**Continuous Deployment Strategies with Kubernetes**

Kubernetes provides a sophisticated platform for making cloud deployments easy. Low-stress deployments are the cornerstone of Continuous Delivery practices. Being able to reliably and safely deploy, rollback and orchestrate software releases allows engineering teams to iterate quickly, experiment and keep the customers happy. All at the same time.

There is a number of industry-standard deployment strategies that make continuous delivery of service-oriented software systems possible. In this series of posts, we will describe a number of existing techniques and strategies and discuss their pros, cons and caveats. For each strategy, we will also provide an example that can be implemented and verified using Codefresh Kubernetes integration features.

# List of strategies:

* Recreate
* Ramped Deployments
* Blue/Green Deployments
* Canary Releases
* A/B Testing

# Setting the Terms Straight

Before we begin it’s important to agree on the terminology.

Kubernetes has a controller object called Deployment. The purpose of this controller is to provide declarative desired state management for underlying Pods and Replica Sets.

Basically a Deployment defines:

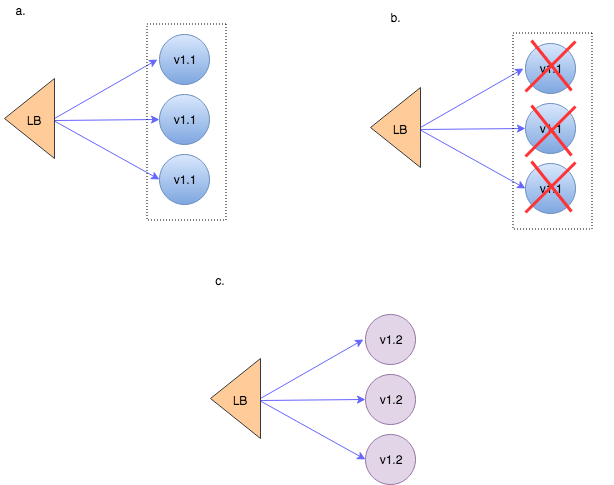
* The Pod to execute (i.e containers, ports, and system resources)
* The number of instances the Pod should have
* All the metadata related to underlying objects

The Deployment object has a property named ‘Strategy’ which can be of 2 types: either ‘Recreate’ or ‘Rolling Update’.

It is important to note that in this post series, when we say deployment strategy we are not referencing the mentioned K8S object property. While we will be showcasing the usage of these K8S technical concepts, what we actually intend to do is describe the general continuous application deployment strategies and how they can be implemented on top of the Kubernetes platform.

With that off our table — let’s begin looking at the strategies.

# Recreate



To be fair — this is not a true continuous deployment strategy. It’s more like the old-fashioned installation/upgrade process. You stop all the running application instances and then spin up the instances with the new version.

(In Kubernetes this is implemented by deployment.spec.strategy.type of the same name — Recreate whereas all existing Pods are killed before new ones are created.)

# The downside:

* This deployment approach involves downtime that occurs, while the old versions are getting brought down and the new versions are starting up.

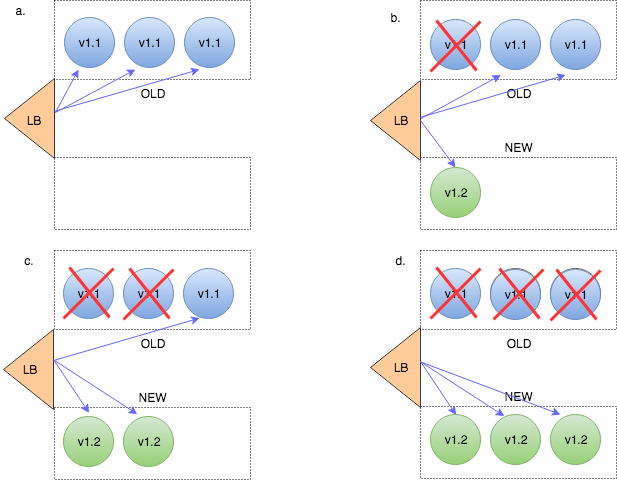
# The upside:

* The upside of this approach is its cleanliness and conceptual simplicity. At no time do we have to manage more than one application version in parallel. No API versioning issues, no data scheme incompatibilities to think about.

# Applicability

This strategy is mostly appropriate for non-critical systems where downtime is acceptable and comes at no significant cost. One case of such a system could be our development environment. It’s important to remember though that if we architect our services for continuous deployment, we should be testing their upgrade and failover behaviours as early as possible. Which means that we better use the same rollout procedure for all environment maturity levels.

# Ramped Deployment



This is basically another term for rolling update. An important thing to note is that this is only applicable when your service is horizontally scaled — i.e running in more than one instance. If this is the case — then instances running the old version get retired as new instances with the new version get spun up.

This should of course be accompanied by at least some kind of a basic health check in order to verify that the new instances are ready to provide services. In Kubernetes these health checks can be defined as a readinessProbe.

Once an instance is considered ready to serve — traffic gets sent to it.

If new instances aren’t found healthy — this strategy provides a smooth rollback procedure: new instances get killed off and old version instances get scaled back to their original capacity.

Note that during the rollout process the old and the new version of our code run side by side. This requires full backward compatibility of our API and schema changes or careful API versioning whenever backward compatibility gets broken.

# The downside:

* Version simultaneousness concerns: Backward compatibility and API versioning aren’t easy to implement and require careful testing.
* System comprehensibility: Issues arising during the transition process will be significantly harder to debug and recreate.
* Resource cost: If we don’t want any impact on the quality of service we’ll need to bring up the new instances while the old ones are still running. This means that there will be a resource usage peak during the rollout process. Dealing with such peaks has become much easier in the age of cloud infrastructure, but still — resources don’t come for free.

# The upside:

* Slow and smooth application code rollout across all servers with integrated health checks and no downtime involved.
* Availability of in-place rollback in case the deployment is unsuccessful.

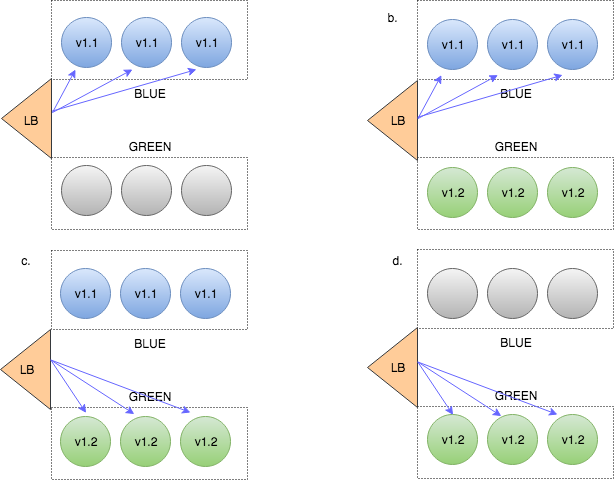
# Applicability

As already stated — ramped deployments only make sense when the deployed application is horizontally scaled.

Additionally, ramped deployments are great when the new versions are backward compatible — both API and data-wise. And when your system is truly service oriented — with clear context boundaries, API versioning and eventual consistency support built-in.

# Blue/Green (aka Red/Black) Deployment

In this approach we always manage 2 versions of our production environment. One of them is considered ‘blue’ — i.e this is the version that is now live. The new versions are always deployed to the ‘green’ replica of the environment. After we run the necessary tests and verifications to make sure the ‘blue’ environment is ready we just flip over the traffic, so ‘green’ becomes ‘blue’ and ‘blue’ becomes ‘green’.



Note that this strategy introduces a new concern we didn’t discuss before — managing multiple deployment environments. While a standard practice in CI/CD — this adds another level of complexity to our deployment pipeline.

# The downside:

* Resource duplication — In this model we’re maintaining at least two full versions of our environment at all times.
* Management overhead: This strategy can become very resource intensive and cumbersome to manage if each environment encompasses multiple service components.
* We can of course create an environment per service, but then the matrix of interrelated environments becomes quite hard to comprehend.
* Data synchronisation between the 2 environments is challenging. Especially while the ‘green’ environment is still under test.

# The upside:

* Very low-stress deployments as we never deploy to production.
* No downtime as traffic redirection occurs on the fly.
* Availability of extensive production-like testing prior to switch-over.
* Rolling back in case something still goes wrong is as easy as just flipping the traffic switch back to the ‘blue’ environment. Data sync taken into account of course.

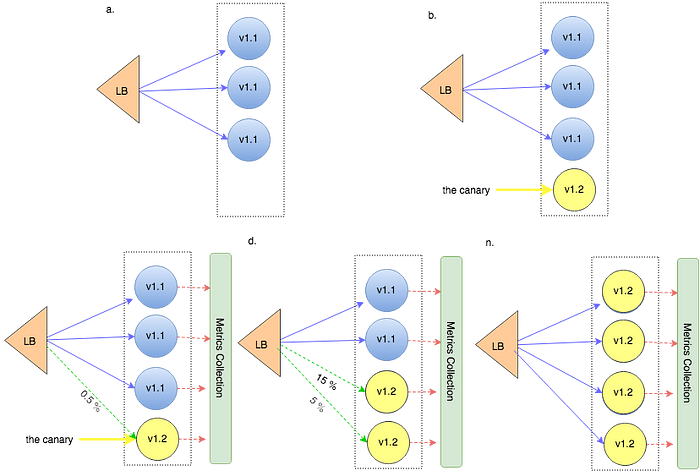
# Applicability

This can be a good fit when you don’t have too many services in your system, or when each service is fully autonomous.

* In the first case we can hold 2 copies of a full application environment. (In K8s each of these could be managed as a separate namespace.)
* In the second case we maintain a pair of environments for each service, test the service in isolation and flip the traffic switch when the new version is verified.

# Canary Releases

Canary releases got their name from coal miners’ practice of carrying a canary in a cage down the mine. If there were toxic gases in the mine — they would kill the canary before killing the humans. Similarly a canary release lets us test for potential problems before impacting our entire production system or user base.



When practicing canary releases –

1. We deploy the new version into the same environment as the production system.
2. We then reroute a small part of production traffic to be served by the new instance. (This is also referred to as ‘testing in production’.) The routing can be purely percentage-based or driven by specific criteria such as user location, type of client and billing properties. This kind of deployment requires careful application performance monitoring and error rate measurements. The collected metrics define the so-called canary quality threshold.
3. If the threshold is not crossed (i.e — application is behaving as expected) we gradually spin up more new version instances transferring even more traffic to the new version. We may introduce additional canary quality gateways along the way.

# The downside:

* Canary can be seen as a variation of blue/green with fine tuned traffic splitting and more meticulous quality threshold enforcement. Therefore, all the complexities of blue/green are present and amplified.
* It requires heightened system observability, which is desirable but entails substantial effort to achieve.

# The upside:

* Observability: It is a must for canary strategy. Once you build up the needed infrastructure — your ability to experiment, test hypotheses and identify issues will provide a lot of power.
* Ability to test on actual production traffic: Creating production-like testing environments is hard. With canary there’s no need to do the same.
* Ability to release a version to subset of users.
* Fail fast: Canary allows one to deploy straight into production and fail fast if something goes wrong. Of course care should be taken so the new app version does not cause data corruption.

# Applicability

Use this if you’ve already invested in modern monitoring and logging tools. May be less applicable if your deployments involve schema changes as you don’t want to change production schemas just to test a new version.

# A/B Testing

Speaking strictly — this isn’t really a deployment strategy. More of a market research approach enabled by a deployment technology very similar to what we described as Canary.

1. We deploy the new version alongside the old one and use traffic splitting techniques to route some of client requests to the new version.
2. We then compare the business, application or system performance metrics between the two versions in order to understand what works best.

A/B testing allows us to not only compare old vs. new but also to deploy multiple versions of the new feature and analyze which one yields better business results.

# The downside:

* Heightened system complexity which is harder to debug and analyze.
* Involves advanced load balancing and traffic splitting techniques.

# The upside:

* Allows for intelligent market and customer behaviour analysis.
* Several application versions can be deployed in parallel.

# Applicability

Use this when you’ve already achieved continuous delivery of features. When your teams are agile enough to quickly produce autonomous features that can be easily rolled back or forward. This strategy works best with application-based continuous deployments techniques such as feature toggling. And of course everything I previously said about observability applies here too.

# Summary

In this post we’ve provided an overview of the major continuous deployment strategies. In the following posts of the series we will go over practical examples of implementing these strategies with Codefresh pipelines and Kubernetes.