O'REILLY®

Serverless Ops

A Beginner's Guide to AWS Lambda and Beyond

Michael Hausenblas

Serverless Ops

A Beginner's Guide to AWS Lambda and Beyond

Michael Hausenblas

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Serverless Ops
by Michael Hausenblas

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Virginia Wilson
Acquisitions Editor: Brian Anderson
Production Editor: Shiny Kalapurakkel
Copyeditor: Amanda Kersey
Proofreader: Rachel Head
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Panzer

November 2016: First Edition

Revision History for the First Edition
2016-11-09: First Release

978-1-491-97079-9

[LSI]

Table of Contents

Preface

The dominant way we deployed and ran applications over the past decade was machine-centric. First, we provisioned physical machines and installed our software on them. Then, to address the low utilization and accelerate the roll-out process, came the age of virtualization. With the emergence of the public cloud, the offerings became more diverse: Infrastructure as a Service (IaaS), again machine-centric; Platform as a Service (PaaS), the first attempt to escape the machine-centric paradigm; and Software as a Service (SaaS), the so far (commercially) most successful offering, operating on a high level of abstraction but offering little control over what is going on.

Over the past couple of years we've also encountered some developments that changed the way we think about running applications and infrastructure as such: the microservices architecture, leading to small-scoped and loosely coupled distributed systems; and the world of containers, providing application-level dependency management in either on-premises or cloud environments.

With the advent of DevOps thinking in the form of Michael T. Nygard's *Release It!* (Pragmatic Programmers) and the twelve-factor manifesto, we've witnessed the transition to immutable infrastructure and the need for organizations to encourage and enable developers and ops folks to work much more closely together, in an automated fashion and with mutual understanding of the motivations and incentives.

In 2016 we started to see the serverless paradigm going mainstream. Starting with the AWS Lambda announcement in 2014, every major cloud player has now introduced such offerings, in addition to many

new players like OpenLambda or Galactic Fog specializing in this space.

Before we dive in, one comment and disclaimer on the term "serverless" itself: catchy as it is, the name is admittedly a misnomer and has attracted a fair amount of criticism, including from people such as AWS CTO Werner Vogels. It is as misleading as "NoSQL" because it defines the concept in terms of what it is not about.[1] There have been a number of attempts to rename it; for example, to Function as a Service(FaaS). Unfortunately, it seems we're stuck with the term because it has gained traction, and the majority of people interested in the paradigm don't seem to have a problem with it.

You and Me

My hope is that this report will be useful for people who are interested in going serverless, people who've just started doing serverless computing, and people who have some experience and are seeking guidance on how to get the maximum value out of it. Notably, the report targets:

- DevOps folks who are exploring serverless computing and want to get a quick overview of the space and its options, and more specifically novice developers and operators of AWS Lambda
- Hands-on software architects who are about to migrate existing workloads to serverless environments or want to apply the paradigm in a new project

This report aims to provide an overview of and introduction to the serverless paradigm, along with best-practice recommendations, rather than concrete implementation details for offerings (other than exemplary cases). I assume that you have a basic familiarity with operations concepts (such as deployment strategies, monitoring, and logging), as well as general knowledge about public cloud offerings.

1 The term NoSQL suggests it's somewhat anti-SQL, but it's not about the SQL language itself. Instead, it's about the fact that relational databases didn't use to do auto-sharding and hence were not easy or able to be used out of the box in a distributed setting (that is, in cluster mode).

Note that true coverage of serverless operations would require a book with many more pages. As such, we will be covering mostly techniques related to AWS Lambda to satisfy curiosity about this emerging technology and provide useful patterns for the infrastructure team that administers these architectures.

As for my background: I'm a developer advocate at Mesosphere working on DC/OS, a distributed operating system for both containerized workloads and elastic data pipelines. I started to dive into serverless offerings in early 2015, doing proofs of concepts, speaking and writing about the topic, as well as helping with the onboarding of serverless offerings onto DC/OS.

Acknowledgments

I'd like to thank Charity Majors for sharing her insights around operations, DevOps, and how developers can get better at operations. Her talks and articles have shaped my understanding of both the technical and organizational aspects of the operations space.

The technical reviewers of this report deserve special thanks too. Eric Windisch (IOpipe, Inc.), Aleksander Slominski (IBM), and Brad Futch (Galactic Fog) haven taken out time of their busy schedules to provide very valuable feedback and certainly shaped it a lot. I owe you all big time (next Velocity conference?).

A number of good folks have supplied me with examples and references and have written timely articles that served as brain food: to Bridget Kromhout, Paul Johnston, and Rotem Tamir, thank you so much for all your input.

A big thank you to the O'Reilly folks who looked after me, providing guidance and managing the process so smoothly: Virginia Wilson and Brian Anderson, you rock!

Last but certainly not least, my deepest gratitude to my awesome family: our sunshine artist Saphira, our sporty girl Ranya, our son Iannis aka "the Magic rower," and my ever-supportive wife Anneliese. Couldn't have done this without you, and the cottage is my second-favorite place when I'm at home. ;)

CHAPTER 1
Overview

Before we get into the inner workings and challenges of serverless computing, or Function as a Service (FaaS), we will first have a look at where it sits in the spectrum of computing paradigms, comparing it with traditional three-tier apps, microservices, and Platform as a Service (PaaS) solutions. We then turn our attention to the concept of serverless computing; that is, dynamically allocated resources for event-driven function execution.

A Spectrum of Computing Paradigms

The basic idea behind serverless computing is to make the unit of computation a function. This effectively provides you with a lightweight and dynamically scalable computing environment with a certain degree of control. What do I mean by this? To start, let's have a look at the spectrum of computing paradigms and some examples in each area, as depicted in Figure 1-1.

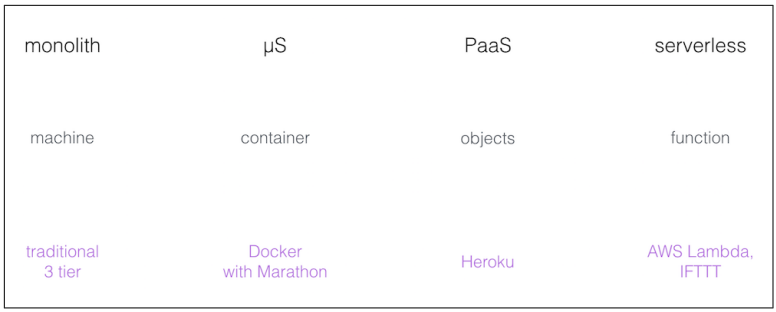

Figure 1-1. A spectrum of compute paradigms

In a monolithic application, the unit of computation is usually a machine (bare-metal or virtual). With microservices we often find containerization, shifting the focus to a more fine-grained but still machine-centric unit of computing. A PaaS offers an environment that includes a collection of APIs and objects (such as job control or storage), essentially eliminating the machine from the picture. The serverless paradigm takes that a step further: the unit of computation is now a single function whose lifecycle you manage, combining many of these functions to build an application.

Looking at some (from an ops perspective), relevant dimensions further sheds light on what the different paradigms bring to the table:

Agility
: In the case of a monolith, the time required to roll out new features into production is usually measured in months; serverless environments allow much more rapid deployments.

Control
: With the machine-centric paradigms, you have a great level of control over the environment. You can set up the machines to your liking, providing exactly what you need for your workload (think libraries, security patches, and networking setup). On the other hand, PaaS and serverless solutions offer little control: the service provider decides how things are set up. The flip side of control is maintenance: with serverless implementations, you essentially outsource the maintenance efforts to the service provider, while with machine-centric approaches the onus is on you. In addition, since autoscaling of functions is typically supported, you have to do less engineering yourself.

Cost per unit
: For many folks, this might be the most attractive aspect of serverless offerings—you only pay for the actual computation. Gone are the days of provisioning for peak load only to experience low resource utilization most of the time. Further, A/B testing is trivial, since you can easily deploy multiple versions of a function without paying the overhead of unused resources.

The Concept of Serverless Computing

With this high-level introduction to serverless computing in the context of the computing paradigms out of the way, we now move on to its core tenents.

At its core, serverless computing is event-driven, as shown in Figure 1-2.

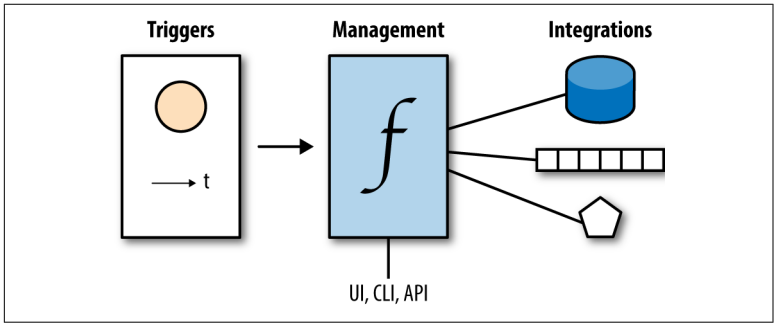

Figure 1-2. The concept of serverless compute

In general, the main components and actors you will find in serverless offerings are:[1]

Management interfaces
: Register, upgrade, and control functions via web UIs, command-line interfaces, or HTTP APIs.

Triggers
: Define when a function is invoked, usually through (external) events, and are scheduled to be executed at a specific time.

1 I've deliberately left routing (mapping, for example, an HTTP API to events) out of the core tenents since different offerings have different approaches for how to achieve this.

Integration points
: Support control and data transfer from function-external systems such as storage.

So, the serverless paradigm boils down to reacting to events by executing code that has been uploaded and configured beforehand.

How Serverless Is Different from PaaS

Quite often, when people start to dig into serverless computing, I hear questions like "How is this different from PaaS?"

Serverless computing (or FaaS), refers to the idea of dynamically allocating resources for an event-driven function execution. A number of related paradigms and technologies exist that you may have come across already. This sidebar aims to compare and delimit them.

PaaS shares a lot with the serverless paradigm, such as no provisioning of machines and autoscaling. However, the unit of computation is much smaller in the latter. Serverless computing is also job-oriented rather than application-oriented. For more on this topic, see Carl Osipov's blog post "Is Serverless Computing Any Different from Cloud Foundry, OpenShift, Heroku, and Other Traditional PaaSes?".

The Remote Procedure Call (RPC) protocol is all about the illusion that one can call a remotely executed function (potentially on a different machine) in the same way as a locally executed function (in the same memory space).

Stored procedures have things in common with serverless computing (including some of the drawbacks, such as lock-in), but they're database-specific and not a general-purpose computing paradigm.

Microservices are not a technology but an architecture and can, among other things, be implemented with serverless offerings.

Containers are typically the basic building blocks used by serverless offering providers to enable rapid provisioning and isolation.

Conclusion

In this chapter we have introduced serverless computing as an event-driven function execution paradigm with its three main components: the triggers that define when a function is executed, the management interfaces that register and configure functions, and integration points that interact with external systems (especially storage). Now we'll take a deeper look at the concrete offerings in this space.

CHAPTER 2
The Ecosystem

In this chapter we will explore the current serverless computing offerings and the wider ecosystem. We'll also try to determine whether serverless computing only makes sense in the context of a public cloud setting or if operating and/or rolling out a serverless offering on-premises also makes sense.

Overview

Many of the serverless offerings at the time of writing of this report (mid-2016) are rather new, and the space is growing quickly.

Table 2-1 gives a brief comparison of the main players. More detailed breakdowns are provided in the following sections.

Table 2-1. Serverless offerings by company

Offering	Cloud offering	On-premises	Launched	Environments
AWS Lambda	Yes	No	2014	Node.js, Python, Java
Azure Functions	Yes	Yes	2016	C#, Node.js, Python, F#, PHP, Java
Google Cloud Functions	Yes	No	2016	JavaScript
iron.io	No	Yes	2012	Ruby, PHP, Python, Java, Node.js, Go, .NET
Galactic Fog's Gestalt	No	Yes	2016	Java, Scala, JavaScript, .NET
IBM OpenWhisk	Yes	Yes	2014	Node.js, Swift

Note that by *cloud offering*, I mean that there's a managed offering in one of the public clouds available, typically with a pay-as-you-go model attached.

AWS Lambda

Introduced in 2014 in an AWS re:Invent keynote, AWS Lambda is the incumbent in the serverless space and makes up an ecosystem in its own right, including frameworks and tooling on top of it, built by folks outside of Amazon. Interestingly, the motivation to introduce Lambda originated in observations of EC2 usage: the AWS team noticed that increasingly event-driven workloads were being deployed, such as infrastructure tasks (log analytics) or batch processing jobs (image manipulation and the like). AWS Lambda started out with support for the Node runtime and currently supports Node.js 4.3, Python 2.7, and Java 8.

The main building blocks of AWS Lambda are:

- The AWS Lambda Web UI (see Figure 2-1) and CLI itself to register, execute, and manage functions
- Event triggers, including, but not limited to, events from S3, SNS, and CloudFormation to trigger the execution of a function
- CloudWatch for logging and monitoring

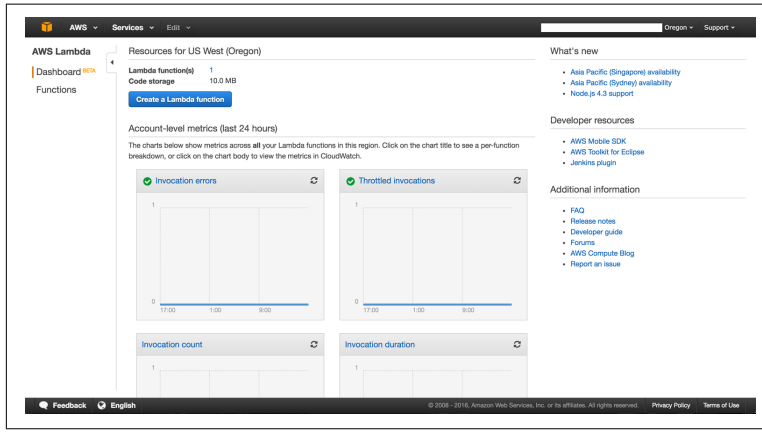

Figure 2-1. AWS Lambda dashboard

Pricing

Pricing of AWS Lambda is based on the total number of requests as well as execution time. The first 1 million requests per month are free; after that, it's $0.20 per 1 million requests. In addition, the free tier includes 400,000 GB-seconds of computation time per month. The minimal duration you'll be billed for is 100 ms, and the actual costs are determined by the amount of RAM you allocate to your function (with a minimum of 128 MB).

Availability

Lambda has been available since 2014 and is a public cloud–only offering.

We will have a closer look at the AWS Lambda offering in Chapter 4, where we will walk through an example from end to end.

Azure Functions

During the Build 2016 conference Microsoft released Azure Functions, supporting functions written with C#, Node.js, Python, F#, PHP, batch, bash, Java, or any executable. The Functions runtime is open source and integrates with Azure-internal and -external services such as Azure Event Hubs, Azure Service Bus, Azure Storage, and GitHub webhooks. The Azure Functions portal, depicted in Figure 2-2, comes with templates and monitoring capabilities.

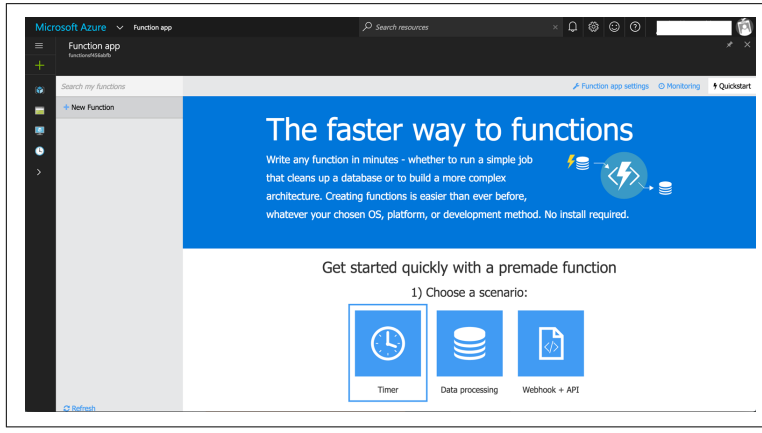

Figure 2-2. Azure Functions portal

As an aside, Microsoft also offers other serverless solutions such as Azure Web Jobs and Microsoft Flow (an "if this, then that" [IFTTT] for business competitors).

Pricing

Pricing of Azure Functions is similar to that of AWS Lambda; you pay based on code execution time and number of executions, at a rate of $0.000008 per GB-second and $0.20 per 1 million executions. As with Lambda, the free tier includes 400,000 GB-seconds and 1 million executions.

Availability

Since early 2016, the Azure Functions service has been available both as a public cloud offering and on-premises as part of the Azure Stack.

Google Cloud Functions

Google Cloud Functions can be triggered by messages on a Cloud Pub/Sub topic or through mutation events on a Cloud Storage bucket (such as "bucket is created"). For now, the service only supports Node.js as the runtime environment. Using Cloud Source Repositories, you can deploy Cloud Functions directly from your GitHub or Bitbucket repository without needing to upload code or manage versions yourself. Logs emitted are automatically written to Stackdriver Logging and performance telemetry is recorded in Stackdriver Monitoring.

Figure 2-3 shows the Google Cloud Functions view in the Google Cloud console. Here you can create a function, including defining a trigger and source code handling.

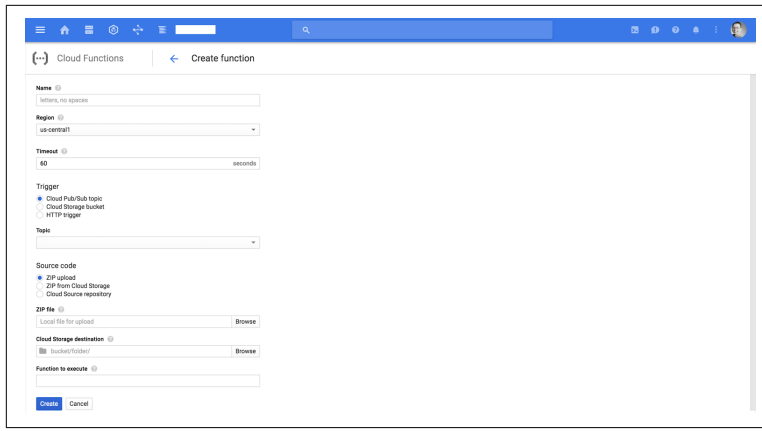

Figure 2-3. Google Cloud Functions

Pricing

Since the Google Cloud Functions service is in Alpha, no pricing has been disclosed yet. However, we can assume that it will be priced competitively with the incumbent, AWS Lambda.

Availability

Google introduced Cloud Functions in February 2016. At the time of writing, it's in Alpha status with access on a per-request basis and is a public cloud–only offering.

Iron.io

Iron.io has supported serverless concepts and frameworks since 2012. Some of the early offerings, such as IronQueue, IronWorker, and IronCache, encouraged developers to bring their code and run it in the Iron.io-managed platform hosted in the public cloud. Written in Go, Iron.io recently embraced Docker and integrated the existing services to offer a cohesive microservices platform. Code-named Project Kratos, the serverless computing framework from Iron.io aims to bring AWS Lambda to enterprises without the vendor lock-in.

In Figure 2-4, the overall Iron.io architecture is depicted: notice the use of containers and container images.

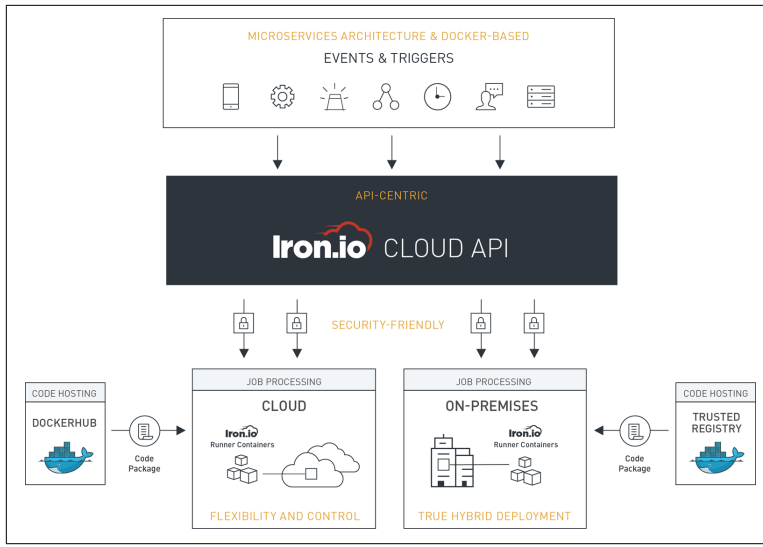

Figure 2-4. Iron.io architecture

Pricing

No public plans are available, but you can use the offering via a number of deployment options, including Microsoft Azure and DC/OS.

Availability

Iron.io has offered its services since 2012, with a recent update around containers and supported environments.

Galactic Fog's Gestalt

Gestalt (see Figure 2-5) is a serverless offering that bundles containers with security and data features, allowing developers to write and deploy microservices on-premises or in the cloud.

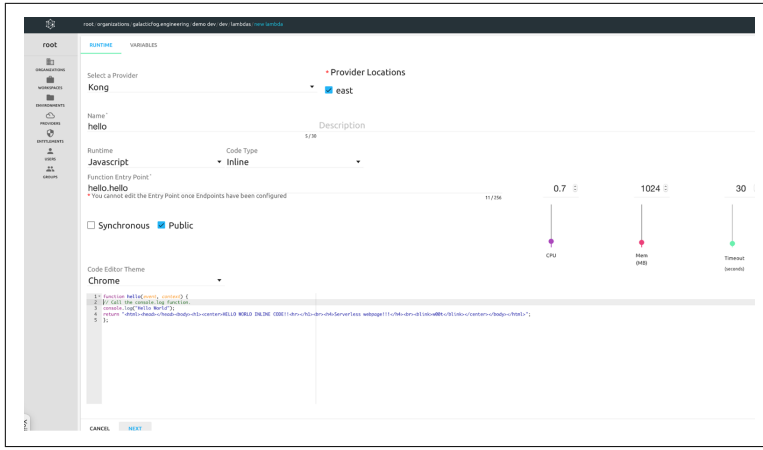

Figure 2-5. Gestalt Lambda

Pricing

No public plans are available.

Availability

Launched in mid-2016, the Gestalt Framework is deployed using DC/OS and is suitable for cloud and on-premises deployments; no hosted service is available yet.

See the MesosCon 2016 talk "Lamba Application Servers on Mesos" by Brad Futch for details on the current state as well as the upcoming rewrite of Gestalt Lambda called LASER.

IBM OpenWhisk

IBM OpenWhisk is an open source alternative to AWS Lambda. As well as supporting Node.js, OpenWhisk can run snippets written in Swift. You can install it on your local machine running Ubuntu. The service is integrated with IBM Bluemix, the PaaS environment powered by Cloud Foundry. Apart from invoking Bluemix services, the framework can be integrated with any third-party service that supports webhooks. Developers can use a CLI to target the OpenWhisk framework.

Figure 2-6shows the high-level architecture of OpenWhisk, including the trigger, management, and integration point options.

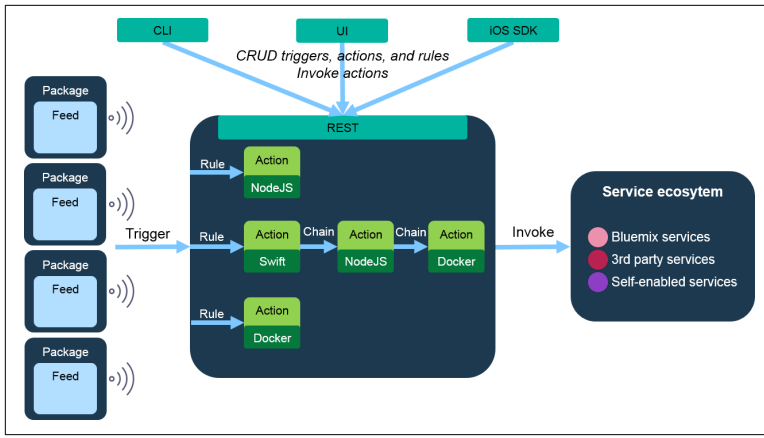

Figure 2-6. OpenWhisk architecture

Pricing

The costs are determined based on Bluemix, at a rate of $0.0288 per GB-hour of RAM and $2.06 per public IP address. The free tier includes 365 GB-hours of RAM, 2 public IP addresses, and 20 GB of external storage.

Availability

Since 2014, OpenWhisk has been available as a hosted service via Bluemix and for on-premises deployments with Bluemix as a dependency.

See "OpenWhisk: a world first in open serverless architecture?" for more details on the offering.

Other Players

In the past few years, the serverless space has seen quite some uptake, not only in terms of end users but also in terms of providers. Some of the new offerings are open source, some leverage or extend existing offerings, and some are specialized offerings from existing providers. They include:

- OpenLambda, an open source serverless computing platform
- Nano Lambda, an automated computing service that runs and scales your microservices
- Webtask by Auth0, a serverless environment supporting Node.js with a focus on security
- Serverless Framework, an application framework for building web, mobile, and Internet of Things (IoT) applications powered by AWS Lambda and AWS API Gateway, with plans to support other providers, such as Azure and Google Cloud
- IOpipe, an analytics and distributed tracing service that allows you to see inside AWS Lambda functions for better insights into the daily operations

Cloud or on-Premises?

A question that often arises is whether serverless computing only makes sense in the context of a public cloud setting, or if rolling out a serverless offering on-premises also makes sense. To answer this question, we will discuss elasticity features, as well as dependencies introduced when using a serverless offering.

So, which one is the better option? A public cloud offering such as AWS Lambda, or one of the existing open source projects, or your home-grown solution on-premises? As with any IT question, the answer depends on many things, but let's have a look at a number of considerations that have been brought up in the community and may be deciding factors for you and your organization.

One big factor that speaks for using one of the (commercial) public cloud offerings is the ecosystem. Look at the supported events (triggers) as well as the integrations with other services, such as S3, Azure SQL Database, and monitoring and security features. Given that the serverless offering is just one tool in your toolbelt, and you might already be using one or more offerings from a certain cloud provider, the ecosystem is an important point to consider.

Oftentimes the argument is put forward that true autoscaling of the functions only applies to public cloud offerings. While this is not black and white, there is a certain point to this claim: the elasticity of the underlying IaaS offerings of public cloud providers will likely

outperform whatever you can achieve in your datacenter. This is, however, mainly relevant for very spiky or unpredictable workloads, since you can certainly add virtual machines (VMs) in an on-premises setup in a reasonable amount of time, especially when you know in advance that you'll need them.

Avoiding lock-in is probably the strongest argument against public cloud serverless deployments, not so much in terms of the actual code (migrating this from one provider to another is a rather straightforward process) but more in terms of the triggers and integration points. At the time of writing, there is no good abstraction that allows you to ignore storage or databases and work around triggers that are available in one offering but not another.

Another consideration is that when you deploy the serverless infrastructure in your datacenter you have full control over, for example how long a function can execute. The public cloud offerings at the current point in time do not disclose details about the underlying implementation, resulting in a lot of guesswork and trial and error when it comes to optimizing the operation. With an on-premises deployment you can go as far as developing your own solution, as discussed in Appendix A; however, you should be aware of the investment (both in terms of development and operations) that is required with this option.

Table 2-1 summarizes the criteria discussed in the previous paragraphs.

Offering	Cloud	On-premises
Ecosystem	Yes	No
True autoscaling	Yes	No
Avoiding lock-in	No	Yes
End-to-end control	No	Yes

Note that depending on what is important to your use case, you'll rank different aspects higher or lower; my intention here is not to categorize these features as positive or negative but simply to point out potential criteria you might want to consider when making a decision.

Conclusion

In this chapter, we looked at the current state of the serverless ecosystem, from the incumbent AWS Lambda to emerging open source projects such as OpenLambda. Further, we discussed the topic of using a serverless offering in the public cloud versus operating (and potentially developing) one on-premises based on decision criteria such as elasticity and integrations with other services such as databases. Next we will discuss serverless computing from an operations perspective and explore how the traditional roles and responsibilities change when applying the serverless paradigm.

CHAPTER 3

Serverless from an Operations Perspective

The serverless paradigm blurs the line between development and operations. On the one hand, certain traditionally necessary steps such as provisioning a machine do not apply anymore; on the other hand, developers can't simply hand off binaries to operations.

In this chapter, we will first discuss roles in the context of a serverless setup and then have a closer look at typical activities, good practices, and antipatterns around serverless ops.

AppOps

With serverless computing, it pays off to rethink roles and responsibilities in the team. To do that, I'm borrowing a term that was first coined by Bryan Liles of Digital Ocean: *AppOps*. The basic idea behind AppOps is that the one who writes a service also operates it in production. This means that AppOps are on call for the services they have developed. In order for this to work, the infrastructure used needs to support service- or app-level monitoring of metrics as well as alerting if the service doesn't perform as expected.

Further, there's another role necessary: a group of people called the *infrastructure team*. This team manages the overall infrastructure, owns global policies, and advises the AppOps.

A sometimes-used alternative label for the serverless paradigm is "NoOps," suggesting that since there are no machines to provision,

the need for operations folks is not given. This term is, however, misleading and best avoided. As discussed, operational skills and practices are not only necessary but pivotal in the serverless context —just not in the traditional sense.

Operations: What's Required and What Isn't

To define operations in the serverless context, I'll start out with Charity Majors's definition:

> Operations is the constellation of your org's technical skills, practices and cultural values around designing, building and maintaining systems, shipping software , and solving problems with technology.
>
> —Serverlessness, NoOps and the Tooth Fairy,, *May 2016*

Building on this definition, we can now understand what is required for successful operations:

Scalability
: Being able to scale parts of the system as well as an understanding of the entire system. The autoscaling support usually found in serverless offerings should not be taken as an excuse to not study and understand this property.

Resilience
: Having a good understanding of the failure modes and self-healing methods. As with scaling, a lot of this is taken care of by the serverless offering; however, one needs to know the limitations of this.

Availability
: Another area where in a serverless setup the control points are limited. The current offerings come with few service-level objectives or agreements, and status pages are typically not provided. The monitoring focus should hence be more on the platform than on the function level.

Maintainability
: Of the function code itself. Since the code is very specific and has a sharp focus, the length of the function shouldn't be a problem. However, understanding how a bunch of functions work together to achieve some goal is vital.

Visibility
: Typically limited by what the serverless provider allows; very often little is known about the underlying infrastructure (OS level, container, etc.).

Interestingly, the way serverless computing addresses many of these aspects seems to be what makes it most attractive. The result of a Twitter poll carried out by DevOps legend Patrick Debois in May 2016 highlights this (see Figure 3-1).

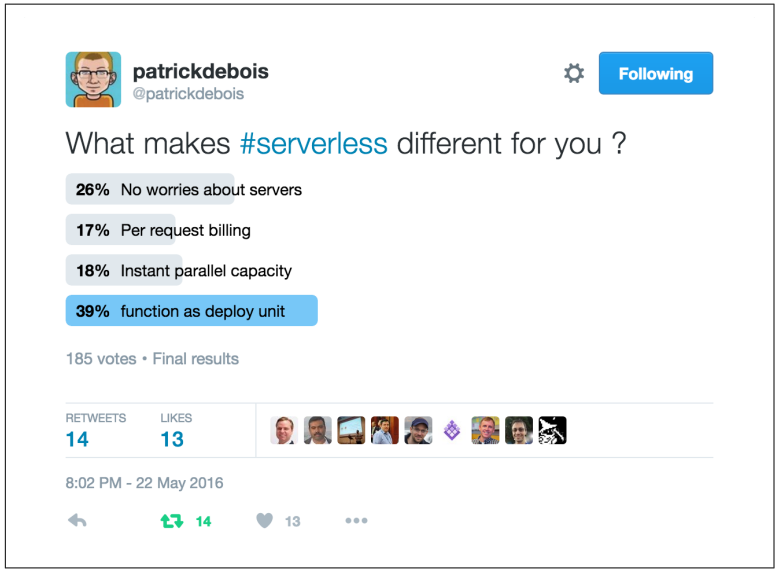

Figure 3-1. Twitter poll: What makes serverless different for you?

As pointed out by Andy Warzon, there are a number of responsibilities found in traditional admin roles that are not applicable in a serverless setup:

- OS-level configuration management and (security) patches are not required, since the execution environment is fixed and managed by the serverless provider.
- Backups are not necessary since functions are supposed to be stateless.
- Service-level scaling is typically a feature of the serverless platform.

Many activities that were traditionally expected to be carried out by the operations team, such as deployments or monitoring, are now the responsibility of the AppOps. However, the infrastructure team has a number of new responsibilities that we will discuss in the next section.

Infrastructure Team Checklist

As a member of the infrastructure team, you act as a coach and guide to AppOps. Here are a couple of ways you can support your colleagues:

- Make sure that the functions are versioned properly. A function's source code should reside in a (ideally distributed) version control system such as Git. This is an infrastructure task that you should manage, along with enforcing the respective policies around access and push rights.
- Keep track of the overall picture—that is, the full set of functions, potentially owned by a number of AppOps—so you can provide recommendations about when to go serverless (as described in Chapter 4) and when it makes more (economic) sense to move back to a dedicated-machine solution.
- Support the troubleshooting process. Since serverless functions typically depend on external systems such as (managed) storage, you can help establish good practices around logging. Further, there may be cases where you can provide insights—for example, in the form of access to additional logs—when an AppOp debugs a function that is either not working correctly or has a higher than normal execution error rate.
- Provide insights regarding load testing of serverless functions. The infrastructure team's holistic view is particularly valuable here.
- Identify potential cost optimizations. While with serverless solutions, there's no capacity planning in the traditional sense, AppOps can make better-informed decisions about the few resource consumption parameters (such as RAM) under their control when the infrastructure team can offer guidance in terms of overall usage.

Conclusion

In this chapter we had a look at the new roles encouraged and to a certain extent required by the serverless paradigm. The traditional developer role morphs into an AppOps role, responsible for not only writing the code but also monitoring and troubleshooting it. In addition, the infrastructure team doesn't have to perform certain tasks required in, say, VM-based deployments, such as patching or scaling, and therefore can take on new responsibilities such as load testing and act as advisors for AppOps. Now we're in a position to look at application areas where serverless computing is a good fit and what the limitations and challenges of this new paradigm are.

CHAPTER 4

Serverless Operations Field Guide

This chapter is meant as a guide to help you decide when and where to use serverless computing. We will talk about application areas and review concrete use cases for it. Then we'll turn our attention to the limitations of serverless computing, potential gotchas, and a migration guide from a monolithic application. Last but not least, we will have a look at a simple walkthrough example to discuss the implications for operations as outlined in the previous chapter.

Latency Versus Access Frequency

Before you embark on the serverless journey, you might want to ask yourself how applicable the serverless paradigm is for the use case at hand. There may be an array of deciding factors for your use case, which can be summed up in two categories: technical and economic. Technical requirements could be supported programming languages, available triggers, or integration points supported by a certain offering. On the other hand, you or the budget holder are probably also interested in the costs of using the service (at least in the context of a public cloud offering, where these are often more transparent).

Figure 4-1 provides a rough guide for the applicability of serverless computing along two dimensions: latency and access frequency.

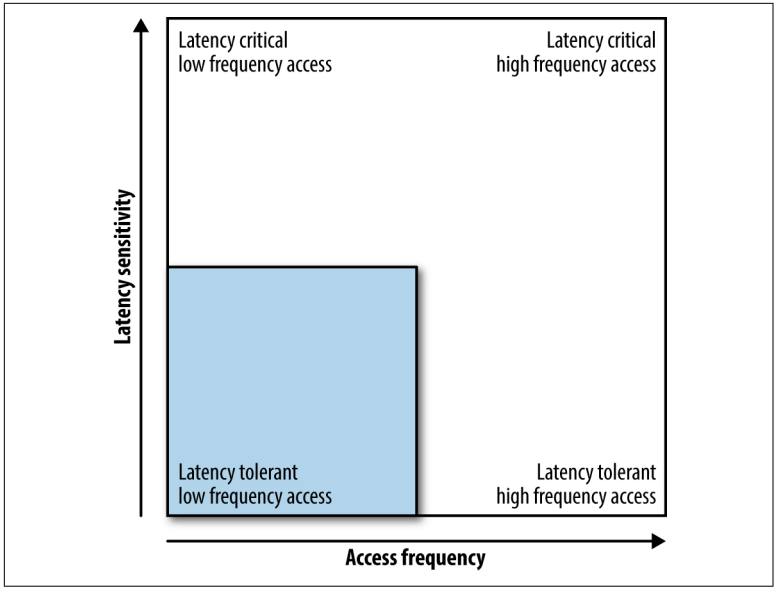

Figure 4-1. Latency sensitivity versus access frequency

By latency, I mean how much time can acceptably elapse between function invocation and termination. It might be important for your use case that you have guarantees around latency—for example, that the 90th percentile cannot exceed 100 ms. It might also be the case that your use case requires an overall low latency. For example, when creating a resized version of a user's profile image, you might not care if it takes 1 second or 5 seconds; on the other hand, when a user wants to check out a shopping basket, you don't want to risk any delays as these might lead to abandonment and loss of revenue.

Independent from the latency and determined by the workload is the access frequency. A certain functionality might only be used once per hour, whereas in another case you're dealing with many concurrent requests, effectively establishing a permanent access pattern. Think of a user checking in at a certain location, triggering an update of a score, versus the case of an online chat environment.

To sum up the guidance that one can derive from the latency-versus-frequency graph, serverless computing is potentially a great fit for workloads that are in the lower-left quadrant of Figure 4-1—that is, use cases that are latency tolerant with a relatively low access frequency. The higher the access frequency and the higher the expectations around latency, the more it usually pays off to have a

dedicated machine or container processing the requests. Granted, I don't provide you with absolute numbers here, and the boundaries will likely be pushed in the future; however, this should provide you with a litmus test to check the general applicability of the paradigm. In addition, if you already have a serverless deployment, the infrastructure team might be able to supply you with data concerning the overall usage and costs. Equipped with this, you'll be in a better position to decide if serverless computing continues to make sense from an economic point of view.

When (Not) to Go Serverless

There are a number of cases where serverless computing is a great fit, mainly centered around rather short-running, stateless jobs in an event-driven setup. These are usually found in mobile apps or IoT applications, such as a sensor updating its value once per day. The reason the paradigm works in this context is that you're dealing with relatively simple operations executing for a short period of time. Let's now have a look at some concrete application areas and use cases.

Application Areas and Use Cases

Typical application areas of serverless computing are:

- Infrastructure and glue tasks, such as reacting to an event triggered from cloud storage or a database
- Mobile and IoT apps to process events, such as user check-in or aggregation functions
- Image processing, for example to create preview versions of an image or extract key frames from a video
- Data processing, like simple extract, transform, load (ETL) pipelines to preprocess datasets

Let's now have a closer look at a concrete example of how the paradigm is applied. LambCI is a serverless continuous integration (CI) system. Michael Hart, the creator of LambCI, was motivated to develop LambCI out of frustration with existing CI systems; in his own words:

> You'll be under- or overutilized, waiting for servers to free up or paying for server power you're not using. And this, for me, is where the advantage of a serverless architecture really comes to light: 100% utilization, coupled with instant invocations.
>
> —Introducing LambCI—a serverless build system,, *July 2016*

The architecture of LambCI is shown in Figure 4-2: it is essentially utilizing the Amazon Simple Notification Service (SNS) to listen to GitHub events and triggering a Lambda function that carries out the actual build, with the resulting build artifacts stored in S3 and build configuration and metadata kept in DynamoDB.

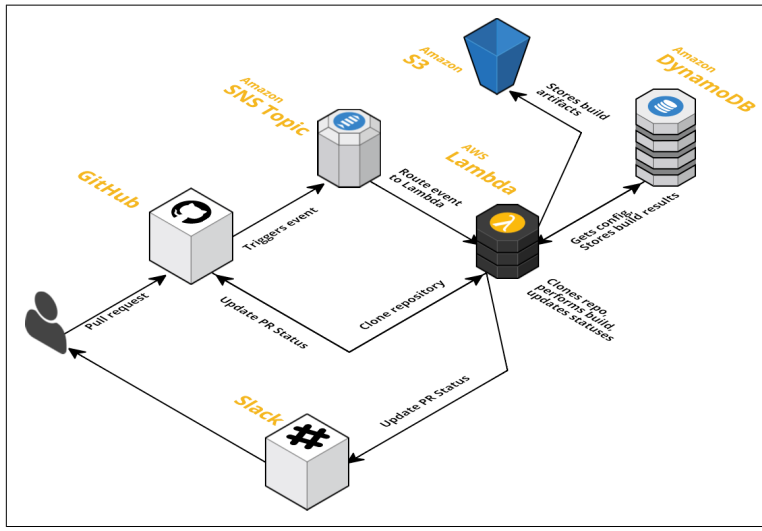

Figure 4-2. LambCI architecture

Limitations of LambCI at the moment are that there is no HTTP interface available (i.e., one has to interface with SNS), no root access can be provided (that is, it's not suitable for building Docker images), and the build time is capped at five minutes. Nevertheless, since LambCI can be deployed based on a CloudFormation stack, using it can save a lot of money, especially for many shorter-running builds.

Other exemplary use cases for serverless architectures include but are not limited to the following:

- Forwarding AWS alerts to Slack to support chatops
- Blocking abusive IP addresses in CloudFlare

- Migrating an email marketing tool for small business
- Providing IRC notifications, as in IRC Hooky
- Powering Slackbots
- Calculating lineups for a fantasy game, as reported in "30K Page Views for $0.21: A Serverless Story"
- Carrying out continuous deployments
- Implementing a ticketing system
- Realizing an IoT service, as in iRobots
- Doing video processing
- Replacing cron jobs
- Fetching nearby Pokemon Go data
- Integrating Keen.io with CloudWatch

Serverless computing is growing in popularity, and as we saw in Chapter 2, the number of offerings is increasing. Does this mean that in the future we will eventually migrate everything to serverless? I don't think so, and next we will have a look at challenges with the serverless paradigm that might help clarify why I don't think this will be the case.

Challenges

While the serverless paradigm without doubt has its use cases and can help simplify certain workloads, there are naturally limitations and challenges. From most pressing to mildly annoying, these include:

- Stateful services are best implemented outside of serverless functions. Integration points with other platform services such as databases, message queues, or storage are therefore extremely important.
- Long-running jobs (in the high minutes to hours range) are usually not a good fit; typically you'll find timeouts in the (high) seconds range.
- Logging and monitoring are a challenge: the current offerings provide little support for these operational necessities, and on top of that, the expectations are quite different than in traditional environments due to the short lifecycle.

- Local development can be challenging: usually developers need to develop and test within the online environment.
- Language support is limited: most serverless offerings support only a handful of programming languages.

Another criticism of serverless computing is the lock-in aspect, as discussed in "Cloud or on-Premises?" on page 15.

In addition to these points, a range of opinions have been voiced on the overall concept and the positioning of the serverless approach (for example, on Hacker News). This can serve as a baseline in terms of expectation management as well as a reminder of how young and fluent the ecosystem is.

Migration Guide

The process of migrating a monolithic application to a serverless architecture is by and large comparable with that of migrating to a microservices architecture, leaving stateful aspects aside. Probably the most important question to ask is: does it make sense? As discussed in "Latency Versus Access Frequency" on page 25 and "Challenges" on page 29, not all parts of a monolith are a good match for the stateless, event-driven, and batch-oriented nature of serverless functions. Furthermore, in comparison to breaking down a monolith into, say, 50 microservices, you might find yourself with hundreds of functions. In this situation, a migration of the whole system can be hard to manage and troubleshoot. A better approach might be to identify the workloads that are a good fit and migrate only this functionality.

Walkthrough Example

In this section, we will be using AWS Lambda for a simple walkthrough example to demonstrate the implications for operations, as outlined in Chapter 3. Note that the goal of the exercise is not to provide you with an in-depth explanation of Lambda but to discuss typical workflows and potential challenges or limitations you might experience. The hope is that, equipped with this knowledge, you'll be better prepared when you decide to apply the serverless paradigm in your own organization or project.

Preparation

For the walkthrough example, I'll be using a blueprint: `s3-get-object-python`. This blueprint, as shown in Figure 4-3, is written in Python and employs an S3 trigger to retrieve metadata for that S3 object when it is updated.

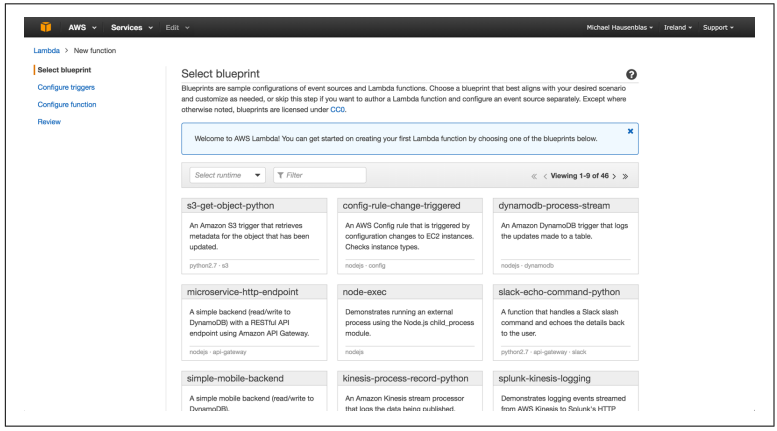

Figure 4-3. AWS Lambda dashboard: selecting a blueprint

Also, as a preparation step, I've created an S3 bucket called `serops-we` that we will be using shortly.

Trigger Configuration

In the first step, depicted in Figure 4-4, I configure and enable the trigger: every time a file is uploaded into the `serops-we` bucket, the trigger should fire. The necessary permissions for S3 to invoke the Lambda function are automatically added in this step.

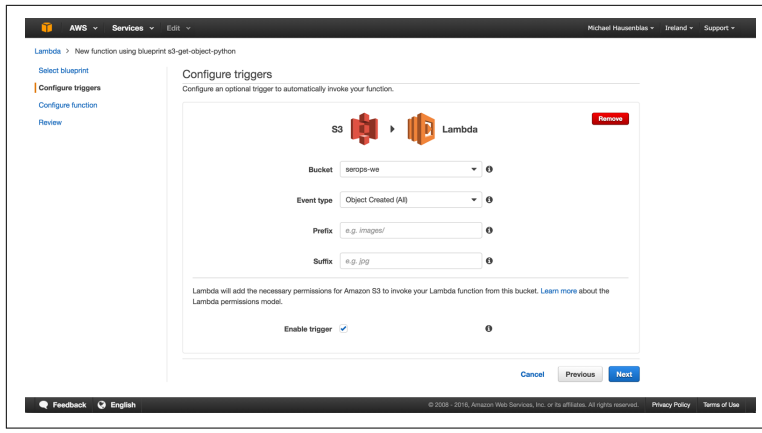

Figure 4-4. Configuring the S3 trigger

Note that in this step I could also have applied certain filters, using the Prefix and Suffix fields, for example, to only react to events from a certain file type.

Function Definition

The next step, configuring the Lambda function, comprises a number of substeps, so let's take these one by one. First we need to provide a name for the function (I'm using `s3-upload-meta` here; see Figure 4-5), and we can enter a description as well as selecting a runtime (Python 2.7 in our case).

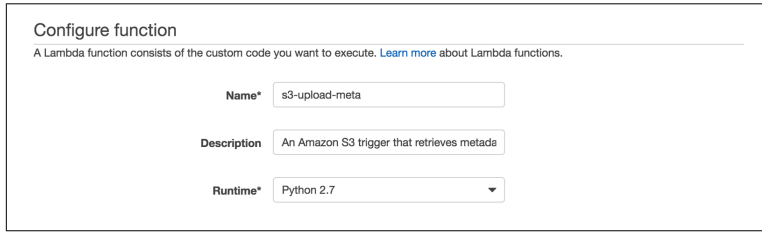

Figure 4-5. Configuring the Lambda function: setting global properties

Next comes the actual definition of the function code, as shown in Figure 4-6. For the purpose of this example, I opted for the most primitive option, defining the code inline. Other options are to upload a ZIP file from local storage or S3. In a production setup, you'd likely have your CI/CD pipeline putting the code on S3.

In this step, also note the function signature, `lambda_handler(event, context)`: while the name of the handler can be arbitrarily chosen, the parameters are fixed in terms of order and type.

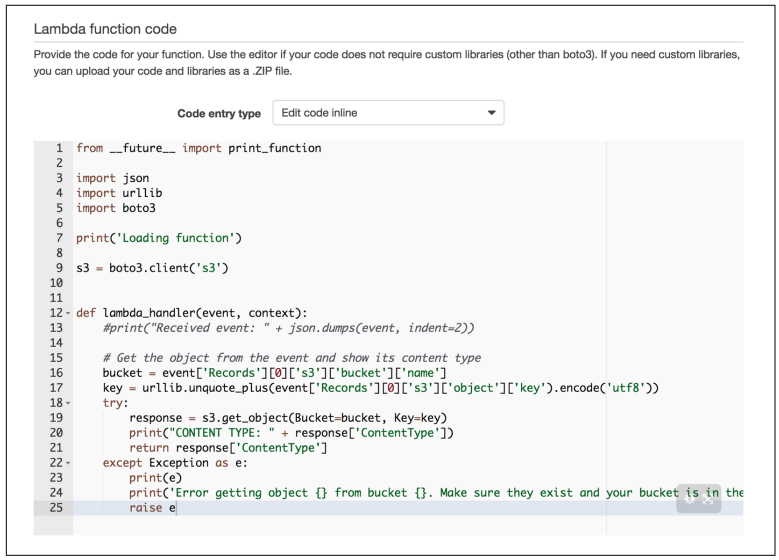

Figure 4-6. Providing the Lambda function code

Now we need to provide some wiring and access information. In this substep, depicted in Figure 4-7, I declare the handler name as chosen in the previous step (`lambda_handler`) as well as the necessary access permissions. For that, I create a new role called `lambda-we` using a template that defines a read-only access policy on the S3 bucket `serops-we` I prepared earlier. This allows the Lambda function to access the specified S3 bucket.

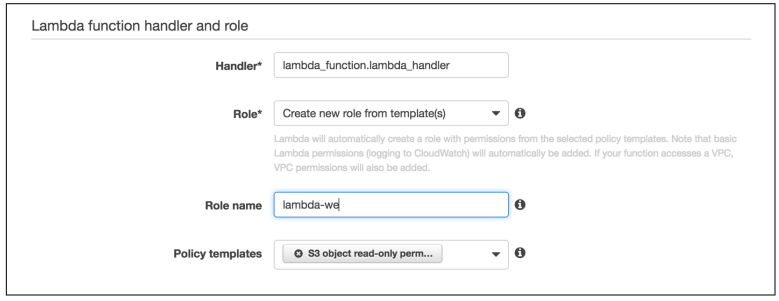

Figure 4-7. Defining the entry point and access control

The last substep to configure the Lambda function is to (optionally) specify the runtime resource consumption behavior (see Figure 4-8).

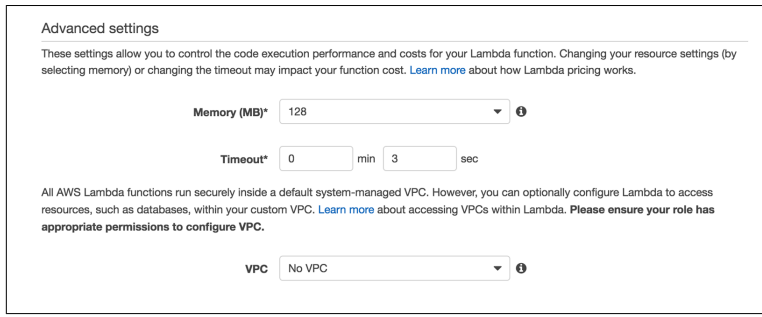

Figure 4-8. Setting the runtime resources

The main parameters here are the amount of available memory you want the function to consume and how long the function is allowed to execute. Both parameters influence the costs, and the (nonconfigurable) CPU share is determined by the amount of RAM you specify.

Review and Deploy

It's now time to review the setup and deploy the function, as shown in Figure 4-9.

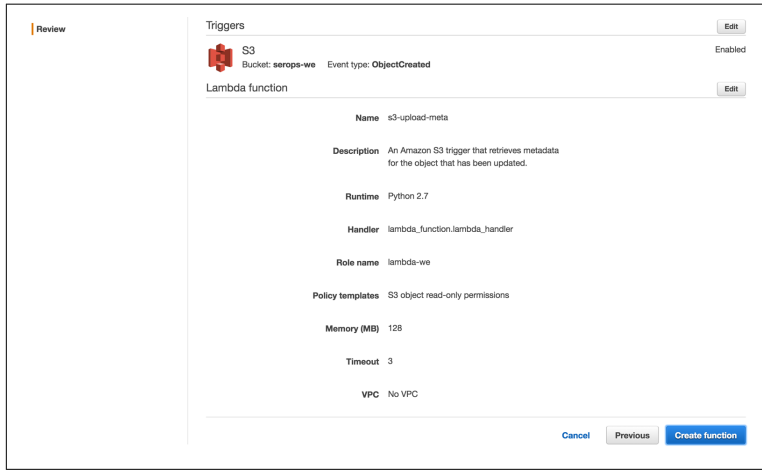

Figure 4-9. Reviewing and deploying the function

The result of the previous steps is a deployed Lambda function like the one in Figure 4-10.

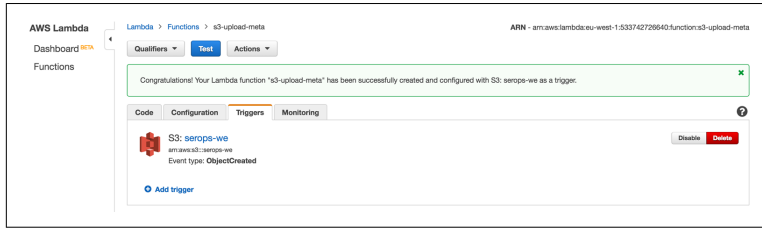

Figure 4-10. The deployed Lambda function

Note the trigger, the S3 bucket `serops-we`, and the available tabs, such as Monitoring.

Invoke

Now we want to invoke our function, `s3-upload-meta`: for this we need to switch to the S3 service dashboard and upload a file to the S3 bucket `serops-we`, as depicted in Figure 4-11.

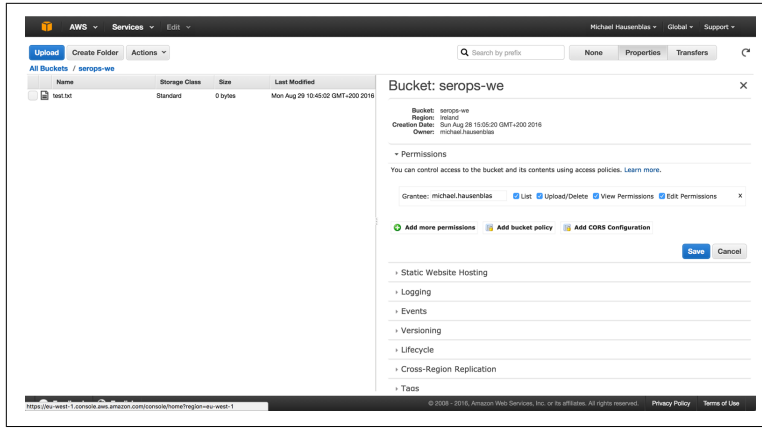

Figure 4-11. Triggering the Lambda function by uploading a file to S3

If we now take a look at the Monitoring tab back in the Lambda dashboard, we can see the function execution there (Figure 4-12). Also available from this tab is the "View logs in CloudWatch" link in the upper-right corner that takes you to the execution logs.

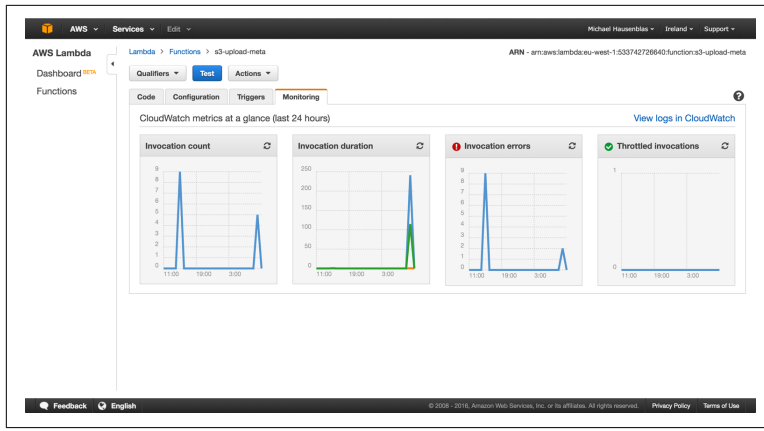

Figure 4-12. Monitoring the function execution

As we can see from the function execution logs in Figure 4-13, the function has executed as expected. Note that the logs are organized in so-called streams, and you can filter and search in them. This is especially relevant for troubleshooting.

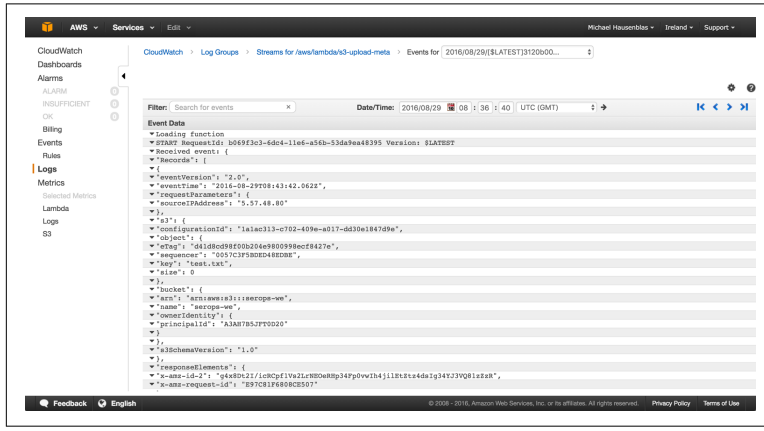

Figure 4-13. Accessing the function execution logs

That's it. A few steps and you have a function deployed and running. But is it really that easy? When applying the serverless paradigm to real-world setups within existing environments or trying to migrate (parts of) an existing application to a serverless architecture, as discussed in "Migration Guide" on page 30, one will likely face a number of questions. Let's now have a closer look at some of the steps

from the walkthrough example from an AppOps and infrastructure team perspective to make this a bit more explicit.

Where Does the Code Come From?

At some point you'll have to specify the source code for the function. No matter what interface you're using to provision the code, be it the command-line interface or, as in Figure 4-6, a graphical user interface, the code comes from somewhere. Ideally this is a (distributed) version control system such as Git and the process to upload the function code is automated through a CI/CD pipeline such as Jenkins or using declarative, templated deployment options such as CloudFormation.

In Figure 4-14 you can see an exemplary setup (focus on the green labels 1 to 3) using Jenkins to deploy AWS Lambda functions. With this setup, you can tell who has introduced a certain change and when, and you can roll back to a previous version if you experience troubles with a newer version.

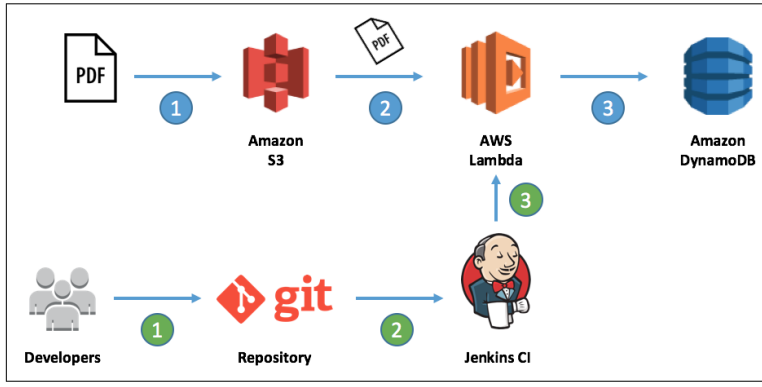

Figure 4-14. Automated deployment of Lambdas using Jenkins (kudos to AWS)

How Is Testing Performed?

If you're using public cloud, fully managed offerings such as Azure Functions or AWS Lambda, you'll typically find some for (automated) testing. Here, self-hosted offerings usually have a slight advantage: while in managed offerings certain things can be tested in a straightforward manner (on the unit test level), you typically don't get to replicate the entire cloud environment, including the triggers

and integration points. The consequence is that you typically end up doing some of the testing online.

Who Takes Care of Troubleshooting?

The current offerings provide you with integrations to monitoring and logging, as I showed you in Figure 4-12 and Figure 4-13. The upside is that, since you're not provisioning machines, you have less to monitor and worry about; however, you're also more restricted in what you get to monitor.

Multiple scenarios are possible: while still in the development phase, you might need to inspect the logs to figure out why a function didn't work as expected; once deployed, your focus shifts more to why a function is performing badly (timing out) or has an increased error count. Oftentimes these runtime issues are due to changes in the triggers or integration points. Both of those scenarios are mainly relevant for someone with an AppOps role.

From the infrastructure team's perspective, studying trends in the metrics might result in recommendations for the AppOps: for example, to split a certain function or to migrate a function out of the serverless implementation if the access patterns have changed drastically (see also the discussion in "Latency Versus Access Frequency" on page 25).

How Do You Handle Multiple Functions?

Using and managing a single function as a single person is fairly easy. Now consider the case where a monolith has been split up into hundreds of functions, if not more. You can imagine the challenges that come with this: you need to figure out a way to keep track of all the functions, potentially using tooling like Netflix Vizceral (originally called Flux).

Conclusion

This chapter covered application areas and use cases for serverless computing to provide guidance about when it's appropriate (and when it's not), highlighting implications for operations as well as potential challenges in the implementation phase through a walkthrough example.

With this chapter, we also conclude this report. The serverless paradigm is a powerful and exciting one, still in its early days but already establishing itself both in terms of support by major cloud players such as AWS, Microsoft, and Google and in the community.

At this juncture, you're equipped with an understanding of the basic inner workings, the requirements, and expectations concerning the team (roles), as well as what offerings are available. I'd suggest that as a next step you check out the collection of resources—from learning material to in-use examples to community activities—in Appendix B. When you and your team feel ready to embark on the serverless journey, you might want to start with a small use case, such as moving an existing batch workload to your serverless platform of choice, to get some experience with it. If you're interested in rolling your own solution, Appendix A gives an example of how this can be done. Just remember: while serverless computing brings a lot of advantages for certain workloads, it is just one tool in your toolbox—and as usual, one size does not fit all.

APPENDIX A

Roll Your Own Serverless Infrastructure

Here we will discuss a simple proof of concept (POC) for a serverless computing implementation using containers.

Note that the following POC is of an educational nature. It serves to demonstrate how one could go about implementing a serverless infrastructure and what logic is typically required; the discussion of its limitations at the end of this appendix will likely be of the most value for you, should you decide to roll your own infrastructure.

Flock of Birds Architecture

So, what is necessary to implement a serverless infrastructure? Astonishingly little, as it turns out: I created a POC called Flock of Birds (FoB), using DC/OS as the underlying platform, in a matter of days.

The underlying design considerations for the FoB proof of concept were:

- The service should be easy to use, and it should be straightforward to integrate the service.
- Executing different functions must not result in side effects; each function must run in its own sandbox.

- Invoking a function should be as fast as possible; that is, long ramp-up times should be avoided when invoking a function.

Taken together, the requirements suggest a container-based implementation. Now let's have a look at how we can address them one by one.

FoB exposes an HTTP API with three public and two internal endpoints:

- `POST /api/gen` with a code fragment as its payload generates a new function; it sets up a language-specific sandbox, stores the user-provided code fragment, and returns a function ID, `$fun_id`.
- `GET /api/call/$fun_id` invokes the function with ID `$fun_id`.
- `GET /api/stats` lists all registered functions.
- `GET /api/meta/$fun_id` is an internal endpoint that provides for service runtime introspection, effectively disclosing the host and port the container with the respective function is running on.
- `GET /api/cs/$fun_id` is an internal endpoint that serves the code fragment that is used by the driver to inject the user-provided code fragment.

The HTTP API makes FoB easy to interact with and also allows for integration, for example, to invoke it programmatically.

Isolation in FoB is achieved through drivers. This is specific code that is dependent on the programming language; it calls the user-provided code fragment. For an example, see the Python driver. The drivers are deployed through sandboxes, which are templated Marathon application specifications using language-specific Docker images. See Example A-1 for an example of the Python sandbox.

Example A-1. Python sandbox in FoB

```
{
    "id": "fob-aviary/$FUN_ID",
    "cpus": 0.1,
    "mem": 100,
    "cmd": "curl $FUN_CODE > fobfun.py && python fob_driver.py",
    "container": {
        "type": "DOCKER",
        "docker": {
            "image": "mhausenblas/fob:pydriver",
            "forcePullImage": true,
            "network": "BRIDGE",
            "portMappings": [
                {
                    "containerPort": 8080,
                    "hostPort": 0
                }
            ]
        }
    },
    "acceptedResourceRoles": [
        "slave_public"
    ],
}
```

At registration time, the `id` of the Marathon app is replaced with the actual UUID of the function, so `fob-aviary/$FUN_ID` turns into something like `fob-aviary/5c2e7f5f-5e57-43b0-ba48-bacf40f666ba`. Similarly, `$FUN_CODE` is replaced with the storage location of the user-provided code, something like *fob.marathon.mesos/api/cs/5c2e7f5f-5e57-43b0-ba48-bacf40f666ba*. When the container is deployed, the `cmd` is executed, along with the injected user-provided code.

Execution speed in FoB is improved by decoupling the registration and execution phases. The registration phase—that is, when the client invokes `/api/gen`—can take anywhere from several seconds to minutes, mainly determined by how fast the sandbox Docker image is pulled from a registry. When the function is invoked, the driver container along with an embedded app server that listens to a certain port simply receives the request and immediately returns the result. In other words, the execution time is almost entirely determined by the properties of the function itself.

Figure A-1 shows the FoB architecture, including its main components, the dispatcher, and the drivers.

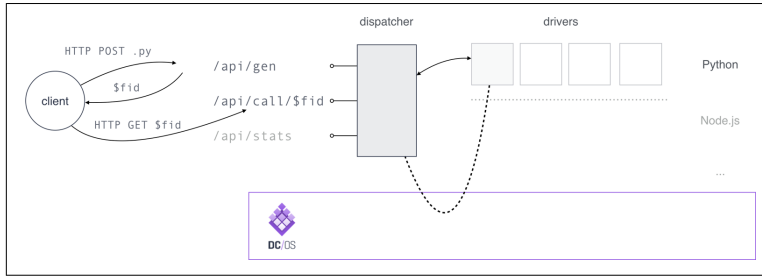

Figure A-1. Flock of Birds architecture

A typical flow would be as follows:

1. A client posts a code snippet to `/api/gen`.
2. The dispatcher launches the matching driver along with the code snippet in a sandbox.
3. The dispatcher returns `$fun_id`, the ID under which the function is registered, to the client.
4. The client calls the function registered above using `/api/call/$fun_id`.
5. The dispatcher routes the function call to the respective driver.
6. The result of the function call is returned to the client.

Both the dispatcher and the drivers are stateless. State is managed through Marathon, using the function ID and a group where all functions live (by default called `fob-aviary`).

Interacting with Flock of Birds

With an understanding of the architecture and the inner workings of FoB, as outlined in the previous section, let's now have a look at the concrete interactions with it from an end user's perspective. The goal is to register two functions and invoke them.

First we need to provide the functions, according to the required signature in the driver. The first function, shown in Example A-2, prints `Hello serverless world!` to standard out and returns `42` as

a value. This code fragment is stored in a file called *helloworld.py*, which we will use shortly to register the function with FoB.

Example A-2. Code fragment for the "hello world" function

```
def callme():
    print("Hello serverless world!")
    return 42
```

The second function, stored in *add.py*, is shown in Example A-3. It takes two numbers as parameters and returns their sum.

Example A-3. Code fragment for the add function

```
def callme(param1, param2):
    if param1 and param2:
        return int(param1) + int(param2)
    else:
        return None
```

For the next steps, we need to figure out where the FoB service is available. The result (IP address and port) is captured in the shell variable `$FOB`.

Now we want to register *helloworld.py* using the `/api/gen` endpoint. Example A-4 shows the outcome of this interaction: the endpoint returns the function ID we will subsequently use to invoke the function.

Example A-4. Registering the "hello world" function

```
$ http POST $FOB/api/gen < helloworld.py
HTTP/1.1 200 OK
Content-Length: 46
Content-Type: application/json; charset=UTF-8
Date: Sat, 02 Apr 2016 23:09:47 GMT
Server: TornadoServer/4.3

{
    "id": "5c2e7f5f-5e57-43b0-ba48-bacf40f666ba"
}
```

We do the same with the second function, stored in *add.py*, and then list the registered functions as shown in Example A-5.

Example A-5. Listing all registered functions

```
$ http $FOB/api/stats
{
    "functions": [
        "5c2e7f5f-5e57-43b0-ba48-bacf40f666ba",
        "fda0c536-2996-41a8-a6eb-693762e4d65b"
    ]
}
```

At this point, the functions are available and are ready to be used. Let's now invoke the `add` function with the ID `fda0c536-2996-41a8-a6eb-693762e4d65b`, which takes two numbers as parameters. Example A-6 shows the interaction with `/api/call`, including the result of the function execution—which is, unsurprisingly and as expected, `2` (since the two parameters we provided were both `1`).

Example A-6. Invoking the add function

```
$ http $FOB/api/call/fda0c536-2996-41a8-a6eb-693762e4d65b?
  param1:1,param2:1
{
    "result": 2
}
```

As you can see in Example A-6, you can also pass parameters when invoking the function. If the cardinality or type of the parameter is incorrect, you'll receive an HTTP 404 status code with the appropriate error message as the JSON payload; otherwise, you'll receive the result of the function invocation.

Limitations of Flock of Birds

Naturally, FoB has a number of limitations, which I'll highlight in this section. If you end up implementing your own solution, you should be aware of these challenges. Ordered from most trivial to most crucial for production-grade operations, the things you'd likely want to address are:

- The only programming language FoB supports is Python. Depending on the requirements of your organization, you'll likely need to support a number of programming languages.

Supporting other interpreted languages, such as Ruby or JavaScript, is straightforward; however, for compiled languages you'll need to figure out a way to inject the user-provided code fragment into the driver.

- If exactly-once execution semantics are required, it's up to the function author to guarantee that the function is idempotent.
- Fault tolerance is limited. While Marathon takes care of container failover, there is one component that needs to be extended to survive machine failures. This component is the dispatcher, which stores the code fragment in local storage, serving it when required via the `/api/meta/$fun_id` endpoint. In order to address this, you could use an NFS or CIFS mount on the host or a solution like Flocker or REX-Ray to make sure that when the dispatcher container fails over to another host, the functions are not lost.
- A rather essential limitation of FoB is that it doesn't support autoscaling of the functions. In serverless computing, this is certainly a feature supported by most commercial offerings. You can add autoscaling to the respective driver container to enable this behavior.
- There are no integration points or explicit triggers. As FoB is currently implemented, the only way to execute a registered function is through knowing the function ID and invoking the HTTP API. In order for it to be useful in a realistic setup, you'd need to implement triggers as well as integrations with external services such as storage.

By now you should have a good idea of what it takes to build your own serverless computing infrastructure.

For a selection of pointers to in-use examples and other useful references, see Appendix B.

APPENDIX B
References

What follows is a collection of links to resources where you can find background information on topics covered in this book or advanced material, such as deep dives, teardowns, example applications, or practitioners' accounts of using serverless offerings.

General

- Serverless: Volume Compute for a New Generation (RedMonk)
- ThoughtWorks Technology Radar
- Five Serverless Computing Frameworks To Watch Out For
- Debunking Serverless Myths
- The Serverless Start-up - Down With Servers!
- 5 killer use cases for AWS Lambda
- Serverless Architectures (Hacker News)
- The Cloudcast #242 - Understanding Serverless Applications

Community and Events

- Serverless on Reddit
- Serverless Meetups
- Serverlessconf

- anaibol/awesome-serverless, a community-curated list of offerings and tools
- JustServerless/awesome-serverless, a community-curated list of posts and talks
- ServerlessHeroes/serverless-resources, a community-curated list of serverless technologies and architectures

Tooling

- Serverless Cost Calculator
- Kappa, a command-line tool for Lambda
- Lever OS
- Vandium, a security layer for your serverless architecture

In-Use Examples

- AWS at SPS Commerce (including Lambda & SWF)
- AWS Lambda: From Curiosity to Production
- A serverless architecture with zero maintenance and infinite scalability
- Introduction to Serverless Architectures with Azure Functions
- Serverless is more than just "nano-compute"
- Observations on AWS Lambda Development Efficiency
- 3 Reasons AWS Lambda Is Not Ready for Prime Time

About the Author

Michael Hausenblas is a developer advocate at Mesosphere, where he helps AppOps to build and operate distributed services. His background is in large-scale data integration, Hadoop/NoSQL, and IoT, and he's experienced in advocacy and standardization (W3C and IETF). Michael contributes to open source software, such as the DC/OS project, and shares his experience with distributed systems and large-scale data processing through code, blog posts, and public speaking engagements.