What Do We Expect from Deployments?

In this lesson we learn about the features of a good deployment strategy.

We'll cover the following

Before we dive into some of the deployment strategies, we might want to set some expectations that will guide us through our choices. But, before we do that, let's try to define what a deployment is.

What is deployment?

Traditionally, a deployment is a process through which we **install new applications** into our servers or **update those that are already running** with new releases.

That was, more or less, what we were doing from the beginning of the history of our industry, and that is, in essence, what we're doing today. But, as we've evolved, our requirements have been changing as well. Today, to say that all we expect is for our releases to run is an understatement. Today we want so much more, and we have technology that can help us fulfill those desires. *So, what does **much more** mean today?*

Requirements and features to expect from

Requirements and features to expect from deployment

Depending on who you speak with, you will get a different list of *desires*. What follows is what I believe to be essential, and what I observed the companies I worked for emphasizing. Without further ado, the requirements, excluding the obvious that applications should be running inside the cluster, are as follows.

Fault-tolerance

Applications should be **fault-tolerant.** If an instance of the application dies, it should be brought back up. If the node where an application is running dies, the application should be moved to a healthy node. Even if a whole data center goes down, the system should be able to move the applications that were running there into a healthy one. An alternative would be to recreate the failed nodes or data centers with precisely the same apps that were running there before the outage. However, that is too slow and, frankly speaking, we moved away from that concept the moment we adopted schedulers. That doesn't mean that failed nodes and failed data centers should not recuperate, but rather that we should not wait for infrastructure to get back to normal. Instead, we should run failed applications, no matter the cause, on healthy nodes as long as there is enough available capacity.

Fault tolerance might be the most crucial requirement of all. If our application is not running, our users cannot use it. That results in dissatisfaction, loss of profit, churn, and quite a few other adverse outcomes. Still, we will not use fault tolerance as a criterion because Kubernetes makes nearly everything fault-tolerant. As long as it has enough available capacity, our applications will run. So, even though that's an essential requirement, it's off the table because we are fulfilling it no matter the deployment strategy we choose. That is not to say that there is no chance for an application not to recuperate from a failure, but instead that Kubernetes provides a reasonable guarantee of fault tolerance. If things do go terribly wrong, we are likely going to have to do some manual actions no matter which deployment strategy we choose.

> Long story short, **fault-tolerance is a given with Kubernetes**, and there's no need to think about it in terms of deployment strategies.

The next in line is **high availability**, and that's a trickier one.

High availability

Being fault-tolerant means that the system will recuperate from failure, not that there will be no downtime. If our application goes down, a few moments later, it will be up-and-running again. Still, those few moments can result in downtime. Depending on many factors, few moments can translate into milliseconds, seconds, minutes, hours, or even days. It is certainly not the same whether our application is unavailable during milliseconds as opposed to hours. Still, for the sake of brevity, we'll assume that any downtime is bad and look at things as black and white. Either there is, or there isn't downtime. What changed over time is what considerable means. In the past, having 99% availability was a worthy goal for many. Today, that figure is unacceptable. Today we are talking about how many nines there are after the decimal. For some, 99.99% uptime is acceptable. For others, that could be 99.99999%.

Now, you might say: "my business is important; therefore, I want 100% uptime." If anyone says that to you, feel free to respond with: "you have no idea what you're talking about." 100% uptime is impossible, assuming that we mean *real* uptime, and not *my application runs all the time*.

Making sure that our application is always running is not that hard. Making sure that not a single request is ever lost or, our users perceive our application as always being available, is impossible. By the nature of HTTP, some requests will fail. Even if that never happens, the network might go down, storage might fail, or something else might happen. Any of those is bound to produce at least one request without a response or with a 4xx or 5xx message.

All in all, high-availability means that our applications are responding to our users most of the time. By *most*, we mean at least 99.99%. Even that is a very pessimistic number that would result in one failure for each ten thousand requests.

Progressive rollout

What are the common causes of unavailability? We already discussed those that tend to be the first associations, hardware and software failures. However, those are often not the primary causes of unavailability. You might have missed something in your tests, and that might cause a malfunction. More often than not, those are not failures caused by "obvious" bugs but rather by those that manifest themselves after a new release is deployed. I will not tell you that you should make

sure that there are no bugs because that is impossible. Instead, I'll tell you that you should focus on detecting those that sneak into production. It's as important to avoid bugs as to minimize their effect on as few users as possible. So, our next requirement will be that our deployments should reduce the number of users affected by bugs. We'll call it **progressive rollout**. Don't worry if you've never heard that term, we'll explain it in more depth later.

Rollback

The progressive rollout, as you'll see later, does allow us to abort upgrades if something goes wrong. But that might not be enough; we might need to abort deployment of a new release and roll back to the one we had before. So, we'll add **rollback** as yet another requirement.

We'll probably find more requirements directly or indirectly related to high-availability or, to inverse it, to unavailability. For now, we'll leave those aside, and move to yet another vital aspect.

Responsive

We should strive to make our applications **responsive**, and there are many ways to accomplish that. We can design our apps in a certain way, we can avoid congestions and memory leaks, and we can do many other things. However, right now, that's not the focus. We're interested in things that are directly or indirectly related to deployments. With such a limited scope, scalability is the key to responsiveness. If we need more replicas of our application, it should scale up. Similarly, if we don't need as many, it should scale down and free the resources for some other processes if cost savings are not a good enough reason.

Cost-effective

Finally, we'll add one more requirement. It would be nice if our applications only use the necessary resources. We can say that scalability provides that as it can scale up and down, but we might want to take it a step further and say that our applications should not use any resources when they are not in use. We can call that "nothing when idle" or, "serverless".

I'll use this as yet another opportunity to express my dislike of that term given that it implies that there are no servers involved. But, since it is a commonly used one, we'll stick with it. After all, it's still better than calling it function-as-a-service since that is just as misleading as serverless. However, serverless is not the real goal,

What matters is that our solution is **cost-effective**, so that will be our last requirement.

Are those all the requirements we care for? They certainly aren't. But, this text cannot contain an infinite number of words, and we need to focus on something. Those, in my experience, are the most important ones, so we'll stick with them, at least for now.

Another thing we might need to note is that those requirements are interconnected. More often than not, one cannot be accomplished without the other or, in some cases, one facilitates the other and makes it easier to accomplish.

Automation of features

Another thing worth noting is that we'll focus only on automation. For example, I know perfectly well that anything can be rolled back through human intervention. I know that we can extend our pipelines with post-deployment tests followed by a rollback step in case they fail. As a matter of fact, anything can be done with enough time and manpower. But that's not what matters in this discussion. We'll ignore humans and focus only on the things that can be automated and are an integral part of deployment processes. I don't want you to scale your applications, I want the system to do it for you. I don't want you to roll back in case of a failure, I wish the system to do that for you. I don't want you to waste your brain capacity on such trivial tasks, I wish you to spend your time on things that matter and leave the rest to the machines.

Summary

After all that, we can summarize our requirements or features by saying we'd like deployments to result in applications that are running and are:

- **Fault-tolerant,**
- **Highly available,**
- **Responsive,**
- **Rolling out progressively,**
- **Rolling back in case of a failure,**
- **Cost-effective.**

We'll remove *fault tolerance* from the future discussions since Kubernetes provides that out-of-the-box. As for the rest, we have yet to see whether we can accomplish them all and whether a single deployment strategy will give us all those benefits.

There is a strong chance that there is no solution that will provide all those features. Even if we do find such a solution, the chances are that it might not be appropriate for your applications and their architecture. We'll worry about that later. For now, we'll explore some of the commonly used deployment strategies and see which of those requirements they fulfill.

Just as in any other chapter, we'll explore the subject in more depth through practical examples. For that, we need a working Jenkins X cluster as well as an application that we'll use as a guinea pig on which we'll experiment with different deployment strategies. Instructions for creating this cluster can be found in the next lesson.