Deployment Patterns for Confidence

Joseph W. Yoder, The Refactory, Inc. – USA
Ademar Aguiar, Faculdade de Engenharia, Universidade do Porto – Portugal
Paulo Merson, Federal Court of Accounts (TCU) – Brasilia, Brazil
Hironori Washizaki, Waseda University – Japan

Many software development processes, such as Agile and Lean, focus on the delivery of working software that meets the needs of the end users. Many of these development processes help teams respond to unpredictability through incremental, iterative work cadences and through empirical feedback. There is a commitment to quickly deliver reliable working software that has the highest value to those using or benefiting from the software. DevOps has become a common practice to assist with quality delivery in these practices, specifically when developing using the microservices architectural style. Delivery options have evolved from the "big bang" approach to those that release new features, or small pieces of them, more safely and reliably, i.e., with more confidence, through techniques such as "Blue-Green" and "Canary" deployments. This paper will focus on these two techniques, presenting patterns for each.

Categories and Subject Descriptors
• Software and its engineering ~ Agile software development • Social and professional topics
• Software and its engineering ~

General Terms
Agile, Sustainable Delivery, Patterns, Deployment, Confidence,

Additional Keywords and Phrases
Agile Software Development, Continuous Integration, Continuous Delivery, Innovation, Reliability, Delivery, Deployment

ACM Reference Format:
Yoder, J.W., Aguiar, A., Merson, P., Washizaki, H., 2019. "Deployment Patterns for Confidence". HILLSIDE Proc. of 8th Asian Conf. on Pattern Lang. of Prog. (March), 9 pages.

Author's email address: joe@refactory.com, ademar.aguiar@fe.up.pt, pmerson@acm.org, washizaki@waseda.jp

Introduction

Nowadays, applications are released more frequently, sometimes many times per day. Deploying frequently helps organizations to be more agile by making changes that deliver business value faster, which includes ways to rapidly experiment with new ideas, testing the impact on the business, and quickly addressing any issues that arise.

DevOps as a software engineering practice unifies software development (Dev) and software operation (Ops). To assist with quality delivery in these practices, you need to provide a "*Quality Delivery Pipeline*" [Yoder18] to help assure the delivery meets the requirements and proper validation and checks are done before releasing into full production. At the end of the pipeline, the validated system will be deployed into production. There are various deployment techniques to help successfully and reliably deploy more quickly. The goal is to give confidence by providing "reliable, working software" to the user (making the user confident in the system). Also, the teams will have more confidence that the system is working.

Monolith architectures generally use a "big bang" deployment approach that updates most of the application at one time, sometimes including database updates as well. This has been the de facto release approach for decades. Big bang deployment approaches require development and operations teams to do extensive development and testing before release, following a ceremonial deployment process that often takes several days.

For example, day 1 of the deployment process could have a deadline for code commit, then code would be built and deployed on a staging environment, acceptance tests would take place until day 3, then a new build and deployment with all fixes would happen on the staging environment, followed by final tests, and finally on day 4 we would have a go/no-go decision and deployment in production. This big bang approach can be slow, error-prone because of cumbersome branching and tagging, thus being less agile.

Feature toggles[1] have become popular with "big bang" deployment to help address some of these problems, especially with reliability. With feature toggles, you can enable or disable features at runtime. This is especially useful for releases where new features might cause issues, thus you can turn off a specific feature and address the eventual issues more quickly. However, this approach is limited to features that can be disabled temporarily, and there are still maintenance issues with using feature toggles, and quite often the toggles become a legacy to the system. This is often because the toggles are not prioritized to be removed from the system after they are no longer needed.

The challenges in any type of deployment are that if you break something, the deployment could negatively affect reliability or customer experience. Another scenario would be that you are testing new features with end users, and unstable new features would negatively affect regular users while power users would want to have them (e.g., stable versus beta versions), so you have parallel development streams. Therefore, having alternative deployment techniques for releasing can provide many benefits. Deployment techniques in modern software development that have recently become more popular are *blue-green deployment* and *canary deployment*.

Blue-green deployment is where you have two "almost" identical environments. One is always live. You release to the non-live environment and validate the release with testing. After verification, you switch all traffic to the newly released environment, and the previous environment is idle and available for rollback or a new release. *Canary deployment* deploys a new application code in a limited part of the production area, visible to a small subset of the

[1] https://en.wikipedia.org/wiki/Feature_toggle

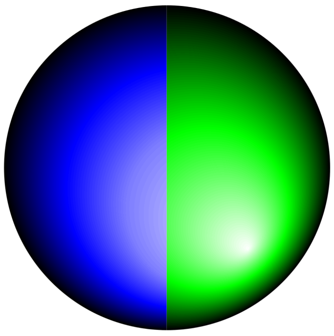
users. You then have these users test and validate the application; if there are no problems, the system is gradually rolled out to the rest of the users.

This paper will start out by looking at the "*blue-green deployment*" and the "*canary deployment"* patterns and how they assist development teams in more reliably releasing more quickly with confidence. The context and the forces for these patterns have similarities as they are both addressing deployment issues. However, there is a variance in the problems they are addressing. Therefore, the patterns duplicate these similar contexts and forces, though some forces have been added as they relate to the different patterns.

Note that there are other patterns related to deployment, such as *Rolling Deployment*, *Chaos Monkey*, and *Feature Toggles*. These could also be written up as patterns, but are not discussed in detail in this paper.

Blue-Green Deployment

aka Red-Black or A/B Deployment



You are doing continuous delivery [Humble10] as part of DevOps by making deployment as automated and quick as possible. You are releasing to a live environment that has potentially many users.

How can we deploy reliably and with confidence without negatively impacting many users?

❖ ❖ ❖

Building and releasing into production environments may require several steps, such as transferring and replacing deployment artifacts (e.g., executable files, JAR files, Docker images), updating property files, altering database structures, and reconfiguring infrastructure elements (e.g., message routing in an API gateway). This process can be tedious and error-prone.

Automating the testing and deployment process is challenging due to its complexity and uncertainty, thus requiring a lot of expertise and effort.

Lack of validation or testing of your release can be dangerous and costly.

It can be expensive to duplicate the entire production environment. Any other alternatives will be only a simpler replica, possibly not emulating all the issues of production.

New releases have various risks that can negatively impact the business (e.g., loss of funds), if problems arise. New releases that cause problems need to be rolled back quickly and reliably.

❖ ❖ ❖

Therefore, when releasing, have two environments that are nearly identical. One is the live environment. Release into the non-live environment, after validating the release, switch all network traffic to the new environment, disabling the previous live environment.

This type of deployment process is referred to as "*blue-green deployment*". *Blue-green deployments* require two nearly identical production environments (called "blue" and "green") where deployments are made The two environments can be, for example, two physical or virtual machines, two nodes on a Kubernetes cluster, to mention a few.

At any time only one of the blue-green environments is live. For example (see Figure 1), let's say the green environment is currently being used for live production. When you have a new release, you deploy your system and do your final testing in the other (blue) environment.

Once the software is working in the new environment (blue), you switch all live access so that all incoming requests now go to the newly tested (blue) environment. The previous production environment (green) is now idle and ready either for a rollback, for emergency use, or for the next release. As you have new releases, you continue to switch back and forth between the blue and the green environment.

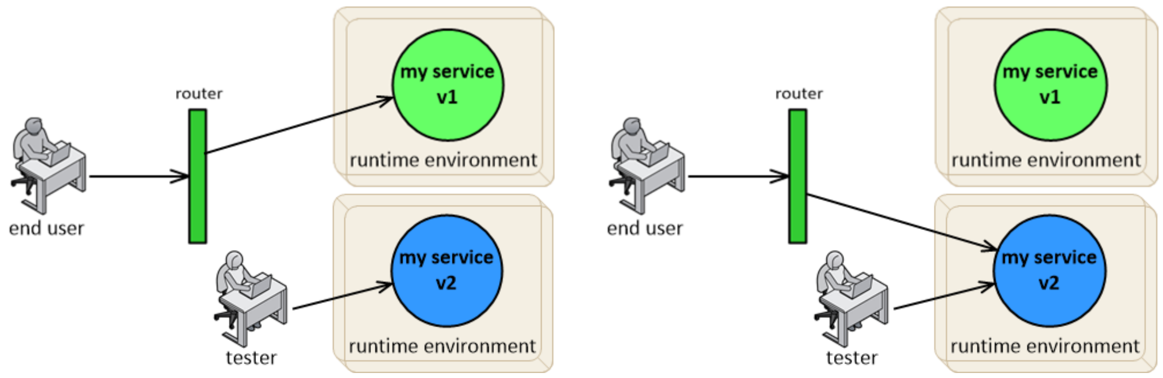


Figure 1: Blue-Green Deployment

Switching the live version from v1 to v2, or rolling back from v2 to v1, can still be complicated, as we should allow requests currently being processed to finish successfully before the transition. Your design might handle this through some replication and ramped release strategy, or possibly putting the systems into a read-only stage during the transition.

Advantages of Blue-Green Deployments

Foremost, *blue-green deployment* significantly decreases the downtime for rolling out a new version.

Blue-green deployment enables quick version rollback: after deploying to one environment, if a problem is discovered, you can easily switch back and start using the previous environment. You may have to clean up some transactions that happened during the rollout of the failed environment.

With *blue-green deployments,* you always have a backup environment ready in case the production environment becomes unavailable. Having the two environments may also allow independent maintenance of the infrastructure. For example, let's say blue and green environments are two separate VMs. When blue is active, you may perform upgrades to the green environment VM. Then, when green becomes active, you can perform the upgrades on the blue environment VM. The application is not affected.

Disadvantages of Blue-Green Deployments

Blue-green deployments require organizations to have two identical sets of production environments, which can lead to significant added costs and overhead without actually adding capacity or improving utilization. As an alternative, there are other strategies that can help, such as *canary* or *rolling deployments*. *Canary deployment* releases the new system to a small, limited number of users, while *rolling deployment* staggers the rollout of new code across servers, usually to a server with a limited number of users first.

When the new version requires database schema changes and/or data migration, employing blue-green poses an additional design challenge. There are two main alternatives:

- When there is a single centralized database, make the database structure changes backward compatible. That is, the old code will be able to access the new database structure. Complementarily, the new version code should be backward compatible, that is, the new code will be able to access the old database structure.
- Make each version access a separate database (separate DB server or separate logical space/owner within the same DB server). In this case, the data is replicated across the old version DB and the new version DB. Therefore, a data synchronization mechanism must be established. This is an eventual consistency setting that has the additional drawback that an application may consume stale data.

Blue-green deployment can be problematic when the new version contains API changes that make old client applications incompatible. In this case, we might need an interceptor placed between the clients and the old and new applications to perform message transformations to deal with the API changes.

* * *

Blue-green deployment can use *feature toggles* to emulate a form of *canary deployment*, by toggling on/off certain features for certain user roles.

Blue-green deployment can be used in conjunction with *canary deployment*. In other words ,you can push the new release completely to the second environment. And then route selected users from the first environment to the second environment in a canary fashion.

Canary Deployment
aka Staged Deployment

You are doing continuous delivery as part of DevOps by making deployments to different environments as automated and as quick as possible. You are releasing to a live environment that has potentially many users.

How can we get feedback on the new release, verify if it is working properly, and get early reactions from users?

❖ ❖ ❖

Building and releasing into production environments can be tedious and error-prone.

Automating the testing and deployment process is challenging and requires a lot of expertise.

Lack of validation or testing of your release can be dangerous and costly.

New releases have various risks that are important to validate before release to all users.

New releases made available to all users can severely hurt the confidence on the application and negatively impact the business if they are flawed,

❖ ❖ ❖

Therefore, first deploy the change to a limited number of users or servers to test and validate the release. This could include verifying the release works properly and/or getting acceptance feedback from your users. After you have validated the release, roll the change out to all servers or users.

This limited release is called *Canary Deployment*. Canaries were used in coal mining as a warning system, making sure there were no toxic gases before miners entered the mine. In a sense, we are doing the same thing with *Canary Deployment*. Before releasing to a wide audience, the system is first deployed to one or more canary servers (see figure 2). These might be for trusted internal users. After the release is validated, make it available to other users and servers. Validation includes getting feedback from canary users and also monitoring runtime properties of the new version. If significant flaws are found, the release of the new version to all users is cancelled.

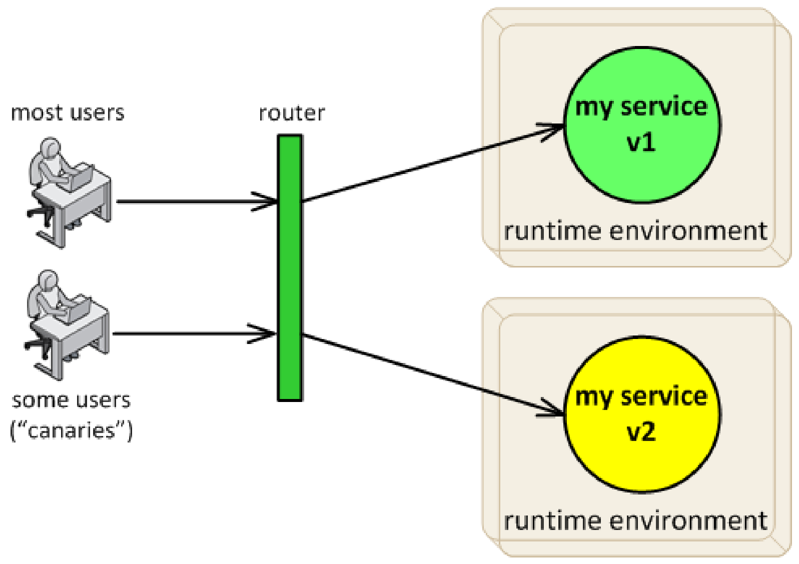


Figure 2: Canary Deployment

Advantages of Canary Deployments

Canary deployment allows finding problems before they are pushed out to all users. These problems can be bugs in the new release, unsatisfactory latency or throughput, poor user experience, security breaches, among others. The final result is improved quality of new releases.

Disadvantages of Canary Deployments

Canary deployments require you to control what users see of the canary release and hence require a supporting infrastructure for moving users from the main system to the canary system, and vice versa. This supporting infrastructure has to be configured and governed, and may include routers, network proxies, authentication mechanisms, and configuration files.

The *canary deployment* represents a delay in the rollout of the new release to general users.

Similar to blue-green deployment, *canary deployment* incurs the cost of having two production environments.

Alternatives to Canary Deployments

An alternative to canary deployment is to deploy the new version to everyone with *feature toggles* turned off for the new features. Then you selectively enable features for different users. As you validate the release, you increase the number of users' access to the new features. This approach requires a special implementation using "*feature toggles*" combined with canary user identification.

* * *

You can also use *blue-green deployment* to push the release out to the green server for example, and then only move a few users from the blue server to the green server for the *canary deployment*. Then, as you validate the releas,e you can move more and more users to the green server.

Summary

The deployment of software applications and microservices to network servers and cloud infrastructure can use different approaches. As usual, no single approach is best for all instances. There are tradeoffs to consider, and you can use variations or a mix of deployment strategies. For example, you could use *feature toggles* with *canary* and/or *blue-green deployment* strategies. Big-bang can use feature toggles to get a canary effect. You can use blue-green for patching a canary version before general release. You can also use feature toggles with blue-green for a canary effect.

Acknowledgements

We'd like to thank our shepherd, Michael Weiss, for his valuable comments and feedback during the AsianLoaf PLoP 2019 shepherding process. Finall,y we'd like to thank our 2019 AsianPLoP Writers Workshop Group, Yuki Kawake, Kai Chang, Ed Fernandez, and Yu Chin Cheng for their valuable comments and suggestions.

References

[Yoder18] Yoder J., Aguilar A., and Washizaki H., "Quality Delivery Pipeline," 12th Latin American Conference on Patterns of Programming Language (SugarLoafPLoP 2018), Valparaíso, Chile, 2018.

[Humble10] Humble, J. & Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.