2016 IEEE 9th International Conference on Cloud Computing

Cloud Event Programming Paradigms

Applications and Analysis

Garrett McGrath, Brenden Judson, Paul Brenner
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, USA
mmcgrat4@nd.edu, bjudson1@nd.edu,
paul.r.brenner@nd.edu

Jared Short, Stephen Ennis
Trek10: Managed Performance Architects for the Cloud
South Bend, USA
jshort@trek10.com, sennis1@nd.edu

***Abstract*—Rapid expansion in cloud event technologies such as Amazon Web Service's Lambda, IBM Bluemix's OpenWhisk, Google Cloud Platform's Cloud Functions, and Microsoft Azure's Functions motivates study of software development in these services and their potential as a disruptive force in commercial cloud technologies. In addition to discussing the current state of cloud event services, this paper presents two real world applications utilizing these platforms: *Lambdefy*, a library designed to make traditional web application run effectively in AWS Lambda, and a performant media management service designed by Trek10, capable of resizing thousands of images per second. Furthermore, we discuss how cloud event technologies enable and/or limit these applications, motivate new software design paradigms in a cloud event environment, and highlight compelling use case scenarios and barriers to entry for cloud event services. AWS cloud technologies are exclusively used due to their maturity and the recent release of the other platforms, while Node.js and the Serverless Framework are utilized for deployment and application development.**

Keywords-*cloud events; AWS; Lambda; Serverless; micro-services; containers; Google Cloud Platform; Cloud Functions; Microsoft Azure; IBM Bluemix; OpenWhisk*

I. INTRODUCTION

The rapid evolution of systems infrastructure technologies, specifically virtualization and cloud, present new opportunities and challenges in software design. In the evolution from bare metal hardware provisioning, to traditional virtualization, to cloud IaaS, to container technology and now cloud events, the software architect has a myriad of IT infrastructure tools upon which to design and deploy his/her software in optimized configurations (price, performance, scale, etc.). The evolution seems to have accelerated with tech hype and maturity cycles condensing from decades (physical infrastructure) to years (virtualization/cloud) to months (containers/micro-services/cloud events). The most recent addition to this continuum is a set of services we will classify as cloud events. Specifically we are referencing new commercial services such as Amazon Web Service's Lambda [1], IBM Bluemix's OpenWhisk [2], Google Cloud Platform's Cloud Functions [3], and Microsoft Azure's Functions [4]. Cloud events in many ways are the next reductive step in IaaS abstraction; replacing coder's concerns about hardware and software dependencies with conceptually simpler function calls to act upon various other cloud services or cloud resident data sets. At this level of reduction, classification within IaaS, PaaS or SaaS blurs and implementation complexities, under the "cloud event" abstraction can have unexpected impacts.

In this work we focus on real world applications initially architected or modified to leverage clouds events. We note where the simple cloud event "function" concept allows designs that scale dynamically with few dependency limitations. At the same time we document where the event does not sufficiently allow for traditional design dependencies inherent in existing architectures. This leads us to a discussion of how application programming paradigms which leverage cloud events will necessarily differ from software designs which leverage more traditional infrastructure. In the following sections we review related work in the field, provide a hypothesis regarding the use of cloud events, describe multiple real world applications, analyze experiences leveraging cloud events and outline plans for future work.

II. BACKGROUND AND RELATED WORK

The fields of IaaS and cloud computing are rapidly evolving from their foundations in hardware, operating systems and distributed systems. To analyze new programming paradigms leveraging cloud events it is important to reference market trends and related work on the traditional and evolving platforms. Cloud oriented business models are a notable driving force behind rapid adoption of public cloud solutions [5]. This adoption has led to the hyper-evolution of cloud technology in recent years [6]. The current leader of the public cloud market is Amazon Web Services with Microsoft's Azure, Google's Cloud Engine and Rackspace as notable commercial competitors [7]; all of whom are competing to create or integrate new cloud market shaping technologies.

The progression to increasingly lightweight and efficient virtualized and abstracted systems (i.e. virtual machines to containers to unikernels) [8] is indicative of the rapid innovation taking place in the public cloud market. Many

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.58

virtual machine based technologies such as Xen and Hyper-V have reached steady state within the cloud computing market [9], [10]. Current trends in cloud infrastructure focus upon container based virtualization like that of Docker and RaaS Clouds [11], [12]. The continual solidification of container based technology within cloud environments underpins a very recent new service, "cloud events". Amazon Web Service's Lambda is the forerunner on cloud events and is the primary cloud event technology leveraged in this paper. Acknowledgement of recent releases of cloud event technologies by IBM [2], Google [3], and Microsoft [4] is also important to demonstrate industry momentum around the cloud event abstraction.

Enterprises are beginning to accept cloud systems as not only viable solutions but more often strategically important solutions. This has led to emergence of many competing or parallel paradigms for architecting efficient and effective cloud solutions. Cloud technology when used as multi-cloud systems in combination with an orchestration framework, such as Roboconf [13], can create powerful automated computing systems that greatly increase the efficiency and productivity of enterprises. Ad Hoc Cloud Computing, a cloud systems rendering of volunteer computing, is another paradigm of cloud events that offers increased productivity and efficiency [14]. The application of cloud events may be most impactful in its combination with the Internet of Things (IoT). As increasingly more sources of data become available, it is becoming evident that our ability to process this data is falling behind. Software-defined ecosystems designed to processes that data make room for numerous diverse and highly effective applications of cloud events [15].

Other application efforts have arisen since the inception of AWS Lambda, and demonstrate other groups testing the capabilities and limitations of cloud event services. A commonality among cloud event service are their efforts, by design, to abstract away the underlying infrastructure and encapsulate the user's code in one of a small set of runtimes. However, although cloud event services strictly maintain security and sandboxing between instances, the types of files and operations allowed within cloud event functions are rather flexible. One application demonstrates a successful attempt to statically cross-compile the Go language and run it within AWS Lambda around a thin Node.js wrapper [16]. Another similar application allows for the easy invocation of shell commands, also in AWS Lambda and around a Node.js wrapper [17]. Other more performance-oriented applications have emerged, such as a MapReduce-like word counter that rapidly parallelizes operations using AWS Lambda [18].

III. Test Platform

Cloud events introduce a new paradigm for architecting cloud based software and IT infrastructure solutions. Through the use of "function" abstracts, software and systems engineers can reduce their software platforms into individual components (functions) with a greatly reduced set of hardware and software dependencies. When the functions are fully integrated with a robust cloud infrastructure, each function has access to a broad range of natively integrated services. Given sufficiently low latency between the end user (client), cloud event function calls and subsequently called cloud services; modern applications can be written as series of largely independent function calls with minimal provisions in place to handle legacy infrastructure dependencies such as hardware resource limits, OS and library dependencies and server/VM boundaries.

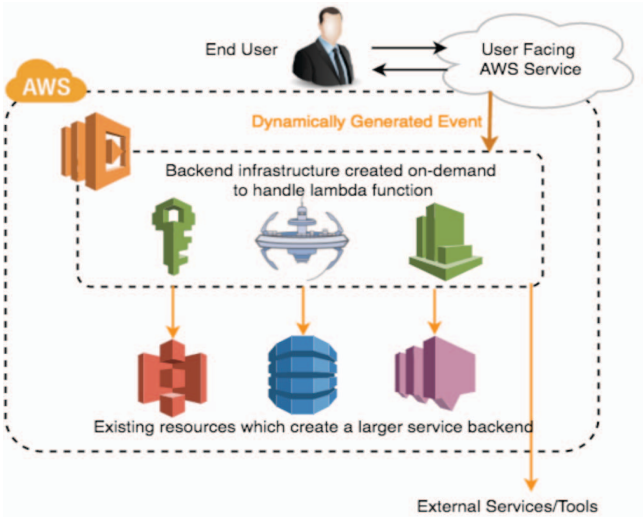

Figure 1. High-level cloud event architecture in AWS

In the following section we discuss two real world evaluations of this hypothesis but first we provide a couple of figures to better outline the general software infrastructure about which we are hypothesizing. Figure 1 graphically illustrates the high-level concept underpinning AWS cloud events. An end user, which may be a software agent or physical user, interacts with an AWS service that is capable of invoking a lambda functions in response to the user's action. This action could have involved uploading an image to S3, making an API request, querying a database or any tangible discrete action upon a Lambda capable AWS resource. The details of the user action upon the AWS resource is recorded in a JSON object and passed as a parameter to a lambda function. The developer only writes the function code to handle the event, all infrastructural resources necessary to run the function's code are dynamically provisioned upon function invocation. Figure 1 shows the Lambda function "internally" utilizing IAM, ECS and CloudWatch. However, this is a thin view of the mechanics involved in dynamically creating a run-time infrastructure for the Lambda function. Once invoked the Lambda function is free to interact with AWS services such as S3, DynamoDB, SNS (shown) or any service that is accessible through the Internet. AWS' large portfolio of products considerably increases the utility of cloud events by allowing Lambda functions to run in response to numerous diverse events. In this sense it is not merely the solidification of lightweight virtualized environments that lead to cloud events but also the diversification of public cloud provider product portfolios.

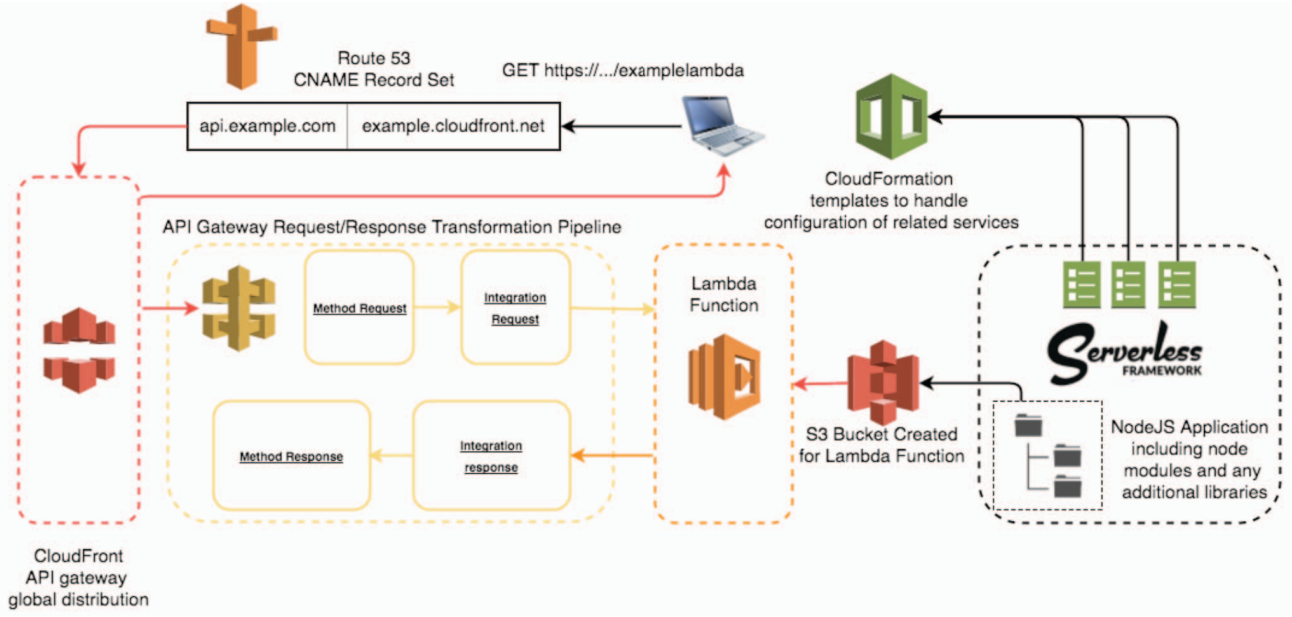

Figure 2. Detailed view of web application architecture implemented with AWS cloud events

Figure 2 gives a more in-depth view of how a Lambda function may be used in a specific context; in this case the context of handling an incoming user API request. Importantly, this figure introduces the open-source Serverless Framework. The Serverless Framework will be discussed at greater length in later sections. In basic terms, the Serverless Framework creates a development environment and several deployment/management features that takes advantage of the structure of Lambda functions and various AWS APIs for services such as S3, Lambda, CloudFormation. In the following example, a series of AWS products are combined and configured in order to create a greater application capable of handling an incoming user web request.

An end user makes a API call to api.example.com. The base of the request URL is resolved through Route53 to a global CloudFront distribution that is associated with the application's API, as defined in API Gateway. CloudFront passes the API request headers and body onto API Gateway. API Gateway applies authentication logic, modifications to the request, and any necessary format transformations before invoking a Lambda function and passing an event JSON representing the incoming request to the function. The function may then interact with a number of AWS based or external services to process and/or respond to the incoming event. Using the Serverless Framework each Lambda function is created from resources deployed through S3. The Serverless Framework project, as illustrated by the directory structure in Figure 3.2, contains all code elements of the invoked Lambda function. The project is also free to provision and configure other component resources of the application through CloudFormation. Once the invoked Lambda function has handled the incoming request a response JSON is returned to API Gateway. This response proceeds through the outbound stage of API Gateway's transformation pipeline before being served to the end user via the CloudFront distribution assigned to the API. Using this approach, independent functions may be written for each API method, and invoked independently.

At the enterprise level there are several benefits to consider. Each Lambda function invocation is given dedicated, isolated, and independent resources at run-time, handling burst loads effortlessly. API Gateway also has considerable capacity to scale considering it use of AWS' CDN, CloudFront, which is hardened to manage bursty and prolonged heavy load conditions. Continual development of Lambda functions is aided by built in versioning functionality in both API Gateway and Lambda. Finally, enterprise level applications may see considerable cost savings as AWS places small costs on API Gateway use and charges fractions of cents per Lambda function invocation.

IV. APPLICATIONS

A. *Blogging in Node.js: Ghost*

AWS Lambda, used in conjunction with API Gateway, is geared towards the creation of new micro-services in the cloud. However, the extreme granularity of Lambda functions and API Gateway resources makes managing the development and deployment of complex APIs untenable. The Serverless Framework aims to solve these issues, and provides organization and complex deployment of applications built on Lambda and API Gateway. Despite these improvements, Lambda, API Gateway, and the Serverless Framework by design only enable ways of writing new applications, rather than ways of deploying existing applications. Unlike advances from bare metal to virtual machines, or VMs to containers, cloud events require new paradigms of software construction, and are incompatible with the traditional tools and frameworks used in building web applications. This section motivates and explains our effort to bridge this gap, Lambdefy, which provides

deployment-time and run-time tools for executing traditional Node.js web applications in AWS Lambda for users who are unfamiliar with the complexities of API Gateway and Lambda.

Traditional web applications follow a familiar pattern of API construction during start-up, followed by API execution once a web server is listening for requests. Numerous frameworks exist to make this process as effortless as possible, and most involve listing method/route combinations and linking them to endpoint logic. These paradigms are problematic in a cloud event context for multiple reasons. Lambdefy allows these conflicting paradigms to coexist, and is composed of two pieces that address different aspects of the cloud event process. The first is a Serverless Framework plugin, which builds API Gateway and Lambda specifications at deployment-time and specifies how API Gateway processes input and output to/from Lambda. The second is a Node.js package that acts as the Lambda event handler for the web application, and translates between cloud events and web requests/responses at run-time.

Cloud event functions are designed to perform specific tasks, and to collectively function as a web application, but a traditional application is not easily split into its individual endpoints. Therefore, to generically deploy traditional web applications to Lambda, the entire application must remain together as a single Lambda function, which can easily be deployed by the Serverless Framework. API Gateway allows web requests to trigger Lambda function executions, to pass information about the web request to the triggered function, and to return a HTTP response. However, API Gateway's design encourages very detailed description of an API's resources, endpoints, methods, headers, and query parameters. This is a tedious and seemingly unnecessary requirement for any API developer, and especially for developers uninterested in low-level interactions with API Gateway. Ideally, API Gateway would extract the information from the incoming web request, pass it to the Lambda function, and return a response with a status code and headers as specified, without requiring specifics about the API beforehand. Lambdefy's deployment-time logic generically achieves this goal, and is configured by Serverless settings files.

A traditional web application receives requests by listening to a port, while a cloud event function receives information as function arguments. Furthermore, the arguments Lambda functions receive are structured differently than web requests, and Lambda functions must handle return values and error conditions instead of returning a HTTP response. Additionally, a traditional web application will not be able to process a Lambda invocation, because it is missing a function designated to handle the incoming events. The run-time layer of Lambdefy generates this handler given the port the application listens on, as well as information about the start-up timing of the application (synchronous vs. asynchronous) and the application's protocol (HTTP vs. HTTPS). When the Lambda function is triggered, this handling function will receive the event, generate a web request from the event information, send this request to the application, process the application's response, and return an object to API Gateway used to generate a web response to the client.

Ghost is a popular open-source blogging platform built on the Express framework in Node.js. This application was used to test the capabilities of the Lambdefy logic, and to experiment with the performance of a traditional web application hosted in AWS Lambda. Using such a heavyweight application was useful for exposing issues in Lambdefy during development, and in highlighting the fundamental differences and limitations of cloud events in comparison to other deployment methods (ex. VMs).

B. High Performance Media Management System

Trek10, Managed Performance Architects for the Cloud; is a company of expert cloud engineers partnered with AWS to bring cloud based enterprise IT solutions. In partnership with University of Notre Dame scientists they regularly explore the most cutting edge new cloud technologies. Their engineers have recently implemented several real world solutions using cloud event paradigms. In particular, Trek10 focuses on the use of AWS API Gateway and AWS Lambda backed by solution-specific supporting AWS services. This section discusses a real world business use case implemented by Trek10.

Large enterprises often consist of multiple internal entities, each operating with a degree of autonomy. Such enterprises often desire to create a media management service (MMS) to unify media asset upload, storage and distribution. Trek10's solution objective is to decrease storage and management costs, and increase and streamline the process of sharing media assets between enterprise entities. Presently, only image assets are being addressed by Trek10's solution. Future work intends to generalize the MMS solution to address video assets and other common asset formats.

A 'serverless' cloud event based approach was selected for the MMS use case due to potential scalability and cost saving improvements over traditional architectures. The MMS solution designed consists of three core features: Asset upload, asset distribution, and asset manipulation. The solution architecture is built upon Amazon API Gateway which routes requests and responses to a back-end. The back end consists of AWS Lambda for event compute, AWS Simple Storage Service (S3) for object storage, AWS' NoSQL database DynamoDB for maintaining state, and AWS CloudFront for asset distribution and low latency asset delivery.

The solution asset upload begins with a request to an API gateway endpoint, associated with the MMS, which dispatches a presigned URL. The asset is then directly uploaded by the requester to S3 via the signed URL. This illustrates a key difference between traditional architectures and cloud event driven architectures. In a traditional architecture, the server receives the upload and can handle any post-processing or metadata storage as a result of receiving the upload. As the uploads for MMS go directly to S3 for storage, a cloud event is necessary to signal some processing agent that there is work to be done. In this case,

on a successful S3 upload, an event is dispatched to a Lambda function with sufficient information to enable the function to process of uploaded object's metadata. The processed metadata is then stored in DynamoDB for later querying and asset management.

Trek10's MMS solution required a method of managing the distribution of uploaded assets. Passing out an S3 object URL is a simple method of enabling asset distribution but is not always appropriate. Trek10's MMS use case outlined requirements for dynamic arbitrary resizing of images, expiration of assets links, and the grouping of assets into sets in order to enable simplified management. When requesting an image asset, an MMS consumer requests a signed CloudFront URL through a known asset set or asset id. An MMS consumer can be considered an entity within an enterprise which will ultimately serve assets to the enterprise's end users. The signed URL allows the MMS to control the duration of time that the asset can be retrieved via that specific URL. The URL is then passed on, via the MMS consumer, to an end user through a browser or mobile app front-end. Through signed CloudFront URLs objects can be sourced directly from S3, thus avoiding the maintenance of any kind of server. CloudFront enables high performance response times by caching assets at multiple CloudFront edge locations.

One of the most compelling aspects of this serverless cloud event based approach is how a task such as image resizing can be handled. Trek10 set out to make the manipulation of entire asset sets a simple as possible. Resizing one to two image assets is a trivial task. For example, a basic solution could pull an image from S3, re-size it and simply write the re-sized image back to S3. For a limited number of assets, one can run such an operation in a serial process. However, in the case of ten or more images, a quick response time may require a multi-threaded application. Scaling even further to a solution which can handle resizing requests for hundreds or potentially thousands of images typically demands the implementation of a queueing mechanism and some form of worker pool.

Using Lambda functions, Trek10's MMS solution can re-size several hundred images in an embarrassingly parallel fashion. This approach results in approximately the same time cost for a single image re-sizing without the significant overheads of managing a queue/worker pool. This is enabled by a Lambda fan-out approach. Lambda fan-out takes advantage of the concept of instance based resources. When the resizing of an asset set is requested, the "master" request function invokes a "worker" lambda function for each image within the asset set. As discussed, one can consider each resizing operation as an independent isolated event with its own CPU, memory, and network resources. This cloud event fan-out paradigm allows for performant results while reducing development and infrastructure cost.

V. Discussion and Results

A. Blogging in Node.js: Ghost

Lambdefy enables direct comparison of web application deployments to both VMs/containers and cloud events. This comparison highlights the strengths and weaknesses of each deployment method, and can guide future development of the services themselves and the applications they host. The Ghost blog's deployment to AWS Lambda is a useful case study for analyzing how application development and structure should change to robustly perform in a serverless context.

The transience of the containers executing in Lambda provides many new challenges to application developers. Most notably, any application deployed to a cloud event service will be required start-up frequently (on the order of the total number of requests). This starkly contrasts more traditional environments, where applications start-up infrequently, and only as the deployment scales up with total application load. Therefore, start-up time becomes a key measure of application performance in a serverless environment, but in a traditional application this delay has little reason for optimization. In the case of Ghost, every time the application starts it must build its API routes and connect to the data layer, but it also performs many checks and even considers making database migrations before it starts its web server. This initialization pipeline is unnecessarily performed in series, and takes hundreds of milliseconds to complete. Of course, Ghost's implementations seem very sane in a world of traditional deployments, but in the world of cloud events, this start-up time is incredibly costly. With technologies like cloud events disrupting cloud programming paradigms, applications that are agnostic of their deployments should begin to optimize start-up latency.

Cloud event transience also forces completely stateless execution, both within code and within the file system. Any changes made to local state will only exist momentarily in a single execution unit, and applications may not even have write access to their working directory. This forces separation between application logic and data layers. Additionally, due to the extreme horizontal scaling capabilities of cloud events, an application's data layer must be capable of performantly handling many concurrent and short-lived connections that may not close gracefully. For example, when experimenting with Ghost, the application used a standard MySQL image in AWS' RDS as its data storage. However, because Lambda containers do not give warning on closure, connections to the RDS instance were ungracefully terminated, and quickly built up due to the long default connection timeout in the RDS instance. Soon, requests were failing due to maximum concurrent connections. In traditional deployments this problem would never exist, but in this cloud event environment, the application's deployment forced changes (connection timeout value) in the data layer to accommodate the behavior of Lambda.

B. High Performance Media Management System

A core component of the serverless cloud event based approach involves pushing workloads and operations to purpose specific services. For example, uploading assets is not readily handled by API Gateway and Lambda alone. Rather, uploading media assets is handled directly through

an AWS S3 presigned URLs. AWS Lambda and API Gateway allow for the federation of application process specific services such as S3, DynamoDB and CloudFront. This separation of concerns is a primary benefit of cloud event paradigms.

The solution architecture discussed was implemented in its entirety, along with additional functionality for asset life-cycle management using the Serverless Framework. As seen in Figure 3, the architecture has been shown to support ~525 requests per second without any signs of performance degradation. Figure 4 compares the latency of cache hits and misses. The response latency for image resizing requests, for non-cached results, for a set of any size is on average 600-700ms from the time of request until assets are re-sized and asset URLs are dispatched. On a cache hit, ~82.5% of requests complete in under 10 milliseconds. This performance is achieved without the need to consider or manage scaling, system resource constraints, queuing, cluster management, and a multitude of further configuration and operational tasks. The adopted approach enabled the entire core system to be built by two engineers on a part time basis within two months, while such engineers attended to several unrelated business responsibilities.

VI. Summary and Future Work

This work describes two applications: one using Lambdefy to demonstrate the differing requirements between applications deployed to IaaS and applications deployed as a cloud event, and a MMS that showcased the fan-out ability of Lambda, easily and performantly solving a large-scale image resizing task. These applications highlight the benefits cloud events can have in problem domains, while arguing that effective cloud event applications must consider different performance metrics than applications deployed to more traditional infrastructure. Moreover, the heavy reliance of these applications on other cloud services shows that effective use of cloud events requires a greater level of cloud competency and experience than deploying applications to IaaS offerings.

Cloud events have enormous potential as a disruptive

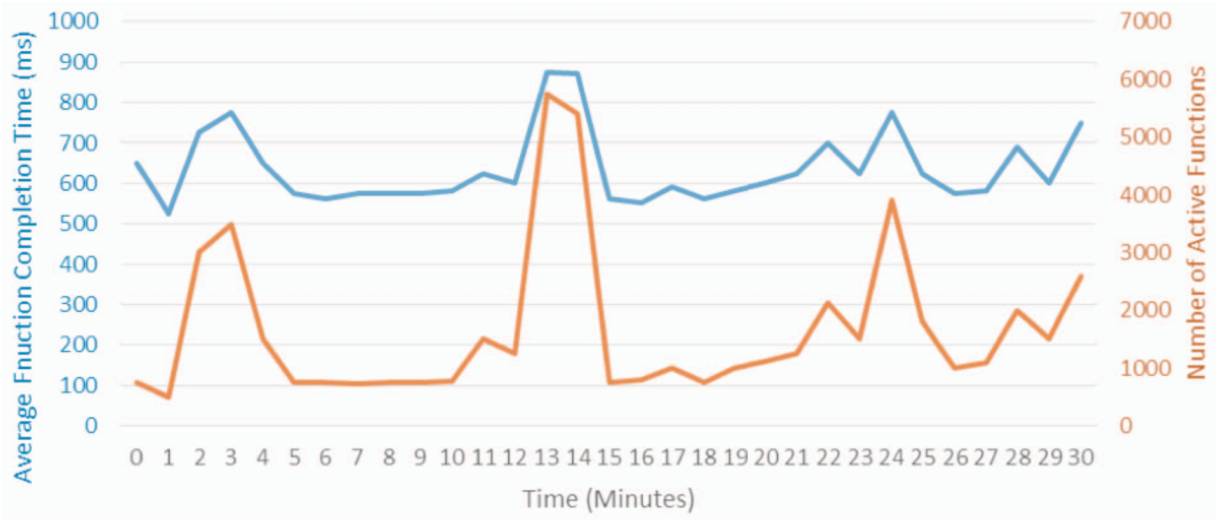

Figure 3. Average worker Lambda function completion time under varying load

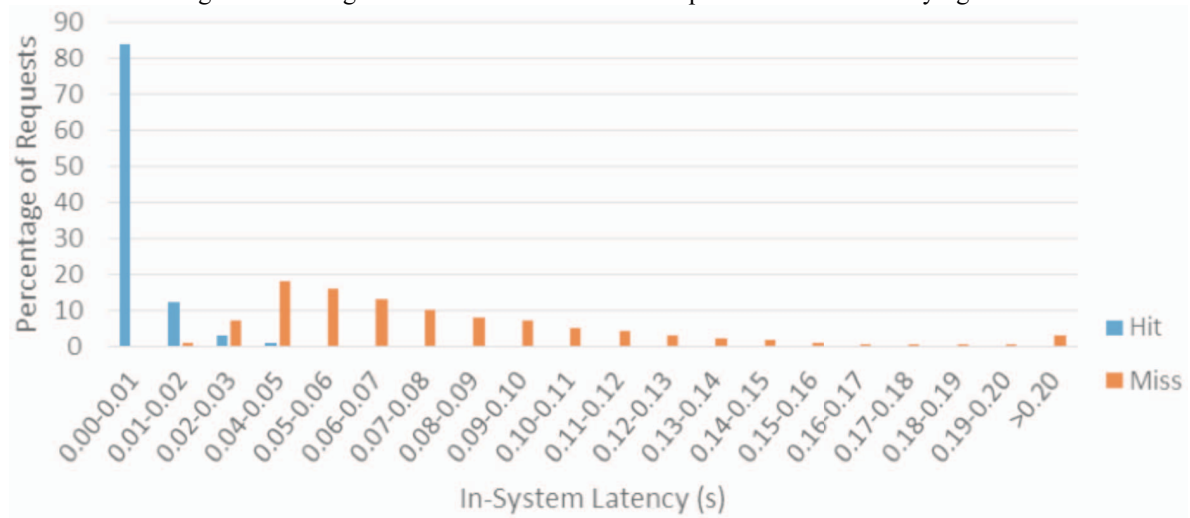

Figure 4. CloudFront in-system latency under 300 requests per second load

cloud technology. Lambda continues to expand capabilities and integrate with other services as it matures, and the release of competing products shows that multiple industry leaders see the potential of cloud events in a commercial cloud environment. Cloud events are low cost, low latency, massively scalable, and capable of powerful parallel and recursive structures. However, while many things are now possible with cloud events, few are easy to understand or learn to develop. Cloud event containers are short-lived but not necessarily single-use, which opens up many non-obvious optimization possibilities. The granular nature of the technology means there are many different pieces composing an application, and understanding how the pieces fit together across multiple cloud services is a large barrier to entry. Versioning becomes more complex when deployments can occur across arbitrary portions of an application, and there are inherent challenges when considering end-to-end testing of such loosely coupled applications, especially when multiple test runs can result in different execution patterns within the cloud event service. Managing deployment across multiple regions and environments is also challenging in a cloud event environment, although third-party frameworks like Serverless can certainly help. Until developing and understanding these complex services becomes easier, it will be difficult for industry players to effectively realize the benefits cloud events can provide.

The ability of the fan-out pattern to rapidly scale and parallelize workloads suggests future work focused on low-latency scaling of cloud event services. The performance results of the MMS show that cloud event services can reliably and performantly parallelize under heavy load, but so far these efforts have focused on improving performance, not optimizing it. An academic approach to the performance of these fan-out patterns could further improve the utility of cloud event services, and provide useful tools for fan-out computation in production cloud event environments. Beyond performance concerns, future efforts in this area should also center on issues of ease and accessibility encountered during use of cloud event services. Significant potential work exists in the development of robust versioning systems, test frameworks that account for unpredictability in cloud event environments, clearer optimization paths, and deployment tooling that understands cloud event application composition across many different services. Furthermore, application development frameworks that are effective and performant across cloud events and IaaS services could be very powerful, and allow cloud events to be viewed as a way to deploy applications and not just a way to develop applications. Additionally, experimenting with application domains could find new uses for cloud event technologies.

ACKNOWLEDGMENT

The authors would like to thank multiple collaborators and financial supporters. Paul Brenner would like to thank the Notre Dame Center for Research Computing and Department of Computer Science and Engineering. Stephen Ennis would like to thank the Notre Dame ESTEEM program and Trek10. Jared Short and Garrett McGrath would also like to thank Trek10. All of the authors are appreciative of support from AWS technical experts and the Serverless developers. Jared and Stephen would also like to thank their commercial customers who partner with Trek10 to leverage new cloud technology for high performance applications.

REFERENCES

[1] Amazon Web Services, Inc., “AWS Lambda: Developer Guide,” 2016.

[2] IBM Bluemix, “Welcome to Bluemix OpenWhisk.” Available: https://new-console.ng.bluemix.net/openwhisk/

[3] Google Cloud Platform, “Cloud Functions,” Available: https://cloud.google.com/functions/

[4] Microsoft Azure, “Functions,” Available: https://azure.microsoft.com/en-us/services/functions/

[5] W. Martorelli, and L. Herbert, “Understanding The Cloud Services Provider Landscape,” Forrester, 23 October 2015

[6] Naveen Chhabra, “Tech Radar: Cloud Computing, Q4 2015,” 24 November 2015.

[7] M. G. McGrath, P. Raycroft, and P. R. Brenner, “Intercloud Networks Performance Analysis,” in IEEE International Conference on Cloud Engineering (IC2E), 2015, pp 487 - 492.

[8] D. Bernstein, “Containers and Cloud: From LXC to Docker to Kubernetes,” IEEE Cloud Computing, September 2014

[9] P. Yu, M. Xia, Q. Lin, M. Zhu, S. Gao, Z. Qi, K. Chen, and H. Guan, “Real-time Enhancement for Xen hypervisor,” in 8th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, 2010.

[10] Trefis, “Growing Competition For VMware In Virtualization Market,” January 2014, retrieved: February 2016. Available: http://www.nasdaq.com/article/growing-competition-for-vmware-in-virtualization-market-cm316783.

[11] L. Li, T. Tang, and W. Chou, “A REST Service Framework for Fine-Grained Resource Management in Container- Based Cloud,” in 8th IEEE International Conference on Cloud Computing, 2015, pp 645 - 652.

[12] S. Gomez Saez, V. Andrikopoulos, R. Jimenez Sanchez, F. Leymann, and J. Wettinger, “Dynamic Tailoring and Cloud-based Deployment of Containerized Service Middleware,” in 8th IEEE International Conference on Cloud Computing, 2015, pp 349 - 356.

[13] L. M. Pham, A. Tchana, D. Donsez, N. Palma, V. Zurczak, and P. Y. Gibello, “Roboconf: a Hybrid Cloud Orchestrator to Deploy Complex Applications,” in 8th IEEE International Conference on Cloud Computing, 2015, pp 365 - 372.

[14] G. A. McGilvary, A. Barker, and M. Atkinson, “Ad hoc Cloud Computing,” in 8th IEEE International Conference on Cloud Computing, 2015, pp 1063 - 1068.

[15] M. Parashar, M. AbdelBaky, M. Zou, A. R. Zamani, and J. Diaz-Montes, “Realizing the Potential of IoT Using Software-Defined Ecosystems,” in 8th IEEE International Conference on Cloud Computing, 2015, pp 1149 - 1158.

[16] Daeli, Jacopo. “Writing AWS Lambda Function in Go,” 9 January 2016, retrieved: May 2016. Available: http://jacopodaeli.com/writing-aws-lambda-functions-in-go/

[17] Hammond, Eric. “lambdash: AWS Lambda Shell Hack: New and Improved!,” 18 June 2015, retrieved: May 2016. Available: https://alestic.com/2015/06/aws-lambda-shell-2/

[18] Holste, Martin, “Building Scalable and Responsible Big Data Interfaces with AWS Lambda,” 10 July 2015, retrieved May 2016. Available: https://blogs.aws.amazon.com/bigdata/post/Tx3KH6BEUL2SGVA/Building-Scalable-and-Responsive-Big-Data-Interfaces-with-AWS-Lambda