SERVERLESS ARCHITECTURE CONFERENCE

WHITEPAPER

Serverless Expert Check 2019

16 Experts from all over the world discuss how serverless is changing the way developers, operators, and administrators work.

@ServerlessCon #ServerlessCon

www.serverless-architecture.io

Content

Serverless & Function as a Service

Kubernetes, Knative & Co

Managed Services, Backend as a Service

Service Mesh & Microservices

Wide adoption of serverless will be led by tomorrow's giants, not today's

Serverless first!

One day soon, serverless will be seen as a primary way to develop new applications. In this article, Erez Berkner explains why it'll take some time for the knowledge and expertise to accumulate.

by Erez Berkner

The advantages of serverless are clear. IT operations are much easier when a company doesn't have to worry about maintaining its own infrastructure. No more patching or working to keep server or container environments running smoothly. Significantly lower costs and the ability to gracefully scale up and down in response to dynamic use patterns.

One day soon. But not today. It'll take some time for the knowledge and expertise to accumulate, so smart developers can learn to tell powerful serverless design patterns from enticing anti-patterns. But within a decade – maybe half that time – serverless will be a choice that every new app designer will at least consider.

Even then, it won't be the only choice or even necessarily the default. Some application workflows will still make more sense to develop in containers or to just take as fully-managed SaaS solutions. However, as the limitations of serverless are solved and cloud providers produce new offerings, serverless will become a viable design choice for more and more companies beginning their digital journeys.

I say 'beginning' because serverless is easiest to adopt from a clean sheet. Today, we're seeing two types of companies dive fully into serverless.

Who adopts serverless?

The first type is startups who have no technical debt and are able to design from scratch without being constrained by existing systems. Examples include retail platform Sqquid [1] and the self-explanatory Vacation Tracker [2]. These companies were able to dive headfirst into serverless architecture, reducing their upfront costs and enabling them to scale gracefully.

The other sort of companies going all-in on serverless aren't new; they're traditional larger firms who undergo

'digital transformation' to software companies in addition to their core business. Legendary guitar company Fender adopted serverless [3] to deliver its instructional content and manage Industrial IoT in its factories, and Roomba maker iRobot went through a "greenfield" re-architecture [4] when it moved to the AWS cloud. Similar to startups, these companies weren't constrained by tech debt and could benefit from the cost reductions and long-term scalability of serverless.

Many other established enterprises are using serverless somewhere in their stacks. Media company Hearst [5], for example, built a serverless data analytics pipeline. Another common use, such as at CapitalOne [6], is for data migration and processing. These companies aren't adopting serverless for everything like the startups or greenfields; breaking up a legacy monolith system into event-driven microservices small enough for a serverless infrastructure can be complicated, expensive and scary. Like replacing a gas-guzzling car with a hybrid, it could take years to feel the benefits so it's understandable that many companies are sticking with what works.

The future of serverless adoption will be found in the future of software companies, not the present.

Today's tech giants may never go serverless. But the startups of today are the tech giants of tomorrow. It is these companies that will be the digital pioneers of serverless, just as the current wave of tech giants like Netflix and Airbnb pioneered public cloud and microservices.

Tomorrow's tech giants are today's growing startups. It's these companies that will drive demand for new ways to use serverless, and encourage new technology options that don't currently exist. They will generate the knowledge, expertise and best practices that help other companies to develop and operate serverless applications and to overcome any technical challenges. They will remove the roadblocks that currently prevent pure serverless adoption and break down the artificial boundaries inside companies, merging business financial decisions and tech design choices into one.

These companies will expect their partners and vendors to have the same agile, scalable, low-touch approach. The desire for familiarity and flexibility, combined with their purchasing power, will drive serverless adoption further into the ecosystem that exists around them.

The developers of tomorrow's tech giants will have the training, experience, and mindset of designing "serverless first". It's these software engineers who'll go on to found the next generation of startups. Many of them will never have known a world in which serverless wasn't the preferred design choice, and will be more familiar with serverless than any other architecture. They will have entered the promised land of serverless and they won't want to go back.

Erez Berkner is the CEO & co-founder of Lumigo, a startup focusing on simplifying serverless applications troubleshooting. Prior to founding Lumigo, Erez was the R&D director of cloud products at Check Point, heading the company's cloud strategy & execution.

Session: Practical Auth in a Serverless World

Andreas Grimm (Akelius GmbH)

It became essential for businesses to protect their applications, services and customer data from attackers. If you want to stay competitive, knowing how to efficiently and easily apply security and auth while being aware of the most common pitfalls is key in today's serverless world.
Traditional machine-to-machine auth approaches where you can rely on a stateful environment fall short in a modern serverless and thus stateless world. After a short recap of some auth fundamentals, you'll learn how to efficiently apply authentication to Azure Functions without compromising security – using an external Identity Provider like "Auth0", "OAuth 2", "JWT", the secrets management system "Azure Key Vault", "Azure Managed Identities" and "Cloudflare Workers".

References

[1] https://sqquid.com

[2] https://vacationtracker.io

[3] https://www.zdnet.com/article/fender-turns-cloud-adoption-up-to-11/

[4] https://read.acloud.guru/serverless-superheroes-ben-kehoe-and-the-roomba-are-vacuuming-up-servers-36bf308670d9

[5] https://www.youtube.com/watch?v=iz90fHia-Wk&feature=youtu.be&t=530

[6] https://www.capitalone.com/tech/cloud/a-serverless-and-go-journey/

Sixteen Serverless Experts weigh in on this year's Number 1 Hype

Serverless Expert Check 2019

„Serverless" is undoubtedly one of the most hyped topics this year. Many developers are experimenting with the possibilities offered by this rather new programming paradigm, and many companies have already jumped on the serverless bandwagon. Startups seem to be especially interested. However, as more people get familiar with it, different ideas about what exactly serverless means are starting to form. We wanted to know more about what's behind the trend, so we confronted 16 experts with a series of questions. In our big Serverless Expert Check 2019, professionals from all over the world discuss how serverless is changing the way developers, operators, and administrators work, whether serverless will soon make container technology obsolete, and what the future holds for this trending topic. Our experts also discussed in what context a serverless approach could be financially worthwhile.

1. Serverless as a term is a rather controversial buzzword – servers are still used. In addition, everyone seems to have a different understanding of serverless – for example FaaS or BaaS. So, first of all: What is serverless to you personally?

Marlon Alagoda: I understand the term serverless to mean "Function as a Service". This is the service provided by Google, Amazon, and Microsoft under the names Google Cloud Functions, Lambda, and Azure Functions. Although these companies operate large data centers with servers, they can make the best possible use of the available computing capacity with this technology.

Kenny Baas: For me personally, serverless is about me not managing the server. I don't want to configure or even be aware that there is a server. I know there is one, but it is not my job to take care of it. Put simply, I just have business logic I create, and it needs to run whenever I want it to. Also, connecting to other systems needs to be easy; I don't want to have to deal with any server configuration like IP addresses, etc.

Rajkumar Balakrishnan: That's a great question. It's just the term "serverless" that is overrated, not the idea behind it. Come on, everyone knows that there are big fat servers behind these services. But wait, have you considered a wireless router or a cordless phone? There are wires and cords behind those instruments, but we agree they are wireLESS and cordless devices. Buzzwords tend to evolve from views and viewpoints. The view serverless sounds about right from the viewpoint of a developer. The developer just needs to care about running a piece of code per se, and not about servers, network, OS, memory, processor, and so on. The serverless platform allows you to operate at a higher level of abstraction. When a particular technology platform (compute, data, storage, etc.) matures right to the level where its utilization does not concern on what and how it should run, then chances are that the platform is serverless. And

like you asked, it can be Functions as a Service (FaaS), Backend as a Service (BaaS), Messaging as a Service, Containers as a Service, and so on. If we look back over the years of computing, we worked our way up from transistors, logic gates, processors, and assembly languages until today where programming a system almost feels natural, and serverless is just another evolution of computing. If you ask me what it means personally, I would say serverless is very similar to playing with Lego blocks, you have all the pieces at your disposal. You just focus on what you want to build and that's about it. For the rest, it just works.

Christian Bannes: For me, serverless means managed services with a „pay-as-you-go" pricing model. Serverless eliminates the need for server provisioning, scaling, or management.

Nikhil Barthwal: To me personally, serverless is an abstraction layer that frees me from worrying about underlying infrastructure. When I look at things from that point of view – FaaS, BaaS, Knative, all fit under the definition of serverless. Of course, there is serverless as a concept and then there is the implementation of that concept. When it comes to implementation, the platform you use to build your applications on can be vendor dependent like AWS Lambda, Azure Functions, Google Cloud Functions; or it could be technology dependent like CloudRun/Knative.

Erez Berkner: The essence of serverless is in handing over the responsibility – and control – of the server, OS, app engine, and scalability maintenance to someone else (the cloud provider). Serverless also entails the concepts of event-driven architecture (your system is reacting to events) and pay-for-consumption (or "don't pay for idle").

That's the technical definition, but to me, serverless is more than the sum of its parts. It's a new software development paradigm, a new engineering mindset, a movement that strives to eliminate maintenance and resource waste and move faster by reusing existing building blocks, avoiding building components that were already built thousands of times around the world.

Azure-Workshop: **Serverless Infrastructure as Code**

Mikhail Shilkov (Freelancer)

Serverless applications utilize multiple cloud services and consist of many moving parts, so they are hard to manage without employing Infrastructure as Code (IaC) approach. During the workshop we will go through three IaC tools: Azure ARM Templates, Terraform, and Pulumi. We will learn the pros and cons of different approaches and see how coding and infrastructure can be blended together in the serverless world. The workshop content uses Azure cloud but the concepts are mostly applicable to AWS and GCP.

Michael Dowden: From FaaS to BaaS to 3rd Party APIs, serverless is comprised of services that run on infrastructure we don't have to manage. Additionally, with serverless what you pay scales with what you use, without any major capital expenditures.

Erwin van Eyk: Serverless indeed remains a controversial term to use. On the one hand, many people argue for various restrictive definitions to comprise a specific cloud model (such as FaaS) or very specific constraints (such as "it is only serverless when it is managed by a cloud provider"). On the other hand, others try to stretch the definition of "serverless" to ensure that the buzzword also applies to a certain service, product, or application.

Personally, I don't think an exact definition is either desirable or achievable. The emergence and subsequent controversy behind serverless feels oddly reminiscent of the early days of the previous buzzword "cloud computing". Back when AWS and other key players popularized that buzzword, there was a lot of controversy around its use. What exactly is cloud computing? When is a service a cloud service, and when is it not? If you look back, various strict definitions were proposed, but we never settled on a clearly delineated definition. Take, for instance, the current cloud computing definition by Merriam-Webster: "The practice of storing regularly used computer data on multiple servers that can be accessed through the Internet" – if that is not a broad, vague definition, I don't know what is.

This is not to say that we don't need a bit more structure in the definition of what serverless constitutes. To introduce a bit more structured definition, I, together with an international team of researchers and industry professionals, proposed a definition of serverless computing based on three key characteristics:

Granular billing: The service only bills the user for actual resources used to execute business logic. For example, a traditional VM does not have this characteristic because users are billed hourly for resources regardless of whether they are used.

Minimal operational logic: Operational logic, such as resource management, provisioning, and autoscaling, should be delegated as much as possible to the cloud provider.

Event driven: User applications (whether they are functions, queries, or containers) should only be active/deployed when they are needed; when an event triggers the need for it.

With this definition, FaaS, BaaS, and even some SaaS services are part of the serverless computing domain. If an application, product or service adheres to these three characteristics, call it serverless.

Vadym Kazulkin: As I see it, serverless is mainly about not having to take care of the infrastructure, letting the app scale automatically and not having to pay for the application's idle time.

Niko Köbler: Serverless is just a name. Admittedly, not a very well-chosen name, but let's get along with it and stop trying this „of course there are still servers in use". I think everyone has understood that at this point.

For me, serverless is first and foremost about the fact that I no longer have to worry about maintaining the infrastructure and can use it easily. Availability, scalability, security, and up-to-dateness of the infrastructure are managed for me, made available to me within certain limits, and I can use this time to bring my software to market better, faster, and more flexibly.

João Rosa: In my opinion, the serverless model provides a way to gather fast feedback. It can provide a short feedback cycle from idea to production, given that the typical hardware/OS/middleware software concerns are not present. It can be presented with different flavors (FaaS, BaaS, etc.), but the aim is the same.

Soenke Ruempler: To me, serverless means "servicefull" first and foremost. This means that modern applications are characterized by the fact that there is as little written (and operated) code as possible, and that managed services are utilized as much as possible.

Kamesh Sampath: Serverless is style of architecture where your application responds on a need basis. I personally like to call this "serve on-demand". Serverless also means that an organization adopting this style can focus on business differentiation rather than worrying about physical or virtual server management.

Ory Segal: I'm actually quite fond of the term "serverless" and find the entire discussion around it amusing. I think it's as good as you can get when attempting to describe a software architecture with one word. I'm not sure why people take it so seriously; it's not as if "cloud" was such a great term, or the "World Wide Web" for that matter. For me, serverless epitomizes what cloud computing is all about – a new architecture to build and deploy software applications, without really caring about the nitty-gritty details of the underlying infrastructure. You concentrate on your core business logic and disregard anything else that isn't relevant.

Mikhail Shilkov: I feel that the discussion has finally matured past the point of "there are servers in serverless" remark. Serverless is a collection of values and technologies with the following characteristics:

- High developer productivity: low ceremony, no plumbing code, no management of servers, rapid value delivery cycle
- Auto-scaling and high availability out of the box
- Pay per value: pay nothing for idle, pay proportional to usage while growing

The ultimate goal is to enable developers to write domain-specific code and let the cloud provider handle all the heavy lifting.

Harald Uebele: Serverless to me is a FaaS platform like OpenWhisk. Serverless is also everything I need for my application that I don't want to run myself, so my cloud provider makes it available to me. This includes object storage ("S3"), database as a service, OAuth services, DevOps pipelines, etc.

2. From a developer's point of view, serverless has many advantages, one of which is that you practically don't have to worry about the infrastructure anymore. In your opinion, how does serverless change the everyday life of developers?

Marlon Alagoda: The available tools for developing serverless applications are still in their infancy, as are the use cases in which the technology is utilized.

The lack of tools makes development a bit tedious at the moment. It feels more like playing around. For many tasks, like automatic testing and deployment, there are no established ways yet. As a developer, you often have to be very creative.

If the technology continues to prevail, the community grows, and if there are more established tools, then I assume that a part of the complexity will be transferred to the serverless provider, and that developers will be able to build new functionality faster.

Kenny Baas: Simple and easy answer: It creates faster feedback and more stateless environments, so I can create more quickly. This makes automated testing, which is currently the bottleneck almost everywhere in a lot of organizations, easier, faster, and more reliable.

Rajkumar Balakrishnan: I believe serverless platform influences the mindset about "How we write code". You don't name a function as one and implement two. You do one thing and do it right. Serverless as a computing platform helps developers write code with a singular purpose. The focus shifts from writing big chunks of classes to simpler functions. In principle, if this all adds up right then we should be delivering greater business value in doing what we are doing.

Christian Bannes: First of all, some of the work that would previously be carried out by a separate operations team is no longer necessary. This of course also eliminates the need for coordination between these teams. But it is also necessary to commission the infrastructure for serverless applications. It no longer makes sense to organize provisioning and development in different teams. Even though automation is already part of most developers' daily work, serverless makes it a must.

In application development, the boundaries between back-end and front-end development are becoming increasingly blurred, enabling end-to-end work and the higher development speed that comes with it.

Nikhil Barthwal: It does make developers' lives significantly easier. However, bear in mind that serverless is a relatively new technology and quite a lot of applications exist today that use containers, virtual machines, or even bare metal machines running in the public/private cloud, and they will continue to exist for a long time. Also, for developing new applications, there are many scenarios where serverless architecture might not be the best fit. So, I don't see imminent change in developers' lives, and we have to continue to deal with "servers" in some form or another for a while.

Erez Berkner: Serverless speeds up development to an unprecedented pace. For the first time, developers are free to focus on building their application and not on maintaining it. From our experience at Lumigo, this results in a pace of development that's more than three

times faster, and which allows us to deliver product features to market much more quickly.

Michael Dowden: We can focus on building our apps, not deploying and scaling them. It greatly decreases the time to market for new apps and allows us to match revenue to costs. Serverless also brings web-scale capability within reach of even small software teams.

Erwin van Eyk: There seems to be a lot of talk about how serverless will radically change the lives of developers. However, serverless computing is simply the next step in the perpetual trend in cloud computing. We are continuously trying to move to higher levels of abstractions with our applications, bringing programming closer to the business domain and away from the operational details.

With serverless, we simply continue this trend of higher levels of abstraction. Cloud computing provided us with an abstraction layer over bare-metal machines. Serverless computing takes the next step, abstracting away virtual machine logic for developers.

Vadym Kazulkin: Since many Ops tasks (infrastructure maintenance, including platform security and scalability) are offered by the provider and you get base logging and monitoring on top, you can concentrate on the business logic instead and achieve more in less time with fewer colleagues. Focusing on the essentials is key.

Niko Köbler: Developers shouldn't have to worry about the infrastructure anyway. They didn't in the past and they shouldn't have to now either if you ask me, even with this artificial word "DevOps". Developers still focuses on creating the software themselves. However, with the serverless model, the structure of the software changes in such a way that there are more components, and these are operated clearly loosely coupled from each other. One of the challenges at the moment (and probably in the future) is staying in control. In other words, without reasonable documentation and administration of the operated components, nothing will work. Those who already have difficulties keeping track of microservices will not have an easier time with serverless.

João Rosa: The serverless model has an impact on developers' lives, as well as their teams and organizations. Serverless enables capabilities that weren't there before. However, there are new challenges that every development team needs to be aware of, given they are responsible for the entire lifecycle, e.g., security, scalability, stability, etc. The new challenges require new practices and skills because the "old" developer stereotype will fade with these new responsibilities.

Soenke Ruempler: That depends on where the developers come from. If they are used to running their own applications ("You built it, you run it"), they will be happy if more and more operational burdens are taken from them, allowing them to concentrate on their core software. If the Dev teams had their software run by an Ops team in the past, serverless can also be negatively received because now more responsibility goes to the Dev team. The DevOps movement has already seen the development of Dev teams taking on more responsibility – voluntarily or involuntarily. However, with serverless, it is strengthened once more. To me, serverless is the logical evolution of DevOps.

Kamesh Sampath: In my opinion, serverless provides lots of advantages to developers. As you said, the primary one being not having to worry about the infrastructure anymore, i.e. no touching of physical/virtual machines. The second one being able to build single-behavior applications. With microservices, developers started to build applications that have "single responsibility", serverless goes one level deeper in building "single behavior" – which means developers become more productive.

Ory Segal: Until serverless came along, developers had to be very skilled in so many different domains other than those related to the core software they were developing, and in addition, they had to re-invent the wheel every single time they needed to interact with back-end services. Public cloud-serverless really hits the point on both issues – developers can now concentrate on what they actually need to build and forget about the underlying stack. Additionally, they get to enjoy the benefits and simplicity of public cloud services. Software development becomes more like building with Lego blocks. It's simple, but you can create wonderful things with it.

Mikhail Shilkov: The changes are numerous. Writing less boilerplate code and get into production faster. Being able to see the impact of performance optimization on the monthly bill. Constantly automating the routine. Spinning up a new environment to run the tests and then tear it down. Learning a new cloud service or feature every week. Drawing numerous diagrams of how small serverless components fit together. Getting to live in the messy world of distributed systems. Loving YAML, hating YAML. The overarching theme is that small teams can write impactful code. That's powerful.

Harald Uebele: I'm becoming much more productive because I don't have to worry about procurement anymore. I click my test environment together and it's available within a few minutes. In contrast, I may have to wait several weeks for classic infrastructure. For example, if I have a bug report that refers to an older version of my code, I can quickly (let it) assemble this environment, and I don't have to have it available at all times.

3. Developers aren't the only ones affected by the new model, especially when you think about DevOps: What are the consequences of the serverless approach for Operators/System Administrators?

Marlon Alagoda: Many of the classic operator and admin tasks would be omitted. The serverless provider ensures that the application is always accessible and that neither too little nor too much computing and storage resources are available. The providers also provide tools for tracing, debugging, and audits, which suddenly nobody needs to operate anymore. Serverless applications, in my opinion, help to make it even easier to implement a DevOps approach in which "You build it, you ship it, you run it" applies, and to make pure operators and administrators obsolete.

Kenny Baas: Ops is still very much needed because we still configure infrastructure. It will get easier as more patterns are created, but the configuration is still a skill for which Ops is needed. In my opinion Ops also needs to evolve more towards engineer/developer skills.

Rajkumar Balakrishnan: In the context of SRE, running every product or service has a percentage of toil associated. Toil is usually repetitive manual work that has no significant value in the long term. A service goes down and there's some manual cogs and wheels to be turned to bring it back again. This can be true for both developers and operators. Adapting to serverless creates space to improve the service's reliability. Monitoring and operations and can shift to metrics that are more focused on customer value than just the amount of memory and CPU being consumed. Thus, I see a potential improvement in reduction of toil and improved reliability in running services.

Christian Bannes: Many tasks such as scaling and high availability, which were previously very complicated, are practically eliminated with serverless. Traditional operations topics such as monitoring remain and will become even more important. Serverless applications are always distributed, event-based applications. Without proper monitoring, log aggregation, and tracing, you will quickly run into problems. But since serverless will not completely replace VMs and containers in the foreseeable future, we will need the expertise of our admins for a long time to come.

Nikhil Barthwal: DevOps is about releasing software faster and more reliability. It has a lot to do with collaboration between developers and operators (hence the term DevOps). I don't see the fundamental principles of DevOps changing because of serverless.

Serverless would surely change the way DevOps principles are applied by using an alternate set of tools & processes. And while you don't have servers to manage in serverless, you still need to deploy code (whichever way you package it) in a reliable and fast way, monitor it, and take action in the event of the unexpected or undesirable.

Erez Berkner: Serverless will finally bring down the wall separating Dev and Ops teams. You simply cannot be a good serverless developer without understanding how to operate, configure, and monitor serverless environments – usually as code. The good news is that there are far fewer of those kinds of operational tasks. Serverless forces Dev and Ops to merge. It does not lead us to a stronger DevOps position, but rather to a more versatile core developer who can also operate his serverless-developed services.

Michael Dowden: The DevOps role will start focusing a lot more on supporting developers (continuous integration, IAM, versioning, etc.), and on administrative support (system monitoring, performance and security testing, etc.).

Erwin van Eyk: Serverless is unlikely to completely remove the need for system administrators or operators. Instead, it will promote more specialization in DevOps. The separation between business logic and operational logic in serverless means that DevOps engineers can specialize in the maintenance and improvement of the common operational layers. On the user side, engineers are needed to monitor, albeit at a higher level, the serverless functions and other services.

Vadym Kazulkin: Serverless allows developers to take care of all aspects of software development (and that also means infrastructure as code) themselves. Thus, the team can work on tasks end to end until release and operation, and they are faster due to less external dependencies. It also means developers must (want to) learn lots of new things. This also applies to typical system admins. Additionally, there are many challenges around serverless concerning performance, scalability, and restrictions, depending on the service, which must be understood in order to make decisions. Just a few examples: When do I use Kinesis, when should I use SQS, SNS, or even DynamoDB Streams instead? When do I call one Lambda from another Lambda, and when is it advisable to place an application load balancer, API gateway or SQS in the middle? Promoting extensive monitoring, alerting, chaos engineering and game days plays a part as well. This is precisely where I believe Ops' responsibility lies.

Niko Köbler: Here we have to draw the line between operations and administration. We should always do this. (System) admins manage the infrastructure and make sure that it works. The people from operations take care of the operation of the applications so that they are error-free and performing correctly. In many companies this is done in personal union, which is not always goal-oriented. That's what DevOps describes – you develop software and have to make sure that it is operated correctly and continuously improved. But that doesn't mean that a developer has to know everything

Session: Playing with Fire : Build a Web Application with Firebase

Michael Dowden (Andromeda), Michael McGinnis (FlexePark)


Come see how easy it can be to use Google Firebase to take your app idea from concept to production. In this workshop you will build your own messaging application, start to finish, with support for images, markdown, and connecting with friends. While building this web app will learn about many Firebase features, including:

- Firestore
- Cloud Functions
- Cloud Storage
- Hosting
- Authentication
- Security Rules
- Client and Admin SDK

Code will be in JavaScript & Node.js from a git repository, so please be prepared to start coding. You will also need a Google account to sign in to Firebase.

about the operation side and vice versa that an Ops person has to know everything about software development. People just have to talk to each other – THAT's what we need.

The serverless generation, on the other hand, no longer speaks of DevOps – it simply lives the principle and the culture without defining itself by a term. According to the motto "we build it – we are responsible for it!" DevOps is not superfluous here at all, it is simply lived without talking about it, because everyone is aware that it is not possible without it. And nobody talks about any DevOps „tools".

João Rosa: I'm a firm believer in breaking silos. Even when we break down the silos (and by silos, I mean the development department, the business department, the operations department), we still need specialists in some tasks (the famous T-shape profile in the DevOps movement). That being said, I believe the roles will transform (like the previously mentioned developer role), towards a cross-functional team that can build, operate, and evolve a system.

Soenke Ruempler: Like developers, Ops people have to adjust to the fact that there are fewer low level tasks. But in return there is more time for (in my opinion) more interesting and value-creating topics other than babysitting servers, VMs, and containers: Automation of the infrastructure (e.g. "Compliance as Code"), recognizing, collecting, and distributing best practices, providing central tooling, etc.

Kamesh Sampath: Serverless is not just for developers, but for Ops as well. To quote the CNCF blog [1], "serverless does not require server management" – this is a big boon for Ops, as it makes their life easier by just deploying and maintaining the right serverless platform. Serverless platforms are built to allow scaling on demand, which eliminates another typical concern for the Ops team – the scalability. The above two points culminate to a better resource and cost optimization, solving another big pain point for Ops. A correctly configured serverless platform can help Ops build an API-driven, quota-based self-service that can serve both the business users as well as their developer users.

Ory Segal: The main change for operators and system administrators is again related to the fact that the issues you have to deal with are now at a much higher level. Less emphasis on operating systems, networking, capacity planning and scalability, and more about how to configure services, monitor them, and make the deployment pipeline more efficient. From a security perspective, much of the tedious work around patching systems is gone – which is a huge benefit.

Mikhail Shilkov: There isn't much system administration to be done anymore. However, the tasks of operational excellence still apply. Serverless applications consist of many ever-changing parts, so monitoring, observability, and smooth rollouts are more useful than ever. "Automate all the things" means that former IT people should get into the developer's mindset too.

Harald Uebele: I don't think that there will be much of a change for the operators of the platforms – as you wrote at the beginning, servers are still in use. For the Ops of my application, however, a lot becomes simpler. The environments do not have to be procured and set up at full scale; they are available within a few minutes – ideally at the push of a button.

4. Simon Wardley has put forward the thesis that containers and Kubernetes are only a marginal phenomenon in the history of software development and could soon become obsolete because "serverless is eating the world." What do you think about that?

Marlon Alagoda: Well, there are also container technologies behind AWS Lambda. Even if in the future we look back at the time when we ourselves still operated Kubernetes clusters and laugh, this technology, or the further development of this technology, will continue to be used to be able to offer serverless services. Today, hardly anyone puts a server in the basement anymore and yet there are servers somewhere. Serverless is another abstraction.

What Simon Wardley means to say in the mentioned interview with Forrest Brazeal, is that as classic software developers we won't come into contact with container technologies if serverless continues to prevail, which I can imagine might well be the case.

Kenny Baas: I hope to skip the whole Kubernetes hype. It is too technically complex. I like to focus on business complexity, not Kubernetes complexity.

Rajkumar Balakrishnan: Actually, I don't know. But if we look at the larger picture, every piece of tech that we use tends to co-exist with another competing tech stack until one of them becomes irrelevant and obsolete. At times people simply don't want to adopt despite the stack being very good, so it's kind of hard to say. I hear more stories about how friends in other companies are building platforms based on container stacks and in the end, it's a matter of choice. I think these stacks will continue to co-exist. Whether or not one will eat the other, time has to say.

Christian Bannes: The trend is clearly moving towards managed services. With serverless, companies can concentrate on the development of the core domain instead of investing in the operation of a Kubernetes cluster. However, I don't believe that containers and Kubernetes will disappear so soon. There are simply too many existing applications that can only be transferred to a serverless model with great effort. Even new applications with special requirements cannot always be implemented as serverless applications, e.g. machine learning applications that require access to the GPU.

Nikhil Barthwal: The meaning of this statement depends a lot on how you interpret it. I would partially agree with this statement, when interpreting it in the broader sense.

There are two dominant trends in in the industry. The first is the trend towards distributed functionality. We are moving from monolithic services to microservices to functions now. The second trend is raising the level of abstractions. We moved from bare metal servers to virtual machines, and then to Docker/containers, and with serverless, towards individual functions.

In that context, both containers & Kubernetes are both passing trends as we move towards more granular functionality and higher level of abstractions. But I would be cautious because not every application will fit into a serverless application model and there will be corner/edge cases where we have to resort to "old" methods that might involve use or containers/Kubernetes.

Erez Berkner: Simon is a forward thinker and is usually looking 10 to 20 years into the future. In that timeframe, the concept of serverless and moving the responsibility for commodity services to someone else makes perfect sense (just look at the adoption of public cloud as a reference to this prediction). However, if we look at the next decade, having containers in parallel to serverless will be a safe haven for many organizations that will find it difficult to abandon the familiar and adopt the new serverless mindset.

But even today we see many organizations that share Simon's vision of the future, and leapfrog from VMs (or even on-premise serverless) directly into serverless, skipping containers in the process.

Michael Dowden: While it's entirely possible that containers and Kubernetes will power some of the serverless platforms of the future, they will have only a passing relevance to application developers. The concept that will remain, however, is the bundling of operational and environmental requirements with the software that will be running.

Erwin van Eyk: While serverless computing is the fastest growing part within the cloud ecosystem and its promise is clear, it is too extreme to state that it will make all other cloud models obsolete. Use cases will remain that require the use of bare-metal machines rather than serverless services. However, existing cloud models and technologies are moving and will move further towards serverless. For example, many projects, such as service meshes (e.g., Istio (https://istio.io/), Linkerd (https://linkerd.io/), etc.) and FaaS platforms (e.g., Knative (https://cloud.google.com/knative/) and the open source Fission), have emerged to further abstract away operational details of containers and Kubernetes. Instead of becoming obsolete, Kubernetes holds the nice promise of a unified ecosystem in which serverless/FaaS services coexist with more traditionally deployed services.

Vadym Kazulkin: Concerning the cloud world, I think Simon Wardley is right. The question is only when this will happen. There are still enough restrictions like execution time, memory consumption, number of parallel executions, etc. But in time, they will weaken or disappear completely. AWS, for instance, is already taking steps towards the future with Lambda Layers and Lambda Runtime API. Hybrid technologies like AWS Fargate, though, will play a major role. As they are based on containers, developers don't come into contact with them aside from creating the Docker file. Those who, for whatever reason, remain in the data center or must pursue a multi-cloud strategy, will not be able to do so without containers and orchestration technologies. With TriggerMesh, some very interesting technologies are entering the market.

Niko Köbler: I wholeheartedly agree with that. The only thing I have to add is that the phrase "...is eating the world" is a little bit overused by now. Yes, I am also guilty of misusing it for an article a year ago, but I regretted it instantly. Nothing is "eating" anything.

João Rosa: It's an interesting remark, and I observe the companies are using containers as a vehicle (1) for the lift-and-shift process to the cloud, where they realized that they could use the same tech stack on-premises as well in the public cloud; (2) I also observe several companies stating the usage of containers is the exit strategy of the chosen cloud provider, given the interoperability (K8s case). The observable trend is that once the software is in the cloud, teams tend to explore alternatives, challenging the status quo. The sensible alternative is to use cloud-native services, and serverless is one of the services that allows focusing on the business outcomes rather than the underlying infrastructure.

Soenke Ruempler: Containers and Kubernetes will not become obsolete, but for most people they will become invisible sub-systems. Compute and FaaS are just a small part in event-based (serverless) applications. The largest parts are building blocks provided by the cloud provider, for e.g. data storage, log-in/identity, queues, event buses, etc.

Kamesh Sampath: I think the other way; Kubernetes and containers have become bigger players in people choosing to go with serverless, especially if you consider that Kubernetes provides two important serverless features such as build (Knative Build) and scaling (Knative Serving) out of the box. The Kubernetes ecosystem has even gone a step further in adding eventing (Knative Eventing), another common style of applications that is used in serverless world, though this style is still maturing. You can also think of Kubernetes as another layer of abstraction for Ops, i.e. as an "application orchestrator".

Ory Segal: I'm in total agreement with Simon. Containers provided temporary technological relief to an Ops/IT problem. They allowed software stacks to be packaged in a slim, lightweight packaging that is easy to replicate, deploy, and teardown. Serverless is not about how you package a software stack – it's about leaving the stack for someone else to manage, and only building, packaging, and deploying applications. It might be that serverless will always run on top of containers, micro VMs, or whatever comes next. The point is that developers will not care anymore, just as they don't care about CPU internals or memory chips.

Mikhail Shilkov: Each business should focus on their unique expertise and competitive edge. If you are not in the infrastructure business, why spend precious hours on building Kubernetes clusters? However, our industry doesn't have a great track record of choosing the best solutions reliably enough. Some organizations might get stuck with Kubernetes for long years to come, which means others will do the impactful bits.

Harald Uebele: Probably something much better than serverless will come along first. Perhaps calculating bacteria…?

5. Serverless is all about scalability and the associated costs – is it still worth using your servers today, or is the price/performance ratio of serverless unbeatable?

Marlon Alagoda: There are use cases for which your own servers are worthwhile and use cases for which serverless is more profitable. Serverless is particularly interesting in monetary terms if you have to be able to manage high workload peaks.

For example, if you need to run pattern recognition once a day on an image that is several gigabytes in size, then renting a server is guaranteed to be more expensive than performing the operation using FaaS, even if you automate your own server and rent it for only a few minutes. Either serverless or your own server will be cheaper in the long run for classic use cases depending on how many hours you really save in the operations area, and as far as that goes, only time will tell.

Kenny Baas: This is really complex. Depending on your transactions per second, serverless can cost you more than containers.

Rajkumar Balakrishnan: An enterprise is backed by different kinds of software products costing from hundreds of thousands to millions of euros. Not every product matures at the same level and every product requires a different execution environment, be it on-premise or cloud-based. And the way some products are built today it cannot be billed on a serverless model. All that it means is we can't just do away with our servers yet. In most cases price is not the only determining factor for driving a project forward. However, new development projects can factor in serverless platforms for building reliable services amongst other stacks.

Christian Bannes: Serverless is not profitable for all scenarios. This is mainly due to the fact that the computing time on a pay-as-you-go (cloud) server is always more expensive than on your own servers or on a VM with the same performance in the cloud. Thus, where there's a constantly high workload, having a server or VM can still be worthwhile.

Nikhil Barthwal: I am not really a believer in one-size-fits all. Sure, serverless has good scalability and price/performance characteristics, but it does also have its downsides. There are problems like cold start that would make it difficult to implement always-on services in serverless (though this is likely to improve in the future). Moreover, since you don't get to manage servers, you cannot tune servers for certain performance characteristics if you need to. Furthermore, serverless can result in vendor lock-in (or in the case of Knative, lock-in to Kubernetes). Serverless thus in some sense also restricts you, which might not be desirable in certain cases.

Erez Berkner: When your workload is not steady (i.e. with spikes), or you utilize less than 30% of your server CPU, serverless pricing is unbeatable.

When you have higher, more predictable CPU utilization it becomes a more interesting debate. But if you factor in the manpower and time saved by serverless it becomes very hard to beat it on pricing. That is, assuming your team is knowledgeable about serverless and is implementing the right architecture design patterns with the right monitoring and alerting tools in place.

Michael Dowden: Obviously the companies providing serverless platforms (Google, Microsoft, Amazon, etc.) all run physical hardware. There's still a value to them. And for some time, we'll see companies opting for managing their own virtual infrastructure (probably in the cloud, on one of these major providers) because they have significant resources and the need to fine-tune things that aren't available in serverless – yet. At some point I see functions being managed by machine-learning controlled systems that correctly identify if a process is I/O, memory, or CPU-limited, allocating resources as necessary. I believe the functions of the future will also select the region to run in based upon current demand.

Erwin van Eyk: The price/performance ratio of serverless platforms is certainly not unbeatable. The internet is full of one-to-one comparisons between serverless and non-serverless alternatives. For example, if you compare the pricing of AWS Lambda to deploying the same service on an EC2 instance, you will find that as you increase the number of function executions the cost of the FaaS function quickly exceeds that of renting a VM.

Yet, even at such discounts one of the key costs remains: operating costs. Deploying a couple of VMs to run your application is only the start. After deployment, you will have to monitor your application; keep the machine, middleware and application up to date; scale the number of VMs up and down according to demand (or monitor the autoscaling policies that do so for you). This maintenance and support will consume the costly time of already scarcely available DevOps engineers.

Still, there are cases in which it is more cost-effective to deploy these applications in traditional close-to-metal cloud models. When the workload is large, relatively stable, or where performance is critical, it can be cheaper and safer to have on-premise dedicated resources. But, in most cases, the microservices that we write are not that performance-critical, nor do they have a consistent, large workload. In such cases, we can save costs by deferring the operations to a cloud provider. The cloud provider (which can simply be an internal operations team, running a self-managed FaaS platform, such as the open source Fission) can benefit from economies of scale, multiplexing the serverless applications on fewer machines, while providing the applications with extensive autoscaling.

Vadym Kazulkin: Serverless is not yet fully developed for all use cases, e.g. in the ML/AI area. In the field of Big Data, there is also some catching up to do, although it's possible to implement a lot in AWS through Kinesis. In some scenarios, like on gaming or bidding platforms, you constantly require a very high level of performance. So-called "cold starts" in the serverless world partly restrict its use.

Niko Köbler: These tiresome costs... „Oh, it costs something, well that's stupid..." This question must not only be seen and answered in the context of serverless, but must be extended to the entire cloud. For me, the cloud is not a cost-saving model. Using the cloud is more expensi-

ve per se than running something yourself. But if you take the aspects into account that make the cloud interesting, namely availability, scalability, managed services, security, etc., then it will quickly turn around. In other words, if you only go into the cloud and do the typical „lift and shift", i.e. operate your own (virtualized) infrastructure without benefiting from the services because you are afraid of this pseudo-ominous „vendor-lock", then you will end up paying more and more because you won't get rid of administration costs. On the other hand, those who use the cloud correctly and do not build and operate an abstraction layer (aka Kubernetes et al.) themselves will benefit from the cloud. Not in terms of absolute costs, but on the revenue side. That is because with the possibilities of the cloud, the company can focus on its core business and generate this so-called „business value" by being able to react more quickly and flexibly to changes in the market and its requirements.

João Rosa: As the industry shows us, it depends on the use case. As mentioned before, the serverless model is one of the most effective ways to experiment and where development teams can quickly validate assumptions.

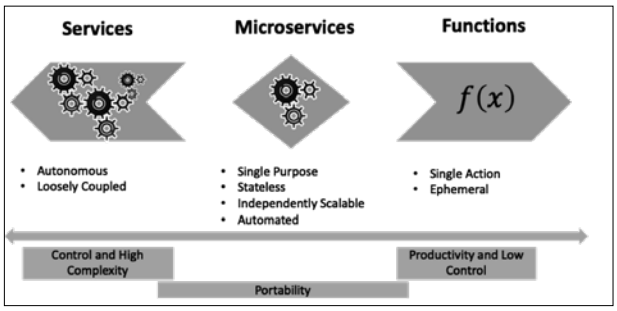
Another great use case for serverless is an event-driven architecture as opposed to a data-driven architecture. An event-driven architecture will promote loosely coupled software and teams, and serverless is a great example of one of the implementation models. It has an effect on the cost. On the other hand, as the Dropbox case shown has, there are a combination of dimensions where cloud services are more expensive than the traditional datacenter. My advice is based in the 3X model from Kent Beck, where we make decisions based on the current contexts of the idea/software, and are always ready to challenge the previous decisions if the context changes.

Soenke Ruempler: Opinions differ widely in this regard. If one compares pure "hardware" costs to cloud provider costs, serverless can seem very expensive from a certain scaling point. However, if you take into account what serverless can do for you – automatic scaling, HA built-in, constant further development of services and components, automatic patching of runtimes, etc. – then it already looks better. This sometimes "replaces" an entire Ops team, which is not exactly cost-effective either. I think that it is also a question of mindset.

Kamesh Sampath: Serverless gives you better "resource and cost optimization" compared with serverfull-based architectures. The fallacy with the information technology industry is that when something new is introduced, we tend to pick up new architecture without doing the need due diligence, i.e. without knowing if the new style will fit the use case or problem we are trying to solve.

I feel serverless can be used independently but that doesn't mean we have to reengineer all applications to new architecture. To get better results I feel we need to mix the architectures.

Ory Segal: I don't think that the current pricing model, and the way people compare it, portrays reality. First, cloud-compute prices will go down as adoption and competition increase. Moreover, there are also a lot of operational expenses that are slashed when adopting serverless



– e.g. the reduction in long-term IT costs, reduction in the prices of backend software licenses, time to market, etc.

Mikhail Shilkov: This topic is very dear to my heart, and some issues like cold starts are yet to be solved. Nonetheless, serverless is in a position to be a sensible default target for most common use cases. From what I've seen, it's unlikely that FaaS will be the most expensive component in your LoB application. On top of that, your cost gains will come from saving resources on infrastructure planning and management, high availability and elasticity, razor focus of developers on business problems, and high innovation velocity. Honestly, reducing the size of the engineering team outweighs any infrastructure costs pretty quickly.

Harald Uebele: At the moment, I don't see an ERP, CRM, PPS, or system that is implemented exclusively serverless; there is still enough space for monoliths and everything in between.

6. Finally, a brief look into the crystal ball – What role will serverless play in 2020?

Marlon Alagoda: I very much hope that the technology will make a leap forward, from being something that feels like a "toy" to a matured tool.

The FaaS, which I know of, are already very powerful and mature. What is needed most of all is more tools around the technology, and strong communities that show how to use serverless successfully in an enterprise world, i.e. beyond one's own pet project.

Kenny Baas: It will grow, but we won't be over the Kubernetes hype in 2020 either.

Rajkumar Balakrishnan: Well 2020 is few months away and I believe serverless adoption will continue to increase. Databases, messaging, and many other components of software building will expand to a serverless model across multiple cloud environments, and while the pace is already accelerating today, I think it will accelerate even further. Again, the number of options for choosing the components of software building will increase. This all in a way democratizes the accessibility of technology to small- and medium-scale businesses, making it all affordable. At the end of the day, it's all about delivering incremental business value to customers. Let's look forward to being part of an inclusive community of people doing great things and building a happier present and future.

Christian Bannes: If you look at the history of software development, you can see that the best technology does not always prevail. A significant part of the success of a technology is its attractiveness to devel-

opers. Not everything can always be implemented in serverless applications in the same way as on your own servers. Java, for example, is usually not the first choice because Java VMs are not optimized for serverless applications. Even great frameworks such as Spring Boot, which many developers have become fond of, simply no longer make sense. Developers will have to change. Therefore it will take some time until serverless will prevail in most areas.

And, of course, there are still many applications that play an important role in companies that cannot be so easily transferred to a serverless model, and so will continue to exist for many years to come. We will therefore see serverless more in new projects than anywhere else.

Nikhil Barthwal: I think serverless will play a bigger role, and I definitely see it being more widely adopted. This would in part be due to improvements in technology (like mitigating cold-start issues). But I don't see serverless replacing server/container-based systems any time soon, and I do think the future is hybrid; traditional services (or microservices) coexisting with serverless.

Erez Berkner: The serverless train has left the station. Now it is just a matter of time. Hundreds of thousands of developers are already practicing serverless. Although a lot of them are still in non-critical workloads, a sizable proportion has moved to production already, and many are running serverless at scale.

The direction the train is moving in is clear. Most developers who get a taste of serverless start using it more and more. It is spreading within organizations from the bottom up, virally spreading from one development team to another. It is just a matter of time until this becomes a standard technology in most organizations, and many will adopt a "serverless first" development approach.

Michael Dowden: In 2020 we'll see the first acquisition of a $1 billion startup that runs serverless. We will also see our first major digital attack that is based upon an infiltration of the platform code from one of the major providers of serverless platforms.

Erwin van Eyk: Serverless will play an increasingly important role in how we use and think about distributed cloud applications. Although there are too many challenges and opportunities in serverless computing to describe, I believe that on a user level, we will see the following trends develop further:

1. The tooling around (open-source) serverless platforms will improve to allow for more complex serverless applications. Systems that allow for the composition of serverless functions and services, such as workflow systems, will mature. This will encourage the reuse of existing functions and simplify the implementation of cross-cloud operations.
2. The emergence of edge computing will further amplify the popularity of serverless computing. The on-demand, lightweight, and ephemeral nature of FaaS functions, for example, makes them ideal candidates to serve workloads at the edge.
3. New programming models will emerge that leverage serverless computing to help developers define their distributed applications.

Overall, I believe that the concepts behind serverless computing, despite all the buzzwords, are here to stay, and will continue to trend towards higher-level abstractions for cloud computing.

Vadym Kazulkin: In my opinion, the importance of serverless will continue to increase. Serverless will be used much more and allow for completely new business models (see FinDev).

Niko Köbler: Sorry, my crystal ball is in the repair shop right now. But seriously – serverless will certainly play a bigger role next year than this. It is not only a nice little trend that people are talking about, serverless is an evolutionary step. And evolution goes on and on. Unfortunately, serverless will not be able to make up ground in Germany as fast as elsewhere since it is known that German IT companies and developers are rather conservative and don't get involved in new things that they can't control or that they think they can't control as quickly. Perhaps people might think differently if they just spoke to the cloud providers – they are quite open when it comes to questions and wishes. Nothing ventured, nothing gained.

João Rosa: I believe that more applications will be based on serverless architecture, and in 2020 we will see growth in this regard. I also hope this paradigm will drive more innovation.

Soenke Ruempler: I think the trend is unstoppable because it is an evolution towards "commodity". More people will realize that with serverless you can get there faster and focus more on your business and core problems. The frameworks are maturing and so are the practices.

Kamesh Sampath: I feel serverless will become one of the de-facto cloud architecture styles by 2020 with many organizations starting to adopt them. Projects like Knative [2] that were started in 2018, will mature more in 2019 allowing organizations to build their serverless solutions on the Kubernetes platform.

Ory Segal: I don't own a crystal ball. However, I can definitely share my vision based on my personal belief and the signals I'm seeing and hearing. In 2014–2017 it was still unclear whether serverless would see mass adoption. However, 2018 brought with it very strong signals that serverless will indeed become mainstream. We're constantly meeting and talking to large organizations around the world, from many different industry verticals, that have already started their journey to adopt serverless. It's very exciting.

Mikhail Shilkov: Some architectural best practices will emerge to address the need to build more complex applications out of fine-grained serverless components. Part of this knowledge will come from larger projects reaching production with serverless and getting enough empirical data to share with the world. The other portion will come from new serverless offerings of major cloud providers and start-ups, providing higher-level abstractions and managed building blocks.

Harald Uebele: A bigger role than today… That's the diplomatic way to put it.

THE EXPERTS

After studying visual computing in his hometown Vienna, **Marlon Alagoda** learned to love Node.js while being involved in start-ups and small businesses. Since the beginning of 2017, he has been working on behalf of Senacor Technologies AG, helping German banks and insurance companies to technologically advance into the 21st century. In addition to classic digitization projects, in which legacy systems were replaced by microservices architectures, he was most recently involved in the design and development of a spin-off of a large German bank.

Kenny Baas-Schwegler is a software engineer and consultant focusing on building quality into software delivery at Xebia. He mentors, coaches, and consults teams by using practices, techniques, and tools from Domain-riven Design, Behaviour-driven Development, Test-driven Development, and Continuous Delivery. Through Aikido training, he learned the most efficient way to work together. To get the outcome that all parties want, energy should not be blocked but should be bent and influenced. The philosophy behind this line of reasoning is not only embedded in his personal life, but also in his work life. He is an advocate for multidisciplinary collaboration in open spaces. By using and combining tools such as EventStorming and Example Mapping, he helps engineer requirements to design and model software. With these approaches, he aims to create a transparent, collaborative space with constant and instant feedback when delivering software.

Rajkumar Balakrishnan graduated in Electronics and Communication Engineering from Anna University, India and is presently based in the Netherlands. He is a Certified Microsoft Azure Solution Architect and a speaker at Microsoft conferences. He is currently working with Royal Agrifirm Group B.V, where he is responsible for Software Development and Integration Architecture focused on Digital Agricultural Products and Services using the Microsoft Cloud Platform.

Christian Bannes works as Lead Developer at ip.labs GmbH and has been working in the professional Java Enterprise environment for over ten years. In recent years, he has been working with AWS Cloud and especially with serverless applications. He is particularly interested in distributed architectures, domain-driven design, and functional programming.

Nikhil Barthwal is a Senior Software Engineer at Google and a Start-up mentor. He works on Cloud Functions, the Google Cloud's Serverless offering, specifically works on developer tooling and aims to make it the best Function-as-a-Service and the best platform for developers. Outside of work, he speaks at local meetups as well as international conferences on several topics related to Distributed systems and Programming Languages. You can get to know more about him via his homepage www.nikhilbarthwal.com.

Erez Berkner is the CEO & co-founder of Lumigo, a startup focusing on simplifying serverless applications troubleshooting, where the entire backend is... 100% serverless. Prior to founding Lumigo, Erez was the R&D director of cloud products at Check Point, heading the company's cloud strategy & execution.

Michael Dowden is the CEO of Andromeda, Product Architect for FlexePark, and a Google Developer Expert in Firebase. For more than twenty years he has been writing code and geeking out over technology. He is passionate about keeping things simple and focusing on what provides real value to the end user. Michael enjoys speaking at conferences and user groups, and mentoring other developers and entrepreneurs. In 2015, he wrote Programming Languages ABC++ to share programming languages with children.

Erwin van Eyk works at the intersection between industry and academia. As a software engineer at Platform9, he contributes to Fission: an open-source, Kubernetes-native, Serverless platform. At the same time, he is a researcher investigating "Function Scheduling and Composition in FaaS Deployments" in the International Research Team @large at the Delft University of Technology. As a part of this, he leads the industry and academia combined serverless research effort at the SPEC Cloud Research Group.

Vadym Kazulkin is Chief Software Architect at ip.labs GmbH, a 100% subsidiary of the FUJIFLM Group, based in Bonn. ip.labs is the world's leading white label e-commerce software imaging company. Vadym has been involved with the Java ecosystem for over fifteen years. His focus and interests currently include the design and implementation of highly scalable and available applications, container and orchestration technologies, and AWS Cloud. Vadym is the co-organizer of the Java User Group Bonn and Serverless Bonn Meetup, and a frequent speaker on various Meetups and conferences.

Niko Köbler is doing something with computers, on-premise, web and in the cloud. He's writing and deploying software and helps others to do so. Besides, he is a co-lead of JUG Darmstadt, speaker at international conferences and author of the book "Serverless Computing in the AWS Cloud".

João Rosa is a Software Developer, focused on delivering quality software that matters. Believes in the software crafts to provide software in sustainable peace; he is a DDD, BDD and TDD practitioner. Can't live without his CI/CD pipeline. During his career, he always pushed the teams and himself to improve the communication, reducing the gap between developers and the business. When he is not on his duties, you can find him traveling with his wife, or laying down on the beach reading a book. João is an amateur cook in his remaining time.

Harald Uebele works as a Developer Advocate in the Digital Business Group of IBM Germany. He is a Fan of Open Source and as a Developer Advocate he helps developers embracing Open Source and cloud technologies. He started working with and using "The Cloud" a couple of years ago with IBM Bluemix which was built around Cloud Foundry. Since then he has worked with Docker, Kubernetes, and Cloud Functions aka OpenWhisk. Bluemix is now IBM Cloud.

Kamesh Sampath is a Principal Software Engineer at Red Hat, as part of his additional role as Director of Developer Experience at Red Hat — he actively educates on Kubernetes/OpenShift, Servicemesh, and Serverless technologies –. With a career spanning close to two decades, most of Kamesh's career was with services industry helping various enterprise customers build Java-based solutions. Kamesh has been a contributor to Open Source projects for more than a decade and he now actively contributes to projects like Knative,Quarkus, Eclipse Che etc., As part of his developer philosophy he strongly believes in LEARN MORE, DO MORE and SHARE MORE!

A world-renowned expert in application security with twenty years of experience in the field. **Ory Segal** is the CTO and co-founder of PureSec, a startup that enables organizations to build and maintain secure and reliable serverless applications. Prior to PureSec, Ory was Sr. Director of Threat Research at Akamai, were he led a team of top web security and big data researchers. Prior to Akamai, Ory worked at IBM as the Security Products Architect and Product Manager for the market leading application security solution IBM Security AppScan. Ory authored twenty patents in the field of application security, static analysis, dynamic analysis, threat reputation systems, etc. Ory is serving as an officer of the Web Application Security Consortium (WASC), is a member of the W3C WebAppSec working group, and was an OWASP Israel board member.

Mikhail Shilkov is a Microsoft Azure MVP, Russian expat living in the Netherlands. He is passionate about cloud technologies, functional programming and the intersection of the two. In his spare time, you can find him answering "azure-functions" questions on StackOverflow, tweeting as @MikhailShilkov about serverless, blogging about functional programming at https://mikhail.io, or presenting the stories of serverless and functional adoption at conferences and meetups.

Soenke Ruempler is co-founder of superluminar, a consultancy located in Hamburg, where he helps organizations to embrace technical and cultural change. He is always curious about the bigger picture, and how everything maps back to Deming all the time.

References

[1] https://www.cncf.io/blog/2018/02/14/cncf-takes-first-step-towards-serverless-computing/

[2] https://cloud.google.com/knative/

Serverless workloads on Kubernetes with Knative

Knative: An Introduction

Serverless is a buzzword that everybody seems to be talking about nowadays. You may like the term or not, but the important thing here is what it describes. In a nutshell, Serverless means the application's scale is constantly adapted to ensure that you always have the exact amount of resources you currently need available. In case of doubt, this may even mean: none at all! For you as a user, this means that you always pay only for the capacity you need to provide a response to the queries to your application. If there are no users or requests, you pay nothing at all.

by Markus Thömmes

Yet another subject matter which cannot be ignored anymore is the is containers. Along with the related area of container orchestration. It deals with the efficient distribution and management of containers within a cluster of machines. You can mention Kubernetes basically in the same breath, since it is the de-facto standard [1] for the orchestration of containers.

What would happen then if you were to put together the possibilities of both worlds and had a platform which would combine the properties and features of Serverless and the sophisticated container and application management of Kubernetes? There are a few solutions which try to do this: On one side, you have the so-called Function-as-a-Service (FaaS) frameworks such as OpenWhisk, Kubeless and OpenFaaS, on the other side there are Platform-as-a-Service-(PaaS) frameworks such as CloudFoundry.

FaaS frameworks give users the possibility to literally deploy a function in the programming sense of the word. The function is then executed as a response to an event. PaaS frameworks on the other hand are more focused on long-running processes which are available via an HTTP interface. Both approaches usually have a mechanism which creates a deployable unit from a piece of source code. Ultimately, all these frameworks therefore concentrate on a specific type of workload and can greatly differ from each other in the manner they are used.

What would you get if you had a platform which would combine long-running applications as well as very short-lived functions and generally applications of any size into a common topology and terminology set?

Welcome to Knative!

Knative is an open source project initiated by Google and supported by a few other giants in the industry. These include Pivotal, IBM, Red Hat and SAP, just to name a few. The participants refer to Knative as a set of Kubernetes-based middleware components which allow you to build modern applications which are container-based and source-code-oriented. The idea was inspired by other Kubernetes-based frameworks in the same environment (see above). It combines all the best practices into a single framework and fulfils the following requirements:

1. It is Kubernetes-native (thus the name Knative) and works like an extension of Kubernetes.
2. It covers any possible type of Serverless workload.
3. It aggregates all these workloads under a common topology and terminology set.

Beyond the Serverless features, it also complies with a number of additional standards required of a modern development platform:

1. It is easy to use for developers (see example)
2. It supports a variety of build methods to create a deployable unit (in this case a container image) from source code.
3. It supports modern deployment methods (such as automated deployment after commits).

How does Knative work?

To go into more detail on how Knative works and how it is structured, we must first determine which part of Knative we are actually talking about. The Knative organisation is made up of several repositories. In terms of content, relevant to the user are three main components: Serving, Build and Eventing.

- Serving: The Serving project is responsible for any features revolving around deployment as well as the scaling of applications to be deployed. This also includes the setup of suitable network topology to provide access to an application under a given hostname. There is certainly a significant overlap in terms of content between this part and the description of Serverless stated above.
- Build: As the name itself suggests, the Build project is responsible for "building" a container image from the program code. This container image can then, for example, be taken from Serving and be deployed. An interesting feature is that Build and Serving run in the same cluster. The built image therefore does need to be transported through an external registry, but is ready and available on the spot.
- Eventing: The Eventing project covers the event-driven nature of serverless applications. It gives you the possibility to establish a buffered connection between an event source and a consumer. This type of consumer can for example be a service managed by Serving.

In this article, we will focus on the Serving project in particular, since it is the most central project of Knative, as can be seen from the description above. Both the Build and Eventing projects can be used as a supplement to Serving or on their own.

Interaction with the Knative API happens by creating entities using Kubernetes' own kubectl CLI. Serving, Build and Eventing each define their own CRDs (Custom Resource Definitions) to map the various functionalities of the respective project. The Serving CRDs are as follows:

Configuration

A Configuration describes what the corresponding deployment of the application should look like. It specifies a variety of parameters, which include the name of the application, the container image to be used or the maximum number of parallel connections which may be open for one instance of the application. The Configuration essentially describes the application to be deployed and its properties.

Revision

As the name already suggests, a Revision represents the state of a Configuration at a specific point in time. A Revision is therefore created from the Configuration. This also means that Revision cannot be modified, while a Configuration very well can. If a user would for example like to deploy a new version of her application, she can update the image. To make this update known to the system, she changes the image of the configuration, which in turn the triggers the creation of a new revision. The user triggers the creation of a new revision with each change that requires a new deployment (such as changing the image or changing the environmental variables) by adapting the configuration. The user never works on the revision itself to make any changes. Whether or not a change requires the creation of a new revision is Serving's job to figure out.

Several revisions per configuration can be active at a given point in time. The name of a revision corresponds to that of the configuration, followed by a suffix (for example *00001* or *00004)*, which reflects the number of currently generated revisions.

Route

A route describes how a particular application can be called and the proportion with which the traffic will be distributed across the different revisions. It couples a

hostname accessible from the outside with potentially multiple revision in the system.

As already mentioned above, several revisions can be active in the system at any given time. If for example the image of an application is updated, depending on the size and importance of the deployment it would make sense not to immediately release the new version of the application to all users, but rather carry out the change step-by-step. For this purpose, a specific proportion of the traffic can be assigned to the different revisions using the Route.

To implement routes, Serving uses Istio, a so-called service mesh. It makes sure that all requests to applications in the system run through a router, no matter if the request originates from inside or outside the system. That way, a dedicated decision can be made as to where the corresponding request ends up. Since these routers are programmable, the proportional distribution of traffic to the revisions described above is possible.

Service

Things get a little hairy here. In the Kubernetes environment, the term "service" is quite overloaded. Knative is no exception here either and uses "service" as a name for one of its CRDs: A Knative Service describes a combination of a Route and a Configuration. It is a higher-level entity that does not provide any additional functionality. It should in fact make it easier to deploy an application quickly (similar to the creation of a configuration) and make it available (similar to the creation of a route).

During the creation of a service, the entities mentioned above are also automatically created. However, they should not be changed, because changes to the Knative Service itself would override the changes applied directly. If fine-grained flexibility is needed for adjustments and changes, the Configuration and Route should be created manually.

Knative Serving also defines a few more CRDs; these however are used internally in the system and do not play a major role from the point of view of the user.

Listing 1

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: knative-example
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/##DOCKERHUB_NAME##/knative-example
```

Theory

If the user would now like to deploy an application with a given container image, for example, she will first create a Configuration with that very image. Once the Configuration has been created, the Knative Serving System will generate a revision from the configuration. Here, we bid farewell to Knative Land and make our way to the usual Kubernetes entities. After all, the system must also be able to create containers and scale their number for the final deployment of an application. In Kubernetes, you refer to this as so-called ReplicaSet, a number of replicas. A ReplicaSet creates a specific number of Pods, depending on how many you would like. The Pods in turn contain a number of containers. In this case, the most important is the user container, which is created from the image that the user had specified when creating the Configuration.

Once the Revision is created, a so-called ReplicaSet is thus generated for the Revision, set with an initial scale of 1. This means that at first, at least one Pod is started with a user container. The purpose of this is to determine whether the Revision can be deployed and used as such at all as well as if the Container can even be started.

If everything has been successful so far, you now need to provide access to the Revision from the outside. For this purpose, a Route is created, which assigns a host name to the configuration (and thus indirectly to the Revision and the deployed Pods) using Istio. That way, the application can be accessed and queried via HTTP.

As soon as requests reach the deployed application, metrics are collected per pod. In addition to the user container, you have the so-called queue-proxy which proxies all the requests to the user container and additionally produces the metrics mentioned above. These metrics are sent to an Autoscaler, which based on the incoming number of requests decides how many Pods (and thus user containers) are needed. The algorithm essentially bases this decision on the incoming number of parallel requests and the configured maximum number of parallel requests per container. If for example, 10 parallel requests per container are set in the Configuration (*containerConcurrency: 10*) and 200 requests reach the application in parallel, the Autoscaler will decide to provide 20 Pods to process the load. Likewise, the number of Pods is reduced as soon as the volume of incoming requests goes down.

If the number of incoming requests goes back to 0, the last Pod with a user container will be deleted after a defined time has passed. Among others, this characteristic referred to as "Scale to Zero" makes Knative serverless. In this state, there is another global proxy called the Activator which accepts all the requests of a Revision that is scaled to 0. Since there are no Pods which can generate metrics for scaling, the Activator generates those very metrics and causes the Revision to be scaled according to the request load once again.

This Autoscaler/Activator team stands out from the usual Kubernetes Horizontal Pod Autoscaler (HPA) in that it allows for this scaling to 0. The HPA works on the basis of metrics such as CPU load or memory usage and performs scaling as soon as these values exceed a certain threshold. However, these values can never reach 0 and there is no way to generate a signal where there are no more Pods. Knative solves this issue with scaling based on HTTP requests, as was already described above. The Activator solves the issue of generating a signal where there are no Pods present anymore.

Practical application

There is nothing more helpful in understanding how such a complex system works than showing an example. We will create a small node.js webserver application, containerize it and then deploy and scale it using Knative. It is important to note here that node.js is only one of many possible examples. Using Knative, you can basically deploy and scale any application based on an HTTP interface. Here it is irrelevant whether the interface complies with specific guidelines such as REST, as long as the application follows a request/response model, meaning that a user makes a request via HTTP and, as soon as the corresponding result is calculated, receives it back as a response to the request. This point is important because the automatic scaling in Knative, as described above, is based on the volume of currently active requests. Applications deployed using Knative usually do not need any kind of adjustment.

To execute the steps, a working installation of Node.js [2] and Docker [3], as well as a valid Docker Hub Account [4] is required. In addition, we assume that a Kubernetes with Knative cluster [5] is already available and that kubectl is correctly configured to communicate with that cluster. A detailed description of the various ways to install Knative is provided in the accompanying documentation.

Creating the application

Back to the example. A minimal web server based on Node.js does not need more than a few lines of code at first:

```
const http = require('http')
const port = process.env.PORT || 8080

const server = http.createServer((_, response) => {
  setTimeout(() => response.end('Hello World!'), 1000)
})
server.listen(port, () => console.log(`server is listening on ${port}`))
```

After putting this in a file called *index.js*, we can start the web server using the following command:

```
$ node index.js
```

The following output will now appear in the console

```
server is listening on 8080
```

And by using a simple *curl* command, we can call the HTTP endpoint of the application:

```
$ curl localhost:8080
Hello World!
```

In this case, the HTTP server will respond after a second with the string *Hello World!* to a request via HTTP. CTRL + C closes the HTTP server. The fact that the application is listening on port 8080 is no accident here. Knative is expecting this port as the default port for the user container.

Creating the container image

To now be able to deploy this HTTP server on Knative, we need to create a container image which holds the application and its runtime. To build a container image with Docker, we first need a so-called Dockerfile. The file contains instructions that Docker needs to know what to put in the image and how the image should behave later on. For our purposes, a very minimalistic file is quite sufficient here:

```
FROM node
ADD index.js /

CMD ["node", "index.js"]
```

The *FROM* line specifies that we want to build a Node.js image. Here, the so-called base image contains all the dependencies needed to be able to start a node.js application. Then we add the newly created *index.js* file to it. Finally, we set the command with which the server is started.

And that's it. Using the following commands, we will now create the container image and publish it in a so-called *Registry*. Here, container images can be made available to allow them to be transported over the network to another machine (our Knative cluster later on). In this simple case, the Registry is Docker Hub. *$DOCKERHUB_NAME* corresponds to the login name to Docker Hub here.

```
$ docker build -t "$DOCKERHUB_NAME/knative-example"
$ docker push "$DOCKERHUB_NAME/knative-example"
```

And that way, we have created the image, given it a meaningful name and published the image under this name.

Creating the Knative Service

Now we get down to business: We will now convert the newly created image into a scalable service using Knative. For the sake of simplicity, we will use the Knative Service described above here, which combines the func-

tionality of Configuration and Route and makes things deployable in a way which is easy to understand. As is usually the case with Kubernetes, a resource is created by applying a YAML file; in our case it looks like Listing 1.

Above all, the *spec*-area is interesting here: In this position, *runLatest* defines that the generated Route always points to the latest revision. After each update, the traffic will therefore point to the updated version.

As the name already suggests, *configuration* contains all the parts needed to create a Configuration. If we were to create a Configuration manually, the now following options would be identical. A configuration generates several revisions - that we have already learned. To generate these Revisions, it uses a so-called *revisionTemplate*. As you can see in Listing 1, it ultimately includes the *container* and all the parameters needed to generate a Container (in a Pod). In this very simple case, we will only specify the *image*. *##DOCKERHUB_NAME##* should be replaced with the login name to Docker Hub, as had already occurred when the image was being built. *docker.io* is the host name, of the Registry belonging to Docker Hub. We put the YAML above into a *app.yaml* file and generate the Knative Service via

```
$ kubectl apply -f app.yaml
service.serving.knative.dev/knative-example created
```

And that's it! We can now observe how the different resources and entities are created by the service:

The Configuration:

```
$ kubectl get configuration
NAME          CREATED AT
knative-example  1h
```

The Revision:

```
$ kubectl get revision
NAME              CREATED AT
knative-example-00001  1h
```

The Deployment/ReplicaSet:

```
$ kubectl get replicaset
NAME                                DESIRED  CURRENT  READY  AGE
knative-example-00001-deployment-d65cfb48d  1       1       1      1h
```

The Route:

```
$ kubectl get route
NAME          CREATED AT
knative-example  1h
```

And last but not least, the Pods in which the user container runs.

```
$ kubectl get pods
NAME                                        READY  STATUS   RESTARTS  AGE
knative-example-00001-deployment-d65cfb48d-plbrq  3/3    Running
0       5s
```

To receive a response from our application, as at the beginning of the example, we now need its host name and the IP address of our Knative instance. The host name basically corresponds to the name of the service (*knative-example* in our case), the namespace in which the service runs (*default* in the example) and a pre-defined top-level domain (*example.com* for a standard installation). The resulting host name should therefore be *knative-example.default.example.com*. It can be called in a program as follows:

```
$ kubectl get route knative-example -ojsonpath="{.status.domain}"
knative-example.default.example.com
```

Things will definitely get trickier with the IP address of the Knative cluster, since it will be different in the various deployment topologies. In most cases though, the following should work:

```
$ kubectl get svc knative-ingressgateway -n istio-system -ojsonpath="{.
status.loadBalancer.ingress[*].ip}"
127.0.0.1
```

Based on both pieces of information, we can now create a *curl* command which calls the application, exactly as described at the beginning of the example:

```
$ curl -H "Host: knative-example.default.example.com" 127.0.0.1
Hello World!
```

Having arrived at this spot, we could now state: Kubernetes can do all of that too. And that is true, as up until now we haven't seen anything groundbreaking, and we haven't shown anything that would make Knative stand out from a base Kubernetes. We'll be coming to that now.

Scaling to 0

As we have already explained above, Knative scales the Pods of a Revision all the way down to 0, if no request is made to the application long enough. To achieve that, we simply wait until the system decides that the resources are no longer needed. In a default setting, this happens if the application does not receive any more requests for 5 minutes.

```
$ kubectl get pods
No resources found.
```

The application is now scaled to 0 instances and no longer needs any resources. And this is, as explained right at the beginning, what Serverless is really all about: If no resources are needed, then none will be consumed.

Scaling from 0

However, as soon the application is used again, meaning that as soon as a request towards application comes into the system, it is immediately scaled to an appropriate number of pods. We can see that by using the familiar command:

```
$ curl -H "Host: knative-example.default.example.com" 127.0.0.1
Hello World!
```

Since scaling needs to occur first and at least one Pod must be created, the requests usually last a bit longer in most cases. Once it has successfully finished, the Pod list looks just like before:

```
$ kubectl get pods
NAME                                          READY  STATUS   RESTARTS  AGE
knative-example-00001-deployment-d65cfb48d-tzcgh  3/3    Running
0        11s
```

You can tell by the Pod name that you are looking at a fresh Pod, because it does not match the previous name.

Scaling above 1 and updating a service

We have already explained in detail how scaling works within Knative Serving. Essentialy the incoming parallel request volume is compared to what the application can process in parallel. This information must be provided by the developer of the Knative Service. The default setting is 100 parallel requests (*containerConcurrency: 100*). To make the effect easier to see, we will decrease this maximum number of parallel requests to 1 (Listing 2)

We apply this change, as is usually the case in Kubernetes, by using the following command

```
$ kubectl apply -f app.yaml
service.serving.knative.dev/knative-example configured
```

And we can then observe how Knative creates a second Revision and generates Pods for it (as can be seen by the *00001* and *00002* suffix for the respective Revision).

```
$ kubectl get pods
NAME                                          READY  STATUS   RESTARTS  AGE
knative-example-00001-deployment-d65cfb48d-tzcgh  3/3
                                                      Running  0        2m
knative-example-00002-deployment-95dfb8f67-mgtd7  3/3
                                                      Running  0        9s
```

Since the service contains the *runLatest* setting, all the requests will from this moment on run against the last revision created. We will now trigger 20 parallel requests against the system:

```
$ for i in `seq 1 20`; do curl -H "Host: knative-example.default.example.com"
                                                      127.0.0.1 &; done
```

As we observe the pod list, we'll see that the application has been scaled accordingly (Listing 3). Warning: Most likely, a scaling of exactly 20 will not be achieved in this example, since Autoscaler is based on the feedback loop and only a sustained load would enable reliable scaling.

In a relatively short time after this small flood of requests, the application will once again be scaled back down to one pod again, to then completely disappear 5 minutes after the last request, as was already described above.

Conclusion

Knative is a project which combines the features of many other projects. It comprises the best practices of frameworks from various areas which specialise in different workloads and combines them into a single, easy-to-use platform. For developers who already use, know and appreciate Kubernetes, Knative as an extension is a solution immediately accessible and understandable. Yet even those developers who have had nothing to do

Listing 2

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: knative-example
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          containerConcurrency: 1
          container:
            image: ##DOCKERHUB_NAME##/knative-example
```

Listing 3

```
$ kubectl get pods
NAME                                          READY  STATUS   RESTARTS  AGE
knative-example-00002-deployment-95dfb8f67-9wz7m  3/3    Running   0        17s
knative-example-00002-deployment-95dfb8f67-fh8vm  3/3    Running   0        12s
knative-example-00002-deployment-95dfb8f67-mgtd7  3/3    Running   0        45s
knative-example-00002-deployment-95dfb8f67-h9t9r  3/3    Running   0        15s
knative-example-00002-deployment-95dfb8f67-hln4j  3/3    Running   0        12s
knative-example-00002-deployment-95dfb8f67-j8s8z  3/3    Running   0        15s
knative-example-00002-deployment-95dfb8f67-lbsgp  3/3    Running   0        17s
knative-example-00002-deployment-95dfb8f67-rx84n  3/3    Running   0        15s
knative-example-00002-deployment-95dfb8f67-tbvk9  3/3    Running   0        12s
```

with Kubernetes do not need to understand all the basics at first to be able to use Knative.

We have learned how Knative Serving works in detail, how it achieves the quick scaling it needs, how it implements the features of Serverless, and how you containerise and deploy a service. The Knative project is still very young. Nevertheless, an extensive group of well-known industry giants has come together and is constantly pushing the project forward along with developers. The atmosphere in the community is very open and the members are extremely helpful, so it is worthwhile even for open source newcomers to take a look at the GitHub repositories. If you're already itching to work on the project: Good entry-level issues are tagged with *good-first-issue*.

All in all, Knative is a very interesting project and thanks to the investments of many companies known in the industry, it is a platform that should not be ignored.

Session: Modern event-driven workloads with Knative

Matthias Wessendorf (Red Hat)

Knative is a Kubernetes-based platform that comes with a set of building blocks to build, deploy, and manage modern serverless workloads. Knative consists of three major areas: Build, Serving and Eventing. The session gives you an introduction of the Knative Eventing component and walks you through an end-to-end demo, showing the lifecycle of event-driven workloads on Knative. You'll see integration of 3rd party events, from systems like Apache Kafka, and how your application can be hooked up to a firehose and connect your service to process incoming events, leveraging built-in Knative features to provide request driven compute, so that services can autoscale, including down to 0, depending on the actual throughput. If you are interested in learning about event-driven serverless developer experience on Kubernetes, this session is for you!

Markus Thömmes is Principal Software Engineer at Red Hat, a specialist in serverless platforms in the open source world, and is generally very passionate about open source software. For years, he has been deeply involved as a committer and PPMC member in the design and development of Apache OpenWhisk. Currently, he is primarily involved in Google's Knative project.

Links & literature

[1] https://techcrunch.com/2018/06/06/four-years-after-release-of-kubernetes-1-0-it-has-come-long-way

[2] https://nodejs.org

[3] https://www.docker.com

[4] https://hub.docker.com

[5] https://github.com/knative/docs/tree/master/install/

Knative

by Markus Thömmes

Build

Build Template

Reusable, parameterized definition of the steps for a Build.

```
apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: jib-gradle
spec:
  parameters:
  - name: IMAGE
    description: The name of the image to push
  - name: DIRECTORY
    description: The directory containing the app
    default: .
  - name: CACHE
    description: The name of the volume for caching
    default: empty-dir-volume
  steps:
  - name: build-and-push
    image: gcr.io/cloud-builders/gradle
    args:
    - jib
    - -Duser.home=/builder/home
    - --image=${IMAGE}
    workingDir: /workspace/${DIRECTORY}
    volumeMounts:
    - name: ${CACHE}
      mountPath: /builder/home/.m2
      subPath: m2-cache
    - name: ${CACHE}
      mountPath: /builder/home/.cache
      subPath: jib-cache
  volumes:
  - name: empty-dir-volume
    emptyDir: {}
```

Build

Defines the input and the steps to fulfill the Build.

```
apiVersion: build.knative.dev/v1alpha1
kind: Build
metadata:
  name: test-build
spec:
  serviceAccountName: test-build
  source:
    git:
      url: https://github.com/GoogleContainerTools/jib
      revision: master
 template:
    name: jib-gradle
    arguments:
    - name: IMAGE
      value: docker.io/dein-name/jib-test
    - name: DIRECTORY
      value: ./examples/vertx
```

Eventing

Channel

Connects a source with a target and buffers Events until they are delivered.

```
apiVersion: eventing.knative.dev/v1alpha1
kind: Channel
metadata:
  name: test-channel
spec:
  provisioner:
    apiVersion: eventing.knative.dev/
                v1alpha1
    kind: ClusterChannelProvisioner
    name: in-memory-channel
```

Source

Emits Events and sends them to a target.

```
apiVersion: sources.eventing.knative.dev/
            v1alpha1
kind: CronJobSource
metadata:
  name: test-cronjob-source
spec:
  schedule: „*/2 * * * *“
  data: ‚{„message“: „Hello world!“}'
  sink:
    apiVersion: eventing.knative.dev/v1alpha1
      kind: Channel
      name: test-channel
```

connects

Subscription

Connects a channel with a target (for example a Route or a Service).

```
apiVersion: eventing.knative.dev/v1alpha1
  kind: Subscription
metadata:
  name: test-subscription
spec:
  channel:
    apiVersion: eventing.knative.dev/
                v1alpha1
    kind: Channel
    name: test-channel
  subscriber:
    ref:
      apiVersion: serving.knative.dev/
                  v1alpha1
      kind: Service
      name: test-service
```

connects

Route

Makes one or multiple applications accessible through a specific hostname.

```
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  name: test-service
spec:
  traffic:
  - configurationName: test-service
    percent: 70
  - revisionName: test-service-00001
    percent: 30
```

creates

Serving

Configuration

Describes the applications and their properties.

```
apiVersion: serving.knative.dev/v1alpha1
kind: Configuration
metadata:
  name: test-service
spec:
  build: see   Build.spec
  revisionTemplate:
    metadata: ...
    spec:
      container:
        image: docker.io/dein-name/jib-test
```

creates

Revision

A Configuration's state at a given point in time. Immutable.

creates

Service

A unified user interface for Routes and Configurations with opinionated roll-out use-cases.

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: test-service
spec:
  runLatest:
    configuration: see Configuration.spec
```

creates

Pod

Will be created based on the revisionTemplate. Runs the application.

Reference

Traffic Flow

Function-as-a-Service with AWS Lambda and Knative

Kubernetes as a multi-cloud operating system

In the future, many companies will try to grow their IT infrastructure with the help of the Cloud or even relocate it completely to the Cloud. Larger companies often request multi-cloud solutions. In terms of serverless frameworks, there are several ways to achieve multi-cloud operation. Using AWS Lambda, a function can be provided and the whole thing can be made cloud-independent with Knative.

by Patrick Arnold

So what exactly is a multi-cloud? A multi-cloud involves the use of several multiple cloud providers / cloud platforms, while providing the user with the feeling that it is a single cloud. For the most part, people try to advance to this evolutionary stage of cloud computing to achieve independence from individual cloud providers.

The use of multiple cloud providers boosts resilience and availability, and of course enables the use of technologies that individual cloud providers do not provide. As an example, deploying your Alexa Skill in the Cloud is relatively difficult if you've decided to use Microsoft Azure as your provider. In addition, a multi-cloud solution gives us the ability to host applications with high requirements in terms of processing power, storage and network performance with a cloud provider that meets these needs. In turn, less critical applications can be hosted by a lower-cost provider to reduce IT costs.

Naturally, the multi-cloud framework is not just all benefit. By using multiple cloud providers, the design of your infrastructure becomes much more complex and more difficult to manage. The number of error sources can increase and the administration of billing for each individual cloud providers becomes more complex.

You should compare both the advantages and disadvantages before you make a decision. If you determine that you don't really need to be afraid of being dependent on a single cloud provider, you should rather invest the time in the use of the cloud services.

What is Function-as-a-Service?

In 2014, the Function-as-a-Service (FaaS) concept first appeared on the market. At that time, the hook.io concept was introduced. In the years that followed,

all the major players in IT jumped on the bandwagon, with products such as AWS Lambda, Google Cloud Functions, IBM OpenWhisk or even Microsoft Azure Functions. The characteristics of such a function are as follows:

- The server, network, operating system, storage, etc. are abstracted by the developer
- Billing is based on usage with accuracy to the second
- FaaS is stateless, meaning that a database or file system is needed to store data or states.
- It is highly scalable

Yet what advantages does the entire package offer? Probably the biggest advantage is that the developer no longer has to worry about the infrastructure, but only needs to address individual functions. The services are highly scalable, enabling accurate and usage-based billing. This allows you to achieve maximum transparency in terms of product costs. The logic of the application can be divided into individual functions, providing considerably more flexibility for the implementation of other requirements. The functions can be used in a variety of scenarios. These often include:

- Web requests
- Scheduled jobs and tasks
- Events
- Manually started tasks

FaaS with AWS Lambda

As a first step, we will create a new Maven project. To be able to use the AWS Lambda-specific functionalities, we need to add the dependency seen in Listing 1 to our project.

The next step is to implement a handler that takes the request and returns a response to the caller. There are two such handlers in the Core Dependency: the *RequestHandler* and the *RequestStreamHandler*. We will use *RequestHandler* in our sample project and declare inbound and outbound as a string (Listing 2).

If you then execute a Maven build, a *.jar* file will be created, which can then be deployed at a later time. Already at this point you can clearly see that the *LambdaDependency* creates permanent wiring to AWS. As I had already mentioned at the beginning, this is not necessarily a bad thing, but the decision should be made consciously. If you now want to use the function to operate an AWS-specific database, this coupling grows even stronger.

Deployment of the function: There are several ways to deploy a Lambda feature, either manually through the AWS console or automatically through a CI server. For this sample project, the automated path was chosen using Travis CI.

Travis is a cloud-based CI server that supports different languages and target platforms. The great advantage of Travis is that it can be connected to your own GitHub account within a matter of seconds. What do we need to do this? Well, Travis is configured through a file called *.travis.yaml* . In our case, we need Maven for the build as well as for the connection to our AWS-hosted Lambda function for the deployment.

As you can see from the configuration in Listing 3, you need the following configuration parameters for a successful deployment:

- **Provider:** In Travis, this is the destination provider to deploy to in the deploy step
- **Runtime:** The runtime needed to execute the deployment
- **Handler name:** Here we have to specify our request handler, including package, class and method names
- **Amazon Credentials:** Last but not least, we need to enter the credentials so that the build can deploy the function. This can be done by using the following commands:
- @Grafik: Please insert second list level:
- AccessKey: *travis encrypt „Your AccessKey“ –add deploy.access_key*
- Secret_AccessKey: *travis encrypt „Your SecretAccessKey“ –add deploy.secret_access_key*
- If these credentials are not passed through the configuration, Travis looks for the environment variables *AWS_ACCESS_KEY* and *AWS_SECRET_ACCESS_KEY*

Interim conclusion: The effort needed to provide a Lambda function is relatively low. The interfaces to the

Listing 1: Core Dependency for AWS Lambda

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-lambda-java-core</artifactId>
  <version>1.2.0</version>
</dependency>
```

Listing 2: Handler class for requests

```
public class LambdaMethodHandler implements RequestHandler<String,
String>
{

  public String handleRequest(String input, Context context) {
    context.getLogger().log("Input: " + input);
    return "Hello World " + input;
  }
}
```

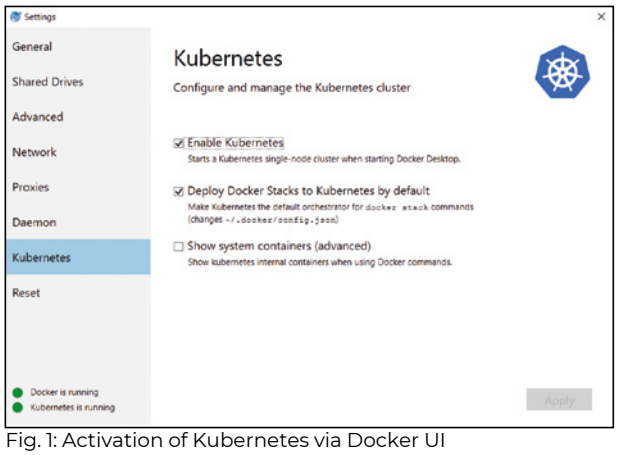


Fig. 1: Activation of Kubernetes via Docker UI

Lambda implementation are clear and easy to understand. Because AWS provides Lambda Runtime as SaaS, we don't have to worry about installation and configuration, and on top of that, we receive certain services right out of the box (such as logging and monitoring). Of course, this implies a very strong connection to AWS. So if for example, you want to switch to Microsoft Azure or to Google Cloud for some reason, you need to migrate the feature and adjust it accordingly in your code.

FaaS with Knative

The open-source Knative project was launched in 2018. The founding fathers were Google and Pivotal, but all the IT celebrities such as IBM and Red Hat have since joined the game. The Knative framework is based on Kubernetes and Istio, which provide the application environment (container-based) and advanced network routing. Knative expands Kubernetes with a suite of middleware components essential for designing modern container-based applications. Since Knative is based on Kubernetes, it is possible to host the applications locally, in the cloud or in a third-party data center.

Pre-Steps: Whether it's AWS, Microsoft, IBM or Google - nowadays every big cloud provider offers a „managed Kubernetes". For test purposes, you can also simply use a local MiniCube or Minishift. For my use case, I use the MiniCube supplied with Docker for Windows, which can be easily activated via the Docker UI (**Fig. 1**).

Unfortunately, the standard MiniCube alone does not give us anything. So how do we install Knative now? Before we can install Knative, we need Istio first. There are two ways to install Istio and Knative - an automated method and a manual one.

Individual installation steps for different cloud providers are provided in the Knative documentation. It should be noted here that the part specifically referring to the respective cloud provider is limited to the provisioning of a Kubernetes cluster. Once you install Istio, the procedure is the same for all cloud providers. The manual steps required for this can be found in the Knative documentation on GitHub [1].

For the automated method, Pivotal has released a framework called riff. We will run into this later on as well during development. riff is designed to simplify the development of Knative applications and support all core components. At the time this article is being written, it is available in version 0.2.0, assuming the use of a kubectl configured for the correct cluster. If you have this, you can use the command from Listing 4 to install Istio and Knative via the CLI.

```
>>riff system install
```

Warning! If you are installing Knative locally, the parameter *--node-port* needs to be added. After you run this command, the message *riff system install completed successfully* should appear within a few minutes.

Now we have our application environment with Kubernetes, Istio and Knative set up. Relatively fast and easy thanks to riff - in my opinion at least.

Development of a function: We now have a number of supported programming languages to develop a function that will then be hosted on Knative. A few sample projects can be found on the GitHub presence [2] of Knative. For my use case, I decided to use the Spring Cloud Function project. This specifically helps deliver

Listing 3: Configuration of „travis.yml"

```
language: java
jdk:
- openjdk8
script: mvn clean install
deploy:
  provider: lambda
  function_name: SimpleFunction
  region: eu-central-1
  role: arn:aws:iam::001843237652:role/lambda_basic_execution
  runtime: java8
  andler_name: de.developerpat.handler.LambdaMethodHandler::handleR
equest
  access_key_id:
    secure: {Your_Access_Key}
  secret_access_key:
    secure: {Your Secret Access Key}
```

Listing 4: "spring-cloud-starter-function-web" dependency

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-function-web</artifactId>
</dependency>
```

business value as a function, while bringing all the benefits of the Spring universe with it (autoconfiguration, dependency injection, metrics, etc.).

As a first step, you add the Dependency described in Listing 4 to your *pom.xml*.

Once we have done that, we can write our little function. To make the Lambda function and the Spring function comparable, we will implement the same logic. Since our logic is very minimal, I will implement it myself in *Spring Boat Application*. Of course, you can also use the Dependency injection as you normally would with Spring. We use the *java.util.function.Function* class to implement the function (Listing 5). This will be returned as an object of our *Hello* method. It is important that the method with the *@Bean* annotation is used, otherwise the end point will not be released.

Knative Deployment: To make sure we can provide our function, we need to first initialize our namespace with riff. Here the user name and the Secret of our Docker registry are stored so that the image can be pushed when creating the function. Since we do not have to write the image ourselves, riff practically takes this over for us. For this purpose, a few CloudFoundry build packs are used. In my case, I use the Docker Hub as a Docker registry. Initialization can be run using the following command:

```
>>riff namespace init default --dockerhub $DOCKER_ID
```

If you want to initialize a namespace other than *default*, just change the label. But now, we want to deploy our function. Of course, we have checked everything into our GitHub repository and we now want to build this state and subsequently deploy it. Once riff has built our Maven project, a Docker image should be created and pushed into our Docker Hub repository. The following command takes care of this for us:

```
>>riff function create springknative --git-repo
https://github.com/developerpat/spring-cloud-function.git --image
developerpat/springknative:v1 --verbose
```

If we want to do all that from a local path, we swap *--git-repo* with *--local path* and add the corresponding path. If you now take a look at how the command runs, you recognize that the project is analyzed, created with the correct build pack, and the finished Docker image is pushed and deployed at the end.

Calling the function: Now we would like to test - which is also relatively easy to do with AWS via the console - whether the call works against our function. We can do that very simply as follows thanks to our riff CLIs:

```
>>riff service invoke springknative --text -- -w '\n' -d Patrick
curl http://localhost:32380/ -H 'Host: springknative.default.example.com' -H
'Content-Type: text/plain' -w '\n' -d Patrick
Hello Patrick
```

Using the *invoke* command, you can generate and execute a *curl* as desired. As you can see now, our function works flawlessly.

Can Knative compete with an integrated Amazon Lambda?

Due to the fact that Knative is based on Kubernetes and Istio, some features are already available natively:

- **Kubernetes:**
- Scaling
- Redundancies
- Rolling out and back
- Health checks
- Service discovery
- Config and Secrets
- Resilience
- **Istio:**
- Logging
- Tracing
- Metrics
- Failover
- Circuit Breaker
- Traffic Flow
- Fault Injection

This range of functionality comes very close to the functional scope of a Function-as-a-Service solution like AWS Lambda. There is one shortcoming, however: There are no UIs to use the functionality. If you want to visualize the monitoring information in a dashboard, you need to set up a solution yourself for it (such as Pro-

Listing 5: Implementation of our function

```
package de.developerpat.springKnative;

import java.util.function.Function;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class SpringKnativeApplication {

  @Bean
  public Function<String, String> hello(){
    return value -> new StringBuilder("Input: " + value).toString();
  }

  public static void main(String[] args) {
    SpringApplication.run(SpringKnativeApplication.class, args);
  }
}
```

metheus). A few tutorials and support documents are available in the Knative documentation (/*TODO*/). With AWS Lambda, you get those UIs innately and you don't need to worry about anything anymore.

Multi cloud capacity

All major cloud providers now offer a managed Kubernetes. Since Knative uses Kubernetes as the base environment, it is completely independent of the Kubernetes provider. So it's no issue to migrate your applications from one environment to another in a very short amount of time. The biggest effort here is to change the target environment in the configuration during deployment. Based on these facts, simple migration and exit strategies can be developed.

Conclusion

Knative does not fall in line with all aspects of the FaaS manifest. So is it an FaaS at all then? From my personal point of view, by all means yes. For me, the most important item in the FaaS manifesto is that no machines, servers, VMs or containers can be visible in the programming model. This item is fulfilled by Knative in conjunction with the riff CLI.

Patrick Arnold is an IT Consultant at Pentasys AG. Technology to him is like a toy for little kids - he's addicted to new things. Having gone through old-school education in the mainframe area, he switched to the decentralized world. His technological focus is on modern architectural and development approaches such as the Cloud, API-Management, Continuous Delivery, DevOps and Microservices.

Links & literature

[1] Knative documentation on GitHub: https://github.com/knative/docs/blob/master/docs/install/README.md

[2] https://github.com/knative/docs/tree/master/docs/serving/samples

Session: **Build a powerful recommendation engine using AWS Rekognition and Serverless in 72 hours**

Samuel James (Architrave GMBH)

One of the advantages of building on Serverless is the drastic reduction in development time and time to market. This session will show you how to build a powerful recommendation engine using image recognition technology and all run on server-less in 72 hours. There will be a demo session. This talk is rated level 200-300 with a target audience of engineers, architects, and developers and assumed you have some knowledge of Amazon web services.

Adapting Testing for Serverless Applications

Serverless Testing

There's no doubt about it, testing serverless applications is a difficult job. Although the fundamental principles you're familiar with from traditional architectures remain the same, the fact that your application exists entirely in the cloud and consists of a whole host of managed services tied together by your code, requires an entirely new testing paradigm.

by Avishai Shafir and Alex Sholem

The new testing pyramid – it's all about the middle

In the traditional testing pyramid [1] we're all familiar with, unit tests sit at the bottom, component testing in the middle, and user interface testing at the top. The reason for this is simple: most emphasis is placed on the cheapest, least complex, and most easily automatable tests.

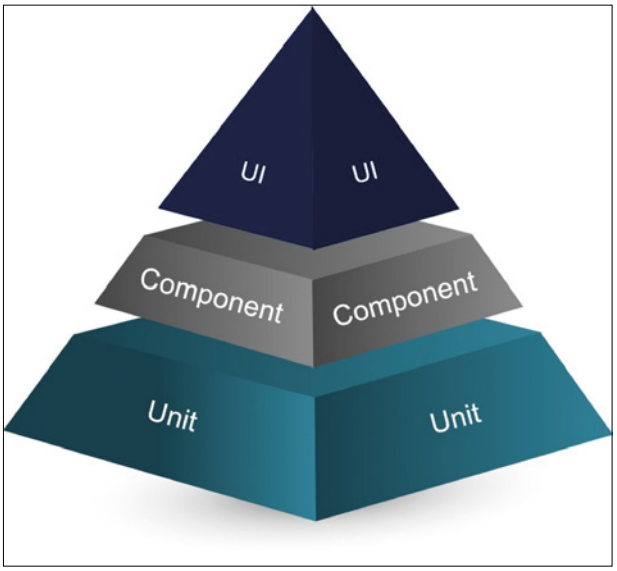
Unit tests are simplest to write and perform, so by focusing on unit tests we're able to get the best possible testing coverage for our software.

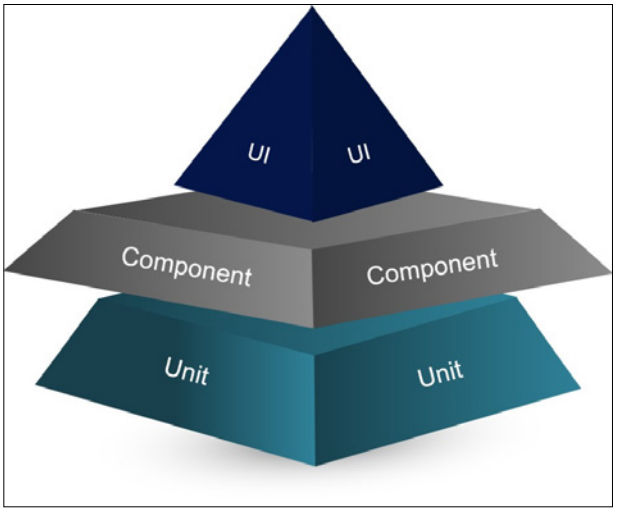
Component – or Integration – testing sits in the middle and focuses on the interaction between the various component parts of our application.

User interface testing is the most difficult of the three types of tests to automate (though there are many tools for the task, these types of tests are the first to break after any small change in the software), and typically has been performed by a separate QA or testing department.

So, how do things change in the world of serverless? You'll be glad to know we still have a pyramid with the same familiar parts. The difference now is that we need to place much more emphasis on component testing than before. Our pyramid has become slightly misshapen.

This is a shift we already saw in the move to microservices, but it's become even more critical in serverless applications (that many refer to as nano services), which are typically comprised of a complex web of ephemeral





functions and third-party services such as DynamoDB, Kinesis, S3 and so on.

Not only is serverless even more granular than microservices, but since your application exists entirely on the cloud, and you do not own all the code that makes it up, it becomes difficult to test locally.

The limits of local testing in serverless

Unit testing for serverless applications can be (and is) conducted locally with the same ease as for other architectures. So, let's skip over our pyramid's foundation and focus on the more interesting question: is it good practice to perform component testing on your local machine?

After all, it's easier to test locally, where you have visibility over your entire environment, not to mention cheaper, since you don't have to pay your cloud service provider for a testing environment.

Tools, such as Lambda-local [2] and LocalStack [3], do allow you to do this, not to mention two of the most popular deployment tools, AWS SAM [4] and Serverless Framework [5]. There are also mocks for some of

Resource	Limit
Function memory allocation	128 MB to 3,008 MB, in 64 MB increments.
Function timeout	900 seconds (15 minutes)
Function environment variables	4 KB
Function resource-based policy	20 KB
Function layers	5 layers
Invocation frequency (requests per second)	10x concurrent executions limit (synchronous – all sources)
	10x concurrent executions limit (asynchronous – non-AWS sources)
	Unlimited (asynchronous – AWS service sources)
Invocation payload (request and response)	6 MB (synchronous)
	256 KB (asynchronous)
Deployment package size	50 MB (zipped, for direct upload)
	250 MB (unzipped, including layers)
	3 MB (console editor)
Test events (console editor)	10
/tmp directory storage	512 MB
File descriptors	1,024
Execution processes/threads	1,024

Quelle: https://docs.aws.amazon.com/lambda/latest/dg/limits.html

the most commonly used managed services. But while it's technically possible to test your application locally, there are some major disadvantages to doing so. The first and most obvious issue is that while official and unofficial mocks exist for many of the managed services we use in production, there are notable absences such as Kinesis and SQS, as well as non-AWS services like Auth0 [6].

And even for those that do exist, mocks may not have all the features that the most updated live version of the services have (the open source mocks are not updated at the pace of the AWS managed services). But one of the biggest reasons to test in the cloud is the importance of configuration to the smooth running of a serverless application. Many of the issues we run into come down to improper configuration of the managed services held together by our Lambdas. When testing against local mocks everything may appear fine, but in production the interaction between various other components may cause issues we didn't foresee.

Serverless component testing should be done in the cloud

Testing in the cloud is the only way to ensure that everything is configured correctly. It's also the only way to test against the various limitations set by the cloud service providers.

There are limitations set by AWS on the number of concurrent functions (this is currently set to a maximum of 1,000 concurrent executions per account) and on how long a Lambda function can run (at the moment the timeout limit is 900 seconds).

Efficiency and cost can also only be tested properly in the cloud. We pay based on the amount of memory we allocate to a function and the duration of execution. Assigning more memory can lead to much faster execution (as the number of cores AWS assign to the task is based on the amount of memory requested), and the trick is to find the right balance to ascertain the most cost-effecti-

Memory Allocation	Execution Time	Cost
128 MB	11.72296 sec	$0.024628
256 MB	6.67894 sec	$0.028035
512 MB	3.194954 sec	$0.026830
1024 MB	1.46598 sec	$0.024638

Quelle: Jeremy Daly https://www.jeremydaly.com/15-key-takeaways-from-the-serverless-talk-at-aws-startup-day/

ve allocation. We can only deduce this by testing in the cloud environment.

While testing in the cloud is more expensive, because every developer will need their own account specifically for testing, it isn't prohibitively so, even for larger teams.

The main problem is with the extra time that it takes to deploy everything to the cloud for testing. At Lumigo [7] we use a simple bash script to automatically check whether any changes have been made to the application. If so, we deploy the whole lot to the cloud for testing. If not, we only need to deploy the function that we wish to test.

Summary

- While we can apply the traditional testing pyramid to serverless application testing, much more emphasis needs to be placed on component testing
- Local testing still suffices for unit tests but for proper component testing it comes with too many limitations.
- Testing in the cloud introduces additional expense and complication but is the only way to ensure our application is configured correctly and operates within the limits set in the cloud environment.

Alex Sholem is Content Marketing Manager at Lumigo. He is a start-up seasoned communications professional, delivering powerful and consistent brand messaging across multiple channels, and driving sales through effective and creative communication plans.

Avishai Shafir is a veteran developer that led development with several global hi-tech companies. Among his roles, he served as the Chief Functional Architect of HP ALM (Quality Center & Performance Center), leading the HP Mercury ALM architects team and meeting with developers around the world to shape state-of-the-art tools for developers. Later in his career, he managed a line of business at Panaya (acquired by Infosys for $200 Million) and led a team of product managers at Cellebrite (the global leader in digital intelligence solutions). He is a technology enthusiast.

Links & Literatur

[1] https://martinfowler.com/articles/practical-test-pyramid.html

[2] https://www.npmjs.com/package/lambda-local

[3] https://localstack.cloud/

[4] https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html

[5] https://serverless.com/

[6] https://auth0.com/

[7] https://lumigo.io/

Want to build a serverless data warehouse? These are the points to consider

Serverless drawbacks and advantages

What do you have to consider when deciding to use a serverless data warehouse? There are both drawbacks and benefits to keep in mind when making this choice. Emily Marchant discusses the advantages and disadvantages to help guide you on your way.

by Emily Marchant

Data warehouses are the places where organizations store and analyse the reams upon reams of data that they collect. Historically, these were eye-wateringly expensive operations that only the largest organizations could afford to build, but with the advent of serverless, third-part hosting sites, data warehousing has become a reality for businesses or all sizes.

These services combine all the necessary aspects of running a data warehouse, from the admin, security and reliability viewpoints, and offer cost-effective pricing plans so organizations can take advantage of their services.

Serverless building blocks

Understanding how a data warehouse works requires understanding of the ever-present building blocks which are integral to data warehouses of every variety. These can be identified as a centralized repository and a data pipeline.

The centralized repository supports storage and analytics. In physical terms, it would be the bricks-and-mortar premises. So if the repository is the physical space, the data pipeline is your transport: how information is sent and received by the repository.

When deciding on the best data pipeline to use (known as the serverless ETL), there are a number of considerations, including:

What the origins and intended destinations of data are, meaning cloud-based, or on-prem to cloud etc.

The usual size of the data being sent.

What type of configurations and usability are required.

The pricing models which are given for the various required functions (and of course there are plenty).

Should you use a serverless data warehouse?

So the question of course is whether a serverless data warehouse is required for your business needs. Unless you are a massive corporation, then you almost certainly do not need to build your own, because then you need to consider the cost and manpower required to undertake such a task.

With many serverless options available that can store and manage your data, which can be accessed with quick queries, there really is no need to follow any other option, states Trevor Jeffries, a tech blogger at Writemyx [1] and Britstudent [2].

Here are just some of the benefits:

It really could not be easier to manage. You do not need to hire (and pay) experts because the third-party service does all this for you. The individual building blocks can be managed independently and you can decide at any stage to add or reduce the services that you require.

It can be scaled on demand. As your data warehouse is stored on the cloud, there is no upfront commitment to what you will require, and if those requirements change over time, you have no already committed to them.

It's cost-effective. For logical reasons already mentioned.

Drawbacks

Like everything, there may be the odd disadvantage. Perhaps the obvious one here is in the question of integration. As Annabell Pieters, a writer at Nextcoursework [3] and 1day2write [4] reasons, "if you have a number of serverless building blocks, integrating these effectively with your existing operation can occasionally be problematic."

Although they remove the need to hire in-house data warehouse experts, just the setup process can be complex and long-winded. A possible solution here is hiring a project manager on contract who can run with this process, but an element of understanding, particular upon existing management and IT staff will be needed, and of course requirements change over time too.

Finally, not all solutions are cost effective, as some solution do require up-front payments. And then there is the all-important issue of vendor lock-in which always provides an inherent risk.

What the future looks like?

Undoubtedly the future for most organizations is a serverless, or at least part-serverless approach. There are just too many benefits and not enough drawbacks to going at it this way to prevent serverless data warehouses becoming the norm.

As third-party vendors become more developed, the technology becomes more advanced and accessible, and in-house staff become more knowledgeable, the current drawbacks may not remain so for long. Indeed, even current cost plans should be greatly reduced upon in the not-too-distant future. That is what serverless data warehouses are here to stay, and why it could be time to migrate your company over to this approach.

Emily Marchant is a marketing manager at Academicbrits and Dissertation Help renewing and retaining existing subscription-holders. From sponsored content to effective advertising campaigns and well-judged PR events, moving through the dissemination of newsletters and the building of fruitful partnerships. Her skillsets include project management and effective team collaboration. She is also a blogger at PhdKingdom.

References

[1] https://writemyx.com/

[2] https://britstudent.com/

[3] https://nextcoursework.com/

[4] https://1day2write.com/