


# WORTHWHILE MODERN DEPLOYMENT STRATEGIES – A FULL GUIDE

BY DEVELOPERS • FOR DEVELOPERS



Red/black, blue/green, A/B, canary, rolling... There are hundreds of articles written on these application deployment strategies. As with any newer approach in tech, it can be difficult to discern practical applications from the hype. We will take a look at these strategies, how they evolved, and whether your application can benefit from them. Finally, we will cover the symbiotic relationship between these deployment strategies and hosting your application in the Cloud.

The evolution of deployment strategies can be divided into three broad, but distinct, “generations” — a time before they existed (“generation 0”), the proliferation of build tools coupled with source control (“generation 1”), and advanced deployment strategies using multiple load-balanced instances (“generation 2”). It’s important to understand that the “unequivocal advantage at no extra cost” improvements stopped after “generation 1”. The latest deployment strategies are highly situational and/or require significant investment. Nonetheless, these strategies are only going to get more popular, as they are invaluable when facing one of the specific problems they are designed to solve. Before attempting to apply advanced deployment strategies, it’s helpful to understand the several decades of deployment strategy evolution which serve as their foundation.

# GENERATION 0: MANUALLY MOVING FILES TO A PHYSICAL MACHINE

Before the 2000s, not much thought was given to deployments; we were living in a Waterfall world with its long development cycles. Efficient deployments weren't as much of a concern, since they happened so rarely. The three defining characteristics of this era are:

- **Everything** is done manually
- No version control (or deployments don't use it)
- The application is deployed to expensive on-prem machines

To really understand this approach, we need to fully understand the amount of manual work involved:



If you aren't wincing, you should be. While the downsides of this approach are obvious, they are listed below for emphasis:

1. **Significant downtime** when the application is stopped, files are copied over, and the app is restarted.
2. **Inability to roll back quickly** if a mistake is found, since the process is manual.
3. **High error rate** caused by tedious, repetitive tasks, requiring meticulous detail during what is often a high-stress deploy.
4. **Increased bus factor risk** due to the tribal knowledge and longer onboarding needed to support archaic manual deployments; developers are also far more likely to leave if they are stuck doing manual tasks which can be easily automated.

# GENERATION 1: AUTOMATED BUILDS FROM SOURCE CONTROL

As mentioned earlier, the listed problems were historically deemed acceptable due to the infrequent deployment cycles under the Waterfall methodology. With the advent of Agile, deployments became more and more frequent, bringing the inefficiencies of manual deployments into the spotlight.

However, despite the demand, fully automating deployments would have been difficult without version control. Without it, files would still need to be manually moved to the build server, which would nullify most of the benefits that come with deploy automation; the timing worked out — just as companies were starting to adopt Agile, Git emerged as the leading source control system and gained widespread use, even among companies that are far behind the bleeding edge in IT.

The demand for faster and simpler builds, coupled with the widespread use of source control systems, paved the way for rapid advancement in build automation tools. The laborious process of making sure the code compiles, the unit tests pass, moving the compiled build artifact, restarting the application with the new code, etc., was offloaded to build automation systems such as Jenkins, TeamCity, Travis CI, etc.

These tools have streamlined deployments — improving speed, while simultaneously decreasing the error rates, developer workload, and eliminating the need for specialized archaic knowledge. Many of these tools are free and easy to set up. There is no longer any reason for a company to deploy manually. This is especially true for companies that have moved their infrastructure to the Cloud, as all the major cloud providers (Azure, AWS, GCP) provide build tools virtually “out of the box”.

The majority of tech-savvy small and mid-size companies today have adopted some sort of CI/CD pipeline. However, there are still businesses with revenue in the hundreds of millions that run without source control or [CI/CD tools](#). Not only is this extremely inefficient, but it is a liability that can compound the stress of an innately difficult situation where a key developer with knowledge of the deployment process is fired or leaves. In the worst case, the company may find itself unable to update a key product, or even restart it properly if an error occurs.

If your company is one of those NOT using a CI/CD pipeline — implementing one is likely the best cost/benefit ratio IT project to undertake, bar none. If you find yourself in that situation, I recommend skipping the rest of this article and looking into build automation tools instead. However, if you have the build pipeline foundation in place, read on to the advanced features which evolved on top of those fundamentals.

## GENERATION 2: SOLVING HIGHLY SPECIFIC PROBLEMS AND COLLECTING USAGE INSIGHTS

Generation 2 is a bit of a misnomer, since the latest deployment strategies are not so much an across-the-board efficiency boost, as they are a loose collection of disparate solutions to very specific problems.

Most of these require a load balancer and the ability to deploy multiple instances. Many also require a person to monitor and interpret the results. Since all deployment strategies solve different problems, we will look at them separately. We will also take a look at A/B testing and feature flags, which are not truly deployment strategies, despite often being labelled as such and having some overlap. The strategies are listed in descending order in terms of utility and benefit for effort for most companies. However, some companies may have a specific need that makes one of the strategies far down on the list invaluable, so it's good to be aware of them.

## BLUE/GREEN (AKA RED/BLACK) DEPLOYMENTS

### Use cases:

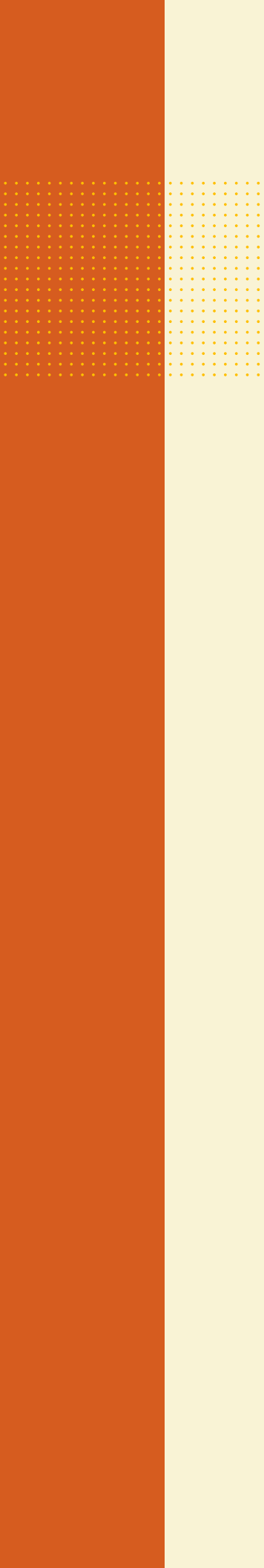
- Zero-downtime deployments
- Instantaneous rollbacks
- Removing failed build risk
- Increasing test confidence
- Reducing off-hours work

### Cons:

- Costs for secondary environment (mitigated by using Cloud platforms)

### Requirements:

- Proxy
- Infrastructure for second environment



Even with an automated build, starting an application after applying changes takes time, resulting in the application being inaccessible to users for up to a couple of minutes. This can cause frustration, especially if a user tries to submit a form during the deployment and loses data. The common approach for avoiding user impact is highly manual — shifting the deploy to a low traffic time of day by having a developer or team deploy when user activity is at a low point, such as at 3 a.m. Naturally, deploying and testing in the middle of the night doesn't make for a happy team. Happiness aside, global companies such as Netflix, which pioneered this approach don't have the luxury of having any truly “low traffic” time and must entirely eliminate the downtime caused by deployments.

A blue/green deployment solves this problem by creating an exact copy of the live environment. For the sake of convention, the original live environment is referred to as the “blue” environment, and the copy containing the changes to be deployed is referred to as the “green” environment. Once the copy is fully built, traffic is switched over to it in an instant by changing which IP the proxy server points to. The added advantage of this approach is that the green environment can be fully tested in place before any traffic is moved over. The older “blue” environment keeps running after traffic is moved over, so that if any issues are found, traffic can be switched back over instantaneously. Usually, the blue environment can be spun down to decrease cost once the green deployment is confirmed to be good. This spin down process is much easier and results in higher cost savings if the underlying infrastructure is hosted in the Cloud, where it can be completely “turned off”, as opposed to an expensive server sitting idle. A word of warning here — if you choose to turn off the old environment immediately after the deploy, ensure that it has finished processing all in-flight requests before doing so. You may hear this referred to as “draining” the environment before it is shut down.

Note that “red/black” deployments are a different name for the exact same thing. There is no nuance — they are exactly the same as “blue/green” deployments. You can read more about the naming convention [here](#) and [here](#), but the important take away is that the colors don't mean anything, they are just there to identify distinct environments.

Blue/Green deployments are, by far, the most practical and “universally applicable” of these deployment strategies. They exist with virtually no downsides, other than the minor cost of duplicating production servers and some set up time. Even if you don't regularly use blue/green deployments, consider setting up the infrastructure to do so for large features which takes months to build and cannot be released until fully complete. The small time investment will save you a lot of stress.

# CANARY DEPLOYMENTS

## Use cases:

- Testing changes on small percentage of users

## Cons:

- Requires manual monitoring or extensive work to automate
- Misunderstood - often used where QA and load testing would suffice

## Requirements:

- Proxy
- Infrastructure for second environment
- Ability to collect user feedback
- Time and resources to analyze feedback

Canaries are birds that were historically taken into mines for safety. If dangerous gasses built up in the mine, the canary would die, warning the miners to evacuate.

Likewise, the purpose of a canary deployment is to alert the dev team to the presence of a problem before the bulk of the users are affected by it. In a canary deployment, a small amount of traffic (usually ~5%) is directed to the new version of the application for a testing period. If problems are discovered, the traffic is redirected back to the old application. If not, all traffic is directed to the updated app. Note that this is the same concept as a blue/green deployment in terms of having two copies of the environment, the difference being that only a small percentage of traffic is moved over initially, rather than all of it at the same time.

However, this is a resource-intensive approach; someone must be available to evaluate whether users are experiencing issues with the application. The most obvious signs of a problem could be discovered by having response speed and error metrics/alerts in place. But those issues often do not require a deployment strategy to catch — they would normally be noticed during testing or automated performance audits.

With that said, some larger companies have perfected canary deployments into an art form and have set up automation to roll back based on metrics such as error rate, latency, throughput, etc. of the newly deployed canary. While this is a nice fail-safe that covers real-world scenarios that load testing doesn't catch, it is also an enormous upfront investment to develop this automation; one that is not cost-efficient for most small and mid-market companies.

The true benefit of canary deployments comes from catching “errors” that could not be anticipated in advance — socially insensitive text / connotations that the team doesn't catch but users notice, a new confusing UI/UX flow that the team didn't catch due to familiarity with the product, and so on. These are qualitative and require both a specialist or team to monitor, as well as some way for users to report problems. As such, this deployment strategy is best left to large companies with a huge number of users, where any such mistake can do irreparable damage.

# ROLLING DEPLOYMENTS

## Use cases:

- Deploying an application served from multiple instances with no service interruption
- Default strategy if using K8S

## Cons:

- Risk of corrupt data if the old and new version of the application aren't compatible

## Requirements:

- Multiple instances
- Health checks
- Load balancer and logic to update the system one instance at a time

In my opinion, green field projects using rolling deployments are only relevant to high-traffic applications served by multiple machines. Let's say a load balancer is directing traffic to 10 machines. In a rolling deployment, each machine is taken out of circulation and updated one at a time, while the other 9 continue to serve traffic. You could, theoretically, perform a rolling deployment on one instance by spinning up an instance with a new version and then turning off the old instance. However, under those circumstances, it's easier and safer to just perform a blue-green deployment.

The purpose of a rolling deployment is to keep serving traffic from multiple load-balanced instances during a deploy. A serious caveat with implementing a rolling deployment is that there should be no conflicts between the new and old version of the application running at the same time, as both versions will be serving traffic during the gradual update. It's easy to dismiss this warning, but this situation arises all the time, both when an object being sent to an API changes or the database schema is updated and an out-of-date API method would be sending the wrong model.

As with most of these strategies, there is nuance here. I stated that rolling deployments only apply to high-traffic applications, which I maintain to be true if you have to write the rolling deployment logic by hand. However, Kubernetes (K8S) uses rolling deployments by default, even for low traffic, single-instance applications, and this requires little to no custom code. When using K8S, rolling deployments are almost always a good choice.



# A/B TESTING (NOT TRULY A DEPLOYMENT STRATEGY)

## Use cases:

- Determining which of two versions users prefer
- Improving conversions

## Cons:

- Requires manual analysis
- Misunderstood - often used where QA and load testing would suffice

## Requirements:

- Proxy
- Infrastructure for second environment
- Time and resources to analyze feedback and/or conversion data

Once again, this approach relies on having two copies of the application running at the same time. In this case, the two deployed applications vary slightly and both serve live traffic simultaneously. Traditionally, half of the traffic is directed to version “A”, and half is directed to version “B”. This is used for testing which version the users prefer. A common use case is seeing which version prompts more conversions — whether that’s registering, clicking an ad, buying a product, making a donation, etc.

Additional code is required to track which version of the application the user is using to make an action, but the code is fairly trivial to write. The larger concern is that someone needs to analyze the data, design the “A/B experiments”, and then write two versions of the code to try out.

I would encourage using A/B deployments in limited circumstances, and only when an important and well defined question is posed. Otherwise, it can be a waste of resources.

# FEATURE TOGGLES (AKA “FEATURE FLAGS”)

## Use cases:

- Continuing to merge code with hidden features
- Turning functionality on/off without a redeploy
- Facilitating A/B testing
- Facilitating targeted canary releases

## Cons:

- Additional, confusing, hard-to-troubleshoot code
- Requires additional clean up after feature completion of A/B

## Requirements:

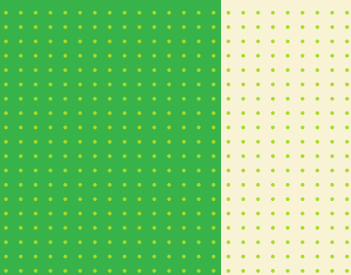
- Additional logic in codebase

Feature toggles are flags in your code which turn specific functionality on or off. This can be used to change an app’s functionality without a deploy (for example, by changing an environment variable). This ability to change how an app behaves without redeploying is often used to enable other deployment strategies. For example, the UI/UX differences seen by users during A/B testing can be achieved by deploying the same codebase with different feature flags enabled. The alternative to this is deploying from two different branches or codebases.

Furthermore, feature flags can allow canary deployments to target a well-defined group of users, rather than a random set. This is achieved by writing a feature flag which checks for a specific role or attribute in the user’s token, and is often used to release new features only to internal users or those who volunteered to try out experimental updates.

Perhaps the most common use of feature flags is to hide incomplete features from users, while continuing to commit code to the release branch (usually “trunk” or “master”). Proponents of this strategy adhere to the popular [trunk-based development methodology](#), which postulates that all work should be committed to the trunk branch as soon as possible, or code will become increasingly stale and result in serious merge conflicts.

I disagree with this view. In my experience, hiding work-in-progress with feature flags is a poor substitute for developing a feature in a separate branch. Advocates of feature flags and trunk-based development often fail to mention just how much additional work and complexity feature flags create. Feature flags not only result in additional code which has to be cleaned up after the feature is released, but often break unit tests, complicate deploys, and confuse testers. This complexity is added twice — both when the feature flag is added and when it is removed.



The dramatically increased code complexity is not worth the benefit. The severity of merge conflicts can be reduced or avoided entirely by breaking new code out into new services (or components if doing UI work), merging the release branch into the feature branch often, and scheduling work in a way that keeps teams from working in the same areas of the codebase. Moreover, writing feature flags slows down work, allowing more unmerged changes to accumulate.

As usual, the key is to consider both sides — if the feature cannot be reasonably isolated and your company already uses trunk-based development, feature flags deserve a second look.

## SHADOW DEPLOYMENT (AKA “TRAFFIC SHADOWING”)

### Use cases:

- Simulating load under real traffic conditions

### Cons:

- A large investment to catch errors that slipped past load and integration tests

### Requirements:

- Traffic forwarding
- Second API environment, and possibly database

A shadow deployment involves the creation of a “shadow” copy of your API layer and forwarding all real application traffic to it, with the purpose of checking the load tolerance of the shadow copy with new features. If this sounds like the job for load testing software — it is. A shadow deployment is a complex set-up for dubious benefit. What adds to the complexity is that the “shadow” copy must also hit a “shadow” database in order to avoid overwriting real data or affecting db performance. This strategy is unlikely to be cost-efficient for small and mid-market companies, despite large corporations such as Twitter using it to catch performance issues which slip past load testing.

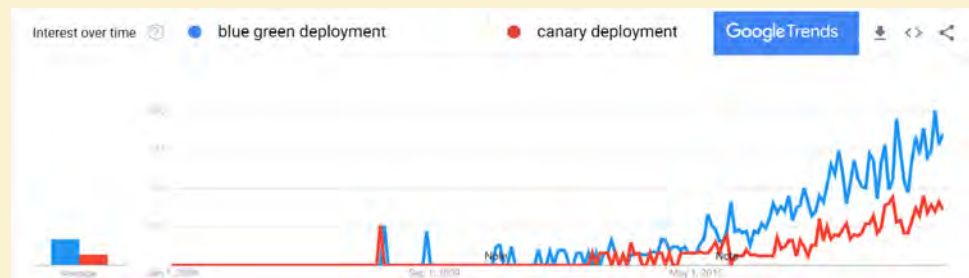
## FINAL THOUGHTS

Having a proper CI/CD pipeline is both easy and essential.

On the other hand, most of the advanced and highly-promoted deployment strategies can be unnecessary and resource intensive, though it's critical to be aware of them if you find yourself facing the specific problem they solve.

Finally, it's important to stress that all cloud platforms provide functionality that was historically handled by CI/CD tools such as Jenkins, TeamCity, etc out of the box, and that alone is a strong argument to consider the cloud for your infrastructure. Furthermore, Cloud providers vastly simplify the implementation of the advanced “generation 2” deployment strategies, especially blue/green deployments, which are among the most frequently used.

Based on Google trends, these strategies continue to gain popularity, and will likely become easier and easier to implement over time.



### AUTHOR

**Victor Chtelmakh**

Senior Software Developer, Callibrity

Victor graduated from the University of Cincinnati with a BBA in Finance and Accounting. He spent several years trying to automate his jobs and crypto-currency trading before moving into programming professionally. MongoDB, AngularJS, and Entity Framework are among his favorite technologies.



**WE ARE ARTISTS.  
WE ARE ENGINEERS.  
WE ARE INNOVATORS.  
WE ARE CALLIBRITY.**

Learn more about Callibrity  
[www.Callibrity.com](http://www.Callibrity.com)

## ABOUT CALLIBRITY

Callibrity is a software consultancy specializing in software engineering, digital transformation, cloud strategy, and data-driven insights. Our national reach serves clients on their digital journey to solve complex problems and create innovative solutions for ever-changing business models. Our technology experience covers a diverse set of industries with a focus on middle-market and enterprise companies.

Callibrity (kə'librətē) is a mashup of two different roots, calli and caliber. Calli means 'beautiful' in Greek, as in Calligraphy - beautiful writing. Caliber means 'a degree of merit or excellence'. We strive to do beautiful work with a high degree of merit and excellence.

< by **developers**  
for **developers** >