About the World We Lived In

This lesson discusses the deployment process that we follow, how different environments are managed and the reason for testing at each stage.

We'll cover the following

The necessity to test new releases before deploying them to production is as old as our industry. Over time, we developed elaborate processes aimed at ensuring that our releases are ready for production.

- We were testing them locally.
- Then, we were deploying them to testing environments where we would test them more.
- When we were comfortable with the quality, we were deploying those releases to integration and pre-production environments.
- The final round of validations was done in these environments.

Typically, the closer we were getting to releasing something to production, the more similar our environments were to production. That was a lengthy process that would last for months, sometimes even years.

Managing different environments

Why are we moving our releases through different environments? The answer lies in the difficulties in maintaining production-like environments.

In the past, it took a lot of effort to manage environments, and the more they looked like production, the more work they required. Later on, we adopted configuration management tools like **CFEngine**, **Chef**, **Puppet**, **Ansible**, and quite a few others. They simplified the management of our environments. Still, we kept

the practice of moving our software from one to another as if it was an abandoned child moving from one foster family to another. The main reason why configuration management tools did not solve many problems lies in a misunderstanding of the root cause of the problem.

What made the management of environments challenging is not that we had many of them, nor that production-like clusters are complicated. Instead, the issue was in mutability. No matter how much effort we put in maintaining the state of our clusters, differences would pile up over time. We could not say that one environment is genuinely the same as the other. Without that guarantee, we could not claim that what was tested in one environment would work in another. The risk of experiencing failure after deploying to production was still too high.

Immutability

Over time, we adopted immutability. We learned that things shouldn't be modified at runtime, but instead created anew whenever we need to update something. We started creating VM images that contained new releases and applying rolling updates that would gradually replace the old. But that was slow. It takes time to create a new VM image, and it takes time to instantiate them. There were many other problems with them, but this is neither time nor place to explore them all. What matters is that immutability applied to the VM level brought quite a few improvements, but also some challenges. Our environments became stable, and it was easy to have as many production-like environments as we needed, even though that approach revealed quite a few other issues.

Immutability through containers

Then came containers that took immutability to the next level. They gave us the ability to say that something running in a laptop is the same as something running in a test environment that happens to behave in the same way as in production. Creating a container based on an image produces the same result no matter where it runs. That's not 100% true, but when compared to what we had in the past, containers bring us as close to repeatability as we can get today.

Testing in different environments

So, if containers provide a reasonable guarantee that a release will behave the same no matter the environment it runs in, we can safely say that if it works in

staging, it should work in production. That is especially true if both environments

are in the same cluster. In such a case, **hardware**, **networking**, **storage**, and other infrastructure components are the same, and the only difference is the Kubernetes Namespace something runs in. That should provide a reasonable guarantee that a release tested in staging should work correctly when promoted to production. *Don't you agree?*

>  Even if environments are just different Namespaces in the same cluster, and if our releases are immutable container images, there is still a reasonable chance that we will detect issues only after we promote releases to production.

No matter how well our performance tests are written, production load cannot be reliably replicated. No matter how good we became writing functional tests, real users are unpredictable, and that cannot be reflected in test automation. Tests look for errors we already know about, and we can't test what we don't know. I can go on and on about the differences between production and non-production environments. It all boils down to one having real users, and the other running simulations of what we think "real" people would do.

I'll assume that we agree that production with real users and non-production with I-hope-this-is-what-real-people-do type of simulations are not the same. We can conclude that the only final and definitive confirmation that a release is successful can come from observing how well it is received by "real" users while running in production. That leads us to the need to:

- monitor our production systems
- observe user behavior
- observe error rates
- observe response times
- observe other metrics

Based on that data, we can conclude whether a new release is truly successful or not. We keep it if it is and we roll back if it isn't. Or, even better, we roll forward with improvements and bug fixes. *That's where progressive delivery kicks in.*

The next lesson will give an introduction to **progressive delivery**.