

SERVERLESS COMPUTING

eMag Issue 53 - May 2017



InfoQ
neue

ARTICLE

Serverless
Takes DevOps to
the Next Level

VIRTUAL PANEL

A Practical Approach
to Serverless
Computing

ARTICLE

The Future
of Serverless
Compute

Serverless Takes DevOps to the Next Level

Serverless doesn't only supplement DevOps, but it goes beyond the current thinking on how IT organisations can achieve greater business agility. It's geared towards the rapid delivery of business value and continuous improvement and learning, and as such has clear potential to drive substantial cultural change, even in organisations that have adopted DevOps culture and practices already.

Examining the Internals of a Serverless Platform: Moving Towards a Zero-Friction PaaS

Platform-as-a-Service (PaaS) architectures simplify how applications are deployed for developers, and serverless architecture or Functions-as-a-Service (FaaS) is the next step in that direction. This article explores how serverless platforms are built, operated and leveraged for rapid application delivery. A conclusion of this work is that the serverless space is ripe for innovation.

The Future of Serverless Compute

As Serverless approaches end of early-adopter phase, Mike Roberts puts on prediction goggles on where this movement is going next and what changes are needed from organizations in order to support it.

Lambda Functions Versus Infrastructure: Are We Trading Apples for Oranges?

Amazon's AWS Lambda service lets us run code without provisioning servers. Serverless offerings have recently been receiving interest due to their simplicity, capabilities and potential for cost reductions. This article compares the tradeoffs of serverless models with VM/Container based models.

Virtual Panel: A Practical Approach to Serverless Computing

Add serverless computing to the growing list of options developers have when building software. Serverless products—more accurately referred to as Functions-as-a-Service—offer incredible simplicity, but at a cost. To learn more about this exciting space and the practical implications, InfoQ reached out to three experienced technologists.

FOLLOW US



facebook.com
/InfoQ



@InfoQ



google.com
/+InfoQ



linkedin.com
company/infoq

CONTACT US

GENERAL FEEDBACK feedback@infoq.com

ADVERTISING sales@infoq.com

EDITORIAL editors@infoq.com

APP **DYNAMICS**

Find & fix app performance issues faster.

Discover how
APM keeps your
customers happier.

Take a Tour

RICHARD SEROTER

is a Senior Director of Product at Pivotal, with a master's degree in Engineering from the University of Colorado. He's also a 9-time Microsoft MVP, trainer for developer-centric training company Pluralsight, speaker, the lead InfoQ editor for cloud computing, and author of multiple books on application integration strategies. Richard maintains a regularly updated blog on topics of architecture and solution design and can be found on Twitter as @rseroter



A LETTER FROM THE EDITOR

Focus on what matters. Avoid distraction. Any credible self-help book tells you that. We're advised to prioritize and pay attention to whatever makes the biggest difference. Serverless computing, also called Function-as-a-Service, fits that mindset. The goal of serverless functions--compared to virtual machines or containers--is for developers to focus onto what matters: their code.

What are examples of serverless platforms? There's the original, AWS Lambda. Quick to follow were Microsoft with Azure Functions and Google with Google Cloud Functions. Other credible players include IBM OpenWhisk and Webtask from Auth0. All these have something in common: they take code and run it for you. As a developer, you don't worry about choosing server types, configuring operating systems, adjusting middleware, scaling, or load balancing. Basically, the only thing you deal with is your code. Sound perfect? Not so fast.

Functions are very exciting and offer intriguing possibilities. But they're not a simple replacement for how you design and host software today. Functions cater to the microservices architecture pattern. They're small, single-purpose chunks of code. Functions are typically event-driven, stateless, and processed asynchronously. Building such Functions requires a different type of development discipline. Operations is another area to consider. While it is improving, the

tooling for monitoring, tracing, and operating Functions is immature. All of this means that serverless Functions fit certain scenarios today, and have potential for even greater impact in the years ahead.

In this InfoQ eMag, we curated some of the best serverless content into a single asset. We want to give you a relevant, pragmatic look at this emerging space. We've chosen five compelling pieces that give you the broadest perspective on serverless. First, we look at early adopter Mike Roberts' take on the future of serverless. Next, you'll find a piece from Rafal Gancarz that looks at how serverless takes DevOps to the next level. How should someone coming from an infrastructure mindset look at serverless Functions? Dan Sheehan considers this point. Some consider serverless platforms to be the next evolution of PaaS. Here, a group of authors evaluate what a zero-friction experience looks like. Finally, we have an insightful virtual panel consisting of a diverse set of experts.

You'll learn a lot from this set of stories, and finish this eMag with a clear-eyed view of the potential and risks of using serverless platforms. Enjoy!



Serverless Takes DevOps to the Next Level



Rafal Gancarz is a lead consultant for OpenCredo, a London-based consultancy specialising in helping clients build and deploy emerging technologies to deliver business value. He is an experienced technologist with expertise in architecting and delivering large-scale distributed systems. Gancarz is also an experienced agile practitioner and a Certified Scrum Master, and is always interested in improving project delivery. He has spent the last 18 months working on a large-scale project using serverless technologies on the AWS platform and has first-hand experience using the serverless stack for building enterprise-grade distributed systems.

[Serverless computing](#) is changing the way that software systems are being built and operated. Despite being a relatively new area within the IT industry, serverless has the potential to massively alter the way that software delivers business value.

It allows running cheaply available and scalable cloud workloads, which is ideal for many product types and business use cases.

But serverless won't only change the way that software is delivered to the user. It will change the organisations that deliver the software too, which I believe is the more profound impact that serverless will have on the IT industry. This article will explore

the ways that serverless will alter the culture of organisations that adopt it for software delivery and how it will affect the whole industry.

The state of DevOps

Anybody who hasn't lived under a rock for the past few years must already have heard about [DevOps](#). The DevOps movement encompasses and extends [agile software development](#) into the

field of IT operations and aims to deliver greater business agility by fostering strong collaboration between development and operations teams as well as the adoption of novel operational practices, particularly around infrastructure provisioning, improved release management, and operational tooling.

DevOps has become widely adopted in the IT industry, often in conjunction with cloud comput- ►

KEY TAKEAWAYS

Many organisations still struggle with DevOps adoption.

Serverless not only embraces the DevOps culture but has a pure form of DevOps in its DNA.

Serverless makes operability a first-class concern and embeds it in the development process.

In the future, broadly skilled cloud engineers will replace many IT specialists.

Serverless will change the culture of IT organisations and impact how companies use cloud computing.

ing and container technologies. At the same time, a lot of organisations are struggling to realise the full benefits of DevOps, mostly due to expertise shortages and organisational challenges. Despite these adoption pains, DevOps is becoming a mainstream standard and is changing the way that IT organisations are delivering software, like the agile movement has done for over a decade.

But how does serverless computing fit into the DevOps culture and how will it influence common DevOps practices?

Why serverless computing?

In order to understand how serverless computing may impact organisations using it, let's first look at key characteristics of this approach to building and running software systems.

Functions-as-a-Service (FaaS) provides a managed runtime for executing any arbitrary code that has been uploaded to this service. This may look identical to simply deploying a runnable artefact to a compute instance (a server) and having an operating

system execute it — but it's not. FaaS takes care of making the function available at the scale required to satisfy the current demand but only charges for the execution count and time. At the same time, it abstracts away the setup of the actual runtime (like the [JVM](#) or [Node.js](#)) and the operating system itself. The runtime process, the OS, and the compute instance are still there (don't let the "serverless" term fool you) but the developer doesn't need to worry about any of these anymore.

And that's the beauty of it: the entire compute stack is completely managed by the cloud provider, including the OS process running the function code. This immensely simplifies the management of the compute infrastructure and, combined with a pay-as-you-go billing model, offers an incredibly flexible and cost-effective compute option compared to a more traditional [Infrastructure-as-a-Service](#) (IaaS) compute model.

Rapid development

Beyond the FaaS compute (such as [AWS Lambda](#), [Azure Functions](#), and [Google Cloud Func-](#)

[tions](#)), public cloud providers offer a range of other services that can be combined to create [serverless architectures](#). From scalable persistent stores and messaging infrastructure to API gateways and content-delivery networks, these days it is possible to build complete systems without ever having to directly deal with servers.

Each cloud provider's service is fully configurable using the provider's [SDK](#) so nothing gets in the way of quickly delivering business value (except a lack of familiarity with available services and their configuration options). Functions tend to be responsible for processing simple events or requests, so usually don't require a lot of coding, resulting in small units of concentrated business logic. For instance, a function could be responsible for publishing a notification of a change to the user's e-mail (based on a trigger from the database table) to the corresponding message topic, so that other subsystems can consume such notifications and update an external system.

Functions then implement the business logic and glue services that provide essential capabili-

ties, such as persistence, messaging, integration, content delivery, machine learning, etc. These services address a lot of complex engineering problems and allow the creation of sophisticated solutions without much difficulty, enabling rapid prototyping and development.

Operability from the start

With the serverless approach, it's virtually impossible (or at least a bit pointless) to write any code without having at least considered how that code will be executed and what other resources it requires to function. After all, in order to see how the code interacts with API gateways, data stores, or messaging infrastructure, code has to be deployed and all dependent resources have to be provisioned. To be fair, function execution can be emulated without deploying it onto the provider platform but this offers only a limited degree of validation. Additionally, it doesn't exercise the entire infrastructure stack that is required by the function or functions.

That serverless computing requires the provisioning of cloud resources can be considered a hindrance or a blessing. Those accustomed to using their own computers to run applications or systems locally in development mode will most likely agonise over the loss of productivity due to slower feedback cycles. Indeed, infrastructure provisioning and code deployments do take more time but not as much time as they would have with an IaaS compute (if you consider launching compute instances on demand).

The main benefit of being forced to look at the infrastructure stack right from the beginning is that

infrastructure setup and provisioning mechanisms are brought forward and tackled at the same time as the code is being produced. This is rather different than a more traditional (and still quite common) approach in which developers write code and perhaps even produce deployable artefacts (with help from the CI tool), then hand these over to the operations team for deployment, assuming that the compute and network infrastructure will be taken care of.

While the DevOps movement promotes collaboration between development and operations teams, with serverless it's simply not possible to separate the two regimes.

Even deploying a simple function requires making a few crucial decisions that have implications from operational as well as financial points of view. The two most basic configuration options are the amount of available memory and the timeout (the time budget for the function invocation). Both of these settings influence the cost of function invocation as providers charge for invocations based on memory consumption and execution time. Additionally, allocated memory is usually linked with the specification of the compute instance that the function is running on — more memory comes with more processing power.

With so much riding on the optimal configuration of the function, it's important to be able to quickly adjust the settings based on available budget as well as the desired and observed performance characteristics. These characteristics can be determined from the metrics collected and exposed by the cloud provider ([AWS CloudWatch](#) is an example of a monitoring ser- ►

While the DevOps movement promotes collaboration between development and operations teams, with serverless it's simply not possible to separate the two regimes.

vice). Having a rich set of metrics for FaaS and other services commonly used for building serverless architectures is essential to properly operate these architectures. Since metrics are available immediately after resources are provisioned, many operational aspects of the architecture, like performance optimisations, capacity planning, monitoring, and logging, can and should be considered even during the development stage.

Security is another aspect of software delivery that is commonly addressed as an afterthought or is delegated to a dedicated security team that has to assess and sign off all software components before deployment to production. With serverless, security has to be considered as soon as the function is deployed during the regular development activities. At a minimum, each function has to have a security policy associated with it. As much as it's possible to give the function access to every other resource in the same provider account, it makes sense to spend a bit more time to determine and configure the right security policy for the function based on its job. Following the [principle of least privilege](#) the function ideally should receive only the minimal set of permissions it needs. For example, a function that needs to query a database table should only have permissions to query this one table and nothing else.

It's becoming evident, I hope, that the serverless model is firmly making operability (including security) a part of the normal development cycle rather than deferring these elements until the time the operations team gets involved, when it's usually too late to properly and easily address any problems.

When it comes to serverless, a DevOps mentality is not something that can be adopted incrementally (usually with much pain) but rather it's written into the DNA of making things work.

Pay for what you use

Serverless computing brings another revolutionary change compared to the IaaS computing model and that's the pricing based on paying for individual function invocations rather than having to pay for keeping servers up.

Organisations using the public cloud are used to treating cloud infrastructure costs as [operating expenditure](#) (OPEX) rather than [capital expenditure](#) (CAPEX), but with IaaS computing they often still end up making substantial up-front investments in order to reduce cost (one example is reserving compute instances or purchasing reserved capacity for other cloud services). With serverless computing, this is usually no longer possible or cost effective as it's much cheaper to just pay for function invocations than to keep servers running all the time.

Considering the pay-per-use pricing model for the majority of services used to build serverless architectures, it becomes feasible to operate many more environments to support the development, testing, and operational activities involved in software delivery. After all, these activities may end up costing very little or nothing if not used. The economic implications of serverless remove many of the obstacles that make the road to DevOps so challenging to many companies.

The ability to have as many environments as may be needed to satisfy the needs of various

teams or business stakeholders offers tremendous possibilities. For instance, each developer could have a personal development environment in the cloud or perhaps each feature in progress could be deployed into a dedicated environment so it can be demoed independently of anything else being worked on. These separate environments can be even hosted on separate provider accounts to deliver the ultimate level of isolation.

Continuous deployment is the new normal

[Continuous delivery](#) is one of the key capabilities that enables DevOps but it still eludes many companies, particularly in the enterprise space. While continuous delivery offers many benefits and enables higher business agility, it fails to release an organisation's full potential.

Serverless computing however can enable the holy grail of business agility: continuous deployment. With continuous deployment, any change merged into the code mainline gets automatically promoted to all environments, including production. Obviously, for this approach to work without negatively affecting users, the system being continuously deployed needs to be subject to stringent quality checks from different angles.

Given that many operational aspects, such as infrastructure provisioning or security, can and should be tackled together with the development of the function code, the resulting infrastructure stacks can be provisioned from scratch or updated in place based on the code and configuration contained in the source-code repository. These stacks can be managed either by the provider's native tools like [CloudFor-](#)

mation on AWS or more generic tools like [Hashicorp Terraform](#).

With fully automated infrastructure stack provisioning and code deployments, it's possible to apply or un-apply (roll back) changes to any environment, including production, as part of the deployment pipeline. Any testing (both functional and non-functional) necessary to ensure the quality of the system should also be automatically executed against relevant environments after the deployment and the pipeline aborted if there are any test failures.

Everybody is a cloud engineer

Serverless is blurring the lines between various technical roles commonly involved in software delivery. Where traditionally architects, developers, testers, database administrators, operations, and security engineers would collaborate to deliver the system and operate it in production, many of these roles in the serverless world are merged into a single one, a cloud engineer.

As many traditional activities have been removed or greatly simplified, it's not longer necessary to involve many specialists. Instead, engineers with broad skillsets and good familiarity with the cloud provider's platform can achieve as much, if not more, and do it much more quickly too. Many development and operational activities can be conflated into the same cycle and costly handoffs or borrowing of external resources can be eliminated altogether.

This doesn't mean that we no longer have individuals who specialise in specific areas; people are naturally attracted to some aspects of software delivery

more than others but ideally everybody in the team should be able to participate in all activities related to delivering a feature, including operating the system in production. This is the best way to make sure all engineers are motivated to produce quality software that is operable from the start.

In effect, rather than talking about bringing development and ops closer together, teams using serverless are the embodiment of pure DevOps culture, where software is forged to be ready for production right from the start.

Survival of the fittest

Serverless can help achieve the ultimate business agility but it depends completely on the organisation's ability to realise the full potential that it offers. While many organisations still struggle to establish some form of DevOps culture and practices, serverless offers a whole new way to create a culture of rapid business-value delivery and operational stability while minimising costs.

Not many organisations can rise to the challenge of embracing the brave new world of serverless as the field is still immature and so requires a lot of extra work to address the gaps or challenges of its infant state. A lot of well-established organisations that might be willing to play with serverless may find themselves trying to use their existing processes and organisational structures and lose any agility that they could have gained or, worse, massively struggle to build and operate serverless architectures.

Those companies able to fully exploit serverless to achieve competitive advantage in the marketplace may be forced to adjust

not only the way they deliver software but also the way they create and sell products.

Conclusion

Serverless doesn't merely supplement DevOps. It goes beyond the current thinking on how IT organisations can achieve greater business agility. It's geared towards rapid delivery of business value and continuous improvement and learning, and as such has clear potential to drive substantial cultural change even in organisations that have already adopted DevOps culture and practices.

Using a serverless approach will not only allow organisations to more quickly and inexpensively deliver new products and features, it will change their cultures in the process. ■



Examining the Internals of a Serverless Platform: Moving Towards a Zero-Friction PaaS



Diptanu Gon Choudhury is an infrastructure engineer at Facebook. He works on infrastructure that powers large-scale distributed systems such as service discovery, RPC, and distributed tracing. He is one of the lead engineers of the Nomad distributed-scheduler project and prior to that he worked in the platform-engineering group at Netflix where he built an Apache Mesos framework for running tens of thousands of Linux containers on AWS.



Sangram Ganguly is a senior research scientist at the Bay Area Environmental Research Institute and at the Biospheric Science Branch at NASA's Ames Research Center. He earned a Ph.D. in remote sensing and his research interests span remote sensing, cloud computing, machine learning, high-performance computing, and advanced signal processing. He has received numerous awards and recognitions for his impactful contributions in Earth sciences and especially in developing a cloud-centric architecture, OpenNEX, for creating workflows and datasets that can be deployed on the cloud.



Andrew Michaelis is a software engineer and has contributed to the NEX, , and TOPS projects at NASA's Ames Research Center. He has been involved in developing large-scale science processing and data-analysis pipelines, with emphasis on remote sensing. He has also co-authored several relevant peer-reviewed publications and received several outstanding awards.



Ramakrishna Nemani is a senior Earth scientist with the NASA Advanced Supercomputing Division at Ames Research Center. He leads the development of NEX, a collaborative computing platform for Earth-system modeling and analysis.

KEY TAKEAWAYS

Platform-as-a-Service (PaaS) architectures simplify how developers deploy applications and serverless architecture, or Functions-as-a-Service (FaaS), is the next step in that direction.

Benefits of the emerging serverless platforms include faster time to market and easier management and operability.

A serverless platform needs the application developers to think and write business logic in the form of functions, which are invoked when an event is dispatched to the system.

Although serverless platforms have improved since their inception, developers still need to focus on latency planning, caching, configuration management, load balancing, traffic shaping, and operational visibility.

Developers can use tools such as distributed cluster schedulers, containers, service discovery, and software load balancers to build a lambda-like platform.

The serverless space is ripe for innovation and provides interesting challenges in areas such as data stores, traffic shaping, messaging systems, cluster schedulers, and application runtimes.

A common trend in application delivery over the last two decades has been a reduction in the time required to move from development to production.

On the operational side of the spectrum, the demand for higher availability, predictable latency, and increased throughput of services has significantly increased. Platform-as-a-Service (PaaS) architectures simplify the deployment of applications and promise a zero-touch operational experience, making application deployment faster and easier to scale. Serverless architecture, also called Functions-as-a-Service (FaaS), is the next step in that direction, where the underlying platform completely alleviates most of the operational concerns of a PaaS, thereby reducing friction between development and

operation. This article explores how serverless technology can be utilised to make this move from zero touch to zero friction possible.

Traditional PaaS versus serverless

Traditional PaaS architectures involved building services that were static operationally with respect to their runtimes, load balancing, logging, and configuration strategies. The most popular architecture was the [12-factor application](#). PaaS has been a great leap forward from deploying snowflake applications by

providing consistency around configuration, operational visibility, operability, scaling heuristics, etc.

In serverless architecture or FaaS, the underlying platform completely alleviates most of the operational concerns of a PaaS. Services like [AWS Lambda](#), [Google Cloud Functions](#), or [Azure Functions](#) allow developers to write only the request-processing logic in a function and the platform automatically handles other aspects of the stack like middleware, bootstrapping, and operational concerns like scaling and sharding. ►

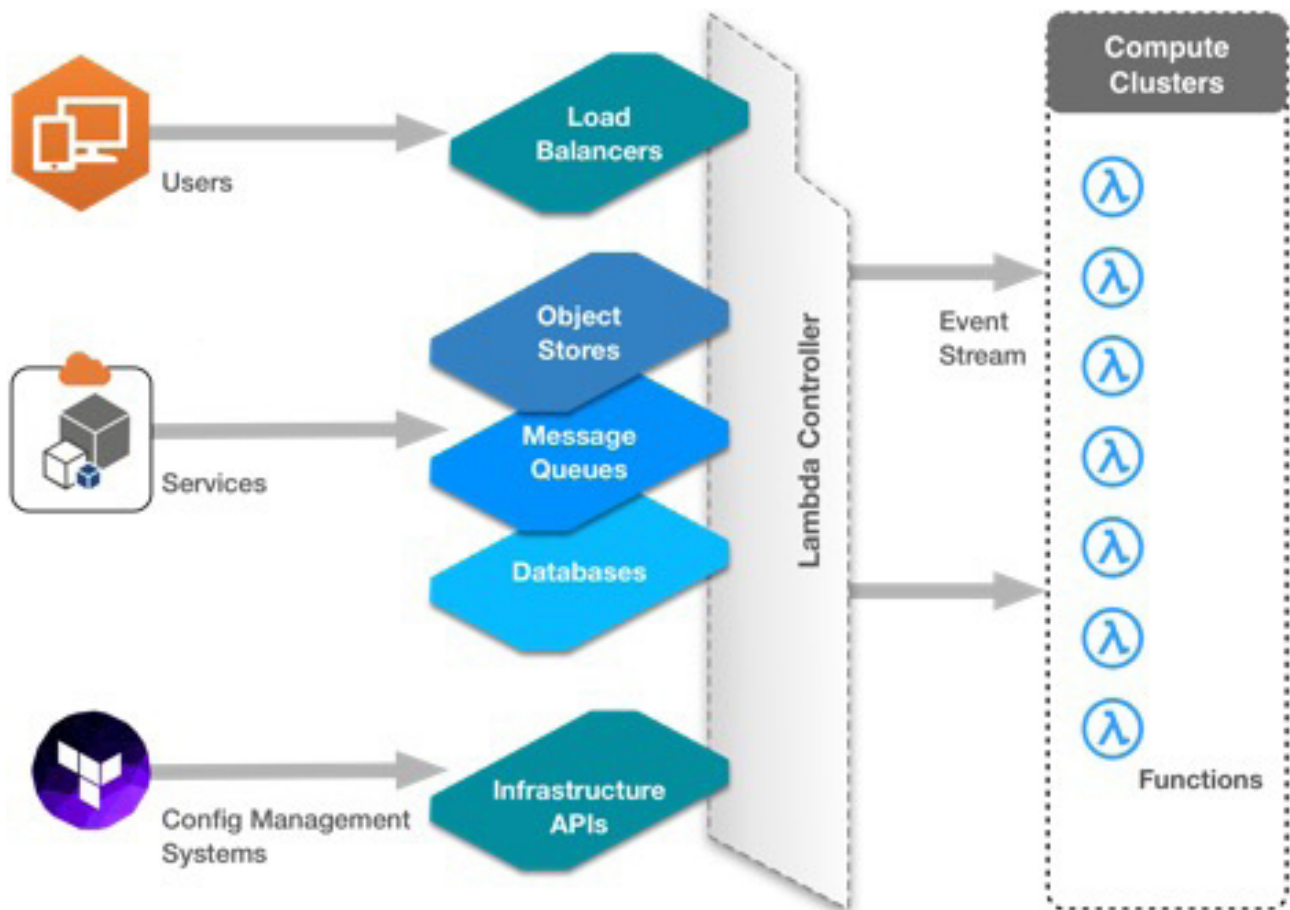


Fig. 1: On receiving a HTTP request from a user, the AWS API Gateway generates an event that invokes an instance of a lambda function that handles HTTP-request events, and the API Gateway responds to the request with the response from the lambda function. Similarly, if a file is written onto a data store or object store like DynamoDB or S3, AWS Lambda generates an event that invokes a lambda function and the function could then asynchronously process the data object.

Two of the benefits of the new emerging serverless platforms are:

1. **Faster time to market** — Serverless platforms allows product engineers to innovate rapidly by abstracting away the problems of system engineering in the underlying platform. Also, regular concerns of Internet-facing applications like identity management, storage, etc. are exposed to FaaS or handled by the underlying middleware. Product engineers can concentrate on develop-

ing the actual business logic of the application.

2. **Easier management and operability** — Serverless platforms provide a clear separation between infrastructure services and applications running on top of the platform. System engineers and SREs can focus on managing and running the underlying platform and core services such as databases and load balancers while product engineers manage the functions running on top of the platform.

Event-driven FaaS

In a nutshell, a serverless platform needs the application developers to think and write business logic in the form of functions that are invoked when an event is dispatched to the system. Event streams are central to serverless architectures, especially in AWS's lambda implementation. Any interaction with the platform like a user request or a mutation of state such as updating an object in the data store generates an event, which is streamed into a user-defined function for processing the event and accomplishes any domain-specific concerns. (see Figure 1 above)

The principle aspects of a serverless platform are:

1. **No need for a server to handle requests**

— Application developers don't need to worry about writing a server that accepts and responds to user requests. Although threaded servers like Apache HTTP Server (HTTPD) and Apache Tomcat and event-driven servers like Netty have been around for a long time, writing high-throughput application servers has always presented challenges. FaaS, by comparison, does not need an application server to respond to requests. Events in a serverless platform are handled by ephemeral functions that are processes spawned by the underlying middleware.

2. **RPC-less**

— Microservices/SOA-based services rely on RPC to fan out to hundreds of services across the network in response to user requests. On one side of the spectrum are load balancers like HAProxy that operate on data streams as opaque byte streams with limited configurability; on the other, RPC libraries like Finagle and gRPC provide extreme configurability around load balancing, request processing, etc. but are quite complex and demand a lot of experience to use. With FaaS, the platform transparently handles RPC as one lambda function invokes another lambda function. A lambda function usually uses the platform provider's SDK to invoke another lambda function or emits an event that is asynchronously fed into another lambda function.

3. **Free of topology and supervision**

— Network topology is an important factor to consider while deploying any modern Internet-scale application. Clusters have to be spread across failure domains such as racks and switches in data centers (DCs) in order to ensure high availability (HA). Also, the platform needs to react when there is an outage of a unit of compute — which could be a node, a rack, or an entire DC. Cluster schedulers such as Nomad, Kubernetes, and Mesos provide features like cluster supervision and HA environments out of the box and allow for the deployment of highly scalable services. One example of a highly scalable service is the Map Reduce infrastructure at Google, which is built on top of the [Borg cluster scheduler \(PDF download\)](#). Serverless platforms alleviate an application developer's concerns for placement details and process supervision. The underlying platform invokes a function whenever an event has to be handled thus reducing operational complexity.

However, as the article "[There's Just No Getting Around It: You're Building a Distributed System](#)" from ACM Queue points out, there is simply no avoiding the inherent complexities of distributed systems while building Internet-scale applications.

Most mission-critical services provide the following service-level agreements (SLAs) to the end consumers:

- predictable throughput,
- high availability of core functionalities,
- tolerable latency, and
- reliable core functionalities.

Companies like Netflix, Google, and Facebook have invested significantly in these aspects as they've built modern platforms for their consumer-facing services and each has attained a reputation for high quality of service despite running on commodity hardware and networks.

Serverless operational concerns

Even though serverless platforms such as AWS Lambda have improved since inception, we still need to focus on some of the problems that we have already solved in traditional platforms.

Let's look at the most relevant ones.

Latency planning

Synchronous user requests usually require response within a predictable time period. If a user request invokes lambda functions in a chained manner, it is important that we factor in the average time and variance it takes for the middleware to do its work such as invoke functions and emitting events. This is hard since the underlying platform is opaque to the application developer.

A cloud platform can solve these kinds of problems with request-cancellation techniques: [Netflix OSS](#), for example, uses the [Ribbon](#) software load balancer. An analogous implementation in a serverless platform would entail a function timing out the invocation of another function once it exceeds a deadline.

Caching

There are various forms of caching, from external caches like [Memcached](#) to application in-memory caches and clustered ►

in-process caches such as [group-cache](#). Caching choices become limited based on the guarantees the serverless platform makes with respect to durability of the instance of a function that responds to an event.

For example, AWS Lambda offers no guarantee that the same instance of a function is going to respond to events so in-process caches might not work well, necessitating the use of external caches like [ElastiCache](#) instead of in-process caches such as GroupCache.

Configuration management

Configuration can be largely divided into two classes: dynamic configuration and static configuration. Static configuration is usually pushed via configuration management systems that are baked into applications or machine images, which take effect when an application process starts. Dynamic configuration is typically a set of runtime parameters that operators can override while an application is already running to change how the application behaves. Most application configuration libraries or management techniques assume a running process so they do not work well with serverless platforms.

Application or environment-related configuration has to be stored on external data stores or caches that the lambda function has to retrieve when it is invoked.

Load balancing

RPC is simply opaque on lambda architectures since invoking another lambda function synchronously usually involves using the SDK. There has been a lot of work in RPC libraries like [Finagle](#) and [Ribbon](#) to improve reliability, la-

tency, and throughput of RPC requests by using techniques for managing outgoing RPC requests such as request cancellation, parallel requests, bulkheads, and batching.

The [AWS Step Functions](#) service is a move in the right direction, but the FaaS space still needs to mature in this regard to ensure that we have the same degree of reliability in handling synchronous requests, which need to fan out to multiple downstream services.

Traffic shaping

It is common to serve Internet-scale applications from more than one geographic region. For example, three different geographic regions within AWS serve the Netflix service. One of the prerequisites for a multi-region architecture is the ability to divide traffic across the different regions so that users can be routed to a nearby DC and the workload is balanced appropriately across all regions. Also it is important that operators retain the ability to shut down a region and steer all the traffic to another region if there is an outage or degradation in a region, which historically hasn't been uncommon. There are many ways to achieve this, such as dividing the continents using geographic DNS routing or software load balancers such as [Zuul](#).

In a serverless architecture, we lose some flexibility in this space because an opaque load balancer such as the [AWS API Gateway](#) handles user requests and the underlying platform usually chooses a lambda function deployed in the same region from which the request arrives. So when there are outages in caching or data store services, it is hard to steer part of the traffic

to other regions. The coupling between a lambda region and its caches and data stores is inherent in the application but not apparent to FaaS providers. With DNS-based traffic shaping, it is still possible for serverless environments to move traffic and indeed this is the most common way of steering traffic to date (versus the use of proxying load balancers).

Operational visibility

Debugging distributed systems is challenging and usually requires access to a significant number of relevant metrics to zero in on the root cause. Distributed tracing can help us understand how a request fans out across multiple services and helps in debugging systems based on the microservices architecture. Due to the newness of serverless architecture, we have yet to see mature debugging tools for it and platforms need to expose more operational metrics than they do today.

Anatomy of a serverless platform

Currently, the leading serverless platforms — AWS Lambda, GCP Cloud Functions, and Azure Cloud Functions — are all hosted services on the public cloud, but similar platforms could be built for a private cloud as well. We can leverage tools such as distributed cluster schedulers, containers, service discovery, and software load balancers to build a lambda-like platform.

There are many ways to build a serverless platform but the backbone usually comprises the following services:

- code or function delivery mechanism,
- request routing,

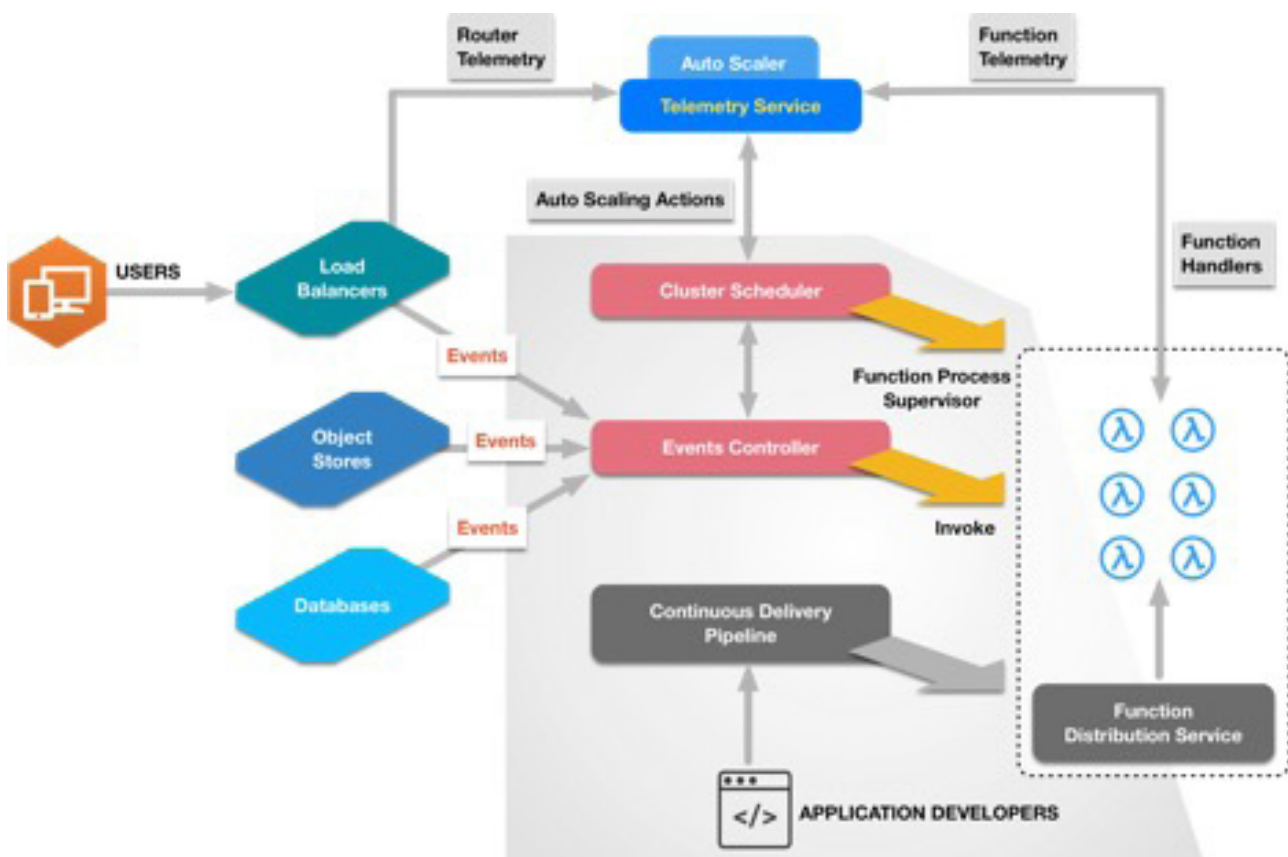


Fig. 2: A generic serverless platform.

- cluster scheduler and quality-of-service (QOS) guarantees,
- auto-scaling engine for scale out/in of services, and
- operational insights.

Function delivery

Functions that application developers write need to be deployed on compute clusters before they can be invoked. The functions also need to be wrapped in a server or an event processor that can receive the events and invoke the function. The build process would typically handle the part of the process that includes making the user functions part of an event processor or a server process. Once the code is built, there are many ways to package the code: OS packages, tarballs, [Docker containers](#), or root-fs archives such as those used by [LXC](#)

and [LXD](#). The methodology we choose here would depend on how we actually run the processes on the compute nodes.

The language and framework choices used to wrap lambda functions by the event processors could influence the choices for the runtimes that are available to application developers. Needless to say, the user functions have to be written with a keen understanding of the underlying platform. For example, if the underlying platform is only capable of invoking functions written in Node.js, the users can only program in JavaScript, targeting a specific VM and using the node SDK. The underlying runtime could also have a whitelisted set of APIs for enhanced security and compatibility to the platform.

Request routing

Request routing is platform specific. The request router at the edge needs to be aware of the registered functions and needs to be able to dispatch events to the underlying middleware that ultimately invokes the functions. For synchronous requests, the router needs to be able to keep a connection open once a request is made and wait for the middleware to return with the response from the function handling the event.

The [Fabio load balancer](#) at eBay can dynamically dispatch requests to services. Fabio was initially built for load-balancing services registered with a dynamic service discovery tool like [Consul](#). Nginx with its Lua bindings offers a lot of flexibility as well; it's possible to extend Nginx to discover services from an ex- ▶

In a nutshell, a serverless platform needs the application developers to think and write business logic in the form of functions that are invoked when an event is dispatched to the system.

ternal-service discovery tool or transform the incoming request and hand it over to the underlying middleware.

Request routers can also add more redundancy to the platform by aiding in traffic shaping. Traffic shaping is an important aspect of managing and operating services at global scale. When a region or data center goes offline, the norm is to transparently shift the user traffic from the affected region to another without degrading the user experience. Serverless platforms should be built with cross-region/data-center operations in mind.

Cluster scheduler and events controller

The cluster scheduler and the events controller, which we will also refer to as the middleware, are collectively the backbone of a serverless platform. The events controller is the service that receives the events from various sources and invokes appropriate user-defined functions. It is important for the events controller to provide an at-least-once delivery of events for the consistency of the platform.

The cluster scheduler supervises the function handlers that process the events or in some cases pre-scales the function handlers. The events controller and cluster scheduler should work in tandem to ensure that there is sufficient capacity to handle the flow of events at all times. In addition, the cluster scheduler has to ensure QOS and HA of the function handlers so that the functions can process the events within acceptable limits of latency.

Serverless platforms are usually multitenant, which means the cluster schedulers have to provide resource isolation (memory,

CPU, disk, and network) to the function handlers. Technologies like [cgroups](#), [namespaces](#), and [ZFS](#) play an important role here.

Cluster schedulers also provide HA to the function handlers running on the platform by migrating functions in a failed node, rack, or data center to somewhere else on the cluster in the same region.

The telemetry generated by schedulers, such as latency of starting function handlers, resource contention among functions in a multitenant system, and dispatch latencies, are important and should be exposed to feedback control systems such as auto-scalers and site reliability engineers.

Auto-scaling

A serverless platform should attempt to alleviate the user's need to plan capacities of compute, storage, and network resources. The platform should scale to the extent that it can handle the events being dispatched by the routers with reasonable response latencies.

There are various ways to achieve this. Usually, the middleware or the cluster scheduler pre-scales the function handlers based on the rate of generated events and thus is able to handle all the events in a reasonable time. One such predictive auto-scaler is Netflix's [Scryer](#) — similar auto-scalers that can consume the telemetry from a scheduler and scale the function handlers should be built for serverless platforms.

Operational insights

There are two principle groups of people who interact with a serverless platform: the applica-

tion developers who write the functions and the SREs who operate the platform.

It is important that the platform exposes the right set of telemetry to these two groups.

Application developers are usually concerned with the following metrics to their functions:

- Throughput of events — The number of events generated by the routers and other sinks of data pipelines such as databases and object stores.
- Time to respond to an event — This metric is usually best expressed as a histogram, which offers the right amount of detail for understanding the long-tail P99 latency.
- Latency at the edge — The latency experienced by the end users of the applications running on the serverless platform.
- Latency between functions — Application developers should understand the latency of invoking functions in a chain when a chain of functions responds to synchronous requests. That latency is usually incurred by the middleware or the request router in the underlying platform but it helps application developers think about the maximum number of functions an event could invoke.
- Distributed tracing — The underlying platform should be able to trace the events flowing into the system as they invoke various function handlers.

SREs benefit from looking at the system from a higher level. Here

are some of the most important metrics for operators:

- Throughput of events being dispatched into the platform — This metric should be tagged with location constraints such as region, data center, etc. They usually indicate the load on a specific cluster and its health. Events should also be tagged with the functions they are invoking since that often points to a particular tenant generating more than the desired load on the platform.
- Functions — The number of active functions on the cluster, and the rate at which they are being invoked. Startup latencies are also important here, and again they should be tagged with nodes, racks, data centers, etc.
- Distribution — Metrics related to function delivery should also be tracked, since increased latencies of function invocation often occur due to an increased interval for replication of function-handler code across the cluster.

Concluding remarks

Serverless platforms are far more approachable for application engineers and provide abstractions that promise to make application delivery faster than traditional PaaS platforms. The space is also ripe for innovation and we are going to see a lot of new entrants. It provides new challenges in areas such as data stores, traffic shaping, messaging systems, cluster schedulers, and application runtimes.

We are in the process of building a serverless batch platform for NASA Earth-science missions through the [OpenNEX](#) platform

on AWS. [OpenNEX](#) is an open version of the [NASA Earth Exchange \(NEX\)](#), a collaborative supercomputing platform for Earth scientists that allows them to perform big-data analytics. Data from NASA missions are processed with community-vetted algorithms on platforms such as NEX to answer various scientific questions.

New data products and scientific workflows are made available to the public through platforms such as OpenNEX and they enable the community to develop better algorithms and create new products. Enabling such experimental products has been a challenge because the processing pipelines are stove-piped. The proposed technology provides a platform so scientists can simply use the setup, including existing functions, and complement it with their own input functions packaged via containers to do new science or generate new products. ■

Deploying at Scale: Chef, Puppet, Ansible, Fabric and SaltStack



By Matt McLarty

The manageability, reliability and powerful technology of remote servers — cloud computing — allows IT managers to deploy hundreds, even thousands of machines. At the same time, the cloud creates a new challenge for sys admins and ops teams: how to maintain and configure all these machines. How do you apply patches, maintain updates and fix security gaps?

The answer is to use powerful tools like Chef, Puppet, Ansible, Fabric or SaltStack for managing Infrastructure As Code (IaC) automation. IaC means deploying and managing infrastructure for computing, including virtual servers and bare-metal servers. Definition files are used instead of physical hardware management. Here is a bit of the history, background, advantages and disadvantages for each of these infrastructure configuration management tools currently on the market.

Puppet

Puppet was founded in 2005 by Luke Kanies, making it one of the earliest infrastructure configuration management tools. It is free software written in Ruby and made available under the Apache Software License 2.0, although it was released using the GNU General Public License up to version 2.7.0. It operates declaratively on Microsoft Windows and UNIX-based systems like AIX, Solaris and Mac OS X. Puppet uses a declarative language to define system configuration. To begin, you set up system resources and relevant state that are stored in files called Puppet Manifests. A resource abstraction layer then lets you use higher-level terms such as packages and services to define configuration.

Because Puppet is model driven, you don't need an extensive programming background to use it. In a model-driven approach, you can set up how you want the in-

frastructure and applications to operate. With the model in place, you can then test and evaluate changes you want to deploy across the system. Constant reporting and feedback allows you to improve processes, show compliance and tweak the results as you go. Puppet is perhaps the most popular infrastructure configuration and management tool among those described here, used by a variety of organizations including:

- Mozilla
- PayPal
- Spotify
- Oracle
- Rackspace
- Wikimedia Foundation
- Chef

Chef is a configuration management tool Adam Jacob developed to use in his consulting company. Seeing a broader use for managing Amazon Web Services operations, he joined with Nathan Haneysmith, Barry Stein-

glass and Joshua Timberman to found a firm called Chef to manage the tool.

Chef is based on “recipes” that describe how the software will configure and manage utilities and server apps like MySQL or Hadoop. These recipes can be combined to form a “cookbook.” Each recipe defines resources used in a state such as what services should be operating, what packages need to be installed and what files need to be created. The resources can be modified to make sure programs are installed in a specific order based on dependencies. Industry commentators often suggest that DevOps and developers usually choose Chef while SysAdmin’s prefer Puppet.

There are two versions of Chef: an open-source basic version and a premium enterprise edition. The enterprise offering has both on-premise and hosted versions. Open-source Chef is available at no charge but lacks many of the add-ons in the enterprise edition as well as ongoing support.

Chef began as a Linux product but later added support for Microsoft Windows. It runs on major platforms including

- Solaris
- Ubuntu
- Microsoft Windows
- FreeBSD

It is used by companies and websites such as:

- Facebook
- Airbnb
- Expedia
- Citi
- Disney

Chef and Puppet are two of the largest infrastructure management tools available to you. They both continue to respond to the needs of enterprise companies

by providing new features, and they are also busy creating partnerships with major vendors like Microsoft to better integrate with their platforms. Puppet has also aligned with software defined networking (SDN) vendors to stay in the forefront of that technology. Choosing between the two is a matter of determining the core advantages of each and figuring out which align with your requirements.

Ansible

Ansible is an open-source software framework for managing and configuring infrastructure. It offers configuration management, software deployment for multiple nodes and ad hoc task execution. You can manage it using PowerShell or through a secure shell (SSH). This software framework was developed by Michael DeHaan, who was also one of the original developers over the Func framework used for administering systems remotely. Ansible is included in distributions of Fedora, and is also available if you use CentOS, Red Hat Enterprise Linux, Scientific Linux and other operating systems. A company of the same name was created to support the software product and help it grow in business markets. Red Hat acquired the company in 2015.

The name Ansible is derived from a communications system in “Ender’s Game,” a 1985 novel by Orson Scott Card. The fictional system was first invented for the 1966 novel “Rocannon’s World” by Ursula K. Le Guin.

Ansible controls two kinds of servers: nodes and controlling machines. The system is based on a single controlling machine, which configures and manages nodes using SSH. Modules are deployed over SSH to orchestrate

notes, which then communicate to the controlling machine using a JSON protocol. Ansible is light on resources because when it is not managing nodes, it does not run any programs or daemons waiting for utilization.

Unlike Puppet and Chef, Ansible has an agentless architecture where nodes need a daemon to talk to the controlling machine. Under this system, nodes do not need to install and operate daemons in the background to communicate. This set-up significantly reduces network overhead because it stops nodes from constantly polling the controlling machine.

Ansible was designed with a minimalist approach, with a focus on making sure managing the system does not create additional dependencies on the system itself. It is secure because it requires OpenSSH. In addition, Ansible playbooks are written in an easy-to-learn, descriptive language. It is used in a variety of private and public clouds including:

- Google Cloud Platform
- OpenStack
- SoftLayer
- Amazon Web Services
- XenServer

Ansible works well with Aerospike, Riak and Hadoop, monitoring resource consumption by every node while using few CPU and memory resources. Organizations and companies deploying Ansible include:

- NASA
- Weight Watchers
- Juniper
- Apple

Its agentless model makes it a popular choice for government divisions such as NASA because it is very secure, a quality highly

valued in federal and state governments.

Fabric

Fabric is an open-source command line tool and Python library used to smooth out SSH utilization for system administration and application deployment. It consists of a suite of operations for launching shell commands, either locally or remotely, via sudo or normally; downloading and uploading files; and asking for input from users, stopping execution and other auxiliary functions. While products like Puppet and Chef focus on organizing and handling system libraries and servers, Fabric is more concerned with deployment and other application-level functions.

Developers like Fabric because it is simple, easy to maintain and you can add any type of job quickly. You can execute Python functions using the command line, and launching shell commands on SSH is simplified due to the extensive library of sub-routines. Companies using Fabric include:

- Snap
- Coursera
- Instagram
- Sosh
- FlightAware
- The Orchard

Fabric development is managed by Jeff Forcier. He is assisted by open-source developers who add suggestions and patches through the Fabric mailing list, on IRC chats or via GitHub.

SaltStack

SaltStack is an open-source platform based on Python, and it is used for managing and configuring cloud infrastructure. It was developed by Thomas S. Hatch using ZeroMQ to create a better

tool for collecting and executing data at high speeds. Initially released in 2011, Reliable Queuing Transport (RAET) was added in 2014. The project has subsequently been developed through a partnership that includes several large enterprises. SaltStack was built from the ground up to be highly modular and flexible, and able to adapt to diverse applications. It creates Python modules that each manage a different part of the Salt system. You can detach and modify the modules to fit the needs of your project. Each module is designed to handle a specific action. The six types of modules include:

- Execution modules which offer functions for directly executing the remote execution engine as well as help manage portability and core API functions.
- Grains detect system static information and keep it in RAM for fast access.
- State modules represent the back end, executing code to configure or change a target system.
- Renderer modules pass information to the state system.
- Returners modules manage the return locations associated with remote execution calls.
- Runners are convenience apps.

SaltStack created a buzz early on by capturing the 2014 InfoWorld Technology of the Year Award as well as the 2013 TechCrunch Award for Most Exciting Project. Organizations and companies using SaltStack include Adobe, Jobspring Partners, Dealertrack

Holdings, JumpCloud and International Game Technology.

This article covered five of the top infrastructure configuration and management tools available. It's a highly dynamic area of enterprise computing, with new tools constantly evolving to solve various challenges. Each of these solutions gives you lots of ways to configure your infrastructure, allowing you to manage digital transformation at scale easily and efficiently.

Learn more

Find out more about our [Infrastructure Monitoring](#) tool. ■

The Future of Serverless Compute



Mike Roberts is an engineering leader and cofounder of Symphonia, a serverless and cloud-technology consultancy. He is a long-time proponent of agile and DevOps values and is excited by the role that cloud technologies have played in enabling such values for many high-functioning software teams. He sees serverless as the next technological evolution of cloud systems and as such is optimistic about their ability to help teams be awesome. Roberts can be reached at mike@symphonia.io and tweets at [@mikebroberts](https://twitter.com/mikebroberts).

It's 2017 and the serverless-compute revolution is a little over two years old. (Do you hear the people sing?) This revolution is not coming with a bang, like Docker did, but with a steady swell.

Amazon is putting out new AWS Lambda features and products on a [regular cadence](#), the [other big vendors](#) are releasing production-ready versions of their offerings one by one, and new open source projects are frequently joining the party.

As we approach the end of the early-adopter period, it's an interesting exercise to put on our prediction goggles and contemplate where this movement is

going, how it's getting there, and most importantly what changes we need from our organizations to support it. So, join me as we look at one possible future of serverless compute.

(Note to readers from the actual future! You'll probably get a good kick out reading this. How far off was I? And how is 2020? Please send me a postcard!)

A vision of serverless capabilities

Compute

The last decade has seen the emergence and meteoric rise of cloud computing. Nine years ago, virtual public cloud servers were toys for startups but in a relatively short time, they have become the de facto platform for large swaths of our industry, which considers any new deployment architecture. ►

KEY TAKEAWAYS

Serverless compute, or functions as a service (FaaS), will be transformational in our industry and organizations that embrace it will have a competitive advantage because of the cost, labor, and time-to-market advantages it brings.

Many serverless applications will combine FaaS functions with significant use of other vendor-provided services that provide pre-managed functionality and state management.

Tooling will significantly improve, especially in the area of deployment and configuration.

Patterns of good serverless architecture will emerge — it's too soon to know what they are now.

Organizations will need to embrace the ideas of “true” DevOps and autonomous, self-sufficient, product teams to reap the full time-to-market benefits that serverless can offer.

Serverless compute, or [Functions-as-a-Service \(FaaS\)](#), is a more recent part of this massive change in how we consider IT. It is the natural evolution of our continuing desire to remove all baggage and infrastructural inventory from how we deliver applications to our customers.

A huge number of the applications we develop consist of many small pieces of behavior. Each of those receive a small input set and informational context, will do some work for a few tens or hundreds of milliseconds, and finally may respond with a result and/or update the world around them. This is the sweet spot of serverless compute.

We predict that many teams will embrace FaaS due to how easy, fast, and cheap it makes deploying, managing, and scaling the infrastructure necessary for this type of logic. We can structure our use of FaaS into various forms, including:

- complete back-end data pipelines consisting of a multitude of sequenced message processors;
- synchronous services fronted by an HTTP API;
- independent pieces of “glue” code to provide custom operations logic for deployment, monitoring, etc.; and
- hybrid systems consisting of traditional “always on” servers directly invoking platform-scaled functions for more computationally intensive tasks.

Businesses that embrace FaaS will have a competitive advantage over those that don't because of the cost and time-to-market benefits it brings.

Managing application state

One of the prerequisites of such a large adoption of FaaS is a solu-

tion, or set of solutions, for fast and simple approaches to state management. Serverless compute is a stateless paradigm. We cannot assume that any useful state exists in the immediate execution environment of our running functions between separate invocations. Some applications are fine with this restriction as it stands. For example, message-driven components that are purely transformational need no access to external state, and web-service components that have liberal response-time requirements may be okay to connect to a remote database on each invocation. But for other applications this is insufficient.

One idea to solve this is a hybrid architecture that manages state in a different type of component than that executing our FaaS code. The most popular such hybrid is to front FaaS functions with other services provided by the cloud infrastructure. We already see this with context-spe-

cific components like [Amazon API Gateway](#), which provides HTTP routing, authorization, and throttling logic that we might typically see programmed in a typical web service but defined instead through configuration. Amazon has also recently shown its hand in a more generic approach to state management with its [AWS Step Functions](#) service, allowing teams to define applications in terms of configured state machines. The Step Functions service itself might not become a winner, but these kinds of codeless solutions are going to become very popular in general.

Where vendor services are insufficient, an alternative approach to a hybrid system is for teams to continue to develop long-lived components that track state. Those might be deployed within a Containers-as-a-Service (CaaS) or a Platform-as-a-Service (PaaS) environment, and will work in concert with FaaS functions.

These hybrid systems combine logic in long-running components and per-request FaaS functions. An alternative is to focus all logic in FaaS functions, but to give those FaaS functions extremely fast retrieval and persistence of state beyond their immediate environment. A possible implementation of this would be to make sure that a particular FaaS function, or a set of FaaS functions, has very-low-latency access to an external cache like Redis. Enabling a feature similar to Amazon's same-zone [placement groups](#) could provide this. While such a solution would still incur more latency than memory — or disk-local state — many applications will find this solution acceptable.

The benefits of the hybrid approach are that frequently ac-

cessed state can stay in-environment with the logic using it, and that it requires no complicated and possibly expensive network colocation of logic and external state. On the other hand, the benefits of a pure-FaaS approach are a more consistent programming model plus a broader use of the scaling and operational benefits that serverless brings. The current momentum suggests that the hybrid approach will win out, but we should keep our eyes open for placement-group-enabled lambdas and the like.

Serverless collaboration services

Beyond orchestration and state management, we see the commoditization and service-ification of other components that we traditionally would develop or at least manage ourselves even within a cloud environment. For instance, we may stop running our own MySQL database server on EC2 machines, and instead use the [Amazon RDS](#) service; we may replace our self-managed Apache Kafka message-bus installation with [Amazon Kinesis](#). Other infrastructural services include [file systems](#) and [data warehouses](#) while more application-oriented examples include [authentication](#) and [speech analysis](#).

This trend will continue, reducing still further the amount of work we need to do to create or maintain our products. We can imagine more pre-built messaging logic (think of a kind of “[Apache Camel](#) as a service” built into Amazon Kinesis or [Amazon SQS](#)) and further developments in generic machine-learning services.

A fun idea here is that FaaS functions, due to their lightweight application model, can themselves be tightly bound to a ser-

vice, leading to ecosystems of FaaS functions calling services that themselves call other FaaS functions, and so on. This leads to “interesting” problems with cascading errors for which we need better monitoring tools, as discussed later in this article.

Beyond the data center

The vast majority of serverless compute so far is on vendor platforms running in their data centers. It gives you an alternative to how you run your code but not where you run your code. An interesting new development from Amazon is to allow customers to run lambda functions in different locations, for instance in a CDN with [Lambda@Edge](#), and even non-server locations, e.g., IoT devices with [AWS Greengrass](#). The reason for this is that AWS Lambda is an extremely lightweight programming model that is inherently event-driven and so it's easy to use the same intellectual ideas and style of code in new locations. Lambda@Edge is a particularly interesting example since it provides an option for programmed customization in a location that never had it before.

Of course, a drawback to this is even deeper vendor lock-in! Those organizations that don't want to use a third-party vendor but do want many of the benefits of serverless compute will be able to do this with an on-premise solution, just like Cloud Foundry has done for PaaS. [Galactic Fog](#), [IronFunctions](#), and [Fission](#), from Kubernetes, are early efforts in this area.

The tools and techniques we'll need

As I [have written](#) previously, there are significant speed bumps, limitations, and tradeoffs when using a serverless approach. This is ►

APPDYNAMICS

Find & fix app performance issues faster.

Discover how APM keeps your customers happier.

Take a Tour

no free lunch. For the serverless user base to grow beyond early adopters, we need to fix or mitigate these concerns. Fortunately, there is good momentum in this area.

Deployment tooling

Deploying functions to Lambda using standard AWS tools can be complex and error-prone. Add in the use of Amazon API Gateway for lambda functions that respond to HTTP requests and you have even more work to do for setup and configuration. The [Serverless](#) and [Clau-dia.js](#) open-source projects have been pushing deployment improvements for over a year, and AWS joined the party with [SAM](#) late in 2016. All these projects simplify the creation, configuration, and deployment of serverless applications by adding considerable automation on top of the AWS standard tooling. But there is still plenty to do here. In the future, two key activities will be heavily automated:

1. initial creation of an application and/or environment (e.g., both initial production environment and temporary testing environments) and
2. continuous delivery/deployment of multicomponent applications.

The first of these is important in order to more widely enable the “conception-to-production lead time” advances that we’ve started seeing. Deploying a new serverless application needs to be as easy as creating a new GitHub repo: fill a small number of fields, press a button, and have some system create everything you need to allow one-click deployment.

However, easy initial deployment is not sufficient. We also need good tools to support continuous delivery and continuous deployment of the type of hybrid application I mentioned earlier. This means we should be able to deploy a suite of compute functions and CaaS/PaaS components, together with changes to any application-encapsulated services (e.g., configured HTTP routes in Amazon API Gateway or a Dynamo table only used by a single “application”), in one click with zero downtime and trivial rollback capability. Furthermore, none of this should be intellectually complex to understand or need days of work to set up and maintain.

This is a tough call, but the tools I mentioned previously (together with hybrid tools like [Terraform](#)) are leading the way to solving these problems, and I fully expect the problems to be largely solved over the coming months and years.

This topic isn’t just about deploying code and configuring services, however. Various other operational concerns are involved. Security is a big one. Right now, getting your AWS credentials, roles, and the like set up and maintained can be a hassle. AWS has a thorough security model but we need tools to make it more developer-friendly.

In short, we need developer UX as good as Auth0 has with its [Webtask](#) product, but for an ecosystem as vast (and as valuable) as AWS.

Monitoring, logging, and debugging

Once our application is deployed, we need good solutions for monitoring and logging, and such tools are under active develop-

ment right now at several organizations. Beyond assessing the behavior of just one component, though, we also need good tools for tracing requests through an entire distributed system of multiple serverless compute functions and supporting services. Amazon is starting to push in this area with [AWS X-Ray](#), but it’s very early days.

Debugging is also important. Programmers have rarely written correct code for every scenario on the first pass in traditional environments and there’s no reason to believe that’s going to change. We rely on monitoring to assess problems in FaaS functions at development time but that’s a Stone Age debugging tool.

When debugging traditional applications, we get a lot of support from IDEs in order to set breakpoints, step through code, etc. With modern Java-based IDEs [you can attach to a remote process that’s already running](#) and perform these debugging operations at a distance. Since we will likely be doing a lot of development using cloud-deployed FaaS functions, expect in the future that your IDE will have similar behavior to connect to a running serverless platform and interrogate the execution of individual functions. This will need collaboration from both tool and platform vendors but it’s necessary if serverless is going to gain widespread adoption. This does imply an amount of cloud-based development, but we’re likely going to need that anyway for testing.

Testing

Of all the serverless tooling topics I’ve considered so far, the one that I think is least advanced is testing. It’s worth pointing out that serverless does have some hefty testing advantages over

traditional solutions in that (a) with serverless compute, individual functions are ripe for unit testing and (b) with serverless services you have less code to write and therefore less to test, at the unit level at least.

But this doesn’t solve the cross-component functional/integration/acceptance/“[journey](#)” test problem. With serverless compute, our logic is spread out across a number of functions and services and so higher-level testing is even more important than with components in something closer to a monolithic approach. But how do we do this when we’re relying so much on execution on cloud infrastructure?

This is probably the most misty prediction for me. I suspect that what will happen is that cloud-based testing will become prevalent. This will occur partly because it will be much easier to deploy, monitor, and debug our serverless apps than it is right now for the reasons I just described.

In other words, to run higher-level tests, we’ll deploy a portion of our ecosystem to the cloud and execute tests against components deployed there, rather than running against a system deployed on our own development machines. This has certain benefits:

- The fidelity of executing cloud-deployed components is much higher than a local simulation.
- We’ll be able to run higher-load/more-data-rich tests than we might otherwise.
- Testing components against production data sources (e.g., a pub-sub message bus or a database) is much easier, al- ▶

though obviously we'll need to be careful of capacity/security concerns.

This solution also has drawbacks, though. First, cycle time to execute tests will likely increase due to both deployment concerns and network latency between the test, which will still run locally, and the remotely executing system under test. Second, we can't run tests when disconnected from the Internet (while on a plane, etc.) Finally, since production and test environments will be so similarly deployed, we'll also need to be very careful about not accidentally changing production when we meant to change test. If using AWS, we can implement such safety through tools like IAM roles or by using entirely different accounts for different types of environment.

Tests are not just about a binary fail/succeed — we also want to know how a test has failed. We should be able to debug the locally running tests and the remote components they are testing, including being able to single-step a lambda function running in AWS as it is responding to a test. And so all of the remote debugging, etc. tools I mentioned in the previous section will be needed for testing too, not only for interactive development.

Note that I'm not implying from this that our development tools need to run in the cloud or that the tests themselves have to run in the cloud, although both of these will occur to a greater or lesser extent. I'm merely expressing that the system under test will only ever run in the cloud, rather than on a non-cloud environment.

Using serverless as a test-driving environment, though, can reap

useful results. One example is [serverless-artillery](#), a load-testing tool made up of running many AWS Lambdas in parallel to perform instant, cheap, easy performance testing at scale.

It's worth pointing out that we may to some extent dodge a bullet here. Traditional higher-level testing is actually becoming less important due to advances in techniques where we (a) [test in production](#) or use [monitoring-driven development](#), (b) significantly reduce our mean-time-to-resolution (MTTR), and (c) embrace continuous deployment. For many serverless apps, extensive unit testing, extensive business-metric-level production monitoring and alerting, and a dedicated approach to reducing MTTR and embracing continuous deployment will be a sufficient strategy for validating code.

Architecture: Many questions to answer

What does a well-formed serverless application look like? How does it evolve?

We're seeing an increasing number of case studies of architectures where serverless is being used effectively but we haven't yet seen something like a pattern grouping for serverless apps. In the early 2000s, we saw books like Fowler's [Patterns of Enterprise Application Architecture](#), and Hohpe and Woolf's [Enterprise Integration Patterns](#). These books looked at whole collections of projects and derived common architectural ideas useful across different domains.

Importantly, these books looked at several elapsed years of experience with the underlying tools before making any unifying opinions. Serverless hasn't even

existed long enough as a technology to warrant such a book, but it's getting closer and within a year or so we'll start seeing some common, useful practices emerge. (Anyone who today uses the phrase "[best practice](#)" when it comes to serverless architecture needs be given a significant raised eyebrow).

Beyond application architecture (how serverless apps are built), we need to think of deployment architecture (how serverless apps are operated) too. I already talked about deployment tools, but how do we use those tools? For instance:

- What do terms like "environment" mean in this world? "Production" seems less clear-cut than it used to be.
- What does a side-by-side deployment of a stack and slowly moving traffic from one set of functions/service versions to a different set of functions/service versions (rolling deployment) look like?
- Is there even such a thing as "blue/green" deployment in this world?
- What does rollback look like now?
- How do we manage upgrading/rolling back databases and other stateful components when we might have multiple different production versions of code running in functions simultaneously?
- What does a [phoenix server](#) look like now when it comes to third-party services that you cannot burn down and redeploy for cleanliness?

Finally, what are useful migration patterns as we move from

one architectural style to something that consists of or includes serverless components? In what ways can our architecture evolve?

Many of these yet-to-be-defined patterns (and anti-patterns) are not obvious, most clearly shown by our very nascent ideas of how best to manage state in serverless systems. There will no doubt be some surprising and fascinating patterns that emerge.

How our organizations will change

While cost benefits are one of the drivers of serverless, the most interesting advantage is the reduction of conception-to-production lead time. Serverless enables this reduction by giving “superpowers” to the vast majority of engineers who aren’t experts in both systems administration and distributed systems development. Those of us who are “merely” skilled application developers are now able to deploy an entire MVP without having to write a single shell script, scale up a platform instance, or configure an

Nginx server. Earlier, I mentioned that deployment tooling was still a work in progress, and so we don’t see this “simple” MVP solution for all types of application right now. However, we do see it for simple web services — and even for other types of apps, deploying a few lambda functions is still often easier than managing operating-system processes or containers.

Beyond the MVP, we also see cycle-time reductions through the ability to redeploy applications without having to be concerned about Chef/Puppet script maintenance, system patch levels, etc.

Serverless gives us the technical means to do this, but that’s not enough to actually realize such improvements in an organization. For that to happen, companies need to grapple with, and embrace, the following.

“True” DevOps

DevOps has come to mean in many quarters “technical operations with the addition of techniques more often seen in

development”. While I’m all for increased automation and testing in system administration, that’s a tiny part of [what Patrick Debois was thinking when he coined the term “DevOps”](#).

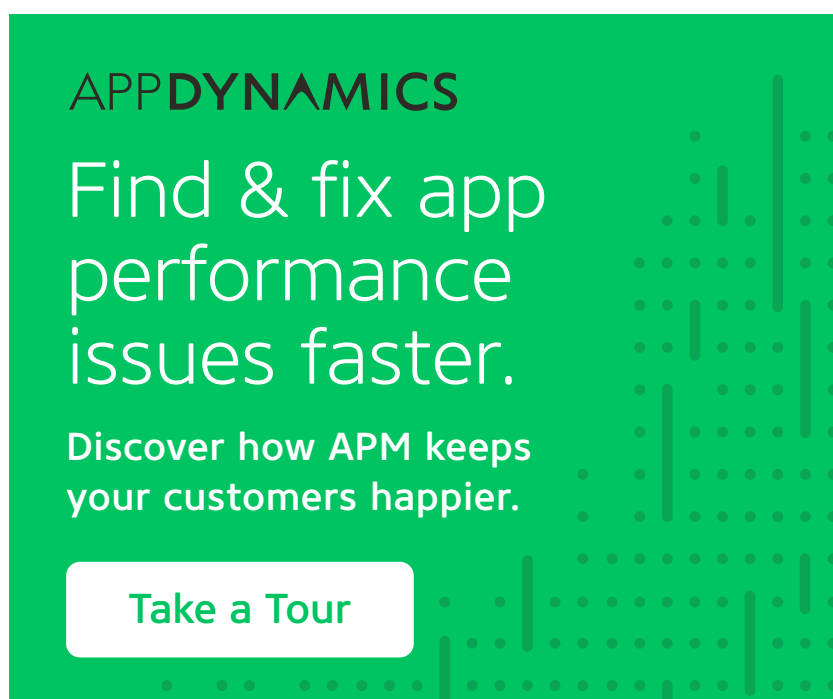
True DevOps is instead about a change in mindset and a change in culture. It’s about having one team whose members work closely together to develop and operate a product. It means collaboration rather than a negotiated work queue. It means developers performing support. It means tech ops folk getting involved with application architecture. It means, in other words, a merging of skill and responsibility.

Organizations won’t see the efficiency gains in serverless if they have separated development and operations teams. If a developer codes an application but then needs someone outside the immediate group to deploy the system, then the feedback gains are wiped out. If an operations engineer is not involved with the development of an application, they won’t be able to adapt a deployment model on the fly.

In other words, in the future the companies that have made the real gains from serverless will be the ones who have embraced true DevOps.

Policy/access control

But even a change in team-by-team culture is not sufficient. In larger companies, an enthusiastic team will often come up against the brick wall of organizational policy. This might mean a lack of ability to deploy new systems without external approval. It might mean data-access restrictions to all but existing applications. It might mean ultra-strict spending controls. ►



APPDYNAMICS

Find & fix app performance issues faster.

Discover how APM keeps your customers happier.

Take a Tour

While I'm not advocating that companies throw all their security and cost concerns out of the window, to make the most of serverless they are going to need to adapt their policies to allow teams to change their operational requirements without needing team-external human approval for every single update. Access-control policies need to be set up not only for the now but also for what might come. Teams need to be given budgetary freedom within a certain allocation. And most of all, experiments should be given as much of a free-reign sandbox as possible while still protecting the truly sensitive needs of an organization.

Access-control tooling is improving, through use of IAM roles and multiple AWS accounts, as I mentioned earlier. However, it is still not simple, and is ripe for better automation. Similarly, rudimentary budget control exists for serverless, again mostly through multiple accounts, but we need easier control of per-team execution limits, and of different execution limits for different environments.

The good news is that all of this is possible through advances in access-control tooling, and we'll see more progress in patterns of budget allocation, etc. as serverless tools continue to advance. In fact, I think automation of access and cost controls will become the new shell scripting. In other words, when teams think of the operational concerns of their software they won't think of start/stop scripts, patch levels, and disk usage; instead, they'll think of precisely what data access they'll need and what budget they require. Because teams will be thinking about this so often, engineers will automate

the heck out of it, just like we did with deployment.

Given this ability and rigor, passionately experimental teams even in the most data-sensitive enterprises will use serverless technologies to try out ideas that would never have made it past the whiteboard before, and will do so knowing that they are protected from doing any real intellectual or financial damage.

Product ownership

Another shift we've seen in many effective engineering teams over the last few years is a change of focus from projects to products. Structurally, this often comes via less focus on project roadmaps, iterations, and burn-down charts in favor of a kanban-style process, lightweight estimates, and continuous delivery. More importantly than the structural changes, though, are the shifts in role and mindset to more overlapping responsibilities, similarly to what we see with (true) DevOps.

For instance, it is likely now that product owners and developers will collaborate closely on the fleshing out of new ideas — developers will prototype something and product owners may dig into some technical data analysis before locking down a final production design. And similarly, the spark of innovation — where a new idea or concept comes into someone's head — could belong to anyone on the team. Many members of the team, not just one, now embrace the idea of [customer affinity](#).

A serverless approach offers a key benefit to those teams embracing a whole-team product mindset. When anyone on the team can come up with an idea and quickly implement a prototype

for it, a new mode of innovation is possible. Now [lean-startup](#)-style experimentation becomes the default way of thinking, not something reserved for hack days, because the cost and time of doing so is massively reduced.

Another way of looking at this is that teams that don't embrace a whole-team product mindset are likely to miss out on this key benefit. If teams aren't encouraged to think beyond a project structure, it's hard for them to make as much use of the accelerated delivery possibilities that serverless brings.

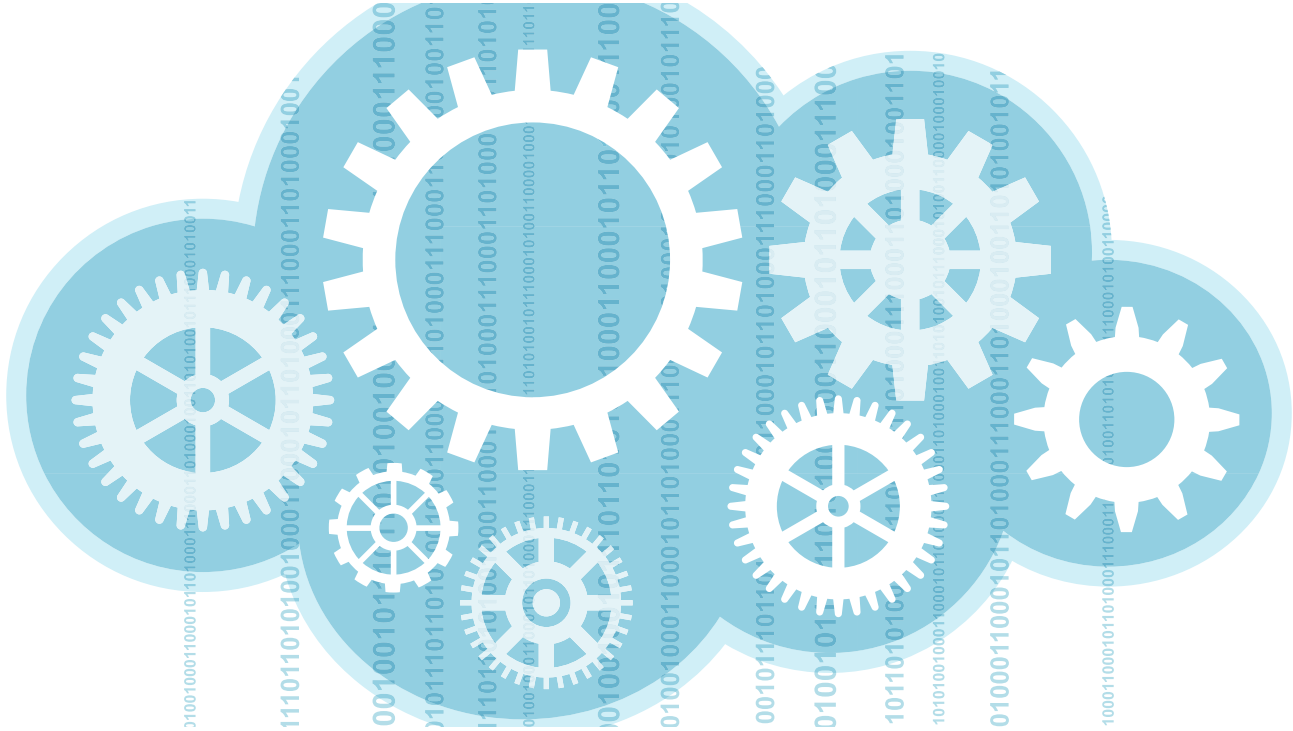
Conclusion

Serverless is a relatively new concept in software architecture, but is one that will probably have an impact as large as other cloud-computing innovations. Through technology advances, tooling improvements, and shared learning in serverless application architecture, many engineering teams will have the building blocks they need to accelerate, and even transform, how they do product development. The companies that adopt serverless, and adapt their culture to support it, are the ones that will lead us into the future.

Acknowledgments

Thanks to the following for their input into this article: John Chapin, Chris Stevenson, Badri Janakiraman, Ben Rady, Ben Kehoe, Nat Pryce. ■

Lambda Functions Versus Infrastructure: Are We Trading Apples for Oranges?



Dean Sheehan has more than 25 years experience in the technology industry covering consulting, product development, product management, and solution deployment throughout the retail, financial, and telecom industries with significant expertise in data-centre automation. He currently serves as director of solutions consulting, EMEA for Apcera. Apcera provides developers with the tools to simplify the process of software creation and operations teams with the control and insight to enable effective, economical, and secure operation of that software.

As technologists, we're constantly bombarded with new paradigms, frameworks, and tools. When adopting any new technology, we need to understand its implications in the operational dimension as well as development dimension: are we really simplifying or are we simply trading one set of problems for another?

Hypervisors enabled virtual machines (VMs), which we created as though memory, compute, network, and storage grow on trees. Then came a suite of management tools to help us shepherd our VMs and gain the most value from this disruptive advance. Docker has popularized containers and once again we are facing a proliferation of operational components that is

driving the need for improved management tooling.

Amazon's AWS Lambda service along with other serverless offerings have recently been receiving interest due to their simplicity, capabilities, and potential for cost reductions.

With serverless models, we are trading the management of less

granular components like virtual machines, containers, or complete applications for the management of very granular functions. Even if your operations team is geared and tooled to manage 100 applications, will they be ready to manage the 500 to 1,000 entities that may result from the functional decomposition? ►

What is AWS Lambda?

Lambda calculus is a branch of computer science and mathematics that deals with the abstraction of functions and their application to create a universal computation model, which may or may not have influenced those responsible for naming within Amazon. Ignoring that, AWS Lambda is a service for running your functions, expressed in a variety of programming languages, with complete disregard for any web server, application server, or framework scaffolding, let alone the notion of some kind of computer in which to execute. Simply write your function, log into your AWS account, and deploy your function into the AWS Lambda service and you are done. Your function is ready to run — now, you just need to decide when it should run.

Before I talk about the execution of your lambda function, here is a complete implementation of a function written in the Node.js programming language for deployment in AWS Lambda:

```
exports.myHandler =  
function(event,context) {  
  context.succeed('hello');  
}
```

Typically, even with the declarative power of Node.js, there would be around 20 more lines of code before this function could be exposed and executed as a service. AWS Lambda has certainly simplified our implementation. What about our operations?

Some refer to this as stateless computing or serverless computing. I prefer the second term, as there is clearly a state somewhere — probably in a database service that the function may leverage — but the function itself is essentially stateless. The

same argument could be held against the serverless term as clearly there are servers floating around in the cloudy background, but their existence is implicit and automatic rather than explicit and manual.

The next area of value in AWS Lambda stems from the ability to easily associate your function with all manner of triggers via both web-based and command-line tools. There are more than 20 different triggers that can be used — most come from other AWS services such as S3, Kinesis, and DynamoDB — including the ability to associate your function with a URL that can be exposed externally via an API gateway. Simply browse the available event sources and integration points and pick the one or more you want to associate with your function and the integration is complete. Your function will now execute automatically in response to, for example, some new data being posted into the selected S3 bucket.

To summarize:

- AWS Lambda functions are easy to build and deploy.
- You only pay for resources used in the execution of your function, not for servers to stand by and be ready to execute your function. This should yield cost savings where you have highly variable load.
- Automatic scaling of the underlying compute resources falls out as part of the approach. In fact, it isn't even really a question or consideration from a technical perspective.
- Integration with other AWS services is trivial as is expos-

ing your function for direct invocation.

- You are trading management of fewer, coarser-grained components for the management of an increased number of finer-grained functions.

To function or not to function?

This all sounds amazing — write a function, deploy it in AWS Lambda, connect it up, let scaling happen automatically, and only pay for the resources used in its execution.

However, AWS Lambda isn't a silver bullet. It needs to be considered as only another tool in our toolbox and we need to avoid seeing nails everywhere so we can wield our shiny new hammer-shaped tool. When selecting the right tool from our toolbox, we need to consider the development and operational dimensions. Just because it is easier to develop and possibly cheaper to deploy, doesn't mean it is easier to manage or even practical to manage at scale.

Functions are best used where there is a simple transformation to be performed. For example, and this is an example that Amazon itself uses frequently, we can use it to create a thumbnail image based on a document loaded into an S3 bucket. Another example would be sending an SMS message when a data stream (maybe from Amazon Kinesis) shows some values falling outside of a normal expected region — an alarm!

Both of these examples share common traits.

Both are stateless; that is, information comes in and is used to create new information (thumb-

nail or alarm) with no need to refer to anything else beyond the input.

Additionally, these examples require limited error handling. If a thumbnail can't be created, then the thumbnail will not be created or a stock thumbnail might be output instead. Similarly, the alarm represents the error case: either there is an alarm to raise or there isn't. There isn't really a third case where the input stream induces an error that needs to be reported in some way beyond the creation of the alarm. (We could consider an error report for not being able to raise the alarm message but that's getting into the weeds.)

Finally, there is little control flow or conditionality that leaks outside of these functions. The data-driven behaviour — if-then-else — resides within the function.

If your requirements are similar then coding them as a function for deployment in AWS Lambda has potential. However, not everything fits this model and there is a need for state, error handling, and complex conditional choreography.

Some would argue that you could always break down a larger problem into small pieces. And this is certainly true, but there is a balance to be struck. Decomposition down to multiple small functions increases the need for inter-function plumbing, expands the surface area — which many consider to be a security vulnerability — and places greater load on operations and management, let alone the cognitive load required to comprehend, for maintenance and change, a complex interwoven web of distributed functions.

How will enterprise organizations adopt cloud functions?

The majority of companies we work with fall into the pre-existing, Global 2000 range rather than the new digital economy and startups, and as such I haven't heard them beating the cloud-function drum too much. However, this is starting to change. At the moment, I would say about 10% of these companies have expressed either a direct interest in cloud-function architectures and platforms or architectures supporting pay per transaction — and the thing to note is that this percentage is increasing.

Aside from the technical sweet spot for cloud functions being relatively small, larger, more mature organizations typically have a lot of regulatory and compliance obligations from a mix of external and internal bodies. Cloud functions, along with microservices-based architectures and agile software-development practices are somewhat at odds with the governance desires of these organizations, with the result of an impedance mismatch. We can create new software and modify existing software much more quickly these days and certainly more quickly than we can push it through an established software-development lifecycle that's living under significant governance.

To be able to benefit from cloud functions at scale, we need to expand their technical sweet spot and appreciate the need for checks and balances that enable managers, operators, CIOs, CSOs, and risk and compliance officers to sleep soundly at night.

To expand the technical sweet spot, we need to increase the error-handling facilities of cloud

functions to make sure that more than just trivial transformation functions can be componentized this way. Being able to trap and expose error conditions in a robust manner then falls neatly into the next area that needs addressing: flow of control and complex choreography. We need to be able to wire together cloud functions in a manner that doesn't just shift the pain from one programming language to another, or from a programming language to a user interface that requires 20 steps to capture the plumbing and then makes it hard to see what you've plumbed afterwards. There needs to be a net improvement in productivity in both creation and comprehension.

With respect to governance, just because it is a cloud function doesn't mean that it should escape controls over which language, runtime, and libraries it is using, or how much compute, network, and IO it is consuming, or what region it is executing in with respect to the data services it might be using (think Safe Harbor etc.) to name just a few of the issues that worry operations teams and managers.

The only constant is change

As mentioned earlier, cloud functions should be one of many tools in our toolbox. Whilst it might be the most recent architectural tool, we can be sure that it will not be the last.

On the vertical axis of Figure 1, cloud functions are absolutely, and forced by the execution framework, intrinsically stateless whereas the other patterns can all move freely between stateful and stateless although there are strong preferences that would be ►

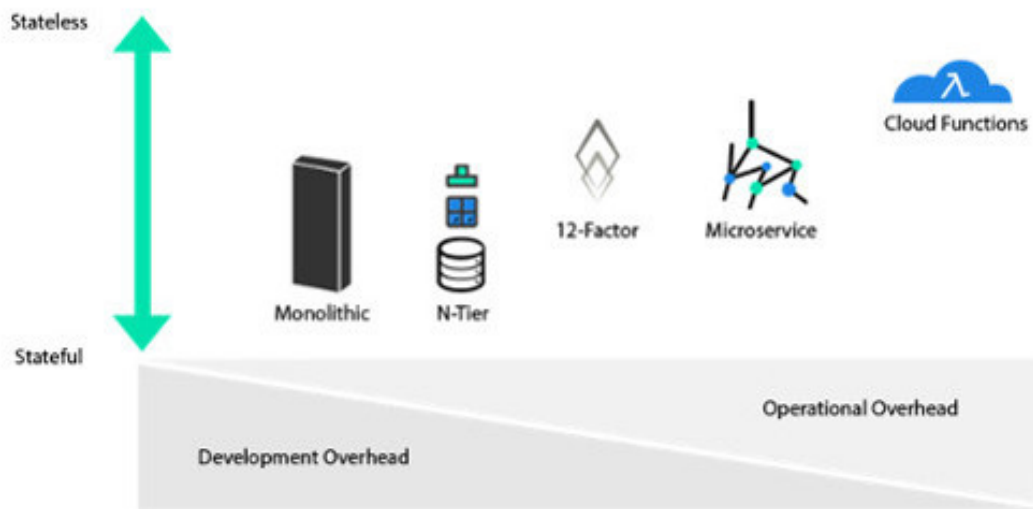


Fig. 1: This figure plots architecture patterns on axes of development/operational overhead versus statelessness. As with all attempts at categorization, there are exceptions but this is broadly correct.

tough, or an anti-pattern, to deviate from.

The horizontal axis illustrates the relative effort/overhead in the development and operational dimensions. At left, monolithic architectures have a lot of development inertia; things that get in the way and slow down initial creation and subsequent alteration of a solution built using this approach. Operationally, monolithic architectures are as simple as can be, one large program that has to be run, secured, and monitored. At the other extreme, cloud functions are arguably the simplest expression of reusable, and cloud-accessible, algorithms from a development perspective. However, a complete solution is made from many cloud functions (and more than likely other non-cloud-function components) that each need to be installed, updated, secured, scaled, governed, and monitored.

Not many people would set out today to develop a solution based on a monolithic architecture — but you never know, considering that it might be the right tool for the right job. Tiered architectures, 12-factor cloud-native applications, microservice architectures, and their logical evolution of cloud functions are all modern options. While architects and engineers select the right approach to solve the problems they face today, they must also address how requirements will change over time. An added pressure is talent retention, recruitment, and the technologies being used within an organization can have a big impact on both of these. It isn't easy to recruit people to work on yesterday's technology — even if yesterday is part of today and part of the foreseeable future.

The only constant is change, and we cannot be prescriptive about how software should be developed. To that end, we should use

platforms and tools that support all of the architecture models mentioned above including a capacity to adapt as we embrace the future — as far as we can envisage it anyway.

Conclusion

Gartner predicted in March 2016 that by 2020 more than 50% of all new applications developed on PaaS will be IoT-centric, disrupting conventional architecture practices.

“IoT adoption will drive additional use of PaaS to implement IoT-centric business applications built around event-driven architecture and IoT data, instead of business applications built around traditional master data,” said Benoit Lheureux, research vice president at Gartner. “New IoT-centric business applications will



Fig. 2. Even in a green-field situation, we need to consider cloud functions only another tool alongside other architecture choices. We need to consider development and operational efforts to make the right overall decision and not optimize for a local minima. We should select platforms and tools that enable choice: choice in programming language, choice in framework, choice in architecture, choice in on/off-premise, mixed infrastructure, etc. Above all else, we should ensure that the chosen platforms and tools embrace and facilitate trust and governance within the choices we make.

drive a transformation in application design practices that focus on real-time contextually rich decisions, event-analysis, lightweight workflow, and broad access to Web-scale data."Delivering enterprise-scale solutions out of cloud functions can be difficult. Technologies change and the universe of services, devices, frameworks, business expectations, and their associated risks are increasingly expanding. It's exciting to be at a point in our industry where tooling is evolving just as rapidly, enabling us to combine yesterday's problems with today's trends and what tomorrow will bring in a reliable, productive manner.

In order to achieve accelerated time to market, developers need to focus their efforts on delivering business value rather than modifying yet another automation script to get their software to build, test, and deploy. Execution assumptions need to be extracted from programs and avoided all together or captured in metadata to enable the runtime platform to understand the operational needs so that it can reason and apply execution decisions at machine speed and against the volume of machine-accessible information.

At the intersection of development and operations lives trust. Trust, and that feeling of comfort and safety, originates from knowing that you've done all you can and that someone or something has your back if you make a mistake. The platform and tools we use to develop new software, maintain older software, and har-

moniously operate both need to evaluate and enforce the governance policies laid down by our organization to ensure maximum creativity and velocity within guardrails and with non-invasive oversight and protection.

If we make workloads smaller, focused, and simpler whilst capturing operational requirements and governance policies in machine-readable forms that can be reasoned upon, then we create the potential for a global compute fabric that embraces all manner of devices and use cases in a safe and secure fashion. ■

InfoQ Virtual Panel: A Practical Approach to Serverless Computing



by Richard Seroter



Kenny Bastani works at Pivotal as a Spring developer advocate. As an open-source contributor and blogger, he engages a community of passionate developers on topics ranging from graph databases to microservices. He is also a co-author of O'Reilly's *Cloud Native Java: Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry*.



Joe Emison is a serial technical entrepreneur, most recently founding BuildFax in 2008, and has consulted with many other companies on their own cloud launches and migrations. He oversees all technology and product management for Xceligent, and regularly contributes articles to *The New Stack* and *InformationWeek* on software development and the cloud. Emison graduated with degrees in English and Mathematics from Williams College and has a law degree from Yale Law School.



Fintan Ryan is an industry analyst at RedMonk, the developer-focused industry-analyst firm. His research focuses on all things related to developers, from tooling to methodologies and the organizational aspects of software development. His primary research areas for 2016 include cloud-native computing architectures, data and analytics, software-defined storage, DevOps, and machine learning. He is an accomplished keynote speaker and panel moderator.

KEY TAKEAWAYS

The sweet spot for serverless functions is where operational costs are unnecessarily high. Developers typically target green-field apps versus rebuild existing ones in a serverless fashion.

High-volume compute is where serverless functions should allow substantial cost benefits but there is no guarantee you're going to save money on compute by adopting a serverless architecture.

Connect your code to the services you want to be locked into. Serverless products today have many convenient — but proprietary — hooks into other cloud products.

Spend time in the proof-of-concept phase to determine whether serverless actually would be meaningfully better for what you are working on.

Add serverless computing to the growing list of options developers have when building software. Serverless products, more accurately referred to as “Functions-as-a-Service” (FaaS), offer incredible simplicity — but at a cost. What should developers know before moving to this stateless, asynchronous paradigm? How should companies evaluate any perceived or actual platform lock-in that accompanies these services? To learn more about this exciting space and the practical implications, InfoQ reached out to three experienced technologists:

InfoQ: If we assume that most enterprises do not (yet) have sophisticated event-driven architectures in place, are serverless functions a drop-in replacement for many existing components? What realistic use cases should developers and operators look at when introducing functions to their environments?

Kenny Bastani: Based on what I've seen, serverless functions are not a drop-in replacement for existing components. I see two roads to adoption for FaaS: green-field applications and

supplemental on-demand workloads.

For green-field applications, building serverless applications dramatically lowers the cost of experimentation. A business or IT might have many ideas they would like to try out, and may have the development resources to execute on them. But until now, the blocker for trying out many ideas has been the misappropriation of critical resources. By critical resources, I mean infrastructure and operations. The infrastructure and support costs that tap into existing operations pipelines for hosting enterprise applications is usually too high to support continuous

experimentation. Serverless applications offer enterprises the ability to try out new ideas that they would normally say no to. If some of the new ideas end up failing, it didn't come at the cost of derailing other ongoing business-critical projects. On the other hand, if one of the ideas ends up a big success, the business can more easily justify investing additional resources.

The other road to adoption for serverless is supplemental on-demand workloads. These kinds of workloads are those that are not meant to be long-lived and do not need a permanent home on premises. One example could be a function that is ►

responsible for moving data between a third-party SaaS product and an application that is hosted on premises. A serverless function could be invoked only when needed and perform an integration task that bridges the two systems. The function would be able to use the developer APIs that are provided by a SaaS vendor to insert or update data in a third-party system using a trigger that is hosted on premises.

Joe Emison: I do think that if you have a microservices-based architecture, you can replace an entire microservice with a single or a cluster of serverless functions, and depending upon how big the microservice is, it might feel fairly trivial. For example, I have built server-based microservices that are called by single-page web apps (SPA) to pull in content for the SPA, and the only reason they were server-based were my need to keep a credential hidden or CORS issues with the service endpoint. In these cases, replacing the microservice with a serverless function was trivial. But outside of this type of situation, I agree with Kenny that you're not going migrate a monolithic application to serverless functions meaningfully.

I disagree with Kenny that serverless functions lower experimentation cost dramatically. I think he both overstates the cost of experimentation without serverless functions, as well as understates the pain of at least the most dominant serverless function service, AWS Lambda. Most experimentation and development happens on developer workstations (laptops). I and my teams try out ideas all the time, and it never involves dependencies upon other departments just to test these things properly. And this is true with basically all of the enterprises I work with,

which includes a decent number in the Fortune 1000 and government agencies. Evaluation speed has not been a huge issue in my experience over the past three or four years at least.

I also find AWS Lambda to be painful in the extreme from a startup standpoint — which is exactly where you want speed in testing an idea. The insane number of services you have to use (IAM, API Gateway, DynamoDB, SNS, SQS, and on and on) is the opposite of fast startup and testing. And if your enterprise allows you to use Lambda, it presumably also allows you to use Heroku or similar services (Digital Ocean, Lightsail if you're AWS only), and those are much better known and easier to launch code and experiment on than Lambda.

I think the sweet spot for serverless functions is where your operational costs are unnecessarily high — either as a direct cost, in time, or as an opportunity cost — and by eliminating server maintenance, you dramatically reduce those costs. If you have any application (probably a small one) that is getting stalled or has too high a cost because of what your organization requires for operational buy-in and support to get it live to customers, then serverless functions may be a great way to handle things.

I would be very wary of writing large green-field applications in serverless functions today. And to be clear, I mean where the serverless-function set is large. If you're writing a large native app or SPA and it relies upon a small set of serverless functions — that's what I do today and have been doing for more than 18 months so I certainly recommend it for that. But if I were planning on building an application that

will likely have more than 10,000 lines of code on the back end or middle tier (and so would be in serverless functions), I would absolutely avoid serverless functions for that application. We just don't know enough about the development and architectural patterns and how we should be structuring those applications as they get big. The likelihood that we build something that is unmaintainable is too high.

Bastani: I'm not one to disagree with the experience and viewpoints of others; we all bring our own perspective. I'm basing my opinion off of what I've learned from industry leaders who are experts in their field. After hosting this year's serverless track at GOTO Chicago, I was able to hear the first-hand experiences of those who have been successful with using serverless for continuous experimentation — namely, Mike Roberts, who writes about the definitions of serverless and FaaS on Martin Fowler's website. In his talk at GOTO Chicago, he talked about how serverless enables continuous experimentation, not just for developers to test things on their local machines, but to stand-up production applications that actual users will interact with. There is no value in experimentation if there are no users to provide feedback. Experimenting with technology is one thing; continuously delivering experimental products and features to customers is what matters, and serverless enables that.

With regards to AWS Lambda, I've used the system extensively myself and I do realize there are some competencies that you must pick up. That's why I have experimented and created reference architectures that allow you to completely bypass using AWS's services with your server-

less functions. In this case, you would be able to create a “micro-repo” of serverless functions that are in the source of your microservice application, in this case Spring Boot. You can use Concourse to continuously deliver changes to your serverless functions, and inject in credentials of third-party services that are managed by other platforms, such as Cloud Foundry.

Finally, I have had opportunity to talk to developers working in the Fortune 100 who are building experimental applications with AWS Lambda. Just because what you’ve seen doesn’t match with the perspective of others does not mean it is not happening.

Emison: I didn’t say that the Fortune 100 isn’t experimenting with Lambda. What I said is that I don’t see a huge experimentation pain in those in the Fortune 1000 who aren’t using Lambda. That is, I don’t see Lambda (or serverless functions) as a way to get dramatically better at experimentation. Experimentation happens on development machines, which developers have. Serverless functions — as far as I have seen — don’t give a huge boost to experimentation and testing beyond what we already have with Heroku and just plain old EC2 on AWS. And I see plenty of great testing and experimentation in Fortune 1000 companies without dependency pain on operations.

It would certainly be interesting to hear Mike Roberts talking about what he thinks Lambda gets one above and beyond traditional VM and container-based ways of deploying code as far as experimentation goes. [His article](#) limits his experimentation benefit to simple functions where the organization is already using core AWS services. Of note,

where he talks about the development agility benefit of serverless, he specifically mentions my ServerlessConf talk where I focused on serviceful architectures (as opposed to serverless functions).

Bastani: I think the misunderstanding here is you may not know what “continuous experimentation” is. It’s a term that was spun out of Etsy that means being able to use continuous delivery to test assumptions (A/B test, experimental features, etc.) If the customer doesn’t find the feature is valuable, it can be rolled back.

Here is [a good talk](#) about continuous experimentation at Etsy. Also, here’s [another good talk by Mike Roberts](#).

Emison: Just watched it. I find it to be incredibly similar to [the speech I gave](#) at the first ServerlessConf on the same topic (and Mike cites this talk of mine in his article that I linked to above).

In particular, his examples are more on the serviceful track than on the serverless-function track (he gives my same examples: Firebase, Auth0, etc.). He’s very concerned about doing product/product management correctly, which is also my central focus, and proper CI/CD. By my rough count, he only spent about one to two minutes of the 30-minute talk addressing serverless functions at all. He talks about blameless post-mortems more than serverless functions.

So I don’t really see how his speech addresses the core issue of how serverless functions really change the game versus Heroku or similar container-based/container-like options for rapid testing/rapid idea-to-deployment.

I would reiterate my argument that I think that where serverless functions really change the game is in situation where you have significant costs — and this includes opportunity costs and dependency costs — related to production operations.

Bastani: I should have watched your talk before today. Thanks for linking it.

I’m in violent agreement with your last two statements.

In many ways, platforms like Heroku and Cloud Foundry are similar to serverless. Each of these platforms gives developers a way to deploy their applications without worrying about setting up and managing servers. The big difference of course is the cost model. Now developers can create workloads that are long-lived but not always on — but are still available. That opens up a range of use cases that developers are just now starting to think about in an enterprise setting.

Fintan Ryan: Most of the FaaS development we have seen has been green-field applications that require some form of basic event handling and initial offload. There are numerous examples of FaaS being used in other ways, but the majority of applications at scale are falling into this relatively simple use case.

As of now, we have not seen FaaS appearing as a replacement for existing components in any significant manner for legacy apps. We do see some replacement of components in newer applications that are already adopting a microservices approach.

Ultimately though, FaaS is just another tool to be used for certain types of workloads; to make proper use of it teams need a lot ►

of other pieces in place. The current user base is, still, very technically savvy, and reasonably well versed in good software practices.

InfoQ: Let's poke further at Joe's comment about costs. One of the characteristics of serverless that people talk about a lot is "pay only for what you use." Do the compute cost savings (versus using a virtual machine or container instance) come into play more for infrequently called services or for high-volume ones? Or is this an overrated talking point, and do the costs that matter relate more to operational cost, as Joe states?

Emison: I don't think that infrequently called services get any cost benefit from serverless functions (and I don't think they get architectural or many time-to-live benefits from serverless functions either). Digital Ocean, Vultr, and even Amazon Lightsail are really cheap options for running code that has a small amount of traffic. Even from an availability standpoint, it's pretty cheap and simple to make these highly available (think multi-AZ Amazon RDS and two app servers in a load balancer). We know systems like this are easy to maintain and we can bring in consultants or new developers to manage them fairly easily.

High-volume — especially unpredictably high-volume — compute is where serverless functions should be able to give really substantial cost benefits, especially where the actual amount of processing you need to do is fairly simple to encapsulate (e.g., a single function) and doesn't take more than a couple

of minutes at most. Many people have been using Hadoop streaming for these cases, but serverless functions are a much better/simpler fit than having to cram something into the map/reduce logic just to get the failover and scheduling benefits that Hadoop gives.

And note that scheduling itself can have a really high cost — if you run into situations where you have to schedule (e.g., you have fewer computer resources than demand for those resources), managing scheduling is usually very expensive in people, opportunity cost, uncertainty cost, and "actual time to develop/deploy a solution" cost. Serverless functions give you an incredibly cheap way to "pay only for your compute" along with an unlimited amount of resources so that you don't have to worry about scheduling anything, and thus all scheduling costs go away.

All that said, I do generally believe that the costs of compute are overstated, and the costs of people are under-considered in the extreme. I can't tell you how many times I've been in conversations about "How do we lower the Amazon bill?" but also be told that we can't cut IT jobs, even though the cuts to the Amazon bill would be, at best, half a FTE. We live in a day and age where you just don't need many (if any) sysadmins or network admins or DBAs, and the vast majority of organizations would be better off if they invested more and spent more on cloud and cloud automation in the interest of removing some of the people in those jobs that can now handle orders of magnitude more infrastructure (if done to modern best practices).

Bastani: I think that talking points around costs as an adop-

tion criteria for serverless should be nuanced. I've actually heard stories from AWS Lambda early adopters that a serverless architecture ended up costing more in compute than the architecture they migrated away from. The compute costs of the new system became cost prohibitive. Because of this, they ended up migrating back to the old system. So, there is no guarantee you're going to save money on compute by adopting a serverless architecture. If anything, it becomes harder to predictably measure the compute costs over time, in the same way that it becomes difficult to accurately predict demand over time. If there is a spike in traffic, it may equate to a spike in compute costs.

The discussion around costs for adopting serverless should be more about how it complements the qualities of your current architecture. It's good to ask yourself questions like "How will adopting serverless help me deliver features faster to production?" or "Will serverless make it easier to maintain my architecture over time and eliminate sources of technical debt?" The answers to most of these questions will really depend on a modular design. If you're bad at architecture, everything will be more expensive in the long run.

InfoQ: The growing consensus about lock-in and serverless platforms seems to be that the code itself is portable, but it's the many ancillary services that pin you to a certain platform. Do you agree? Does it matter? And what should those who aren't ready to pick winners and losers in the serverless space do in order to maintain host flexibility?

Emison: I think the ancillary services are only part of the issue, and probably unique to Amazon right now. Amazon has baked Lambda so thoroughly into its ecosystem that you can't actually use Lambda without connecting it to several (maybe even half a dozen) different AWS services. And it is meant to be used and works best within that ecosystem. But the same thing is also somewhat true with something like Google Cloud Functions, which works really well with Firebase, and there really is not a good Firebase equivalent at AWS (Dynamo is a poor competitor to Firebase right now), so that has a lock-in element at Google.

Beyond ancillary services, it's certainly possible to get quite locked into a FaaS framework by the actual code you have, the framework(s) you are using, how you have it organized in your code repository, and how you deploy it. For example, many people on Lambda are choosing the Serverless framework, and may be choosing to execute arbitrary Linux code on Lambda. Switching costs from Lambda and the Serverless framework will be high — assuming that the target FaaS platform even supports the language/code you want to run on it.

I think the vendor lock-in issue with FaaS is much less scary than the architectural lock-in you'll have that will likely go hand in hand with the vendor lock-in. That is, the application will get increasingly harder to add on to, to debug, and to train new developers on. FaaS essentially requires a microservices architecture, and to a large extent, each function should be viewed as its own microservice. So instead of reading all of the functions in a monolithic code base (for which we have good style guides on

how to organize and with which developers are generally familiar), you will have a large collection of functions that likely don't have coherent overarching organization or documentation that explains how you're supposed to work with all of them. (See, e.g., [this Register piece on BA's 200 systems in the critical path.](#))

As a side note, I think that FaaS will follow a similar path to that of database normalization 15 years ago, which is that everyone kept seeking true third-normal-form database structures as a best practice. We don't do that anymore, because it created unmaintainable garbage that was incredibly hard to write good interfaces for, or import data into, etc. Instead, we will selectively de-normalize pieces of databases for sanity (at least using enumerated values in tables instead of lookup tables). We're about to see the over-function-ification of software development with FaaS, where — like you'd end up with 350 tables in a database structure — you'll end up with 350 functions in an application that would have only had 75-100 in a monolithic application (or in well-constructed microservices not on FaaS).

All this is to say that the best way to maintain flexibility in this world is to reduce the actual amount of code (and certainly your cyclomatic complexity of applications) within FaaS platforms. I don't see huge problems with using great services like RDS or Firebase — they give you a competitive advantage so who cares if they're proprietary? But you can and should certainly avoid building large applications that call tons of different functions within any FaaS platform today. I think it is a near certainty that you'll regret it in 18-24 months.

Ryan: I think Joe hits the nail on the head here re architecture, which I'll return to in a moment.

There are certainly concerns about ancillary services, but to Joe's point earlier: for most people that are using FaaS at scale this far from a concern, if anything it is an advantage. Products like RDS, Firebase, etc., simplify the longer-term operational maintenance, albeit for lock-in and cost. Many of the heavy users of FaaS we have spoken with are more than happy to take that trade-off in at least the medium term for the efficiency benefits they are currently gaining.

On the architecture aspect, a jumble of code is still a jumble of code if it is not well thought out, written in a coherent and consistent manner, and structured properly. There is already a tendency towards overcomplicating things with tons of different functions making up an application, and the application management overhead that that entails.

Bastani: For a while now I've thought that the idea that you have to use the ancillary services of a provider with FaaS is a myth. A while ago, I decided to do some compatibility experiments with AWS Lambda and Cloud Foundry. One of the things that I've found is that you can absolutely use services provisioned using Cloud Foundry with your functions deployed to AWS Lambda.

I disagree with Joe that you have to use AWS's services with Lambda. But I do understand why most people think that this is the case. First of all, AWS makes it quite easy for you to use their services inside of a Lambda function. Secondly, the event sources that are available to Lambda are mostly all attached to AWS services. But ►

the truth is that it's entirely possible to use a CI/CD tool like Concourse to stage service credentials provided by Cloud Foundry inside of the Lambda function that you're deploying. You can even cache the connection details of a third-party service between function invocations, which keeps things performant.

As for portability, with Cloud Foundry you're given a portable abstraction that can run on top of multiple cloud providers. That's because under the hood there is a tool called BOSH that speaks to the IaaS of the different providers. If you want to switch vendors, BOSH will allow you to move your VMs to a different provider. That is where portability with FaaS becomes tricky.

At face value, Lambda appears to be a platform service. That is, Lambda is not a resource abstraction at the IaaS layer. I think there is a good reason for this. Because the execution model for FaaS has a unique cost model, functions as a resource cannot be a part of the IaaS. That is the same reason that we don't use containers as a core abstraction of an IaaS, but instead we use VMs. Why FaaS is unique is that it is a service that only becomes possible to provide when you control the pricing model of the underlying IaaS. Other than that, the code that you host with a provider is portable, just like the code that sits inside of a container is portable. Your functions are portable as long as the services that you connect to it are also portable.

My recommendation is to connect your code to the services you want to be locked into. If you use a platform solution like Cloud Foundry, you're still locked into whatever sits above the IaaS layer, but you're not necessarily

locked in to the underlying provider.

Emison: Just to clarify, when I say you have to use AWS's services with Lambda, I mean that in order to make a web-accessible endpoint, you at least have to use IAM and API Gateway — otherwise your code is unreachable. And if you want to have any kind of authenticated access across a set of users, you'll have to use Cognito as well. If you want to have any kind of state, you'll need to write and read data, likely from either S3 or Dynamo or RDS. And you'll almost certainly end up using both SNS and SQS if you're using multiple functions. And if you want to debug, you likely end up at least testing X-Ray. And this is by design — AWS is essentially a collection of IaaS microservices that AWS spends a lot of time and money (and its partners help) making life much more functional if you just embrace and adopt lots of them.

Bastani: Thanks for clarifying. Yes, IAM and API Gateway are required. Authentication can be handled by the invoking application. If you're using Spring Boot, you can invoke the function through API Gateway while still managing security as a part of the Spring Boot application. I do however think that security should be handled inside the function, which is being made possible by a new project called Spring Cloud Function. As for SNS and SQS, you can instead use your own messaging broker by injecting access credentials into the function from Cloud Foundry. The main problem with this approach is that you'll have to use an always-on event source, such as a Spring Boot application, that will invoke lambda functions in response to messages being inserted to a queue. In this kind

of architecture, the Spring Boot application becomes the event source.

InfoQ: What programming language do you prefer to use in serverless platforms —keeping in mind that only Node.js is universally supported in AWS Lambda, Google Cloud Functions, and Azure Functions? Are there attributes of one language or another that complement the serverless mindset? Are there languages you'd like to see more readily available in serverless platforms?

Emison: I'm increasingly of the belief that JavaScript, for all of its awfulness, has won and since we can't beat it, we should join it. We could choose to write in a language like CoffeeScript that transpiles to JavaScript to avoid some of the awfulness. Or just have strict style guides and linters and do your best.

JavaScript also lends itself to serverless platforms because it has functional elements (although it is not a functional language) — at least more so than other commonly used languages for web development like PHP, Ruby, and Python. By this, I mean that it feels more natural to write stateless functions in JavaScript than in PHP or Ruby or Python.

I suppose that true functional languages might make even more sense to support on FaaS platforms, and I know that at least Lambda has some tutorials for running Clojure code, but you'd have to have a pretty compelling reason to pick a language that has so many fewer people who can write in it...

Ryan: JavaScript is by far the most popular at the moment, but Python is definitely making some inroads. We also see Java appearing in different guises, be it on AWS or in newer platforms such as Fungatron.

I think that the fact that we now have first-class functions in Java 8 could ease the transition to serverless for many enterprise developers. For the hipsters, there is definitely some interest in using Go (e.g., with the Sparta framework), but like all things Go-related the skillsets are not exactly widely available, and this will slow any shift in that direction.

Finally, we shouldn't discount C# — Amazon didn't invest in making it a first-class citizen for no reason.

Bastani: I think Joe and Fintan have covered the best points. I would add that JavaScript is a bit of a pain when you have complex functions that have a multi-step sequential database transaction where the results of the previous step affect the outcome

of the next step. That's because functions, at least on AWS Lambda, require that you call back to a function provided as a parameter to the entry point when done.

Let's think about that for a second. With an asynchronous flow control in Node.js, you're going to need to keep track of the entry point's callback function and then pass that function along to each asynchronous method call. You can of course assign the callback method to a global variable, but that's a bit of an anti-pattern in JavaScript (or in any language for that matter). Each database transaction is also an asynchronous method, so if you have two transactions that are serial then you're going to need to nest each callback function as a parameter without losing the one you started with. I've found this to be a severe deficit in practice, as you'll end up trying to debug the flow control through log statements that become riddled throughout an asynchronous cascade of callback functions. The symptom of this is that your Lambda function will just time out, without any information about where the flow

control was before the invocation ended.

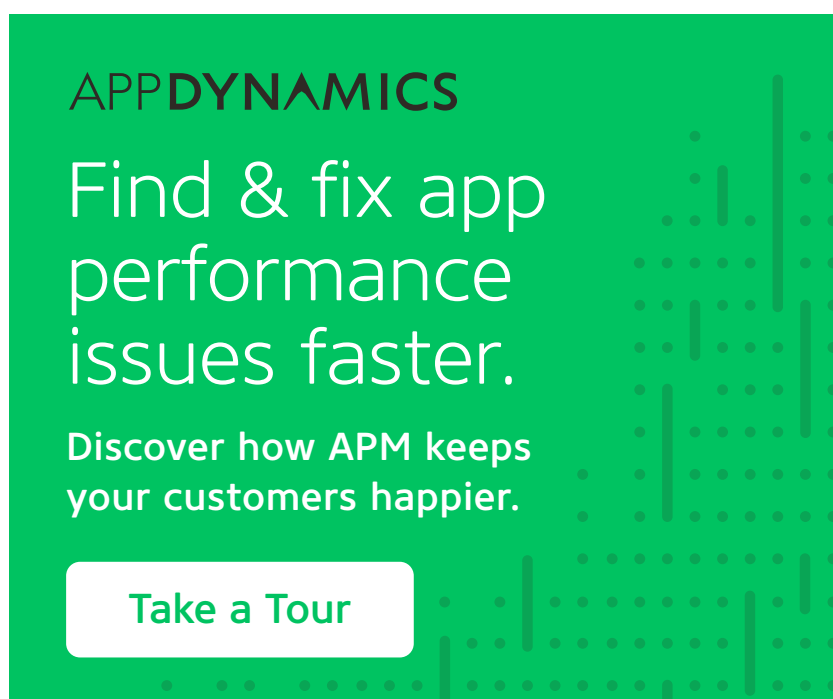
There are ways to work around the callback problems with JavaScript, but each one is a trade-off that sacrifices the quality of an easy-to-maintain modular function.

InfoQ: Is there pre-work that you recommend for developers and system admins getting started with serverless functions? Should they dig into event-driven architecture patterns? Decide on an instrumentation strategy? Set up a CI/CD chain?

Emison: I would sign up for Webtask.io and do their super basic tutorial (which if you have done Node.js or really any Linux development will take you less than five minutes to build and deploy Hello World), and then also their [Slack tutorial](#), which lets you build a chatbot in about five minutes as well. All of this is free and you can easily do it while you're waiting for people to join a conference call, and you'll get the idea and power behind serverless immediately.

I would then recommend reading [Mike Roberts's great page on serverless](#) that is on Martin Fowler's website. Mike does the best job out there at talking about pros and cons and concerns you should have going into serverless. This won't take long, and is well worth looking at before you commit tons of time to serverless.

Once you've done those things, which are quick and (honestly) delightful, then I'd recommend checking out Serverless.com and all of its great quick starts, dem- ▶



APPDYNAMICS

Find & fix app performance issues faster.

Discover how APM keeps your customers happier.

Take a Tour

os, and extensive documentation ([starting here](#)). Most of the “serious” deployments will look more like this and have to deal with the complexities herein (although the Serverless framework does a good job of trying to make it all a lot better).

I would then spend a decent amount of time in the proof-of-concept phase, and see whether serverless actually seems like something that would be meaningfully better for things you are working on, as opposed to something cool and in the upward swing on a hype cycle.

Ryan: The only thing I can add to Joe’s comments is to ensure they have CI/CD set up when getting started.

Instrumentation is definitely something that should be thought about from the outset for any serious production deployment, but right now we are very limited in options.

Bastani: Joe’s list of resources is an excellent starting point.

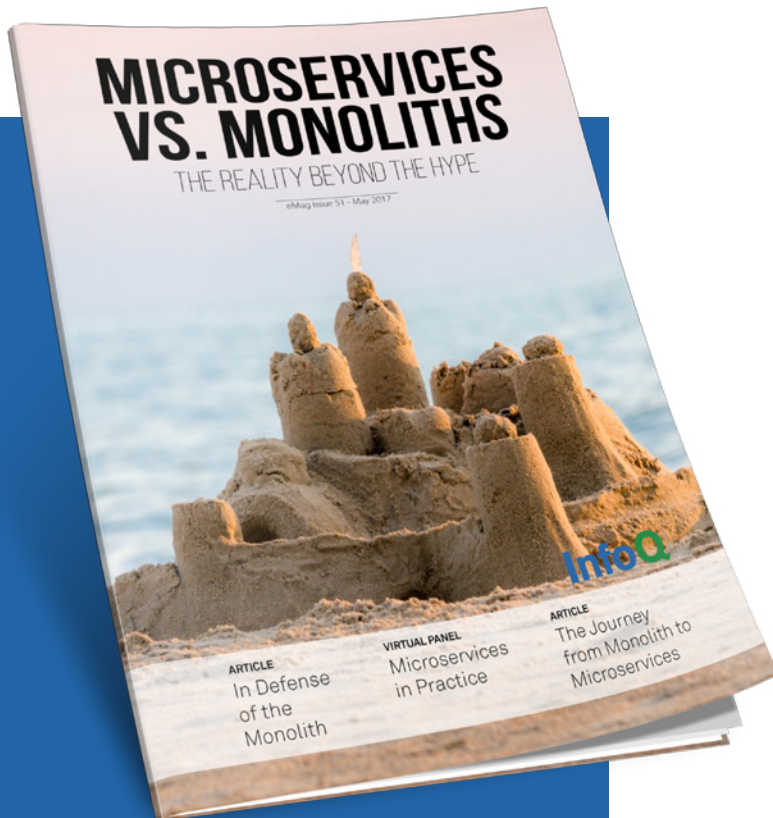
I do recommend to anyone who wants to get started with serverless to experiment with an application you can have fun with. One of the more exciting use cases of serverless functions I’ve come across is conversational interfaces (or chatbots). [Amazon Lex](#) is a conversational interface for building chatbots using either voice or text. What I really love about this service is that it takes something that was relatively inaccessible on the client side and bakes it into a programming model that thrives in a serverless environment. Lex is great because it frames a set of problems that are well suited to a serverless architecture. Lex’s programming model is an excellent playground for those who are in-

terested in figuring out where else serverless is a valuable fit.

My last piece of advice is that serverless is not a panacea. Things like CI/CD and testing are not easy when your execution model is truly cloud-first. Rapid feedback loops in a development environment remain critical to developer productivity. When you shift your execution model to the cloud and start needing to coordinate a deployment of multiple serverless functions — just to test a small change — you can lose your ability to iterate in your local environment rapidly.

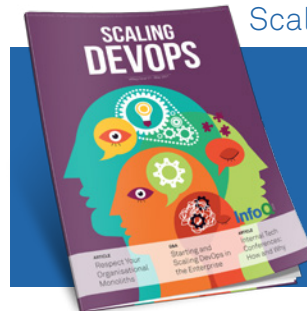
Serverless can help you deploy code to production much faster, but can also lengthen the time it takes to iterate on changes locally. ■

PREVIOUS ISSUES



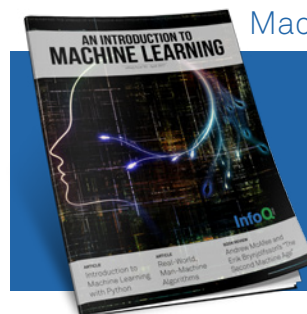
Microservices vs. Monoliths - The Reality Beyond the Hype

This eMag includes articles written by experts who have implemented successful, maintainable systems across both microservices and monoliths.



Scaling DevOps

This eMag collects articles that explore how to go about scaling DevOps in large organizations – effectively identifying cultural challenges that were blocking faster and safer delivery – and the lessons learned along the way. We include a couple of practices that can help disseminate those lessons.



Introduction to Machine Learning

InfoQ has curated a series of articles for this introduction to machine learning eMagazine, covering everything from the very basics of machine learning (what are typical classifiers and how do you measure their performance?) and production considerations (how do you deal with changing patterns in data after you've deployed your model?), to newer techniques in deep learning.



Getting a Handle on Data Science

This eMag looks at data science from the ground up, across technology selection, assembling raw and unstructured data, statistical thinking, machine learning basics, and the ethics of applying these new weapons.