A Short Introduction to Progressive Delivery

This lesson gives an introduction to progressive delivery and its encompassing methodologies. Canary deployments are also discussed.

We'll cover the following

What is progressive delivery?

Progressive delivery is a term that includes a group of deployment strategies that try to avoid the pitfalls of the all-or-nothing approach. New versions being deployed do not replace existing versions but run in parallel for some time while receiving live production traffic. They are evaluated in terms of correctness and performance before the rollout is considered successful.

Progressive Delivery encompasses methodologies such as:

- *rolling updates*
- *blue-green*
- *canary deployments*

We already used rolling updates for most of our deployments so you should be familiar with at least one flavor of progressive delivery. What is common to all of them is that monitoring and metrics are used to evaluate whether a new version is "safe" or needs to be rolled back. That's the part that our deployments were missing so far or, at least, did not do very well. Even though we did add tests that run during and after deployments to staging and production environments, they

were not communicating findings to the deployment process.

We did manage to have a system that can decide whether the deployment was successful or not, but we need more. We need a system that will run validations during the deployment process and let it decide whether to proceed, to halt, or to roll back. We should be able to roll out a release to a fraction of users, evaluate whether it works well and whether the users are finding it useful. If everything goes well, we should be able to continue extending the percentage of users affected by the new release. All in all, we should be able to roll out gradually, let's say ten percent at a time, run some tests and evaluate results, and, depending on the outcome, choose whether to proceed or to roll back.

Progressive delivery process in commonly used deployment strategies

To make progressive delivery easier to grasp, we should probably go through the high-level process followed for the three most commonly used flavors.

Rolling updates

With rolling updates, not all the instances of our application are updated at the same time, but they are rolled out incrementally. If we have several replicas (containers, virtual machines, etc.) of our application, we would update one at a time and check the metrics before updating the next. In case of issues, we would remove them from the pool and increase the number of instances running the previous version.

Blue-green deployments

Blue-green deployments temporarily create a parallel duplicate set of our application with both the old and new versions running at the same time. We would reconfigure a load balancer or a DNS to start routing all traffic to the new release. Both versions coexist until the new version is validated in production. In some cases, we keep the old release until it is replaced with the new. If there are problems with the new version, the load balancer or DNS is just pointed back to the previous version.

Canary deployments

With canary deployments, new versions are deployed, and only a subset of users

are directed to it using traffic rules in a load balancer or more advanced solutions like service mesh. Users of the new version can be chosen randomly as a

percentage of the total users or using other criteria such as geographic location, headers, employees instead of general users, etc. The new version is evaluated in terms of correctness and performance and, if successful, more users are gradually directed to the new version. If there are issues with the new version or if it doesn't match the expected metrics, the traffic rules are updated to send all traffic back to the previous one.

Canary releases are very similar to rolling updates. In both cases, the new release is gradually rolled out to users. The significant difference is that canary deployments allow us more control over the process. They allow us to decide who sees the new release and who is using the old. They allow us to gradually extend the reach based on the outcome of validations and on evaluating metrics. There are quite a few other differences we'll explore in more detail later through practical examples. For now, what matters is that canary releases bring additional levers to continuous delivery.

Progressive delivery in continuous delivery

> **Progressive delivery makes it easier to adopt continuous delivery**.

It reduces risks of new deployments by limiting the blast radius of any possible issues, known or unknown. It also provides automated ways to rollback to an existing working version.

No matter how much we test a release before deploying it to production, we can never be entirely sure that it will work. Our tests could never fully simulate "real" users' behavior. So, if being 100% certain that a release is valid and will be well received by our users is impossible, the best we can do it to provide a safety net in the form of gradual rollout to production that depends on results of tests, evaluation of metrics, and observation how users receive the new release.

How will we implement canary deployments?

There are quite a few ways we can implement canary deployments, ranging from custom scripts to using ready-to-go tools that can facilitate the process. Given that we do not have time or space to evaluate all the tools we could use, we'll have to

pick a combination.

We will explore how Jenkins X integrates with **Flagger**, **Istio**, and **Prometheus** which, when combined, can be used to facilitate canary deployments. Each will start by getting a small percentage of the traffic and analyzing metrics such as response errors and duration. If these metrics fit a predefined requirement, the deployment of the new release will continue, and more and more traffic will be forwarded to it until everything goes through the new release. If these metrics are not successful for any reason, our deployment will be rolled back and marked as a failure. To do all that, we'll start with a rapid overview of those tools. Just remember that what you'll read next is a high-level overview, not an in-depth description. A whole book can be written on **Istio** alone, and this chapter is already too big.

The next lesson will introduce us to **Flagger**, **Istio**, **Grafana** and **Prometheus**.