CONTENTS

1 Introduction **1**

1.1 Post Release Testing supports developers to efficiently do operations work 1

2 Continuous Delivery only covers practices until release **3**

2.1 Continous Delivery disregards security and operations topics 3
2.2 Fast time to market is crucial 3
2.3 Continuous monitoring is hard 3
2.4 Simple day to day work must be automated 3
2.5 How to read this masterthesis 3

3 State of the art technologies and practices are the foundations for NPRT **5**

3.1 The Continuous Delivery Pipeline consists of commit, automated testing and deployment 5
3.2 Docker packages applications 5
3.3 Kubernetes is a cluster operating system 5
3.4 Monitoring a highly dynamic infrastructure is role centric 5

4 Nonfunctional Production Regression Testing extends the Continuous Delivery Pipeline **7**

5 Deployer implements and automates NPRT **11**

5.1 Version centric testing via commit hashes 11
5.2 A canary and its testing metrics know about themselves 11
5.3 Controlling deploys in the pipeline and manually 11
5.4 Comparison of versions triggers webhooks for further actions 11

6 Evaluation 13

6.1 How NPRT changes the behaviour of development teams 13
6.1.1 Deploys 13
6.1.2 Cycletime 13
6.1.3 Change 13
6.1.4 True/False Positives/Negatives 13
6.2 NPRT compared to other in production testing strategies 13
6.2.1 Netflix Simian Army to intensify NPRT 13
6.2.2 Synthetic Monitoring is functional post release testing 13

7 Conclusion 15

7.1 Resume 15
7.2 Outlook and future work 15

CHAPTER 1

INTRODUCTION

1.1 Post Release Testing supports developers to efficiently do operations work

Many people talk about DevOps as well as there are multiple definitions and interpretations of the term DevOps. DevOps is referred as a philosophy, a culture, practices and specific tools. For my research, I will focus on two different aspects of the term DevOps:

The first one is the perspective of operation teams. Operation teams traditionally modeled infrastructure by installing physical hardware and by manually installing software components. With the rise of virtual machines and the cloud, it became possible to model infrastructure in software[1]. Modelling via software enables operation teams to use tools and practices[2] as seen in software engineering. Infrastructure code is version controlled, tested and can be automatically deployed.

The other aspect of DevOps[3] is the perspective of developer teams. Previously developer teams were only responsible for developing new features. Software engineering practices got established and proven. One of those practices is the continuous delivery pipeline[4]. The last step of the continuous delivery pipeline is the deployment. Formerly operation teams were responsible for deploying new features. The deployment as last step of the continuous delivery pipeline shifts a responsibility from operation to de-

[1] "Infrastructure as Code" describes different dynamic infrastructure types [3, p. 30] and how to model those by code [3, p. 42].
[2] In the chapter "Software Engineering Practices for Infrastrucure" [3, p. 179-194] practices like version controlling, continuous integration are described.
[3] The book "DevOps" [1] is written in the view of a developer running a system.
[4] For theoretical details on the continuous delivery pipeline read Part II of "Continuous Delivery" [2, p. 103-140] or a more practical approach by Wolff [4].

velopment. This shows that developer teams are becoming more and more responsible for running the software, they built.

CHAPTER 2

CONTINUOUS DELIVERY ONLY COVERS PRACTICES UNTIL RELEASE

2.1 Continous Delivery disregards security and operations topics

2.2 Fast time to market is crucial

2.3 Continuous monitoring is hard

monitoring change and trying to predict the future from data

2.4 Simple day to day work must be automated

2.5 How to read this masterthesis

CHAPTER 3

STATE OF THE ART TECHNOLOGIES AND PRACTICES ARE THE FOUNDATIONS FOR NPRT

3.1 The Continuous Delivery Pipeline consists of commit, automated testing and deployment

3.2 Docker packages applications

3.3 Kubernetes is a cluster operating system

3.4 Monitoring a highly dynamic infrastructure is role centric

CHAPTER 4

NONFUNCTIONAL PRODUCTION REGRESSION TESTING EXTENDS THE CONTINUOUS DELIVERY PIPELINE

Nonfunctional Production Regression Testing is the topic of the masterthesis and we will discuss the core of it in this chapter. The designation itself describes precisely what the technique and what the new practice is about. Therefore we will go shorty through every single term in the following.

Nonfunctional refers to the metrics, which we're evaluating in the test; These metrics are only non-functional and as a consequence generically applicable to multiple applications. The term production refers to the environment. The metrics, which are collected are collected in the production environment. And finally the term regression refering to the testing strategy. The metrics will be compared between two different versions and the latter version is tested for a regression, concretely a decline of the monitored metrics.

The testing technique provides some further features, which are not included in the designation. Indeed the testing technique is completely automatable and you can continuously apply it to the new versions. The testing technique is designed in respect to failing as fast as possible and inform developers.

When it comes to automation and continuous, the thoughtful reader is now probably reminded of continuous integration, delivery and deployment. This testing techniques evolved naturally from those practices and extends those. Those already established practices support the developers until the software is deployed. In contrast to that, nonfunctional production regression testing, supports the developers during after the deploy and while the software runs in production.

To understand the testing procedure in a whole and completely, it is necessary to show a complete overview of the whole testing and production environment. We will go through the steps of the pipeline and discuss it in a nutshell as you can see in the figures.

The steps in the very beginning are known from the established practices continuous delivery and deployment respectively. But it is necessary to touch them and integrate them to the whole picture and outline the special characteristics for the new testing technique.

At the very beginning, there is a developer, who changed the code locally on his working machine. There is also a version control system. And in the first step the developer commits the code to the version control system. The main focus is on git. Subversion is possible, too though. In the following description we will stick to the terms and notions of git.

Next there is a continuous integration system. After the commit happened, the second step is a message to the continuous integration system. The message holds the information that a new commit exists and the continuous integration server clones the code from the version control system and checks out the specified version. Now the continuous integration system has three major jobs. The first one is to start a build process, the second is to run the tests and the third is to give the deploy signal.

In step four, namely running the build, it is typical to compile binaries, render assets and further artifacts. For our purposes it is especially necessary to build at least one or multiple docker images.

The import thing about this is, that we need to identify every docker image to a specific build. Therefore we use the commit hash, the version created by the version control system. This commit hash will follow us through the whole pipeline. This is important to be able to trace every step in the pipeline for a specific version. With this thought in mind, the docker image is tagged with the commit hash of this version and the name of the branch. Along the way it is mentioned, that the branch name is not absolutely necessary to definitely determine the version. The branch name is included for better readability for a developer and approximately recognize what the image version is about.

The continuous integration server then pushes the ready build docker image to an image registry, such as docker hub. Nevertheless this can be a private registry as well. This registry serves later as an artifact repository.

The second continuous integration step, or in total step six, the continuous integration server runs the tests. There can be multiple stages, such as unit, feature or smoke tests. Yet we do net need to recall all the details here.

The last step of the continuous integration system is to send a deploy signal. In the shown figure this is step seven. When you look at the tests, the result could on the one hand be a failure or on the other hand be successful. If the test have failed, the remaining pipeline will be cancelled and the developer will be informed. Just as you know it from a typical continuous integration system. If the tests have been successful and accordingly the build including all test stages have been successful, the continuous integration system sends the signal to deploy.

It makes sense to deploy only specific versions and not every commit. The practice which is pretty common, is that you develop new features in a seperate branch. For those version it is common to not send a deploy signal even though the branch build and tests are successful. Usually after there has been a review and a decision to deploy the changes to production, even though it is a very small change. But when the decision is made and merged into a specified branch, for instance the master branch, this version will go to production.

However, just to clarify, each built image for every single version is sent, independently of successful tests and independently of the intention to go to production, to the image repository. The reason could be a staging system and even running the tests inside the build image. But this is just a side note.

So the deploy signal is given when two requirements are fulfilled: the build and tests are successful and it is a version which is planned to go to production.

Until this point, as it was already mentioned, it is just a usual continuous delivery or deployment pipeline, which is commonly used in the development process. But since nonfunctional production regression testing is a technique, which is supposed to be completely automated, such a prior described delivery pipeline including automated deployments is precondition. From now on it is becoming interest-

ing, hence the testing technique supports the developers post deployment in production instead of the old practices before the deploy.

The next unit is the deployer. It is the software, which is particularly implemented for this masterthesis. In the next chapter the deployer is described in detail. This chapter demonstrates how the deployer is embedded in the pipeline or in other words in the environment amongst all other tools. In the meantime tools exist, which have a similar purpose. It is crucial to have full control over the whole deployment process and as a consequence it was necessary to implement the software and have it customizable.

CHAPTER 5

DEPLOYER IMPLEMENTS AND AUTOMATES NPRT

5.1 Version centric testing via commit hashes

build, test, deploy, only 2 versions in production. undeploy a canary.

5.2 A canary and its testing metrics know about themselves

5.3 Controlling deploys in the pipeline and manually

5.4 Comparison of versions triggers webhooks for further actions

monitoring validation, fail

CHAPTER 6

EVALUATION

6.1 How NPRT changes the behaviour of development teams

6.1.1 Deploys

6.1.2 Cycletime

6.1.3 Change

6.1.4 True/False Positives/Negatives

6.2 NPRT compared to other in production testing strategies

6.2.1 Netflix Simian Army to intensify NPRT

6.2.2 Synthetic Monitoring is functional post release testing

CHAPTER 7

CONCLUSION

7.1 Resume

7.2 Outlook and future work

Bibliography

[1] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: a software architect's perspective*. Addison-Wesley, 2015.

[2] Jez Humble and David Farley. *Continuous Delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley, 2010.

[3] Kief Morris. *Infrastructure as code : managing servers in the cloud*. O'Reilly, 2016.

[4] Eberhard Wolff. *Continuous Delivery: der pragmatische Einstieg*. 2. dpunkt.verlag, 2016.