



ServerLess Guide

Table of Contents

Introduction	1.1
Introduction	1.2
The Evolution	1.2.1
What Is Serverless?	1.2.2
Serverless In The Cloud Computing Paradigm	1.2.3
Serverless vs. PaaS	1.2.4
Architecture	1.3
A Look Back	1.3.1
The Monolithic Architecture	1.3.1.1
The Service Oriented Architecture	1.3.1.2
The Microservices Architecture	1.3.1.3
The Serverless Architecture	1.3.2
Patterns for Partitioning Code	1.3.2.1
Adoption	1.4
Development	1.5
Culture	1.5.1
Pathway to Serverless Development	1.5.2
Project Structure	1.5.3
Deployment	1.5.4
Testing	1.5.5
Unit Testing	1.5.5.1
Integration Testing	1.5.5.2
Debugging	1.5.6
Examples	1.5.7
Toolsets	1.6
Frameworks	1.6.1
Apex	1.6.1.1
Chalice	1.6.1.2
Serverless Application Model (SAM)	1.6.1.3
Serverless Framework	1.6.1.4
Operations	1.7
Security	1.8
General	1.8.1
What are the new concerns and challenges?	1.8.1.1
Are existing security best practices relevant?	1.8.1.2

Pros Of Serverless	1.8.2
A Dynamic Attack Surface Area	1.8.3
Data At Rest And Data In Transit	1.8.4
Application Vulnerabilities	1.8.5
Access Management	1.8.6
Access Segmentation	1.8.7
Best Practices	1.8.8
Providers	1.9
Commercial Hosted Platforms	1.9.1
A Comparative Look	1.9.1.1
AWS Lambda	1.9.1.2
Azure Functions	1.9.1.3
Google Cloud Functions	1.9.1.4
IBM Cloud Functions	1.9.1.5
Opensource Platforms	1.9.2
A Comparative View	1.9.2.1
Kubeless	1.9.2.2
Apache OpenWhisk	1.9.2.3
OpenFaaS	1.9.2.4
Case Studies	1.10
Glossary	1.11



Authored by the community, curated by [Serverless, Inc.](#)

Vision

This is your definitive guide to serverless architectures. Inside, you will find everything you need to know about serverless development and how to be a serverless organization: patterns, best practices, case studies and everything in-between.

Aside from implementation, this guide is a collection and discussion of concerns around the state and maturity of serverless. A lot is being written about serverless technology all over the web, and right now that information is fragmented. As a community, we're bringing it together.

The goal is to create a one-of-a-kind, trusted resource for developers, architects and thought leaders. This should be a valuable resource that will help drive adoption and innovation of serverless architectures.

Are you with us? We want [you to be part](#) of this journey.

What Will Be Covered

- **Intro to Serverless:** A little bit of history, evolution, misconceptions around serverless
- **Benefits:** Get into why serverless, its benefits and characteristics
- **Adoption:** Insight into the adoption by enterprises and the tech community in general
- **Architecture:** Discussions around serverless architecture, FaaS, evolving patterns and solutions
- **Security:** Auth. services, access controls, surface areas for attack, data isolation...
- **Development, Deployment & Testing:** Rethinking around developing, deploying and testing serverless applications and services
- **Toolsets:** Evolving toolsets, frameworks and methodologies
- **Changing DevOps:** A look into the changing roles of DevOps teams and the mindshift
- **Challenges:** Concerns around debugging, logging, and monitoring, of serverless applications
- **Providers:** A comparative look at the serverless providers out there
- **Case Studies:** Examples of real-life implementations of serverless technologies

How to Contribute

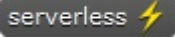
- Are you excited about the serverless movement and talk about its future?
- Do you have an example of a serverless usecase that you want to highlight?
- Are you a developer who has expertise in a particular serverless provider?
- Do you work for a provider and see some inconsistencies in the content?
- Is there an architectural pattern that you feel solves a specific usecase?
- Do you have a success story that you would like to share?
- Have you been successful with a specific vendor or a platform?
- Do you have an opinion/quote that you would like to share?

Then you have lots to share with the community.

We want your contributions to this guide. We would like you to bring in your expertise to showcase them in this guide. We are aiming to create a world-class guide that excites everyone to join the serverless movement. We feel we can achieve that by creating authentic content that is driven by the community. See the [contributor guidelines](#) and the [contributor code of conduct](#) for details.

Credits

To recognize your contributions to the guide, we encourage you to add your name to the Credits section on a page. We will add you to the 'Guide Authors' team on our Github org. You also get our serverless badge

 for your Github profile.

Get in Touch

We would love to have your feedback. Or, if you have any questions, please let us know at hello@serverless.com.

License



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

Thanks in Advance

The serverless guide is a community-driven effort, and we thank you for your contributions.

Thanks,

The Serverless Team

Introduction

Introduction

After a decade of cloud computing, we are at the crossroads of a paradigm shift in technology. Barring the confusion and the misconception of the term "serverless", serverless computing is the talk of the town. Serverless is being heralded as the pinnacle of cloud computing.

Serverless is a new way to approach cloud computing and AWS Lambda trailblazed the path with its serverless compute platform. It provided an event-driven, functions based, pay-per-execution, auto-scaling serverless computing platform. It is liberating the developers from constantly thinking about infrastructure and the means to manage them. It is set to bring the focus back on building and shipping products in an agile and iterative manner.

From an application development perspective, serverless computing makes functions as the unit of development. Based on event-driven architecture, it is creating an ecosystem where events will flow seamlessly across product boundaries, ushering a new era of composability and sharing. Heterogenous systems and disparate services will be talking to each other based on universal event specs, forming the basis for a large-scale, distributed system. The services and their functions could be discovered, shared and eventually run across infrastructure or provider boundaries.

Serverless computing is *not* here to replace every other computing platform out there. It is here to augment them. It will help us re-think how we build applications for tomorrow. It is a disruptive technology that will change how we code.

The Evolution

Back in the sixties, the concept of delivering computing capabilities over a global network was being envisioned. The vision was to have everyone interconnected and access applications & data, from any site, across the globe. In the nineties, with the internet gaining more popularity, the web technologies started maturing. It created a need for hosting these websites. ISPs mushroomed the landscape and hosting providers lined up infrastructure, renting shared resources to satisfy the need. Soon enough, a larger scale and distributed hosting strategy gave rise to global data-centers. **Traditional data-centers abstracted the hosting environment with limited elasticity and resource pooling. Scaling was achieved by adding more hardware.** Consumers managed the application stack, OS, data, storage, networking and the hardware.

As cloud technologies evolved, Infrastructure-as-a-Service (IaaS) platforms like Amazon Web Services (AWS), Microsoft Azure, Google Compute Engine (GCE), Joyent, and OpenStack became prevalent. They abstracted away infrastructure components into self-service models for accessing and managing compute (VMs or bare-metal), object storage, block storage, and networking services via APIs, and billed based on consumption. **Virtualized data-centers abstracted the underlying infrastructure. Scaling was achieved by allocating more compute (VMs) and other infrastructure resources.** Consumers managed the application stack, data, and OS, while the IaaS provider managed the virtualization, servers, hard drives, storage, and networking.

Platform-as-a-Service (PaaS), added another layer of abstraction on top of the IaaS components, by providing a unified computing platform with a self-service portal to deploy applications. PaaS software platforms like AWS Elastic Beanstalk, Windows Azure, Heroku, Force.com, Google App Engine, Pivotal CF, Apprenda, RedHat Openshift are some popular examples. **PaaS abstracted the management of infrastructure services, with scalability, high-availability, and multi-tenancy as it's core tenets.** Consumers managed the application stack and data, while the PaaS provider managed the OS, virtualization, servers, storage, and networking.

Containerization extended the virtualization solution by making it lightweight using fewer resources and faster boot times. It provided a portable runtime across OSs and a lightweight distribution & packaging mechanism for applications along with its dependencies. PaaS platforms use containers to manage and orchestrate applications. Containerization promoted the micro-services architecture pattern, by encapsulating & isolating reusable functional components as services. Container service providers like Docker Data Center, Amazon ECS, Google Cloud Platform, and Microsoft Azure are some popular examples. **Container runtime engines abstracted the OS.** Consumers manage the application stack and data, while the container service provider manages the container engine, host OS, servers, storage, and networking.

In the last few years, AWS Lambda made the serverless computing platform accessible to the masses, thus bringing another round of abstraction in play. The Functions-as-a-service (FaaS) paradigm promotes self-contained, stateless chunks of code packaged into 'functions', that can be run or triggered, in ephemeral containers. All done, without managing the underlying infrastructure or the language runtime needed by the code. The unit of deployment is functions. Microsoft Azure Functions, Google Cloud Functions, IBM Openwhisk, Webtask.io, and Iron.io are few examples of serverless providers. **Serverless computing abstracted the language runtime.** Consumers manage the application code in units of functions, while the serverless cloud provider manages the execution environment and everything else under it.

What Is Serverless?

Just like wireless internet has wires somewhere, serverless architectures still have servers somewhere. What makes something serverless is, that as a developer you don't have to worry or think about those servers. You can just focus on code.

There are 4 core tenets you should know about:

1. **Zero Administration:** This is the most exciting thing about serverless. Whereas previous abstractions like VMs and containers still shared a lot of the same configuration and administration properties of servers, serverless is a completely different experience. When you're ready to deploy code, you don't have to provision anything beforehand, or manage anything afterward. There is no concept of a fleet, an instance, or even an operating system. Everything runs in the cloud and the provider manages scaling for you.
2. **Pay-per-execution:** This is what typically incentivizes developers to try serverless for the first time. It's alluring to have complete resource utilization without paying a cent for idle time. This tenet alone results in over 90% cost-savings over a cloud VM and immeasurable developer satisfaction in knowing that you never have to pay for resources that you don't use.
3. **Function as unit of deployment:** Serverless architectures are composed of very small, independent bits of code (functions) that are loosely coupled and collaborative—also known as a microservice architecture. The main advantage? Pieces of the system are contained. They can be developed and deployed independently. The result is fewer blockers and far greater developer autonomy and productivity.
4. **Event-Driven:** This aspect of serverless is the most under-the-radar right now, but is shaping up to be the most important in the long-term. Serverless functions are stateless, and essentially dormant, until they have an event to react to. The event is what brings them to life and provides them with the data/context to do their job. Event-driven architectures are nothing new, but the rise of serverless compute has renewed interest in them because serverless architectures are by definition event-driven.

Credits: Rupak Ganguly (@rupakg), Nick Gottlieb (@worldsoup)

Architecture

Architecture

A Look Back

Over years of building software, distinct architectural patterns have emerged. As newer technologies get introduced, we move to keep pace. Here, we compare and contrast some of the characteristics and benefits of each iteration.

The Monolithic Architecture

Applications based on [monolithic](#) architectures were developed and deployed as one large unit on shared infrastructure. The code might have been split up into components in the development phase, but they were tightly-coupled, meaning deployments are all-or-nothing. Versioning of releases was done at the application level. When problems arose in one area of the application, the whole application needed to be rolled back. Deploying new functionality required long planning and tedious approval processes across different teams. Deployments followed longer release cycles.

The Service Oriented Architecture (SOA)

With the advent of Web Services, applications based on [SOA](#) became popular and gradually replaced monolithic architectures. In practical use, the ubiquitous Web services standards enhanced the mainstream appeal of SOA design. With beginnings in [client-server](#) or n-tier architecture, the 3-tier architecture became most popular, with distinct separation of concerns via presentation, business logic, and data tiers.

The presentation tier encapsulated the UI or the front-end, the business logic was kept decoupled in the logic tier, and the data sources abstracted behind the data tier. With each layer decoupled from the other and remotely accessible via Web Service APIs, they could be deployed and scaled independently of each other. This also encouraged reuse of the business logic and data tiers across different presentation views, i.e. a web site and a mobile front-end.

Smaller groups with specialized skill sets formed teams that were aligned to the tier they owned. Decisions regarding infrastructure, planning, and approval were scoped to fewer individuals. Releases for each tier could be versioned and deployed independently. In the case of failure, rollbacks were limited to the tier in question. Deployments of new functionality were quicker, less risky and each tier could follow its own release cycle.

The Microservices Architecture

The last couple of years has seen the advent of [microservices architecture](#), which still preserves many tenets of SOA. In essence, a lot of time and effort was being wasted in managing servers and subsequent scaling of applications; microservices architecture allows teams to break an application into fine-grained, smaller services. They can communicate using a lightweight protocol. Microservices mean teams can develop, test, deploy and scale each service independently. As we saw with SOA before, even smaller, more autonomous teams tend to emerge.

The Serverless Architecture

The notion of being serverless took off when [AWS launched Lambda](#) in late 2014. The benefits were too good to ignore: serverless computing let you have a microservices architecture in which it wasn't necessary to manage underlying infrastructure. The provider would auto-scale your application in response to load.

Serverless architectures are really opinionated microservices architectures. It's a combination of [FaaS](#) (Functions-as-a-Service) for compute and [MBaaS](#) (Mobile Backend-as-a-Service) for everything else: authentication, databases, search, cache, CDN etc. The focus of application development changed from being infrastructure-centric to being code-centric.

Serverless architectures have radically changed how applications are being built and deployed at scale. It has caused teams to interact with increased autonomy and higher productivity. Decomposing microservices into functions has led to decoupling at a finely-grained level, and given developers the flexibility to map functions to events that bubble up from disparate sources.

Taking advantage of the event-driven nature of serverless applications, many use cases became relevant. Widely popular use cases being data processing, SPAs, Mobile and IoT applications, chat bots, event workflows to name a few.

Credits: Rupak Ganguly (@rupakg)

Adoption

Adoption

To contribute to this section of the guide, please see suggested content breakup in [Issue #28](#)

Development

Development

Bringing a product to life usually starts with an idea, which transforms into a code spike. A short sprint that manifests the core idea into code. Most developers want to share the idea and deploy the code as soon as they can. The excitement is usually curbed by the decisions that follow. The decisions around infrastructure, scalability, costs... That is usually a big chasm to cross.

Reimagined Ops

Longer the time from ideation to the users using the product, more chances of the idea dying on the vine.

Today, the DevOps teams spend a lot of time managing the pre-deployment phase of a product life-cycle. Most of the mundane work is standing up infrastructure and then managing the resources around it. PaaS systems and teams adhering to Infrastructure as Code, find it a little easier to manage the infrastructure, but it is still never ending. Managing hardware resource metrics, uptime of infrastructure, upgrades, and patches, takes up all the time. No time for collecting and analyzing the product usage data.

Over-allocation of hardware resources or auto-scaling tactics are used for future proofing product loads. The focus is on the scalability of core infrastructure or maintaining cost thresholds.

There is huge complexity involved in maintaining highly scalable, fault-tolerant, self-healing, performant systems. It requires highly trained teams with deep systems knowledge, course correcting at all times. Mundane and busy work. No time for enhancing core business value.

That is just the production side of things. Multiply all that pain at least three times - for development, QA and staging environments.

On the product development side of the house, developers are burdened by technical debt managing monolith applications. Larger build-test-deploy cycles take up time and focus away from incorporating user feedback. RAD is non-existent. POCs take too much time to showcase. Packaging and release processes are too cumbersome. No time to listen to the users who matter the most.

Product development is walled inside the confines of the infrastructure fortress. The developers chained to the boundaries established for them.

Numbers Don't Lie

- Serverless saves roughly 40%-60% of typical infrastructure development time

With AWS Lambda, we eliminate the need to worry about operations. We just write code, deploy it, and it scales infinitely; no one really has to deal with infrastructure management. The size of our team is half of what is normally needed to build and operate a site of this scale. - *Tyler Love, CTO at Bustle*

- 100%+ MoM growth in the serverless workloads
- 20x increase in search popularity for serverless

By using AWS Lambda, we've cut our CRISPR off-target search times by 90% and scaled to hundreds of genomes. With faster searches, scientists using our platform can spend more time focusing on their research. - *Vineet Gopal, Engineering Manager at Benchling*

- Zero to minimal cost, with AWS Lambda including 1 million **free** requests and up to 3.2 million seconds of compute time per month

The operational cost estimates are roughly \$1700/month to service approximately 40 million page requests, with the vast majority of that going towards CloudFront data transfer costs. At scale, Trek10 estimates the Lambda and DynamoDB costs to be less than \$200/month! - *Trek10 case study*

Look Maa... No Servers

NoOps is not practical. LessOps is where we are headed. Less ops, more dev. More code equals more innovation. Less servers, means less ops, which in turn means significant cost savings.

No more multiple physical environments to manage, to support different stages of product releases. Serverless enables simple configuration and tagging of functions, to realize complex combinations of function execution. Different versions of same functions can be targeted to run on different staging environments. This allows for parallel experimentation without stepping on production release cycles.

Serverless computing has all the tenets to realize that promise. Code execution in small bite sized functions is the driver. No touch provisioning of infrastructure and auto-scaling on a need-basis liberates the developer from operational complexity. A new generation of event-driven services and applications,

Zero administration of infrastructure and no responsibility towards auto-scaling, clustering or load-balancing applications will help evolve the DevOps team. A different kind of DevOps. The goals for the future DevOps teams will align on delivering immediate business value. Keeping a finger on the pulse of the core business through monitoring user activity & product usage will be the key. Developing tools and services for logging, detecting errors, distributed tracing will be the focus.

Zero to Sixty...

An approach to application development incorporating microservices architecture and FaaS, coupled with serverless computing, is the new wave.

Functions being unit of deployment, with minimal packaging, makes rapid prototyping and agile feature release cycles possible. Quicker implementation and shorter user feedback loops.

Functions being the smallest computational unit allows for pay-per-execution and micro-billing. Instead of auto-scaling VMs, small tweaks in function run times exponentially decreases the overall cost of ownership. Small knobs, big gains, at the developer level.

Room to Grow

Serverless computing is at its early stages and comes with its own challenges. By nature, since the infrastructure is not managed directly, applications that need server optimizations, counter that with tweaking RAM and CPU usage. A new problem that has emerged with serverless, is long cold start times. It has been resolved by clever techniques such as scheduled health checks to keep functions warm. The serverless tooling landscape is still immature but several frameworks have emerged to help with that. Application with very low-latency requirements may suffer performance. Serverless architecture being completely stateless presents challenges in caching, connection pooling and data storage in general. Because of the distributed and event-driven nature of the serverless system, support for debugging, troubleshooting, distributed tracing, and monitoring across functions and services needs to improve.

The Future is Promising...

The serverless style of computing is opening up the gates to a distributed, event-driven, collaborative ecosystem of functions and services. Serverless applications will interact with various event sinks and consume events from publishers. Developers will have access to a palette of services that they can discover, and integrate into their applications in real-time. Services made up of functions, possibly written in different programming languages, possibly running on multiple compute providers. All done in a transparent, seamless, cohesive manner.

The possibilities are endless, and we are on the threshold of unleashing the power of serverless computing.

Credits: Rupak Ganguly (@rupakg)

Testing

Testing

When thinking about testing serverless functions, it's useful to think in terms of unit tests that are performed against a function in isolation, and integration tests that test the system as a whole. Unit tests can be performed locally because they just require the code for the function, but integration tests involving SaaS can really only be performed on the deployed system (some platforms may offer local executions as part of a deployed system, which mitigates this somewhat).

Unit Testing

Lambda functions are ideally small—a few hundred lines of code at the most—taken up mostly by error handling; the happy path should be very short, or at least relatively straightforward. Thus, introducing abstractions can create a lot of code bloat. So what should serverless function unit tests look like? A serverless function, by definition, can only have side effects by using other services. Unlike traditional (read: serverfull) systems, it's less necessary to abstract out the service invocations. There are two reasons for this:

- Abstraction isn't worth it. Serverless architectures come bundled with a level of vendor lock-in; a lowest-common denominator interface that can talk to both AWS DynamoDB and Google Cloud Bigtable is going to have limited functionality, and basically no opportunities to take advantage of either service's optimization techniques.
- Abstraction isn't necessary for testing. If anything, it creates extra work! The AWS SDK provides mechanisms for stubbing out SDK calls. Using recorded responses as a mocking technique, the function can be tricked into believing it is making the live call.
 - There are options for the JavaScript SDK in [aws-sdk-mock](#) and [mock-aws](#), but if you code in Python, you can use [placebo](#) (though there is also [moto](#)). We'll use the AWS Python SDK, boto3 in our examples. With placebo, you passively record SDK calls on a real session, and then for testing you can instruct boto3 to use the recorded response instead of actually making the call.

Create an abstraction (`Boto3Wrapper` class) that provides factory methods for sessions, clients, and resources will enable caching, which will then reduce the overhead across successive function invocations. For example:

```
# in package boto3wrapper
import boto3

class Boto3Wrapper(object):
    _SESSION_CACHE = {}
    SESSION_CREATION_HOOK = None
    @classmethod
    def get_session(cls, **kwargs):
        key = tuple(sorted(kwargs.items()))
        if key in cls._SESSION_CACHE:
            return cls._SESSION_CACHE[key]
        session = boto3.Session(**kwargs)
        if cls.SESSION_CREATION_HOOK:
            session = cls.SESSION_CREATION_HOOK(session)
        cls._SESSION_CACHE[key] = session
        return session
    # similar for client and resource, using get_session to obtain
    # a session, and also caching the objects
```

In a function, you use it in place of directly creating Session, Client, and Resource objects:

```
from boto3wrapper import Boto3Wrapper

def handler(event, context):
    # replacing dynamodb = boto3.resource('dynamodb')
    dynamodb = Boto3Wrapper.get_resource('dynamodb')
    # use as normal
    table = dynamodb.Table('MyTable')
```

Note that since the caching is done at the class level, it persists inside a given function container between invocations.

Unit tests can then use this functionality:

```
import unittest2, os.path
from boto3wrapper import Boto3Wrapper

class MyTest(unittest2.TestCase):
    def setUp(self):
        def attach_placebo(session):
            path = os.path.join(
                os.path.dirname(__file__),
                'placebo')
            pill = placebo.attach(session, data_path=path)
            return session
        Boto3Wrapper.SESSION_CREATE_HOOK = attach_placebo

    def test_function_requirement_1(self):
        # perform test, Lambda function will automatically get
        # placebo injected on its sessions
```

This approach allows for functions to be written as concisely as possible, focusing on business logic, and letting abstraction take place at the architecture level, in the separation of code and APIs between functions.

Integration Testing

In serverless architectures, control over many—or even most—components is given up. This is generally true of using SaaS products, but with a fully serverless system, the number of points where the developer has full control is further reduced. On AWS, user code is limited to Lambda functions, API Gateway mappings, and IoT rules, which gives no ability to, for example, induce a premature shutdown of the underlying EC2 instance handling an API Gateway connection, or cause SNS to fail when invoked by an event on S3. While the compute components of serverless systems are generally stateless (a good practice), this doesn't mean that, in a degraded system, they will meet performance requirements (e.g., latency, data loss, management of distributed transactions, etc.).

While unit testing of serverless function code is fairly straightforward, as we've seen above, this does not suffice for verifying that a full system is production-ready; integration testing is required. However, integration testing for serverless architectures presents a problem. For the purpose of this section, we will assume the system uses solely AWS services. How can we test the situation where DynamoDB has less-than-perfect reliability? Does our system degrade gracefully? Does our logging and monitoring system adequately inform us of the problems?

In traditional architectures, a system like Netflix's [Chaos Monkey](#) (and related pieces of the [simian army](#)) serves this purpose, by randomly shutting down VMs and interfering with network traffic. If a system has no SaaS components, nearly every error condition can be tested this way.

Using SaaS components, we have no way to induce those components to behave abnormally. In a fully serverless system, the only control we have is over the code we put in. Given that constraint, how can we do integration testing similar to Chaos Monkey? What would Monkeyless Chaos look like?

With the starting assumption that we are using only AWS services, and the further assumption that we are using Python (just to pick a particular SDK; the requirements work for all languages), we could establish some requirements for such a system:

Requirements for Monkeyless Chaos

- A system for injecting errors into boto3 SDK calls
 - This exists, and botocore's [Stubber](#) class provides a template for implementing a more focused error-injection class
- A system for intercepting the creation of boto3 Sessions, Clients, and Resources
 - The same injection system in boto3 will work for this
 - This is so we can inject the error injector when the chaos library is loaded
- A system for specifying the errors to inject and how often (and where) they should appear
 - Service errors can be referred to by name, as the service definitions in botocore suffice to translate that into an actual exception
 - We also need network errors, such as latency or timeouts, as well as, perhaps, corrupted data
 - Allow placebo's pill format for direct specification of return
 - This system should allow for varying degrees of specificity. For example, from "this particular Lambda can't reach Kinesis 60% of the time" to "all requests to Kinesis from all Lambdas fail"
 - The error specifications should be able to be changed at run time, without requiring redeployment, to allow simulating outage scenarios (e.g., how long does recovery take once an outage is over?).

It should be possible to deploy the system without any of this code included at all, so that it would be impossible to use it to cause system degradation by accidental or malicious means. This system would likely use a DynamoDB table, shared by all components of the system, to satisfy requirement #3. The table name would be provided to the Lambda function through environment variables. The error specifications themselves could be provided through environment variables, but are then not adjustable at runtime.

To extend this beyond the use of AWS services, the first logical step is HTTP calls. The system should allow similar specifications for HTTP errors, and a way to inject these errors into common HTTP libraries like requests.

Credits: Ben Kehoe (@benkehoe)

Operations

Operations

To contribute to this section of the guide, please see suggested content breakup in [Issue #26](#)

Security

Security

Serverless is ushering in a new age of application development, but this revolution is not problem free. Most notably, when dealing with serverless, we must completely rethink the way we secure our applications.

- How do we add security controls when we ourselves don't have control over the operating system that executes our code?
- How do we add security controls when the network is abstracted?

In the following piece, we'll try to provide some useful answers to these questions.

What are the new concerns and challenges?

The main challenge when it comes to serverless security is that traditional security solutions don't have the same efficacy with this new architecture. Traditional security solutions monitor servers and the network communication between them. In serverless however, both the servers and the network are abstracted. Furthermore, serverless functions are inherently distributed and therefore visibility is limited and normal flows are difficult to define.

Additionally, implementing serverless leads to a significant increase in the amount of resources in our cloud environment that require monitoring.

Other concerns include the faster pace at which the CI/CD processes are executed, and the sudden simplicity of attaching new input sources (such as creating a new trigger) to an application which can serve as a potential entry point for attackers.

Are existing security best practices relevant?

Not only are existing best practices still relevant, but there are certain best practices that become more critical than they were in the past. These include keeping the functions' permissions least privileged and maintaining a least privileged build system.

There are many other existing best practices for security that are indeed relevant for serverless. For example, during the development and application design process, it is highly recommended that the amount of code that can access sensitive data be reduced, exceptions are handled and input is validated. It is also a good practice to avoid embedding secrets and access keys in code.

Regarding the CI/CD processes, it is still best to integrate AppSec tests, both static and dynamic. And it's also still important to integrate tools that scan your third party libraries for potential vulnerabilities and try to keep them up-to-date. It's recommended to use the popular frameworks for the development and deployment processes.

Pros Of Serverless

There are many benefits to being serverless. First and foremost, the cloud provider is responsible for managing the infrastructure. The code is executed in a pre-patched container image, and thus OS and Kernel vulnerabilities are no longer an operative issue. When implementing serverless architecture, we place full trust in the cloud provider to maintain the image in which the code is executed.

Secondly, serverless functions are ephemeral and often limited in their execution time. This fact makes it harder, **however not impossible**, for attackers to gain persistency in the execution environment.

And a third advantage in serverless is the features provided by the SaaS services that are connected to the functions. For example, API Gateway services provide the ability to throttle and define a quota on the amount of requests that go through.

A Dynamic Attack Surface Area

The attack surface in serverless changes dramatically. The complex data flows that define a fully serverless architecture, combined with the proliferation of functions and the lack of visibility, make it much harder to understand, map and test the behavior of an application. And as the amount of resources in an account grows, it becomes harder to monitor the normal state and allows attackers to potentially hide their activity.

Data At Rest And Data In Transit

Serverless functions are stateless, thus data must be stored in the cloud provider's storage service or databases. It's best to keep this data at rest encrypted. Alternatively, data can be passed from one function to another and even between different

executions of the same function, by storing it in an appropriate folder in the container. This is one of the advantages of “warm” containers.

Data can still be compromised in transit if an attacker is able to leverage a vulnerability in the function’s code, exploit the cloud provider’s virtualization technology or gain access to the account. A third party API service or a malicious third party library might also be the reason for a data leakage.

Application Vulnerabilities

In serverless, application vulnerabilities are even more relevant than before. As the application now consists of mostly code and configurations, attackers will naturally focus on exploiting these layers.

Other possible attack scenarios aside from exploiting application vulnerabilities include a compromised developer machine or [MiTM](#) attacks, which may result in adding a backdoor function or in hijacking existing functions in the account.

Once a function has been compromised, an attacker will typically try to identify the access available to him by brute-forcing the cloud provider’s APIs. An attacker will attempt to use the permissions to either gain persistency, execute lateral movement or exfiltrate data from the account.

Additionally, source code repositories such as Github and Bitbucket are well-known targets for crawlers seeking access keys. An attacker could potentially gain access to these publicly available keys and then tamper with your account.

Access Management

As is always the case, access management is a crucial part of maintaining a secured application. Dividing the cloud provider’s account users into groups, such as developers and administrators allows you to easily control the multiple users in the account and their allowed activity. Carefully examine granting permissions that allow creation, modification or removal of the resources that belong to your application.

Access Segmentation

What also happens in serverless is a unique opportunity for the creation of well defined segmentation in an application. This is due to the fact that the application is divided into granular functions and each function’s behavior can be controlled using the appropriate permissions.

In AWS for example, the entity that is responsible for the function’s permissions is an IAM role. An IAM role grants permissions to attached trusted entities. Such entities might be AWS services, AWS accounts or AWS users. Each Lambda function has an IAM role which grants permissions to this specific function and is trusted by the Lambda service entity. A Lambda function in AWS is allowed to do only what the IAM role permits. It’s a highly recommended best practice to keep the role and it’s permissions least privileged.

Another best practice is to only allow the build system to create or update resources in your production account. This limits the possibility of having unfamiliar resources in the account and the ability of an attacker to gain significant access to the system.

Best Practices

To summarize, these are a few recommendations that will help you secure your serverless applications:

- Scrutinize and research before attaching new input/event sources
- Keep the functions’ permissions least privileged and maintain a least privileged build system
- Amount of code that can access sensitive data be reduced, exceptions are handled and input is validated
- Avoid embedding secrets and access keys in code
- Do not store access keys or credentials in source code repositories
- Throttle and define quotas on the amount of requests that go through
- Keep data encrypted that is stored at rest
- Scrutinize and keep tab on third party API services for vulnerabilities
- Scan third party libraries for potential vulnerabilities and try to keep them up-to-date
- Carefully examine granting permissions that allow creation, modification or removal of the resources

Credits: Zohar Einy (@ZoharEiny), Avi Shulman (@avi-puresec)

Providers

Providers

The serverless technology has raised a lot of eyebrows in the community in the last few years and a few big players have stepped up to release their own serverless compute infrastructure platforms.

The Serverless Framework recognizes the concerns involving provider lock-in and complexity to adopt these platforms. The Serverless Framework provides flexibility for the developer to pick an infrastructure provider of their choosing, develop and deploy applications using it.

Commercial Hosted Platforms

We will take a look at some popular commercial hosted platforms, namely, **AWS Lambda**, **Microsoft Azure Functions**, **Google Cloud Functions**, and **IBM Cloud Functions** in more detail.

A Comparative Look

Features/Providers	AWS Lambda	Azure Functions	Google Cloud Functions	IBM Cloud Functions
Language Support	Node.js, Java, C#, Python	Node.js, C#, F#, Python, PHP	Node.js	Node.js, Python, Java, Swift, Docker
Security	AWS IAM , VPC support	OAuth providers such as Azure Active Directory, Facebook, Google, Twitter, and Microsoft Account	Cloud IAM	IBM Cloud IAM , OAuth providers such as Google, Facebook, and GitHub.
Monitoring	AWS CloudWatch	Azure Application Insights	Stackdriver Monitoring	IBM Cloud Functions Dashboard
Logging	AWS CloudWatch	Azure Application Insights Analytics	Stackdriver Logging	IBM Cloud Functions Dashboard
Auditing	AWS CloudTrail	Azure Audit Logs	Cloud Audit Logging	
Alerts	AWS CloudWatch Alarms	Azure Application Insights , Log Analytics, and Azure Monitor	Stackdriver Monitoring	IBM Cloud Functions Dashboard
Tooling Support	AWS CodePipeline , AWS CodeBuild	Azure Portal , Azure Powershell , Azure CLI , Azure SDK	gcloud CLI (beta) for functions	IBM Cloud Functions UI , IBM Cloud Functions CLI , OpenWhisk Shell
Debugging Support	AWS X-Ray	Azure CLI - local debugging, Azure App Service - remote debugging	Stackdriver Debugger	wskdb: The OpenWhisk Debugger openwhisk-light
Pricing	* \$0.20/million requests with 1 million requests per month free. More details...	Execution Time: \$0.000016/GBs, 400,000 GBs/month are free Total Executions: \$0.20/million executions, with 1 million executions/month	Invocations: \$0.40/million invocations with 2 million invocations free Compute Time: \$0.0000025/GB-sec with 400,00 GB-sec/month free & \$0.0000100/GHz-sec with 200,000 GHz-	Execution Time: \$0.000017 GB-s, 400,000 GBs/month are free No extra charge per invocation or API gateway

		free More details...	More details...	call More details...
Limits	<p><i>Memory allocation range: Min. 128 MB / Max. 1536 MB (with 64 MB increments)</i></p> <p>Ephemeral disk capacity ("/tmp" space): 512 MB</p> <p><i>Number of file descriptors: 1,024</i></p> <p>Number of processes and threads (combined total): 1,024</p> <p><i>Maximum execution duration per request: 300 seconds</i></p> <p>Invoke request body payload size (RequestResponse): 6 MB</p> <p><i>Invoke request body payload size (Event): 128 K</i></p> <p>Invoke response body payload size (RequestResponse): 6 MB</p> <p>More details...</p>	<p><i>Allow only 10 concurrent executions per function</i></p> <p>No limitations on max. execution time limit</p>	<p>Resource, Time and Rate Limits are defined under Google Cloud Functions Quota limits</p>	<p>IBM Cloud Functions System Limits</p>

AWS Lambda

AWS Lambda is Amazon's serverless compute offering, announced in 2015.

AWS Lambda lets you run code without provisioning or managing servers. You pay only for the compute time you consume - there is no charge when your code is not running. With Lambda, you can run code for virtually any type of application or backend service - all with zero administration. Just upload your code and Lambda takes care of everything required to run and scale your code with high availability. You can set up your code to automatically trigger from other AWS services or call it directly from any web or mobile app.

[More details...](#)

Microsoft Azure Functions

Azure Functions is Microsoft's serverless compute offering, announced in spring of 2016.

Process events with a serverless code architecture. An event-based serverless compute experience to accelerate your development. Scale based on demand and pay only for the resources you consume.

[More details...](#)

Google Cloud Functions

Cloud Functions is Google's serverless compute offering, with Beta announced in spring of 2017.

A serverless environment to build and connect cloud services. Construct applications from bite-sized business logic billed to the nearest 100 milliseconds, only while your code is running. Serve users from zero to planet-scale, all without managing any infrastructure.

[More details...](#)

IBM Cloud Functions

IBM Cloud Functions is IBM's serverless compute offering, based on Apache OpenWhisk, which launched in early 2016.

IBM Cloud Functions is an event-driven compute platform that executes application logic in response to events or through direct invocations—from web/mobile apps or other endpoints. The IBM Cloud Functions serverless architecture accelerates development as a set of small, distinct, and independent actions. By abstracting away infrastructure, IBM Cloud Functions frees members of small teams to rapidly work on different pieces of code simultaneously, keeping the overall focus on creating user experiences customers want.

[More details...](#)

Opensource Platforms

We will take a look at some popular opensource platforms, namely, **Kubeless**, **OpenFaaS**, and **Apache OpenWhisk** in more detail.

A Comparative View

Features/Providers	Kubeless	OpenFaaS	Apache OpenWhisk
Language Support	Node.js, Python, Ruby		
Security	Kubernetes Role Based Access Control (RBAC)		
Monitoring	Prometheus		
Logging	Fluentd		
Auditing			
Alerts	Prometheus Alert Manager		
Tooling Support	Kubeless UI , Kubeless CLI , Kubeless serverless plugin		
Debugging Support			
Pricing	Open-source solution to be deployed for free on any Kubernetes cluster		
Limits	Memory limits using Pod limits		

Kubeless

Kubeless is an open source project initiated by [Bitnami](#), first introduced in December 2016.

Kubeless is a Kubernetes native serverless solution. It leverages Kubernetes API primitives to deploy functions within Kubernetes Pods and expose them via Kubernetes services. Functions can be triggered by events or via regular HTTP calls. Monitoring and scaling come from the underlying Kubernetes features.

[More details...](#)

Credits: Rupak Ganguly (@rupakg), James Thomas (@thomasj)

AWS Lambda

AWS Lambda

AWS Lambda is Amazon's serverless compute service offering, announced in 2015.

AWS Lambda lets you run code without provisioning or managing servers. You pay only for the compute time you consume - there is no charge when your code is not running. With Lambda, you can run code for virtually any type of application or backend service - all with zero administration. Just upload your code and Lambda takes care of everything required to run and scale your code with high availability. You can set up your code to automatically trigger from other AWS services or call it directly from any web or mobile app.

The platform includes the following products and services:

- Lambda Functions - serverless compute
- Step Functions - state management
- API Gateway - API proxy
- IAM/VPC - security
- DynamoDB - NoSql database
- S3 - storage
- CloudFront - edge locations
- Kinesis - data streaming
- SNS/SQS - messaging and queuing
- CloudWatch - monitoring and logging
- X-Ray - diagnostics

How It Works

[insert diagram to showcase usage]

Language Support

Supports Java, Node.js, C#, and Python code.

Security

Secure access to other AWS services via built-in AWS SDK and integration with [AWS IAM](#). The code is run within a VPC by default, so the code is isolated. Support for custom VPC, security groups and network access control lists as well.

Tooling

AWS has a variety of developer tools that enable developers to securely store and version control application source code, automatically build, test and deploy their applications.

- [AWS CodePipeline](#)
- [AWS CodeBuild](#)

See [AWS Developer Tools](#) for details.

Monitoring, Logging & Alerting

Automatic monitoring of Lambda functions with metrics pushed to [AWS CloudWatch](#). The [Lambda metrics](#) include number of requests, the latency per request, requests with errors, request and error rates.

Code can be instrumented with logging statements, and the logs are automatically pushed to [CloudWatch Logs](#).

All the data about metrics and logs can be viewed from the CloudWatch console.

CloudWatch also provides alerting capabilities to create alarms and receive notifications of API activity that are captured by [AWS CloudTrail](#).

Debugging & Diagnostics

[AWS X-Ray](#), helps identify, analyze and debug issues in production for applications built with microservices architecture. It

helps in finding root cause and troubleshooting issues.

Pricing

See [AWS Lambda pricing page](#) for details.

Limitations

[AWS Lambda limits](#) are very clearly documented.

Resources

- [AWS Lambda](#)
- [Getting Started Guide](#)
- [AWS Lambda Documentation](#)

Credits: Rupak Ganguly (@rupakg)

Azure Functions

Microsoft Azure Functions

Azure Functions is Microsoft's serverless compute offering, announced in spring of 2016.

Process events with a serverless code architecture. An event-based serverless compute experience to accelerate your development. Scale based on demand and pay only for the resources you consume.

The platform includes the following products and services:

- Functions - serverless compute
- LogicApps - orchestrated workflows visual designer
- Flow - higher abstraction on top of LogicApps
- WebJobs - run scripts or programs as background processes

How It Works

[insert diagram to showcase usage]

Language Support

Supports C#, F#, Node.js, Python, PHP, batch, bash, or any executable.

Security

Protect HTTP-triggered functions with OAuth providers such as Azure Active Directory, Facebook, Google, Twitter, and Microsoft Account.

Tooling

[Azure Portal](#), [Azure Powershell](#), [Azure CLI](#), and [Azure SDK](#)

Supports coding functions directly in the portal, or through Visual Studio Team Services or others IDEs like Xcode, Eclipse, and IntelliJ IDEA. Visual Studio supports all three deployment processes (FTP, Git, and Web Deploy), while other IDEs can deploy to App Service if they have FTP or Git integration. See [deployment processes overview](#) for details.

Monitoring, Logging & Alerting

Functions integrates with [Azure Application Insights](#), the Azure APM service. It includes metrics, traces, exception tracking, dependencies and user data. Despite the serverless abstraction, App Insights lets users see server-level metrics down into individual VMs such as the CPU usage. See [monitoring overview](#) for details.

[Application Insights Analytics](#) gives detailed information about diagnostic data for an application.

[Azure Audit Logs](#) is a data source that provides a wealth of information on the operations on your Azure resources. The most important data within Azure Audit Logs is the operational logs from all your resources.

Alerts are available across different services, including, Application Insights, Log Analytics, and Azure Monitor.

Debugging & Diagnostics

The [Azure Functions CLI](#) provides local debugging support. But, since Azure Functions is built on top of [Azure App Service](#), remote debugging support is built-in.

With the help of a few tools and components like Visual Studio, [Visual Studio Tools for Azure Functions](#) and [Cloud Explorer extension](#), it is possible to set breakpoints and step through code via the debugger. **Note:** Although these tooling experience is currently in preview.

Pricing

See [Azure Functions pricing page](#) for details.

Limitations

The limitations are not really documented but gathering from user experiences, it seems Azure Functions allow only **10** concurrent executions per function. There is also **no** limitations on max. execution time limit, but you will be billed for any accidental loops.

Resources

- [Azure Functions](#)
- [Azure Functions Source](#)
- [Azure Functions Documentation](#)

Credits: Rupak Ganguly (@rupakg)

Google Cloud Functions

Google Cloud Functions

Cloud Functions is Google's serverless compute offering, with Beta announced in spring of 2017.

A serverless environment to build and connect cloud services. Construct applications from bite-sized business logic billed to the nearest 100 milliseconds, only while your code is running. Serve users from zero to planet-scale, all without managing any infrastructure.

The Google serverless platform consists of the compute service called Cloud Functions. It integrates well with other Google cloud services like:

- Cloud Storage - object storage
- Cloud Pub/Sub - real-time messaging service
- Stackdriver Logging - log management and analysis

There is a full list of [services and event providers](#), that are supported.

How It Works

[insert diagram to showcase usage]

Language Support

Supports Node.js.

Security

Google Cloud Functions provide the ability to provide access control to create and manage functions by adding users to teams and by granting them permissions using [Cloud IAM](#) roles. Cloud Functions currently only supports primitive roles.

Tooling

[gcloud CLI](#) (beta) for functions.

Monitoring, Logging & Alerting

Logs emitted from Cloud Functions are automatically written to [Stackdriver Logging](#). Stackdriver Logging allows storage, search, analysis, monitoring, and alerting of log data and events from various sources.

[Stackdriver Monitoring](#) provides visibility into the performance, uptime, and overall health of cloud-powered applications. It collects metrics, events, and metadata from various sources, ingests that data and generates insights via dashboards, charts, and alerts. It allows setting alerts on logs events based on user-defined log based metrics.

[Cloud Audit Logging](#) maintains two audit logs for each project and organization: Admin Activity and Data Access.

Debugging & Diagnostics

[Stackdriver Debugger](#) helps debug and inspect state of an application without using logging statements. It can capture the local variables & call stack and link it back to a specific line location in the source code.

Pricing

See [Google Cloud Functions pricing page](#) for details.

Limitations

Google Cloud Functions define quota limits in three different areas:

- Resource Limits: Defines the limits on the total amount of resources the functions can consume.
- Time Limits: Defines the limits on how long functions and builds can run.
- Rate Limits: Defines the limits on the rate at which the Cloud Functions API can be called. It also defines the limits on the rate at which resources can be used.

[Google Cloud Functions Quota limits](#) are very clearly documented. But, the quota limits defined by default can be increased on request.

Resources

- [Google Cloud Functions](#)
- [Quickstart](#)
- [Google Cloud Functions Documentation](#)

Credits: Rupak Ganguly (@rupakg)

IBM Cloud Functions

IBM Cloud Functions

IBM Cloud Functions is IBM's serverless compute offering, based on Apache OpenWhisk, which launched in early 2016.

IBM Cloud Functions is an event-driven compute platform that executes application logic in response to events or through direct invocations—from web/mobile apps or other endpoints. The IBM Cloud Functions serverless architecture accelerates development as a set of small, distinct, and independent actions. By abstracting away infrastructure, IBM Cloud Functions frees members of small teams to rapidly work on different pieces of code simultaneously, keeping the overall focus on creating user experiences customers want.

IBM's serverless platform is a managed instance of [Apache OpenWhisk](#), the open-source serverless project. It extends the open-source platform with additional tooling and integrates well with other IBM Cloud services like API Connect, Cloudant and Message Hub.

The platform includes the following products and services:

- [IBM API Gateway](#)
- [IBM Cloudant](#)
- [IBM Message Hub](#)
- [IBM Watson APIs](#)

How It Works



Language Support

Supports Node.js, Python, Java, Swift and raw binary runtimes. Binaries compiled for the platform can be provided in a zip file. Docker images can be specified to customise the runtime. Docker images can be used as action source or in conjunction with serverless function code.

Security

IBM Cloud Functions integrates IBM Cloud's IAM service into the OpenWhisk platform API user-based authentication scheme. Resources are shareable using the IBM cloud account management service.

API Gateway integration for IBM Cloud Functions supports a variety of authentication sources for controlling access to public API endpoints. Methods being supported include API key, API secret, and OAuth validation. Supported OAuth providers include Google, Facebook, and GitHub.

Tooling

[IBM Cloud Functions UI](#), [IBM Cloud Functions CLI](#) [openwhisk-shell](#)

Monitoring, Logging & Alerting

The [IBM Cloud Functions Dashboard](#) provides a graphical summary of the activity and can determine the performance and health of the OpenWhisk actions. It provides Activity summary, timeline, a histogram view, and logs.

Debugging & Diagnostics

The [wskdbg: The OpenWhisk Debugger](#), currently supports debugging OpenWhisk actions written in NodeJS, Python, and Swift. The debugger will arrange things so that the actions you wish to debug will be offloaded from the main OpenWhisk servers and instead run on your laptop. You can then, from within the debugger, inspect and modify values. For NodeJS actions, you can even modify code and publish those changes back to the OpenWhisk servers.

The [openwhisk-light](#) project provides a "lightweight" openwhisk runtime using local Docker runtimes. This Node.js project implements the full OpenWhisk platform API. Actions are instantiated and executed using a local Docker engine. Developers can access containers running in their development environment to resolve issues.

IBM Cloud Functions is built upon the open-source serverless platform, Apache Openwhisk. Running the [full platform](#)

[locally](#) by be achieved using a VM, Docker Compose or Kubernetes. Developers can target their `wsk` cli to point to the local endpoint for debugging production issues.

Pricing

IBM Cloud Functions only charges for execution time (GB/s), there is no extra charge for invocations or API gateway access. 400,000GBs per month is included for free.

See [IBM Cloud Functions pricing page](#) for details.

Limitations

Resources

- [IBM Cloud Functions](#)
- [Apache OpenWhisk](#)
- [Getting Started](#)

Credits: Rupak Ganguly (@rupakg), James Thomas (@thomasj)

Kubeless

Kubeless

Kubeless is an open source effort. Mostly maintained by engineers at Bitnami, it was first showcased at the Kubernetes community meeting in [December 2016](#).

Kubeless is a Kubernetes native serverless solution. It let's your go from source to deployment extremely quickly without having to care about the infrastructure plumbing underneath and without having to build containers. It extends the Kubernetes API to provide an AWS Lambda clone on-prem using Kubernetes primitives. Kubeless offers HTTP and event based triggers and supports Python, Node.js and Ruby functions.

[Kubeless](#) uses Prometheus for monitoring and fluentd for logging. A default event-based broker can be used to publish events that trigger functions. The CLI mimics the Google Cloud Functions and AWS Lambda CLI.

How It Works

Kubeless is a Kubernetes API server extension which defines a *function* primitive in a Kubernetes cluster. When you deploy a kubeless function, a *controller* injects the function source code into a Kubernetes pod. For HTTP triggers, the functions are exposed via Kubernetes services and if the user decides they can also be exposed via an ingress resource (which provides an API gateway). Functions can also be triggered by events in a Kafka broker topic.

All language runtimes are instrumented for metrics collection which allows kubeless to leverage the Kubernetes horizontal pod autoscaler (HPA) primitives.

[insert diagram to showcase usage]

Language Support

Supports Node.js, Python and Ruby.

Security

Kubeless is a Kubernetes API extension. To deploy functions, users need to have access to a Kubernetes cluster and have enough privileges in a given Kubernetes namespace. Kubernetes [RBAC](#) is the main security mechanism. Functions can be exposed to the public internet using Kubernetes ingress rules which can be secured via TLS.

Tooling

Most of the kubeless tooling is developed in the [kubeless](#) GitHub organization, except the serverless plugin which is hosted upstream in the [serverless](#) GitHub organization.

- The [Kubeless UI](#) can be run locally or in-cluster.
- The [Kubeless CLI](#) is available on the release page.
- The [Serverless plugin](#) developed upstream.

Monitoring, Logging & Alerting

Kubeless monitoring relies on [Prometheus](#). The language runtimes are instrumented to automatically collect metrics for each function. Prometheus will scrape those metrics and display them in the default Prometheus dashboard or a Grafana dashboard.

Logging is available through [fluentd](#). It stores logs in [elastic](#) and displays them through Kibana.

Alerting is configurable using the [Prometheus Alert Manager](#).

Debugging & Diagnostics

Standard Kubernetes debugging and troubleshooting mechanisms.

Pricing

Kubeless is open-source and free, there is no commercial version.

Limitations

[Memory limits](#) can be set via the CLI or the serverless plugin function definition.

Resources

- [Kubeless](#) web site
- [Kubeless](#) GitHub organization

Credits: Nguyen Anh-Tu (ng.tuna@gmail.com)

Case Studies

Case Studies

AWS Lambda and serverless technologies in general, has led to a lot of excitement in the enterprises. A lot of use cases have emerged where serverless is the right fit. There is a need to showcase specific implementations out there in the field, and to highlight success stories around them. The early adopters have paved the path with to innovative solutions to real-life problems using serverless. We want to celebrate those successes. Build trust in serverless. Share our excitement.

To contribute to this section of the guide, please see suggested content breakup in [Issue #29](#)

Glossary

Serverless Terminology

Serverless Platform

- The platform supports administration, permission and user management associated with the development of [Serverless Services](#).

User Account

- A user account represents a single user in the [Serverless Platform](#).
- A user account stores sensitive identifying information such as email, password, username, etc
- A user account is considered the private representation of a user.

User Profile

- A user profile represents the public presentation of a user in the [Serverless Platform](#).
- A user profile stores information that is displayed publicly about a user such as name, bio, github username, etc

Serverless Service

- A Serverless Service consists of source code, a description, and service configuration.
- A service configuration is defined by a single `serverless.yml`

Serverless Framework

- Drives the development/deployment lifecycle of a [Serverless Service](#)
- Used to install [Functions](#) and [Plugins](#)
- Makes use of Plugins to extend the functionality of the framework.

Plugin

- A Plugin is where the business logic lives for the functionality of the [Serverless Framework](#).

Registry

- A service for registering and retrieving code packages for use in the [Serverless Framework](#).
- Registry is primarily used by framework for building [Serverless Services](#).

FDK

- Provides a simple middleware abstraction
- Enables runtime interaction with [Serverless Services](#)
 - Executing functions
 - Dispatching events

Gateway

- Every [Serverless Service](#) automatically has a Gateway provisioned on deployment.
- The Gateway enables execution of the [Functions](#) and handles propagation of [Events](#).

Discovery

- A service for registering and retrieving information on how to dispatch requests to [Serverless Services](#).
- Discovery is primarily used by the [FDK](#) for retrieving info on how to make calls and emit events to [Serverless Services](#) within code.
- A [Serverless Service](#) is automatically registered with Discovery when it is registered with the [Serverless Platform](#).

Function

- Functions represent a basic unit of executable code.

Unprovisioned Functions

- A zip file of a Function which you can download from the [Registry](#) and provision on your own infrastructure (e.g., AWS account).

Provisioned Functions

- A [Function](#) that is provisioned on someone's infrastructure and able to be invoked by someone else.
- All previous versions are expected to be kept available (which is possible w/ FaaS since all versions can be kept on a provider, awaiting an invocation for no additional cost/maintenance).

Event

- Events are a unit of data that are sent between services