majors. MongoDB + RocksDB at Parse. http://blog.parse.com/announcements/mongodb-rocksdb-parse/, April 2015.

[22] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 137–150, San Francisco, California, December 2004.

[23] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[24] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 78–91, Saint-Malo, France, October 1997.

[25] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. Trace-based Just-in-Time Type Specialization for Dynamic Languages. *ACM Sigplan Notices*, 44(6):465–478, 2009.

[26] Clinton Gormley and Zachary Tong. Elasticsearch: The Definitive Guide. https://www.elastic.co/guide/en/elasticsearch/guide/current/inverted-index.html, 2015.

[27] Jim Gray. Why Do Computers Stop and What Can We Do About It? In *6th International Conference on Reliability and Distributed Databases*, June 1987.

[28] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc., 1st edition, 2014.

[29] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 181–195, Santa Clara, CA, February 2016. USENIX Association.

[30] Galen Hunt and Michael Scott. The Coign Automatic Distributed Partitioning System. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 187–200, New Orleans, Louisiana, February 1999.

[31] Ian Fette and Alexey Melnikov. The WebSocket Protocol. Technical Report 6455, Internet Engineering Task Force, December 2011.

[32] Igor Canadi. Integrating RocksDB with MongoDB. http://rocksdb.org/blog/1967/integrating-rocksdb-with-mongodb-2/, April 2015.

[33] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the EuroSys Conference (EuroSys '07)*, Lisbon, Portugal, March 2007.

[34] James Mickens and Mohan Dhawan. Atlantis: Robust, Extensible Execution Environments for Web Applications. In *Proceedings of SOSP*. ACM, October 2011.

[35] Avinash Lakshman and Prashant Malik. Cassandra – A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, Big Sky Resort, Montana, Oct 2009.

[36] Bryan Liston. Simply Serverless: Using AWS Lambda to Expose Custom Cookies with API Gateway. https://aws.amazon.com/blogs/compute/simply-serverless-using-aws-lambda-to-expose-custom-cookies-with-api-gateway/, April 2016.

[37] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th*

USENIX Conference on File and Storage Technologies (FAST 16), pages 133–148, Santa Clara, CA, February 2016. USENIX Association.

[38] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[39] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.

[40] Oracle Inc. Consolidating Applications with Oracle Solaris Containers. http://www.oracle.com/technetwork/server-storage/solaris/documentation/consolidating-apps-163572.pdf, Jul 2011.

[41] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[42] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich M. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 205–216, San Jose, California, October 1998.

[43] Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Fractured Processes: Adaptive, Fine-Grained Process Abstractions. In *Proceedings of the 2014 Conference on Timely Results in Operating Systems (TRIOS '14)*, Broomfield, CO, October 2014.

[44] Raj Srinivasan. RPC: Remote procedure call protocol specification version 2. Technical Report 1831, Internet Engineering Task Force, August 1995.

[45] Tim Wagner. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, December 2014.

[46] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration Debugging As Search: Finding the Needle in the Haystack. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[47] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A Smart Hill-Climbing Algorithm for Application Server Configuration. In *Proceedings of the 13th international conference on World Wide Web*, pages 287–296. ACM, 2004.