Twitter Thread by Oliver Jumpertz

Oliver Jumpertz
@oliverjumpertz

Kubernetes vs Serverless offerings

Why would you need Kubernetes when there are offerings like Vercel, Netlify, or AWS Lambda/Amplify that basically manage everything for you and offer even more?

Well, let's try to look at both approaches and draw our own conclusions!

■■

1■■ A quick look at Kubernetes

Kubernetes is a container orchestrator and thus needs containers to begin with. It's a paradigm shift to more traditional software development, where components are developed, and then deployed to bare metal machines or VMs.

There are additional steps now: Making sure your application is suited to be containerized (12-factor apps, I look at you: https://t.co/nuH4dmpUmf), containerizing the application, following some pretty well-proven standards, and then pushing the image to a registry.

After all that, you need to write specs which instruct Kubernetes what the desired state of your application is, and finally let Kubernetes do its work. It's certainly not a NoOps platform, as you'll still need people knowing what they do and how to handle Kubernetes.

■

2■■ A quick look at (some!) serverless offerings

The offer is pretty simple: You write the code, the platform handles everything else for you. It's basically leaning far to the NoOps side. There is not much to manage anymore.

Take your Next.js / Nuxt.js app, point the ...

... platform (Vercel e.g.) to the repository and then it does its magic. It handles deployment, it handles scaling, it offers metrics, it offers a lot. Sounds like a great offering, doesn't it?

Or take Amplify for example. It offers an integrated full-stack experience.

Simply use the CLI to add an API, or drop your functions into a special folder and let Amplify handle setting up Lambda for you. No need to do everything manually anymore, and also nearly a NoOps experience. Devs can focus on what they want to focus on.

■

3■■ Understanding the author's perspective

Before we're going into a comparison, I want you to understand my perspective. As objective as I try to be, I have my own experience which may differ vastly from yours.

So always keep this in mind when reading on!

I've designed and deployed many production-grade systems on Kubernetes, and I've designed and deployed fewer serverless apps.

The workloads I dealt with were more often suited to a Kubernetes deployment, with traditional micro services, handling infrastructure ...

... components like databases, message brokers, and such on our own.

I used serverless especially when we were building the surrounding systems of mobile or web apps, integrating with existing systems which did the heavy lifting.

■

4■■ Some Pros and Cons of serverless

There is of course more to serverless than only Vercel, Netlify, AWS Amplify/Lambda/CDK, Azure Functions or whatever the offer is.

You can have serverless databases, serverless message brokers, and much more.

Simply look at the catalogue, pick what you need, and with most offerings: Only pay for what you use!

The developer experience is great, because using those offerings is relatively simple. Sometimes there's a good UI or a CLI which helps a lot.

Devs can focus on the code while the provider handles everything else for you. The simpler your project, the easier the usage gets.

A Next.js / Nuxt.js app without any backend functionality besides SSR can be deployed in 5 minutes and scales and scales and scales.

But there are also downsides.

You can either decide to pick wildly through the whole catalogue and have your database hosted here, while your app is hosted there, and your messaging service is somewhere completely different.

Or you go the vendor lock-in route.

Already on AWS?
Need a database? DynamoDB!
Need a message queue? SQS!
Need backend functionality? Lambda!
Need storage? S3!

And now you have branded code. You won't be able to move from AWS to let's say Azure easily.

As long as you don't plan to or need to move, that's no problem at all. But what if the need ever arises?

Well, that's a whole migration project then which may, depending on the size of your system, take quite some time.

Another problem arises when you have backend-heavy applications.

If you ever considered the management of 50 micro services to be difficult, what about 360 lambda functions? You'll need good documentation to be able to recall all of them and what each and every one does.

■

5■■ Some Pros and Cons of Kubernetes

Kubernetes itself is a container orchestrator, as we already learned. It works with all sorts of containers. Every application that can be containerized can be run on K8s.

You won't need to set it up manually nowadays. There is EKS, a fully managed K8s service on AWS, AKS (K8s on Azure), or GKE (K8s on Google Cloud). So at least distributing the infrastructure comes close to a serverless offering.

Where Kubernetes shines is especially standardization. As long as you don't use any vendor-specifics, you can take your EKS cluster and simply move it to AKS or GKE. All your yaml specs will still work and you'll have the same system running in the end.

Another advantage is flexibility.

View Kubernetes as a blank sheet of papers. You can write nearly everything on it, or remove it again if you don't want it anymore. And this offers a lot more flexibility.

Need a database? You can use one offered by your cloud provider or deploy your own MySQL (e.g.) instances.

Need messaging? Deploy your own RabbitMQ or Kafka or, again, use something offered by your cloud provider.

Need...? I think you see where this is going.

In the end, you can deploy your favorite (open source) software (as long as its license permits it).

And if traditional micro services aren't for you, you can even deploy your own serverless function infrastructure (https://t.co/ynaXXliREv)!

But of course, all of this comes at a cost. You don't get all of that for free. You'll need to work for it. There is mostly no one-click solution to deploy all of those awesome components you want within your cluster.

You'll need to take care of storing your logs somewhere yourself, you'll need to deploy your databases yourself, you'll ... I once again think you see where this is going.

Simply put: You need people doing this for you and it's complex.

It's nothing an application developer should do next to writing business logic.

So you'll need DevOps / Site Reliability Engineers that handle at least CI/CD and all those infrastructure components for you. And optimally, they'll take care of everything related to the cluster.

Which means:
- Monitoring
- Automating
- Deployments
- Disaster recovery
- Alerting
- Upgrading components
- And a lot more

■

6■■ And...which one is...better now?

You hate those answers, I know, but it is how it is.
Neither does Kubernetes win, nor does serverless.

They each fill a specific niche in which they are good at.

Serverless offerings are pretty great when you need that development speed and ease of use. If you plan to make that system really big later, and can plan in some time to do a rewrite at some time, start with serverless.

Sometimes you can even stay there forever and the need to migrate never arises. In this case you've saved a lot of time and probably money and made a good decision.

And it's especially a good choice to use serverless when you're developing a mobile or web app.

Kubernetes on the other hand will shine when you have very large systems with many teams involved. It will offer you the flexibility to chose your components to your likings. No need to wait for a provider to offer something as a service that you'd like to have.

Through its flexibility, you'll get many things like A/B testing for very little cost (it's a mere gateway config setting).

It's also great for backend-heavy systems, with a lot of APIs that do not only exist to fulfill the needs of mobile and a web app.

And many serverless offerings you see could be running on Kubernetes under the hood. So you would be right to assume that Kubernetes is, of course, more low-level.

■

7■■ Final conclusion

This was only a very high-level overview and comparison. Going very in-depth is nearly impossible, as the Kubernetes and serverless ecosystems are both pretty large nowadays.

It would simply be nearly impossible to compare everything highly detailed.

The statement set, however, is still true in my humble opinion.

Both have their right to exist, and both cover their own niche, so choose what suits your use case and your team's skills best when you have to decide!