Serverless Computing

This lesson discusses how serverless computing evolved, what problems it has solved, and what are some drawbacks of it.

We'll cover the following

> ⚠ At the time of this writing (December 2019), the examples in this chapter work only in a **GKE** cluster. Feel free to monitor the issue 4668 for more info.

We already saw how we could run the serverless flavor of Jenkins X. That helped with many things, like better resource utilization and scalability.

- *Can we do something similar with our applications?*
- *Can we scale them to zero when no one is using them?*
- *Can we scale them up when the number of concurrent requests increases?*
- *Can we make our applications serverless?*

Let's start from the beginning and discuss serverless computing.

What is serverless computing?

To understand serverless computing, we need to understand the challenges we're facing with more *traditional* types of deployments of our applications.

Challenges faced in traditional deployments

A long time ago, most of us were deploying our apps directly to servers.

- We had to decide the size (**memory and CPU**) of the nodes where our applications would run,
- We had to **create** those servers,
- And we had to **maintain** them.

The emergence of cloud computing

The situation improved with the emergence of **cloud computing**. We still had to do all those things, but they were much easier due to the simplicity of the APIs and the services cloud vendors gave us. Suddenly, we had (a perception of) *infinite resources*, and all we had to do was run a command, and a few minutes later, the servers (VMs) we needed would materialize.

Things became faster and easier, but that didn't remove the tasks of **creating** and **maintaining** servers. Instead, that made them more straightforward. Concepts like **immutability** become mainstream. As a result, we got the following:

- We got much-needed **reliability**,
- We drastically reduced drastically **lean time**,
- We started to reap the benefits of **elasticity**.

Still, some important questions were left unanswered.

- *Should we keep our servers running even when our applications are not serving any requests?*
- *If we shouldn't, how can we ensure that they are readily available when we do need them?*
- *Who should be responsible for the maintenance of those servers?*
- *Is it our infrastructure department, is it our cloud provider, or can we build a system that will do that for us without human intervention?*

Emergence of containers and schedulers

Things changed with the emergence of containers and schedulers. After a few years of uncertainty from having too many options on the table, the situation stabilized around Kubernetes that became the *de-facto standard*.

Solutions for serverless computing

At roughly the same time as the rise in the popularity of containers and schedulers, solutions for serverless computing concepts started to materialize. Those solutions were not related to each other during those first few years.

- **Kubernetes** provided us with the means to run **microservices** as well as more **traditional types of applications**.
- While **serverless** focused on running **functions**, often only a few lines of code.

> 🔍 The name *serverless* is **misleading**, giving the impression that there are no servers involved. They certainly still are, but the concept and the solutions implementing them allow us to ignore their existence.

The major cloud providers (**AWS**, **Microsoft Azure**, and **Google**) all came up with solutions for serverless computing. Developers could focus on writing functions with a few additional lines of code specific to the serverless computing vendor we choose. Everything else required for running and scaling those functions becomes transparent.

Downside of the serverless computing

Not everything is amazing in the serverless world. The number of use-cases that can be fulfilled with functions, as opposed to applications, is limited. Even when we do have enough use-cases to make serverless computing a worthwhile effort, a more significant concern is lurking just around the corner.

We are likely going to be locked into a vendor, given that none of them implement any type of industry standard. No matter whether we choose **AWS Lambda**, **Azure Functions**, or **Google Cloud Functions**, the code we write will not be portable from one vendor to another.

That does not mean that there are no serverless frameworks that are not tied to a specific cloud provider. There are, but we'd need to maintain them ourselves, be it on-prem or inside clusters running in a public cloud, and that removes one of the most essential benefits of serverless concepts.

That's where Kubernetes comes into play. We will discuss that in the next lesson.