Containers and Serverless

Oscar Rosquist

DD2482 - Automated Software Testing and DevOps

2019-04-30

1 Introduction

During the last couple of years, running applications in containers have grown more and more popular. Another popular way of hosting an application is by doing it with serverless computing. In this paper I intend to discuss the difference between these two. The different benefits and drawbacks they both have. I will however limit this paper to only discuss and compare the differences between them when planning a new application. To be able to achieve this I will first explain what both of them are, how they work and what use cases they have. At the end I will also hypothesize about a possible future of these tools/technologies.

2 Container Virtualization

So the first idea on how you would run an application is to simply run it directly on a computer/server. There is not much of virtualization at this point. Lets say you're working for a company that's going to deploy a new application. Then you would ideally want a new server environment so that you are sure that no old application will disrupt the environment, clog the resources or depend on different versions of a specific software making the environment incompatible with the application. So the logical solution here is to buy a new server for your app to run on. This obviously brings with it the cost of buying the server, having it be delivered, set up, maintained and powered. The next step up would be virtualizing the environment. This is where containers can provide a great benefit. They offer a virtualized environment for the app to run in. There exists however, different types of commonly used technologies for virtualization. Namely Virtual Machines and Containers. They both offer different types of virtualization using different methodologies illustrated in figure 1

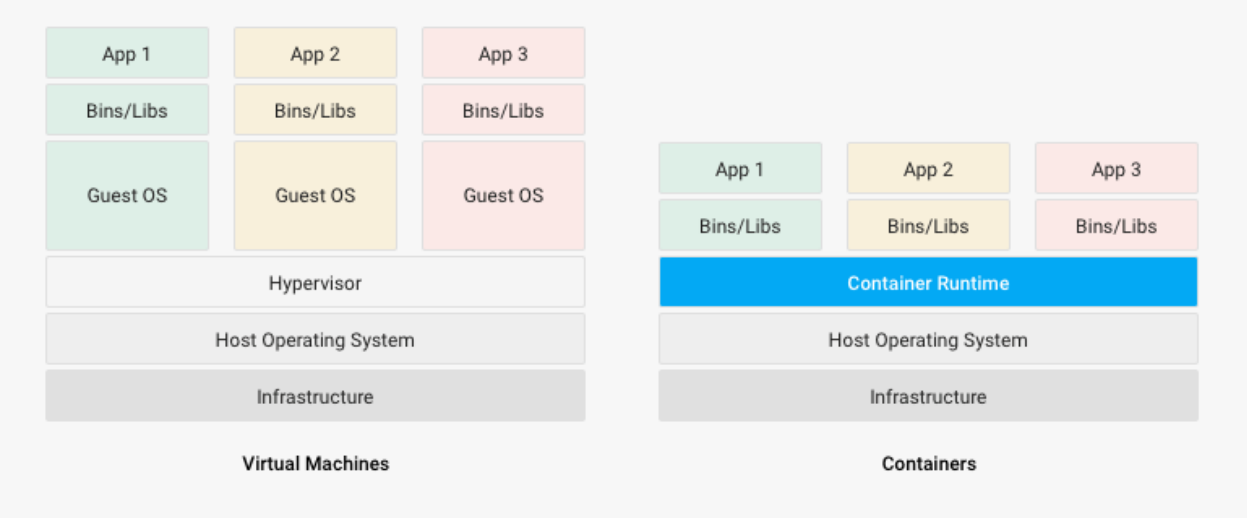

Figure 1: Difference between VMs and Containers [6]

The left part shows three Virtual Machines running on a single physical machine. They utilizes hypervisor virtualization which splits up the physical resources and turns it into virtual hardware which becomes the virtual machines. On top of the virtual hardware, an OS is installed and then the environment with the application at the top just like if it were running directly on the machine. So a VM virtualizes the hardware resources.

This is a great improvement on top of the earlier system with just running the application directly on the server. It isolates the apps from the host machine which reduces the risk of any app making it unstable. They can be stopped and started with the ability to save the state of it. In case of the app causing an unrecoverable problem in the Virtual Machine, it's possible to revert the VM to a previously known stable snapshot of it. They are also easy to replicate in case the user want to scale up the app or share it. [8]

The right side of the figure illustrates how containers work and shows three containers running on one physical machine. Instead of splitting the physical resources, the container runtime split the operating systems resources. They split the process name space, the network stack, the file system hierarchy and so on. It creates a virtualized operating system for the application to run in. [6] So every container gets their own Process ID 1 and their own root file system etc. To sum this up: hypervisor virtualization virtualizes physical resources and builds a Virtual Machine while the container runtime creates virtual operating systems and assigns one to each container.

Apart from possessing the same advantages as Virtual Machines, containers are also very lightweight compared to VMs since they only virtualize the parts of the OS needed. They also have more of the native performance of the host machine available to them, a very short start up time, usually in a few seconds and requires less memory space. [2]

Running your app in a container makes it more portable because it will be able to run anywhere the container runtime works. The app will always be running in the same environment even though the underlying physical machine could be vastly different, thanks to the container. The container wraps your app with everything it needs to run like configuration files and dependencies. [7]

Containers provide effective process isolation and resource sharing. If one application within one container crash, the other containers will be unaffected.[7]

They are also very easy and fast to start, stop, replicate or destroy which also makes it very easy to scale an application by replicating a container. Having the environment for an app configured as a container makes it easier for developers to run and test in an environment that's the same as real live production environment. [7]

So for a company that wants to deploy a new application or service, the logical solution would not be to buy a new server anymore, but rather to deploy it with a container, next to an already existing container. This saves cost by not needing to buy hardware, not having to wait for it to be set up and having fewer servers overall to power and maintain.

Switching over to containers may however not bring the same advantage to your company if you're already using another virtualization method like Virtual Machines. Moving applications from virtual machines to containers may give some benefits but not substantial enough in my point of view to justify it. Here the effort to change infrastructure needs to be compared with the improvement working with containers bring. It could however be a good strategy for new applications.

Adopting the use of containers do however require a great investment in knowledge about the container technologies and tool sets. The developers within a company will need to learn how to use containers and their tools to effectively and securely use it. This can bring a lot of cost in either training courses for the developers, new hires or simply a lower work output when adjusting to the change. The modern container technology and tools are also new and still in development which means they are subject to change.[9] The benefits and drawbacks of containers can be summarized in the following way:

Benefits

- Portability. The containers can run on almost any platform, as long as it can run the container runtime. Since containers wraps all the application needs to be able to run, it's enough to build it once to be able to run it anywhere.
- Process isolation and resource sharing. Containers are able to run on the same hardware and share the same resources without interfering with each other in the case of technical problem och security issue.
- Scaling. You can easily scale an application during times of heavier load by running more containers.
- Environment remains consistent. Because the container itself doesn't change based on which platform it runs on, there will be less complications when switching the underlying environment.

Drawbacks

- Knowledge of containers needed. The developers within the company will need to learn how to work efficiently with containers.
- Performance will not reflect 100% of the machine payed for.
- Persistent data storage is complicated. Since containers run in memory, once shut down, the data disappears.

One example of one of the most popular container technologies is Docker. They utilize something they call images which are a specification of a how a container should be built and what to include. Example:

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

The image created by this specification file will consist of four layers, one being created by each line. Each layer can then be reused which saves time when for example changing an image. In the example above, when changing the last line, only the last layer will need to be rebuilt, and the first three will instead be reused. Another idea the was popularized with Docker is having an image repository called Docker hub. Using this repository, the users can download other peoples images and build their own on top of them. This again is a strategy that makes images much more reusable. [4]

3 Serverless Computing

Serverless computing is really a misnomer because it still involves servers. The name refers to a cloud execution model where it's the cloud provider that runs all the server and dynamically manages allocation of machine resources.[1][11][3] Another definition I liked personally is how Rachel Stephens from Redmonk describes it, "Serverless: managed services that scale to zero" [10]. This says a lot with very little. Serverless are managed services, so somebody else are running them for you, and they are able to be turned off completely. Serverless computing comes in several different forms, of which some are the following:

- Software as a Service (SaaS) - Provides software without any installations
- Infrastructure as a Service (IaaS) - On demand infrastructure
- Function as a Service (FaaS) - Pay-per-use code execution

Serverless then becomes the idea to abstract away the operations of the infrastructure and environment around your application. The goal is to give the consumer more opportunity to focus on building the software and time to market rather then building and managing infrastructure and services like authentication, databases or Email senders. What you do in a serverless system is the following [1]:

- Writing code - but not much, the more code you write means the more you have to maintain, the more vulnerabilities you have and the more things can go wrong. A lot of the code is not actual business logic but instead components to handle authentication, authorization, communication between components etc. All of which are available as SaaS.
- Define triggers - Many managed services within a serverless system works with triggers and creates sort of a chain that can for example instantiate instances of compute.
- Write functions that connect managed services together - this is what forms complex applications and systems.
- Pay for consumption and not allocation. A lot of the managed services are pay-per-use, so when nobody uses your app, services cost nothing.

So you are only responsible for your application itself while the infrastructure is fully managed. This also means that if there is downtime of the infrastructure, you will rely on someone else to fix it, but you also get a modern, up to date environment where you only need to deploy code. Your architecture depends on events and stateless computing which means you most likely have to re-architect an existing app to get it up and running in a serverless environment. And when nothing happens, you pay nothing. You also get a very fine-grained billing visibility so you can clearly see what each component cost, something that can be difficult otherwise. [1][3]

So if comparing with the original setup with managing your own server and application discussed in 2, they differ in the following ways:

Serverless setup	**Original setup**
Program runs based on defined events and triggers	Program runs until it's stopped
Maximum memory size of usually a few gigabytes	Maximum potential memory size of many terabytes
Maximum runtime in a few minutes	No limit of runtime
OS and machine determined by provider	OS and machine determined by user
Provider responsible for scaling	User responsible for scaling

A serverless setup brings the same issues as switching to using containers with needing your developers to have a deep knowledge of the tools and services to be effective with them. Maybe even more so since this changes the development environment more than running an application in containers. We do however get some additional benefits and drawback when working with serverless that are important to mention.

Benefits of serverless

- Cost based upon code execution and service usage measured in milliseconds. In other words, cost goes down with application usage and the billing is fine-grained.
- Less infrastructure to manage and be responsible for. This means less cost from employees and a reduced liability.
- Reduced operational costs.
- Encourages a microservice architecture. Because of how serverless and FaaS works, the logical way of building an application will be with small services that typically only has one purpose.
- Fast to set up and fast to fail. When no hardware needs to be delivered and configured, you can quickly set up and testing stuff out.
- Scalable. The services all scale dependent on how much usage there is.

Drawbacks of serverless

- Reduced overall control. Giving up managing the infrastructure also means relying on a third part to handle everything related to it. From decision making to downtime handling to security.

- Vendor lock-in. The cloud providers naturally wants you to only use their cloud services, which means there will be some vendor lock-in happening. Both from services not being as compatible with other cloud providers and also from the developers only getting knowledge about how to use one providers' services.
- New and still developing technology which results in unclear best-practices.
- Architectural complexity. More services integrating together usually means a higher complexity of the system as a whole.
- More difficult to test locally. If the application builds and rely upon cloud services, running tests on the app can become more tricky than if you could have the hosting environment running locally.
- Duration of runtime is limited. It limits what can be done through functions if they can only run for a few minutes.
- Increased latency. The services within the application will be communicating in a network and having a large app with many different components can add up to a rather large latency.
- Deep knowledge of the cloud providers' tool set and offered services is needed by the developers to be able to sufficiently develop new application in it.

4 Discussion

It may seem like comparing containers and serverless is like comparing apples and oranges and it is somewhat correct. Containers is a virtualization technology that virtualizes the operative systems resources while serverless is a cloud execution model that offers developers and companies a way of abstracting away managing any infrastructure. In fact you could run containers in the cloud and having the infrastructure of you containers being operated by a cloud provider. This would however not be a serverless application since it would only move the application to another server that you don't own. It's also not scalable in the same way. For example, it doesn't scale down to zero if nobody is using it, it still continues to run, and it doesn't necessarily run based on triggers and events either.

We can however compare these two when comparing which one to choose for development of a new application. If the plan is a service based application with no need for controlling the hardware that is running, maybe a web app, serverless is the better choice. Serverless will have it so that the developer can set up very fast, doesn't have to manage any infrastructure and the services will scale up and down depending on the load on the system. When the services is not used, they also doesn't cost anything. If we would instead deploy it using our own server with containers, it would still be a fast set up except for the fact that we need to configure the server. But to actually get it running in a container is also fast and this factor is mostly dependent on the developers experience and expertise within the two technologies. But because needing to allocate a server which then needs to be configured, serverless is according to me, the winner. You would also have to maintain and operate your own server which means having employees to do that and paying for a constantly running server. So using containers for this purpose would also cost more. However, running serverless is less portable compared to a container solution. Because it will entail some vendor-lock in to the cloud providers platform and you will also have a more difficult time to test it locally. Both of these are strong points for the container solution but all in all I think serverless is the overall better solution in this case.

If we instead look at an application that requires running for a longer amount of time, maybe some batch program or an application that performs fuzzing jobs on other applications. So you know it will be running for a long time and it wont be handling any events or triggers. You may also need to have a large amount of memory and maybe you also need to be able to control the hardware and OS running the app because the information is highly confidential. In this case being locked in by a third part provider, having them being in control of that hardware and security of it and their network could pose an issue. Large cloud providers spend a lot of money and resources securing their cloud infrastructure but it also makes them larger targets. Cloud providers can also offer batch processing services but the reduced control, reduced portability,

vendor lock, and runtime limits of serverless core functions means that for these applications, a container solution makes more sense than serverless.

A lot of large modern systems nowadays are being built as microservices and the need for large, always running applications are more of a niche solution than the norm. Therefore running applications using a serverless solution, I believe will become more and more popular in the future. The cost saving, speed of development, scalability and reduced liability will be driving this change. Containers will still grow and see their use cases but with additional frameworks like Serverless Framework [5], that supports multiple cloud providers which reduces the vendor lock in for the consumer, Serverless solutions become more and more reliable.

The take home of this paper should be that both of these technologies are powerful and will most likely have a large spot in the market in the future. While serverless is more limiting in terms of what is possible to do with it right now, it can still offer great value when used properly and perhaps even more in the future. Containers are a great bet to go with at the moment but I believe that because of the perks of outsourcing the infrastructure operations, running them on your own servers will most likely be left in the past unless the data needs to be handled locally.

References

[1] Faizan Bashir. *What is Serverless Architecture? What are its Pros and Cons?* URL: https://hackernoon.com/what-is-serverless-architecture-what-are-its-pros-and-cons-cc4b804022e9.

[2] Roderick Bauer. *What's the Diff: VMs vs Containers*. URL: https://www.backblaze.com/blog/vm-vs-containers/.

[3] Cloudflare. *What Is Serverless Computing? | Serverless Definition*. URL: https://www.cloudflare.com/learning/serverless/what-is-serverless/.

[4] Docker documentation. *About images, containers, and storage drivers*. URL: https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/.

[5] Serverless Framework. *Serverless Framework*. URL: https://serverless.com/.

[6] Google. *Containers at Google*. URL: https://cloud.google.com/containers/.

[7] Kumina. *Top 7 benefits of using containers*. URL: https://blog.kumina.nl/2017/04/the-benefits-of-containers-and-container-technology/.

[8] Brian Linkletter. *Why use a Virtual Machine?* URL: https://www.brianlinkletter.com/why-use-a-virtual-machine/.

[9] Tom Smith. *Container Issues*. URL: https://dzone.com/articles/container-issues.

[10] Rachel Stephens. *What is serverless*. URL: https://twitter.com/rstephensme/status/1073669252320772102.

[11] Wikipedia. *Serverless computing*. URL: https://en.wikipedia.org/wiki/Serverless_computing.