Guides & Tutorials
Operations & Observability
Engineering Culture

Serverless (FaaS) vs. Containers – when to pick which?

Philipp Müns Oct 6, 2017

Contrary to popular thought, Serverless (FaaS) and Containers (Container Orchestration) have some pretty important things in common.

You want a modern, future-proof architecture? They both have it. You want to build that slick architecture while also leveraging the latest innovations in distributed systems and large-scale application development? Yep, they both have that too.

It makes it hard to decide which one is best for *you*. But friend, you deserve to know. So we're taking off the gloves and laying it all on the line.

What are the commonalities and distinctions? What are the advantages and disadvantages of each?

It's serverless computing vs containerization, right now. Read on.

How did we get here?

Before we jump right into the details, let's cover some very important history.

1. Physical servers

We used to build our own infrastructure in the form of physical servers. We set up those machines, deployed our code on them, scaled them and maintained them. The whole thing was a manual process, and pretty slow to boot.

2. Server clusters and VMs

Using a single physical server for one application was a waste of resources. So, we evolved our infrastructure thinking and combined multiple physical servers into a cluster.

We used those so-called 'virtual machines' to run multiple applications in isolation on top of this infrastructure. Deployment and management got way faster and easier. However, server administration was still necessary and largely very manual.

3. Entering the cloud (IaaS)

Setting up and operating your own datacenter came with new operational challenges; cloud computing began to tackle those issues.

Why not rent your servers and operational services individually, for a monthly fee? This approach made it way easier to scale up or down, and let teams move faster.

4. PaaS

While cloud environments made it convenient to build large-scale applications, they still came saddled with the downsides of manual administration:

"Are the latest security fixes installed?"

"When should we scale down/up?"

"How many more servers do we need?"

Wouldn't it be great if all those administrative hassles were taken off of our plates, and we could simply focus on applications and business value?

Yep! That's what some other folks started thinking, too.

In corner 1: Containers

Wouldn't it be nice if one could pack the application, with alllllll its dependencies, into a dedicated box and run it anywhere? No matter what software dependencies the host system has installed, or where and what the host system actually is?

That's the idea of containerization. Create a container which has all the required dependencies pre-installed, put your application code inside of it and run it everywhere the container runtime is installed. No more devs saying: "Well, it works on my machine!"

Containerization gained attention when it came to light that Google used such technologies to power some of their services (such as Gmail or Maps). Using containers was initially pretty cumbersome, however; it required deep knowledge about Linux kernel internals and making home-grown scripts to put an application in a container and run it on a host machine.

Then Dotcloud (a PaaS startup from San Francisco) announced a new tool called Docker at Pycon US 2013. Docker was an easy to use CLI tool which made it possible to manage software containers easily.

Dotcloud then pivoted to become Docker, and Google worked on an OpenSource implementation of the "Borg" container orchestration service, which is called Kubernetes.

More and more enterprises adopted containers, and standards around this new technology got defined. Nowadays, nearly every cloud provider offers a way to host containerized applications on their infrastructure.

Advantages of containers

- Control and Flexibility
- Vendor-agnostic
- Easier migration path
- Portability

Disadvantages of containers

- Administrative work (e.g. apply security fixes for containers)
- Scaling is slower
- Running costs
- Hard to get started
- More manual intervention

In corner 2: Serverless compute (FaaS)

About a year later, AWS introduced the first serverless compute service ever: AWS Lambda.

The most basic premise of a serverless setup is that the whole application--all its business logic--is implemented as *functions* and *events*.

Here's the full break-down. Applications get split up into different functionalities (or services), which are in turn triggered by events. You upload your function code and attach an event source to it.

That's basically it. The cloud provider takes care of the rest and ensures that your functions will always be available and usable, no matter what.

When serverless compute was first introduced in 2014, the workloads were pretty limited and focused around smaller jobs such as image/data manipulation. But then AWS introduced the API Gateway as an event source for Lambda functions.

That changed everything. It became possible to create whole APIs that were powered by serverless compute. More and more AWS services integrated with the Lambda compute offering, making it possible to build even larger, more complex, fully serverless applications.

But what is a serverless application, exactly? In sum, an architecture is serverless if it has these characteristics:

- Event-driven workflow ("If X then Y")
- Pay-per-execution
- Zero administration
- Auto-scaling
- Short-lived, stateless functions

Advantages of serverless

- Zero administration
- Pay-per-execution
- Zero cost for idle time
- Auto-scaling
- Faster time-to-market
- Microservice nature —> Clear codebase separation
- Significantly reduced administration and maintenance burden

Disadvantages of serverless

- No standardization (though the CNCF is working on this)
- "Black box" environment
- Vendor lock-in
- Cold starts
- Complex apps can be hard to build

When to choose what?

Now it's time for the big question:

> "Which technology should I pick for my next project"?

Truthfully, it depends.

When to choose containerization

Containers are great if you need the flexibility to install and use software with specific version requirements. With containers, you can choose the underlying operating system and have full control of the installed programming language and runtime version.

It's even possible to operate containers with different software stacks throughout a large container fleet--especially interesting if you need to migrate an old, legacy system into a containerized environment. As an added bonus, many tools for managing large-scale container set-ups (like Kubernetes) come with all the best practices already baked in.

This flexibility *does* come with an operational price tag, though. Containers still require a lot of maintenance and set-up.

For maximum benefit, you'll need to split up your monolithic application into separate microservices, which in turn need to be rolled out as individual groups of containers. That means you'll need tooling that allows all those containers to talk to each other. You'll also need to do the grunt work of keeping their operating systems current with regular security fixes and other updates.

While you *can* configure the container orchestration platform to automatically handle traffic fluctuations for you (a.k.a, self-healing and auto-scaling), the process of detecting those traffic pattern changes and spinning the containers up or down won't be instantaneous. A complete shutdown where no container-related infrastructure is running at all (e.g. when there's no traffic) will also not be possible. There will always be runtime costs.

When to choose serverless

In that vein, serverless is great if you need traffic pattern changes to be automatically detected and handled instantly. The application is even completely shut down if there's no traffic at all. With serverless applications, you pay only for the resources you use; no usage, no costs.

The serverless developer doesn't have to care about administrating underlying infrastructure; they just need to care about the code and the business value to end users. Iteration can be more rapid, as code can be shipped faster, without set-up or provisioning. In fact, because the underlying infrastructure is abstracted, the developer may not even know what it looks like. They won't really need to.

But currently, there are some limitations with vendor support and ecosystem lock-in. Programming languages and runtimes are limited to whichever the provider supports (though there are some workarounds (or "shims") available to overcome those restrictions). Event sources (which trigger all your functions) are usually services that the specific cloud provider offers.

Reasoning about all the individual pieces of the application stack becomes harder when the infrastructure and the code are so separate. Serverless is a bit more new, and its tools still have room to evolve. That's what we're actively working on here at Serverless.com, anyway. 😉

Final verdict?

Of course, this will be an oversimplification. The real world is always more complex. But your rule of thumb?

Choose containers and container orchestrators when you need flexibility, or when you need to migrate legacy services. Choose serverless when you need speed of development, automatic scaling and significantly lowered runtime costs.

Related articles:

- Why we switched from Docker to Serverless