Forms of software deployment

Ralfs Zangis: zangis@kth.se

April 2021

1 Introduction

Nowadays applications have ever-increasing requirements, whose satisfaction is paramount for the product's successful mass adoption. One of these requirements is the effective delivery of new services to the customer, which over the last few decades has been happening ever more frequently. However, keeping up with change can be a difficult process due to the exceedingly large scale of many applications. For these solutions, a new release must be deployed on multiple concurrently running server instances, rather than just one.

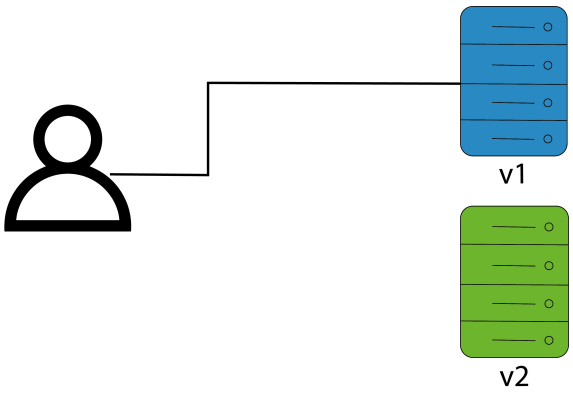

Figure 1: Example deployment

The task of deployment and the subsequent release regards the process of how a new build can be made accessible to the userbase, this has evolved a lot during the last few years, with the growth of the DevOps [1]. Due to ever more agile working approaches [2] the users have started to anticipate quick and seamless changes to the products that they are using, with companies that can not provide such services often being left on the backfoot, as they are not able to compete on the count of being less efficient/adaptive. That is most noticeable when looking at information giants such as Amazon, Google, and Netflix that deploy changes to their key products thousands of times per day [3].

Often the process of solutions release is automated and performed by a load balancer [4]. However, there are many strategies that could be utilized when making changes available, meaning that there is no best approach, just solution that is better suited for the fulfilment of the needs.

2 Types of deployment

The deployment could be utilized for product release and testing purposes [5, 6]. The following sections will give an introduction to the different types of these operations while providing a short overview of their benefits/drawbacks.

2.1 Deployment strategies

The deployment strategies are used to introduce changes in an environment, these could be upgrades, deletions, or additions. However, it is worth noting that deployed service does not imply its accessibility to the users, often this is a separate step. The following patterns allow for different forms of version deployments, with each having varying degrees of disturbance to the service, experienced by the user. The end goal often being the user not noticing change taking place, enabling for customer's operation to continue unimpeded. This, however, was not always the case as in the early days of the software deployment the service being down for maintenance was a loathed but expected occurrence.

2.1.1 Big Bang Deployment

The Big Bang deployment strategy essentially follows 2 steps, which are: the removal of the previous application's version and subsequent addition of the desired edition. This approach is closely associated with plan-driven development and thus not well suited for more agile development teams. Furthermore, releases often are big events throughout the company and multiple teams could be expected to coordinate their actions before, during, and after the deployment.

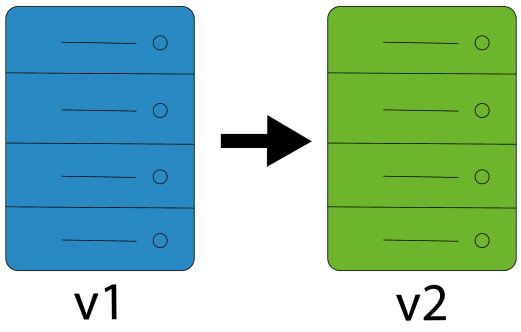

Figure 2: Example big bang deployment

Pros:

1. Simple to implement.

Cons:

1. Includes downtime during update.
2. Not easy to roll back changes if an issue occurs.

2.1.2 Rolling Deployment

The rolling deployment follows similar steps to the already discussed big bang deployment, however, the main difference is that the changes are gradually introduced to an ever-growing subset of application instances. The number of nodes simultaneously updated being called "window size", which could change depending on the instance cluster size or company policy.

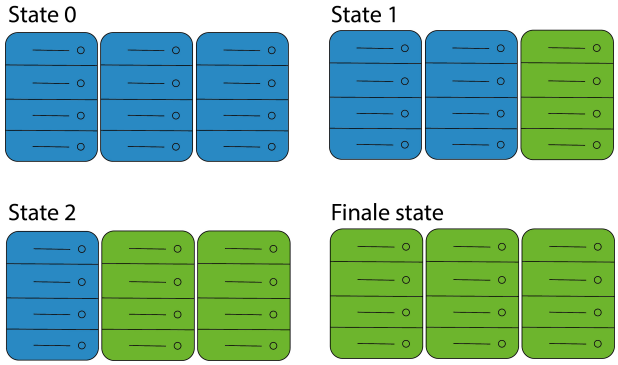

Figure 3: Example rolling release deployment

Pros:

1. No downtime.
2. Lower issue impact, since it would affect a smaller portion of users than big bang.

Cons:

1. Change rollback could be a slow process.
2. There is a need to consider backward compatibility, as both the new and the old versions coexist during deployment, with users using either one of the versions indiscriminately.

3. Session persistence would require stickiness [7] and connection draining (Existing connections continue until they naturally end, only then the server can be taken offline).

2.1.3 Blue-Green

In the Blue-Green deployment, there are two identical environments, with users by the usage of load balancers being routed to one of them while the other one is idle. Meaning that it is like the Big Bang deployment, but instead of having a single production environment, an additional staging location is provided. When the staging environment is ready for usage (the newest version has been deployed and tested), it could be released to the public, by making it production, while the previously used environment would be made into staging and decommissioned or left for possible future rollback.

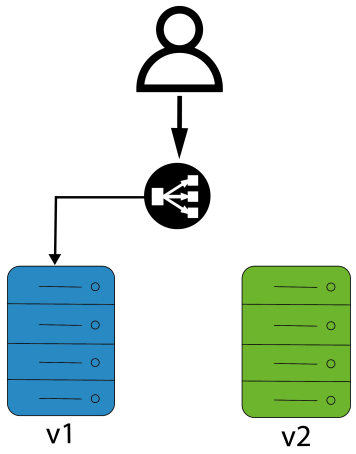

Figure 4: Example Blue-Green deployment

Pros:

1. No downtime.
2. Rollback can be as simple are routing traffic back to the original version.
3. The separation of the two environments ensures that one does not affect the other.

Cons:

1. Increased costs, due to maintenance of duplicate environments.
2. Backward compatibility is a must-have to seamlessly switch the version.

3. In case of environment decommission, considerations must be made on the connection draining.

2.1.4 Usage

Based on the provided descriptions it could be gathered that the Big Bang solution is not a preferable answer when the aim is placed at maximizing the service's uptime. However, the simplicity associated with this approach could come useful for some companies that can handle downtimes in their operations. When downtime is not acceptable, the rolling release and Blue-green deployment are better solutions, but one must decide on the tradeoffs they are willing to make when choosing between the two.

2.2 Testing strategies

Increased speed of deployments could be a double-edged sword, as frequent new version delivery might negatively impact the reliability of the product, by introducing unexpected issues. Therefore, companies often have developed strategies for deploying code while minimizing the risks.

The most common way of assessing code is of course by running tests and simulations; however, they might not predict all steps and changes that users could perform in the real world, thus, deploying for testing is often invaluable to the company that wants to provide the best product.

The patterns discussed in this section can be used to validate the requirements of the solution, through regulated products exposure, this is usually done over some time period and includes monitoring [8].

2.2.1 Canary Deployment

Canary testing works similarly to the rolling release, however, after rolling out change it is evaluated against the previous version, this is done by choosing some key metrics to look for. The new release could be made accessible to any subset of users, for example, based on their location, application usage trends, user affiliation to the company, and so forth. After satisfactory results, the next step could be taken, and update released to an ever expending userbase.

Pros:

1. Can do non-intrusive live testing.
2. Rollback, by simply redirecting traffic.
3. No downtime.

Cons:

1. Slow rollout due to the required monitoring at each sample size.

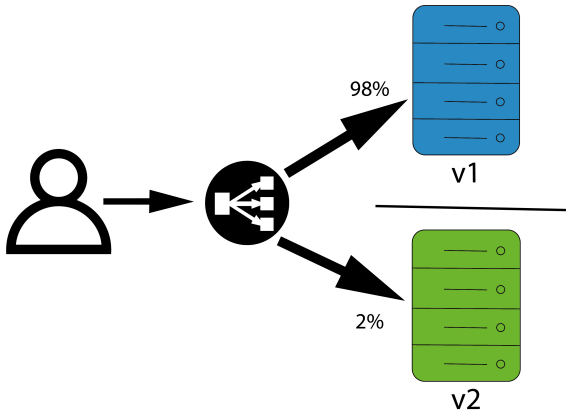

Figure 5: Example Canary deployment

2. Monitoring setup could be a significant endeavor.
3. Support for backward compatibility.

2.2.2 A/B Deployment

This testing strategy is used to observe changes in user behavior, it works by routing part of the userbase to the new version, while the rest keep using an old solution. When enough data has been collected decision could be made to use the version that generates the best results (for example, increased retail numbers).

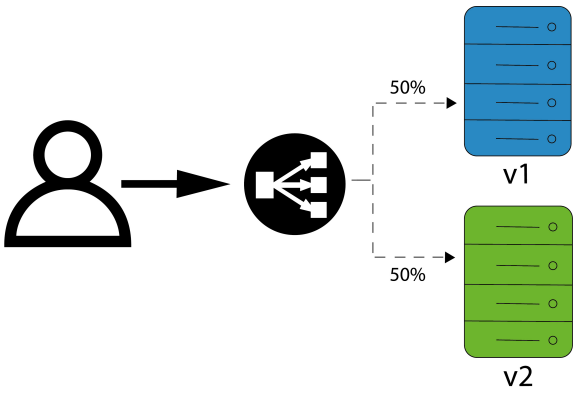

Figure 6: Example A/B deployment

Pros:

1. Can be used to measure the benefits of changes between versions.

Cons:

1. It could be complex to implement and to prove that one version is better.
2. Results could easily be invalidated by multiple external or internal reasons.

2.2.3 Shadow Deployment

While canary release could expose portion of users to an inferior application's version, the shadow deployment instead opts out to deploy the new version in tandem with the old one and mirror incoming traffic (can happen in real-time or by replaying previous traffic). By doing so there is no need for simulation, as both versions operate based on the same inputs, but only the production system responds.

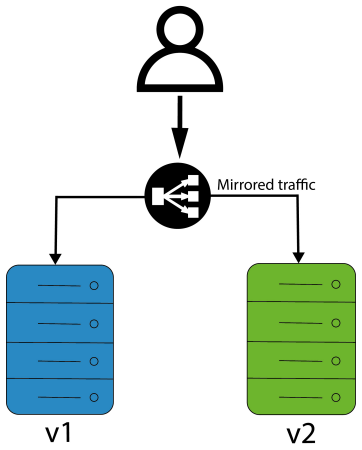

Figure 7: Example Shadow deployment

Pros:

1. Bugs experienced in the shadow environment have no impact on the user.
2. Can see the changed logic performance using real-life traffic.

Cons:

1. Developers always must keep in mind the potential side effects, that could arise from calling the same operation twice (such as paying for the item), in those cases stubs [9] might be the preferred solution for interfaces.
2. It could be expensive to run two environments in tandem.

2.2.4 Usage

The usage of testing strategies is not a prerequisite for product success, but it most certainly would help the company to avoid/deal with issues that could arise if products were released with major fault to a huge number of consumers. The chosen testing strategy should be decided based on the desired goal, while not forgetting the costs associated with the choice (both if a strategy is chosen or neglected in favor of another approach). With many organizations opting to utilize more than one approach, often even all three. This is most noticeable with products such as chrome, where there are multiple channels (canary, dev, beta) a build must go through before being called stable and made accessible to all users [10].

2.3 Examples from the industry

One notable example of a company embracing quick and automated deployments is the beloved movie broadcasting service: Netflix. They from the beginning have attempted and succeeded in being fast adopters of the newest trends and technologies, with this often being attributed to their success. And when deploying solutions, they do not shy away from using many of the testing strategies already discussed and are looking forward to further improving several processes [11].

However, the fear of the bad could be as much a motivator for change, as the good that could come from it. One of the main things to consider being the cost of outages, which could be a major hurdle to the company, by impacting not only the income but also the reputation of the firm. The two most notable cases in recent times are: google blackout of December 14, 2020, which lasted for whole 45 minutes and cost according some 2.3 million USD, and the amazon 5-hour debacle that happened in the same year in November, resulting in thousands of companies being affected. While neither of these issues can be attributed to problems with deployments, they serve to showcase the stakes involved when failure could arise, especially in service on such a scale and impact.

2.4 Tools

Often used solutions for implementing the aforementioned deployment strategies are docker swarm and Kubernetes. With Kubernetes being the most famous and regularly used solution for container orchestration. However, there are many tools, such as Istio, that can be applied to secure, connect, and monitor microservices deployed on Kubernetes and other platforms, which is important for proper testing strategies implementation.

2.5 Conclusion

Deployment is an integral part of any systems lifecycle and when properly done it could allow for delivery of an improved product while ignoring it could cause

far-reaching consequences to not on the companies' balance, but also reputations (which could be even worse).

Even though the deployment might look like a straightforward process, it oftentimes is more complicated than initially anticipated by the developer and requires monitoring, which might not always be simple to do, especially properly.

As a final thought, it is worth mentioning that deployment is a balancing act in which we are looking for maximizing benefits and minimizing costs when releasing an update for our application. With most important goals being:

1. Minimizing downtime.
2. Dealing with arising issues without impacting the customers.
3. Maximizing repeatability by automating.

References

[1] What is devops? URL https://aws.amazon.com/devops/what-is-devops/.

[2] Agile methodology: What is agile software development model? URL https://www.guru99.com/agile-scrum-extreme-testing.html.

[3] Jez Humble Nicole Forsgren, Dustin Smith and Jessie Frazelle. State of devops. URL https://services.google.com/fh/files/misc/state-of-devops-2019.pdf.

[4] Johan von Hacht. Introduction to load balancers, 11 2020. URL https://github.com/KTH/devops-course/blob/2021/attic/2020/contributions-2020/essay/johvh/Introduction_to_Load_Balancers.pdf.

[5] Application deployment and testing strategies, 2 2021. URL https://cloud.google.com/solutions/application-deployment-and-testing-strategies#testing_strategies.

[6] Jason Skowronski. Intro to deployment strategies: blue-green, canary, and more, 1 2018. URL https://dev.to/mostlyjason/intro-to-deployment-strategies-blue-green-canary-and-more-3a3#:~:text=Canary%20deployment%20is%20like%20blue,part%20of%20the%20production%20infrastructure.

[7] Session persistence. URL https://docs.oracle.com/en-us/iaas/Content/Balance/Reference/sessionpersistence.htm.

[8] Monitoring software, . URL https://www.techopedia.com/definition/4313/monitoring-software#:~:text=Explains%20Monitoring%20Software-,What%20Does%20Monitoring%20Software%20Mean%3F,a%20computer%20or%20enterprise%20systems.&text=Monitoring%20software%20is%20also%20known%20as%20computer.

[9] Method stub, . URL https://www.techopedia.com/definition/3731/method-stub-software-development.

[10] Chrome release channels. URL https://www.chromium.org/getting-involved/dev-channel.

[11] Brian Moyles Ed Bukoski and Mike McGarr. How we build code at netflix, 3 2016. URL https://netflixtechblog.com/how-we-build-code-at-netflix-c5d9bd727f15.