Evaluating Whether Blue-Green Deployments Are Useful

This lesson discusses the Blue-Green strategy, the concept behind it and how to use it.

We'll cover the following

The blue-green deployment strategy

Blue-green deployment is probably the most commonly mentioned "modern" deployment strategy. It was made known by Martin Fowler.

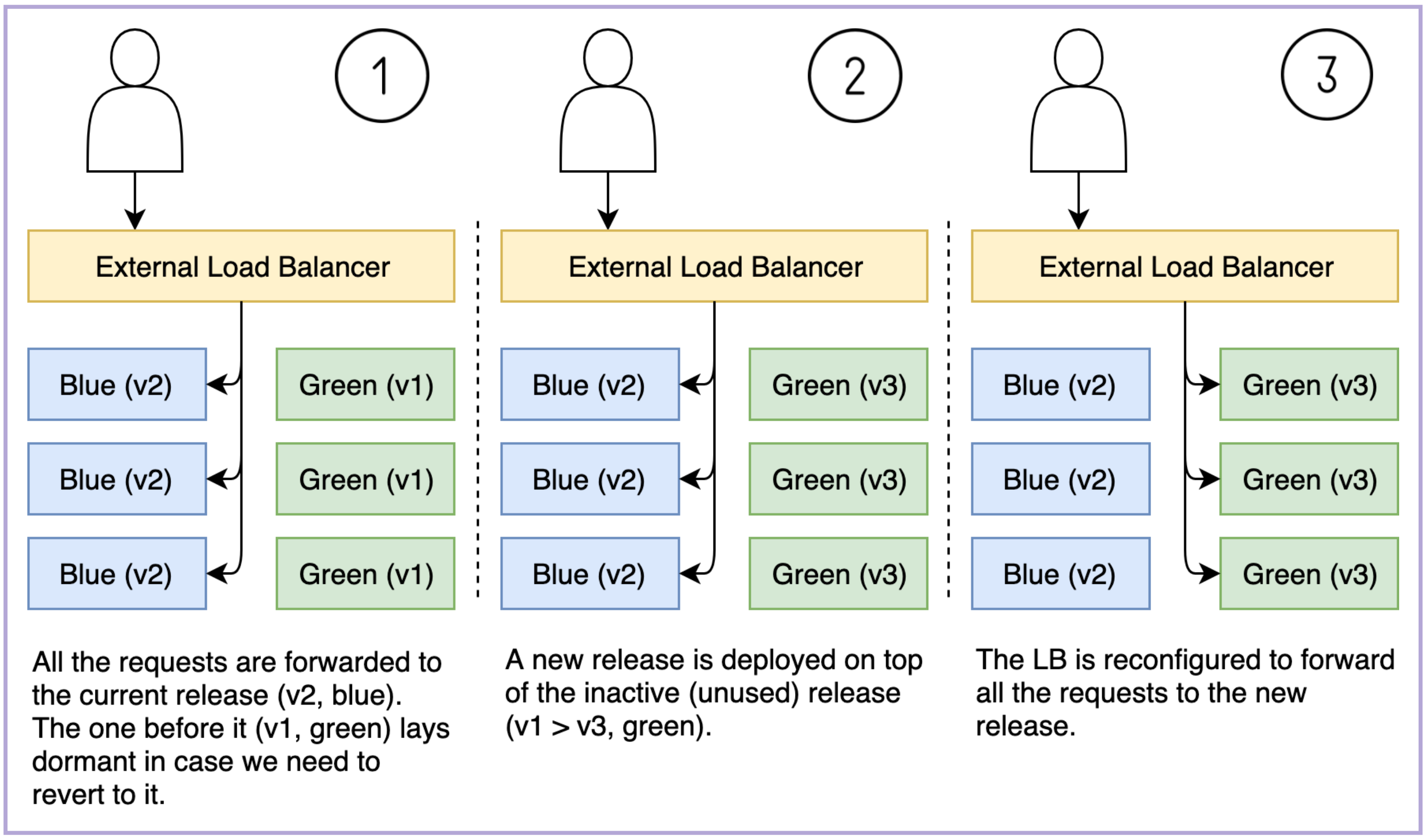
The idea behind blue-green deployments is to run two production releases in parallel. If, for example, the current release is called "blue", the new one would be called "green", and vice versa. Assuming that the load balancer (LB) is currently pointing to the blue release, all we'd need to do to start redirecting users to the new one would be to change the LB to point to green.

We can describe the process through three different stages.

1. Let's say that, right now, all the requests are forwarded to the current release. Let's that that's the blue release with version v2. We'll also imagine that the release before it is running as green and that the version is v1. The green

release lays dormant, mostly wasting resources.

2. When we decide to deploy a new release, we do it by replacing the inactive (dormant) instances. In our case, that would be green instances that are currently running v1 and will be replaced with v3.
3. When all the green instances are running the new release, all that's left is to reconfigure the LB to forward all the requests to green instead of the blue instances. Now the blue release is dormant (unused) and will be the target of the next deployment.


1
External Load Balancer
Blue (v2)
Green (v1)
Blue (v2)
Green (v1)
Blue (v2)
Green (v1)
All the requests are forwarded to the current release (v2, blue). The one before it (v1, green) lays dormant in case we need to revert to it.
2
External Load Balancer
Blue (v2)
Green (v3)
Blue (v2)
Green (v3)
Blue (v2)
Green (v3)
A new release is deployed on top of the inactive (unused) release (v1 > v3, green).
3
External Load Balancer
Blue (v2)
Green (v3)
Blue (v2)
Green (v3)
Blue (v2)
Green (v3)
The LB is reconfigured to forward all the requests to the new release.

Blue-green deployment strategy

If we'd like to revert the release, all we'd have to do is change the LB to point from the active to the inactive set of instances. Or, to use different terminology, we switch it from one color to another.

Blue-green deployments made a lot of sense before. If each of the replicas of our application were running in a separate VM, rolling updates would be much harder to accomplish. On top of that, rolling back (manual or automated) is indeed relatively easy with blue-green given that both releases are running in parallel. All we'd have to do is to reconfigure the LB to point to the old release.

However, we do not live in the past. We are not deploying binaries to VMs but containers to Kubernetes which schedules them inside virtual machines that

constitute the cluster. Running any release is easy and fast. Rolling back containers is as easy as reconfiguring the LB.

Blue-green strategy vs serverless deployments and the `RollingUpdate` strategy

When compared to the serverless deployments and the `RollingUpdate` strategy, blue-green does not bring anything new to the table. In all the cases, multiple replicas are running in parallel, even though that might not be that obvious with blue-green deployments.

People tend to think that switching from blue to green deployment is instant, but that is far from the truth. The moment we change the load balancer to point to the new release, both are being accessed by users, at least for a while. The requests initiated before the switch will continue being processed by the old release, and the new ones will be handled by the new release. In that aspect, blue-green deployments are no different from the `RollingUpdate` strategy. The significant difference is in the cost.

Let's imagine that we have five replicas of our application. If we're using rolling updates, during the deployment process, we will have six, unless we configure it differently. A replica of the new release is created so we have six replicas. After that, a replica of the old release is destroyed, so we're back to five. And so on, the process keeps alternating between five and six replicas, until the whole new release is rolled out and the old one is destroyed. With blue-green deployments, the number of replicas is duplicated. If we keep with five replicas as the example, during the deployment process, we'd have ten (five of the old and five of the new). As you can imagine, the increase in resource consumption is much lower if we increase the number of replicas by one than if we double them. Now, you can say that the increase is not that big given that it lasts only for the duration of the deployment process, but that would not necessarily be true.

One of the cornerstones of blue-green strategy is the ability to roll back by reconfiguring the load balancer to point to the old release. For that, we need the old release to be always up-and-running, and thus have the whole infrastructure requirements doubled permanently. Now, I do not believe that's the reason good enough today. Replacing a container based on one image with a container based on another is almost instant. Instead of running two releases in parallel, we can just as easily and rapidly roll forward to the old release. Today, running two releases in parallel forever and ever is just a waste of resources for no good reason.

Such a waste of resources (for no good reason) is even more evident if we're dealing with a large scale. Imagine that your application requires hundreds of CPU and hundreds of gigabytes of memory. *Do you want to double that knowing that rolling updates give you all the same benefits without such a high cost associated with it?*

Frankly, I think that blue-green was a short blink in the history of deployments. They were instrumental, and they provided the base from which others like rolling updates and canaries were born. Both are much better implementations of the same objectives. Nevertheless, blue-green is so popular that there is a high chance that you will not listen to me and that you want it anyways. I will not show you how to do "strict" blue-green deployments. Instead, I will argue that you were already doing a variation of it through quite a few chapters. I will assume that you want to deploy to production what you already tested so no new builds of binaries should be involved. I will also expect that you do understand that there is no need to keep the old release running so that you can roll back to it. Finally, I will assume that you do not want to have any downtime during the deployment process.

Applying a variation of the blue-green deployment

With all the above-mentioned discussion in mind, let's do a variation of blue-green deployments without actually employing any new strategy or using any additional tools.

Checking the applications running in staging

Now, let's take a look at what's running in the staging environment.

```
jx get applications --env staging
```

The output is as follows.

```
APPLICATION    STAGING PODS URL
jx-progressive 0.0.7   3/3  http://jx-progressive.jx-staging...
```

Defining the blue and the green deployment

For now, we'll assume that whatever is running in production is our blue release and that staging is green. At this moment, you can say that both releases should be running in production to qualify for blue-green deployments. If that's what's going through your brain, remember that "staging" is just the name. It is running in the same cluster as production unless you choose to run Jenkins X environments in different clusters. The only thing that makes the release in staging different from production (apart from different Namespaces) is that it might not be using the production database or to be connected with other applications in production, but to their equivalents in staging. Even that would be an exaggeration since you are (and should be) running in staging the same setup as production. The only difference should be that one has production candidates while the other is the "real" production. If that bothers you, you can easily change the configuration so that an application in staging is using the database in production, be connected with other applications in production, and whatever else you have there.

With the differences between production and staging out of the way, we can say that the application running in staging is the candidate to be deployed in production. We can just as easily call one **blue (production)** and the other one **green (staging)**.

Promoting the staging to production

Now, what comes next will be a slight deviation behind the idea of blue-green deployments. Instead of changing the load balancer (in our case **Ingress**) to point to the staging release (green), we'll promote it to production.

> Please replace `[...]` with the version from the previous output.

```
VERSION=[...]

jx promote jx-progressive \
    --version $VERSION \
    --env production \
    --batch-mode
```

After a while, the promotion will end, and the new (green) release will be running in production. All that's left, if you're running serverless Jenkins X, is to confirm that by watching the activities associated with the production repository.

```
jx get activities \
    --filter environment-jx-rocks-production/master \
    --watch
```

Please press *ctrl+c* to stop watching the activity once you confirm that the build initiated by the previous activity's push of the changes to the master branch.

Inspecting the applications running in production

Now we can take a look at the applications running in production.

```
jx get applications --env production
```

The output should not be a surprise since you already saw the promotion process quite a few times before. It shows that the release version we have in staging is now running in production and that it is accessible through a specific address.

Testing if the release is running

Finally, we'll confirm that the release is indeed running correctly by sending a request.

> Please replace `[...]` with the address of the release in production. You can get it from the output of the previous command.

```
PRODUCTION_ADDR=[...]

curl "$PRODUCTION_ADDR"
```

Surprise, surprise... The output is `Hello from: Jenkins X golang http rolling update`. If you got `503 Service Temporarily Unavailable`, the application is not yet fully operational because you probably did not have anything running in production before the promotion. If that's the case, please wait for a few moments and re-run the `curl` command.

Was that blue-green deployment?

It was, of sorts. We had a release (in staging) running in precisely the same way as if it would run in production. We had the opportunity to test it. We switched our production load from the old to the new release without downtime. The significant difference is that we used `RollingUpdate` for that, but that's not a bad thing. Quite the contrary.

What we did has many of the characteristics of blue-green deployments. On the other hand, we did not strictly follow the blue-green deployment process. We didn't because I believe that it is silly. Kubernetes opened quite a few new possibilities that make blue-green deployments obsolete, inefficient, and wasteful.

Did that make you mad? Are you disappointed that I bashed blue-green deployments? Did you hope to see examples of the "real" blue-green process? If that's what's going through your mind, the only thing I can say is to stick with me for a while longer. We're about to explore progressive delivery and the tools we'll explore can just as well be used to perform blue-green deployments. By the end of this chapter, all you'll have to do is read a bit of documentation and change a few things in a `YAML` file. You'll have "real" blue-green deployment. However, by the time you finish reading this chapter, especially what's coming next, the chances are that you will discard blue-green as well.

Given that we did not execute the *"strict"* blue-green process and that what we used is `RollingUpdate` combined with promotion from one environment to another, we will not discuss the pros and cons of the blue-green strategy. We will not have the table that evaluates which requirements we fulfilled. Instead, we'll jump into progressive delivery as a way to try to address progressive rollout and automated rollbacks. Those are the only two requirements we did not yet obtain fully.

The next lesson will discuss the deployment process that we follow and how different environments are managed.