

Contents

1	Introduction	3
1.1	Post Release Testing supports developers to efficiently do operations work	3
2	Continuous Delivery only covers practices until release	5
2.1	Continuous Delivery disregards security and operations topics . .	5
2.2	Fast time to market is crucial	5
2.3	Continuous monitoring is hard	5
2.4	Simple day to day work must be automated	5
2.5	How to read this masterthesis	5
3	State of the art technologies and practices are the foundations for NPRT	6
3.1	The Continuous Delivery Pipeline consists of commit, automated testing and deployment	6
3.2	Docker packages applications	6
3.3	Kubernetes is a cluster operating system	6
3.4	Monitoring a highly dynamic infrastructure is role centric . . .	6
4	Nonfunctional Production Regression Testing extends the Continuous Delivery Pipeline	7
5	Deployer implements and automates NPRT	15
5.1	Version centric testing via commit hashes	21
5.2	A canary and its testing metrics know about themselves	22
5.3	Controlling deploys in the pipeline and manually	22
5.4	Comparison of versions triggers webhooks for further actions . .	22
6	Evaluation	23
6.1	How NPRT changes the behaviour of development teams	23
6.1.1	Deploys	23
6.1.2	Cycletime	23
6.1.3	Change	23
6.1.4	True/False Positives/Negatives	23

6.2	NPRT compared to other in production testing strategies . . .	23
6.2.1	Netflix Simian Army to intensify NPRT	23
6.2.2	Synthetic Monitoring is functional post release testing .	23
7	Conclusion	24
7.1	Resume	24
7.2	Outlook and future work	24

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 software¹. Modelling via software enables operation teams to use tools and practices² as seen in software engineering. Infrastructure code is version controlled, tested and can be automatically deployed.

The other aspect of DevOps³ 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 pipeline⁴. 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 development. This shows that developer teams are becoming more and more responsible for running the software, they built.

¹“Infrastructure as Code” describes different dynamic infrastructure types [3, p. 30] and how to model those by code [3, p. 42].

²In the chapter “Software Engineering Practices for Infrastructure” [3, p. 179-194] practices like version controlling, continuous integration are described.

³The book “DevOps” [1] is written in the view of a developer running a system.

⁴For 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].

In the following the structure of the thesis is outlined. Every chapter is briefly discussed, what it is about.

In the first chapter we will walk through the foundations. The chapter mentions technologies, which are used for the thesis and gives references. Furthermore it gives references to the practices which are used and are crucial for the thesis. The references are properly selected, to understand the details if they are not known and to understand what the technologies and techniques are used for. In summary those are kubernetes, continuous delivery, continuous deployment and techniques from infrastructure as code and site reliability engineering.

The second chapter is conceptual macro view to the technique nonfunctional production regression testing. The text walks through the general environment and discusses the most important concepts and how they communicate with each other. The most important steps of the continuous delivery pipeline are discussed and it explains how the pipeline is extended in order to have the technique of nonfunctional production regression testing. The text argues how the methodologies of nonfunctional production are embedded in the pipeline and how the pipeline must be extended.

In the third chapter we will get to the concrete implementation of the nonfunctional production regression testing. The chapter will go into the details of how the concept is implemented. Concretely the software deployer is described, which was implemented in the context of this thesis.

Chapter four is about the evaluation of the new approach. We will investigate the use of the technique and customized software in two different companies. The first company is Gapfish, a four year old startup, and the software department of DIN, a company which is established for a hundred years. We are going to evaluate positive outcomes, still problematic concerns and their improvements. Another part of the evaluation is the comparison to other techniques which other companies and groups developed and tested. We differentiate in their features, advantages and disadvantages.

In the last chapter, the conclusion, the whole thesis is summarized and all the chapters are resumed. Important is the second part of the conclusion, in which we have an outlook to further improvements and how the technique can be extended to have further upgrades to delivery pipelines.

Chapter 2

Continuous Delivery only covers practices until release

2.1 Continuous 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 shortly through every single term in the following.

Nonfunctional refers to the metrics, which we're evaluating in the test; These metrics are only nonfunctional 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 referring 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 not 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 separate

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 interesting, 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.

We could also implement the logic of the deploy deploy in the continuous integration system. But we had to decide against that, because the deploy needs full access to the production system and the continuous integration system is in our case outsourced to a third party company. We don not want to give other companies full access to another company. However this meant, that we had to implement some steps again, which a continuous integration server already implements.

The deploy message, which the continuous integration system sends to the deployer, includes the commit hash again. And again we use the commit hash to identify the version. Now the deployer executes three major steps: In the first step the deployer fetches the code from the version control system. This is the same thing, which the continuous integration system does. The repository is necessary, because it holds the files which describe our infrastructure. We want to version control the definitions of our infrastructure in order to be able to relate the version of the infrastructure to the version of the code and the version of the artifacts. At kubernetes those infrastructure definitions are made up of different resources, which were already mentioned in the foundations chapter.

In the second step deployer modifies those infrastructure resources. The

infrastructure should use the related docker image and be aware of its own version.

And the third step or the ninth step in the figure, deployer applies the modified resources to the production system, which is shown in the figure as well. We note that not only application code changes are deployed, but also infrastructure changes. We note that we deploy the infrastructure changes in a continuous fashion as well.

Next we go on with the process of how the production system updates itself. The production system is a kubernetes cluster, which the previous chapter foundations describes. For now, we assume, that the production system runs a typical three tier webapp. The webapp is made up out of a loadbalancer, multiple stateless webserver and a stateful database. The loadbalancer balances the requests between the webserver. And the webserver communicate with the database. And the database consists of a replicated cluster.

We explain firmly how the typical three tier webapp is implemented in kubernetes. For more details refer to the references given in the foundations chapter. With kubernetes we define the mentioned elements with the kubernetes resources: service, deployment and its pods and statefulset. For simplification, we imagine the service as a loadbalancer. Then there is a deployment, which manages the existence of the identical pods. And lastly the statefulset, which manages the stateful pods, with their unique name and disk.

So the loadbalancer receives a request from the client. The loadbalancer selects a pod via round robin and proxies the request to the pod. The pod probably communicates with the database and sends the request back to the client, where the loadbalancer acts again as proxy.

We now look at the change, which happens to the production system. Earlier, deployer made changes to the definitions of the kubernetes resources in the deployers memory and communicates those changes to the production system. We are interested in particular in the deployments. As an illustration we only look at what happens to the deployment and the pods.

The deployer talks to the kubernetes master api and sends the changes. The master manages the concrete changes. It swaps out one pod by another by stopping the pods in the old version and starting the pods in the new version. The procedure is called rolling update. Now the pods are all swapped out and run in the new version.

Another part of the cluster are the monitoring agents. They pick up monitoring metrics in different ways. They collect the data, we are interested in, for the regression test in production, as well. We are talking about the non-functional metrics. We selected the metrics, defined by the four golden metrics of google's sre (identified). The metrics are throughput, latency, errorrate and utilization.

With utilization we are lucky, because kubernetes already implements a collection of cpu and memory. But we need to instrument the application to collect the other metrics, throughput, latency and errorrate. So the pods send

the instrumentation data to the monitoring agent. The monitoring agent is an statsd server, which collection the data and aggregates the data and forwards it to the monitoring system.

The monitoring system consists basically out of a timeseriesdatabase, a graphing user interface and a alarm system. The timeseriesdatabase persists the metrics. And the user can define graphs from those metrics, which the user interface presents visually. You can define rules in the alarm system, which monitor the metrics in the timeseriesdatabase and then, in case the rule is violated, sends notifications.

We are interested in the monitoring data of specific versions. Consequently the monitoring agents needs to send the monitoring labeled with the specific version to the monitoring system. This is important, since we want to compare the metrics of the different version.

We have different possibilities to compare those version. One possibility is, that we compare current and historical data. For instance to compare the metrics of the current production system with the metrics of the production system of the day before or even the week before and compare the different versions of those times.

We are following a different approach, because when we are comparing the current production system with the production system of last week, we have lots of different changes. The current traffic must not be the same traffic as last week, the load of the production system must not be the same load and other system with which the application is interacting with must not be the same.

That is why we decided to compare two different versions which run in the production system concurrently. This brings not only the advantage, that you have the very same traffic, but also the advantage, that there is less risk involved. We illustrate the advantage of less risk now by demonstrating the process of deploying the second version and comparing it to the old version.

Ok, if you compare the two versions with each other, you will do at as follows. Deployer create another deployment resource from the one that already exists. Deployer calls this other deployment resource canary deployment. The creation of the canary deployment resource has the effect, that not only pods of version I are running in production, but there is pods running in version II as well. Similar to the regular deployment, the canary deployment defines how many pods in which version are supposed to be running.

We want to test, if there is a regression respectively a degradation between the two versions. On account to the fact, that a regression is possible and when introducing change, a regression is very likely, we at least want to affect as little users. So what do we do for that? In our example there three pods running in version I and only one pod in version II. This is a ratio of three to one and due to the fact that the loadbalancer uses round robin as the scheduling algorithm, only one in four requests, so 25% of the total traffic is sent to the pods in version II, which is to test.

This certainly lowers the risk of failure and that users are affected by a regression. Even if the request of specific single user hits the degraded pod, the next request of the same user has the probability 75% to hit the old stable version.

A limitation to this technique is that the new version II needs to be able to run side by side with the old version I. In most cases, that means that the new version needs to be semantically almost identical to the old version. So version II should not provide functional changes compared to version I, but only nonfunctional changes. However that means we cannot test new feature like in an A/B test. Instead we can test performance improvements, refactoring or updates.

They call this technique canary releasing. Again, you change would only change a part of the production system, the canary instead of the whole. Devops TODO examines this technique in more detail.

Assuming we would want to test features in production, the current implementation of the technique is not suitable. If we wanted to do that, we would need to include the loadbalancer. The loadbalancer would need to remember which user is proxied to which version, so that the next request of that user goes to the same version, thus the user sees the same set of features as before. The design of the database could potentially be affected as well and could be needed to be loadbalanced for the users. The technique we just described is usually called an A/B test. The disadvantage of the A/B test is that the same user will hit on the same potentially degraded service and it is not that simple to automatically provide a stable service to the user. Due to simplification, we did decide to not include the implementation of the loadbalancing.

We want to state that it is suboptimal to run multiple versions in the cluster like also mentioned in devops TODO. Rolling updates require it to be able to have two versions in production, though. And kubernetes utilizes rolling updates as a technique to provide zero downtime deployments. Accordingly our proposed technique does not introduce a worsening to that. But as in devops mentioned, you should avoid to run more than two versions at the same time in production. Deployer ensures that by either updating a deployment, creating a canary deployment, or creating a deployment in a new version, just before it deleted the canary deployment.

Especially to test the latter, security updates, is absolutely appealing, since we can fully automate the procedure of updating the dependencies of our application in a fully automated and in a way, which would have a very low risk. We could have a job, which checks frequently for any new version, pushes the updates to the version control system, the continuous integration system runs the pre deploy tests, deployer deploys the update and even in production we check the update for an regression. We could save a lot of developer time, who would usually need to take care of the whole updating procedure. And even if there is a degradation in production, a small amount requests is affected, because we send only a reasonable amount of traffic, which arrives at the same

time, to the potentially degraded version. Further more we limit the time the degraded version is in production, because we automate detection of the degradation and the rollback to the old stable version.

Now version I and version II send metrics via the monitoring agent to the monitoring system. We tag the metrics with the specific version as well. The monitoring system stores the two comparable metrics of the two different versions in the timeseriesdatabase. Now you can define a graph to the metrics of version I and you can, too, define a graph of the metrics in version II. We then compare the two graphs by for instance subtracting one from the other and monitor the result. We let this running for a specified time in production. We need to decide on how long we want to compare the versions. That depends on how much traffic is in production, because when we would have few traffic in production, we wanted to compare for a longer time. We suggest to have a well balanced test scenario in terms of load. The time depends on how much traffic there is in production and how often a team wants to deploy its application. A team which is working with a monolithic application has the disadvantage, that every change in every part of the software causes a deploy and deploys are more frequent. This limits the time in production for the canary. Instead if we have a microservice environment, the deploy affects only a specific service, hence little part of the whole application. As a result deploys are less frequent and we have more time for the canary in production.

We do not need to generate the test traffic, we do not need to weight traffic and we do not need to think about edge cases. These are all advantages, that we get for free from the production traffic. We save time and work, because the users generate the test data, instead of us. The users create more requests and with that test data for parts of the application, which are more important. Consequently the users reasonably weight the test data. And lastly the longer we run the comparison in production, users will produce more of those edge cases, which would be hard to make up.

We are aware of that the two compared versions do not receive the very same requests. Hence the comparison is not perfect. In future work we could extend the technique to achieve that. We could simply clone the requests, send the original request to the stable version and send a cloned request to the canary. The loadbalancer could then differentiate between the two responses of the two versions. We would reject the response of the canary. And we would forward the response of the stable version.

As a result we even lower the risk, because the potentially degraded version does not even respond to real users. Ergo we do not have any risk of a degradation of our production service which we cause by testing the new version.

Now there are two different scenarios. The first one would be, that the canary runs in production for a certain amount time and the monitoring system does not identify any regression. Afterwards the monitoring system triggers the full rollout of the new version. It sends a deploy message of the new version

to deployer, and deployer deletes the canary and modifies the other deployment to have the new version and the production system proceeds and stops and starts the pods accordingly.

The other scenario is that the new version turns out to be a regression to the old version, or in other words, that we have a deviation, which is above a certain threshold and which is regression instead of an improvement. Then we want to rollback the canary. Our monitoring system identifies the deviation and it sends a message to deployer, as shown in step 12 of the figure. Accordingly our test for regression is failed.

Deployer receives the rollback message and sends a deploy to the production system. Just to be precise, it is actually a deploy message with the commit hash of the stable version, which the monitoring system knows about because of the metrics it monitors. Deployer now modifies the infrastructure definitions that there is no canary deployment anymore and the production system itself takes care of deleting the canary. Since the version to deploy is already in production, the production system will not touch the other deployment and its pods.

We illustrated that we are able to automate the whole testing procedure, which is the advantage of nonfunctional production regression testing. So we can extend the continuous delivery pipeline in a natural way and we support the developers not only until the deploy, furthermore we support the developers during run time. The pipeline is now advanced in a way, that even jobs, which for instance are able to update dependencies of an application at a very low risk. The update would go through the whole pipeline including tests in a testing environment, an automated deploy and tests in the production environment. If the nonfunctional production regression test and automated update is successful, developers would be completely free of work. The changes of bots would be robust enough to be able to act fully automated and unguarded, but we would still test the changes they make and can be sure to not have a major degradation to our production system.

Finally we want to summarize, what we examined so far: The whole process of nonfunctional production regression testing is fully automatable, every step is determined and traceable throughout and even reproducible until the deploy on the basis of the commit hash. We can save time not only by automization, but also by not having to write load and integration tests with edge cases, which occur in production. And ultimately the amount of work and time saved comes at a low risk.

Chapter 5

Deployer implements and automates NPRT

This chapter discusses the software deployer. Deployer is the implementation in account to the new testing procedure nonfunctional production regression testing. The chapter does not go into implementation details, but describes how deployer works and how to use deployer from a user's perspective. The chapter does not discuss the surrounding applications, such as implementations of version control systems, continuous integration systems. Whereas we discuss the monitoring system, since the testing rules for the metrics are implemented with it. And we will go into more detail how deployer is integrated with kubernetes, since this is an important part.

Initially we want to mention in short the technologies we used to implement deployer. It is implemented in ruby and since it is a webserver, sinatra is it's framework. Deployer uses git and svn to communicate with the control version system. Moreover deployer has a plugin which sends messages to bugsnag and slack, to inform the developer in case it identifies a problem. Also to mention is that deployer uses kubectl to interact with kubernetes master. Deployer itself is containerized with docker and its natural hosting solution would be kubernetes. Deployer is published on github under GPL-3.0, an free software license.

The monitoring system, which we used in our evaluation, is datadog, a commercial service. Nevertheless prometheus is also a suitable open source solution as monitoring system.

Similar to the previous chapter, we demonstrate the implementation with the help of a figure, which shows the whole picture of the implementation. And we are going through the parts of the figure.

As mentioned earlier, deployer is a webserver, which runs in kubernetes itself. The ... in the figure is the kuberntes cluster, which is also our production environment. There is a kubernetes resource service. The service is, as we already know, a loadbalancer and clients can reach deployer via an http

interface. The figure show us that deployer is composed of multiple pods. The pods are replicated and identical in their behaviour. The pods are basically stateless, yet every pod has its own caching layer, which we will discuss in a later section. We assert, that because of the statelessness, we can scale deployer horizontally and can just add more pods. Along the way, deployer uses approximately 50 mega bytes ram and very few cpu, so resources shouldn't be a problem. A deploy takes about 10 seconds. This depends on the size of the repository and the download rate from the version control system. A request is blocking and waits for the deployment process to complete. Certainly we do not need a queuing system here and can scale deployer to the concurrent deploys we need.

Next we want to go through the steps involved in the deployment process. Initially a client sends a http request to deployer. Therefore the client needs to authenticate via http basic authentication. In other words, the client authenticates via a username and password. As we mentioned earlier, our continuous integration system is a software as a service solution, and one of the reasons why we could not implement a deploy logic inside the continuous integration system, were security concerns. In this case the token authorizes to only deploy a specific version from the repository. What that means for us, is that the client is not able to do everything to our production system. It is for example not able to deploy other code than ours or read credentials from the production cluster.

So the loadbalancer proxies the request to a pod and after deployer authenticated a valid user, deployer starts the deploy procedure itself. At first deployer validates the given arguments by the client. The deploy request requires the service to deploy and a version of the service. The client needs to specify either the commit hash or the tag name or both. The tag would additionally include the branch name, which is used for readability reasons.

Deployer initially validates the arguments from the request. To be specific, it checks if the service exists in the configuration. Furthermore Deployer checks if the commit exists in the repository and deployer checks checks if a the tag, which is associated to the commit, exists for all the images, which are necessary in order to deploy the service. Deployer executes the latter validation by communicating with the docker registry. The client, which can be depctl or curl, do in contrast not have any validations. The reason for that is that we want to have a single definition for the validations.

We designed the deployer deploy interface to have the option to only give either a commit hash or a tag, because it makes things a lot easier when a developer needs to deploy manually. It makes the development flow more efficient. The commit is totally sufficient to determine the version, so the clients usually pass the commit hash only.

After deployer validated the deploy request, it then fetches the code in the version control system lies. When there is git as the version control system, we are using commit hashes and branches to determine the application

version. And when there is subversion, we are using revision numbers. We implemented subversion, which is more of an ancient technology, to be able to do the evaluation with DIN legacy system.

As we know, the process of cloning a repository includes persistence and disk interaction. That is true, but deployer uses its volume just as cache. The reason why there is a cache at all is of course performance. One of the bottlenecks of the deploy is to download the repository. If a repository is big, for instance because of images or it is a repository with a long history of commits, it lasts quite an amount of time to download it. And if the download rate, the throughput, is very low the duration is even longer, which is bad. Therefore deployer keeps the already downloaded repositories on volume, as a cache, and the next time it deploys the same repository, but in a different version, it only downloads the changes.

As a simplification, every pod has its own cache and it is not very long living. So every pod utilizes its own volume as a cache, so the cache is integrated in the pod and there is no extra instance or communication to a caching service. Since docker containers are immutable, every time kubernetes recreates a pod, docker destroys the volume. Thus the old data is not available anymore and in other words the cache is empty again.

The version control system contains not only application code, moreover it contains the infrastructure code as well. In kubernetes especially these infrastructure definitions are different resource definitions. These resources are deployments and statefulsets and so on. These resource definitions should be stored in the 'kubernetes' directory. This is a convention and deployer assumes that this is the location. If an application is not able to locate it there or does not want to locate it there for any reason, the configuration of deployer provides an option to reconfigure this location.

The configuration of deployer is very little, because we designed deployer to have conventions. This is the design, which we know from the rails community as well. It has the advantage of good defaults instead and there is not the need for everyone to invent a good configuration.

The configuration of the kubernetes resource path enables two different methodologies in the microservice approach. The first one is the multiple repository methodology. In that methodology we have for every single service or microservice a dedicated repository. Deployers configuration would have for each service the repository url, which would be sufficient. Second methodology is the single repository, which contains multiple microservices. There we would have multiple services defined with the identical repository url. However deployer can differentiate between services, due to the fact that the configuration for each service has a different location for the kubernetes resource definitions. An example for these different configurations are shown below TODO.

After having the specific commit hash of the repository checked out, deployer takes the resources and modifies them in a next step. Especially deployments and statefulsets are relevant to that modification the deployer makes.

In contrast to those resources, deployer does not modify all other resources, such as services.

Deployer applies two modifications. The most important is the image version. The version of a docker image is specified by the docker tag. And if the deployment already specifies a tag, such as 'debian:wheezy', deployer does not change that tag. The reason for that is, that there may be containers, which the continuous integration system does not build and tag with the specific commit hash. Commonly another party maintains that docker image. An example is an additional statsd container running in pod side by side to the application container.

The other case would be that the tag is not specified in the kubernetes resource. Then deployer appends the tag to the container image according to the either given commit hash or given tag.

Sometimes something unexpected is happening during the deployment. This can be for example that a verification of deployer fails, such as deployer does not find the tag corresponding to the commit. Or maybe kubernetes api server returns an error for any reason. If something like the described happens, deployer will inform the developers. We distinguish between two scenarios. One is, when the continuous integration system requested the deploy. There our already mentioned bugsnag plugin takes hold. Deployer raises an error and the error is send to bugsnag, which collects all the errors of systems. Bugsnag informs the developers via sending a message to a specified slack channel. Slack notifies the developer who is on call about the bugsnag message and the developer knows about the problem. Another possibility would be, that the one who created the commit and triggered the deploy is informed.

The other scenario is that a developer manually sends a deploy request. In that case the developer typically uses depctl. Deployer discriminates usual curl requests from requests with depctl. Instead of utilizing bugsnag, deployer answers with an error in the http response. We can do this due to the fact that the http requests are synchronous.

The sixth step is the communication with kubernetes. Deployer now sends the modified resources to the kubernetes api, which the last chapter already described. In this chapter we focus on the comparison between the stable version and new version and how deployer deploys the canary for the new version.

To deploy the canary, the client sends the deploy canary request to a separate http path, which is different from the usual deploy and which deployer defines as well. The client needs to provide the same arguments as for the deploy request. Namely those arguments are the service, which is required, and either a commit hash, a tag or both. When deployer received the canary request, it proceeds almost identical to the deploy request. The steps are verification, version checkout, modification and application to the kubernetes api. The difference to the deploy lies in the modification step. Deployer create a new canary deployment resource.

In the modification step deployer does not only change the container image version, but it also changes the name of the deployment, which would then represent another deployment. Deployer names this new deployment resource with the suffix canary. Since deployer changes the name, kubernetes does not identify the canary deployment as the original deployment. Instead it treats the canary deployment as a separate resource and creates pods for that deployment in the different version, which deployer specified.

The service though, which does our loadbalancing, selects both. It selects the pods which were created from the original deployment. And it selects the pods, which were created from the canary deployment. Hence the service selects its pods on account of the label. And the original and the canary deployment share some of their labels, but not all the labels. In our example the shared label is 'deploy=webserver' and the different labels are 'track=stable' respectively 'track=canary'.

As it was mentioned in the previous chapter, we want to have fewer pods in the canary version than in the stable version. Deploy does this simply by scaling the canary deployment to only one replica.

Now we want to focus on how we collect the metrics during the two versions are in production. We differentiate between two different collection mechanisms. There is the group of metrics, which are picked up on the host. The host has information about the pods' utilization of cpu and memory. So on each kubernetes host, there is a monitoring agent running, which watches the proc directory and the docker daemon, picks up the information frequently and then sends it to our monitoring system.

The other group is the metrics, which the application sends. We need to have the application instrumented in order to collect metrics like throughput, latency and errorrate. In practice we use the statsd protocol and statsd server for that purpose. There are statsd libraries for the most languages and frameworks and we get the instrumentation out of the box. To summarize the process of the data collection shortly, the instrumented application sends the data to the statsd server in each request and the statsd server aggregates the data. From there the statsd server forwards the aggregated metrics data to our monitoring system.

It is important to correctly label the metrics independently from which version we collect the metric. The monitoring system is then able to distinguish between the metrics of the stable version and the metrics of the canary version. For the first group of metrics collection, the monitoring agent can pick up the label from the labeled pod. And for the second group of metrics collection the instrumentation code of the application picks up the label from environment variables, which have been set by deployer, either stable or canary.

Requests to deployer are very simple. We can easily create a curl request and execute it from our continuous integration system. Another way to interact with deployer is to use the depctl command line interface. Developers use this tool to manually deploy as well as during development and to stage versions.

Depctl is a very helpful tool, because it provides shortcuts to deploy a version to a staging system, without having the need to wait for the whole delivery pipeline. With depctl we can easily skip steps like tests, which would run on the continuous integration system. This makes the development flow more fluently.

In the following we will go through the interface of deployer. Deployer provides different endpoints: ls, show, tags, deploy, canary and version. The design of the deployer api feels more like remote procedure calls instead of a restful api. This simplifies the calls with curl and approaches a more intuitive way of deploying a service. Nevertheless the endpoints are very similar to a restful api. With index, show and tags you can read the information services, whereas deploy and canary update the services to a specific version. Deployer does not provide create or delete, though. To add or remove services from deployer, you edit the configuration.

A client speaks to deployer by either http calls or by simply using depctl command line interface, which wraps the http calls and assists the developer by automatically completing information such as the repository to deploy and the version to deploy. You find the http call and the corresponding depctl command in the following tables.

As mentioned earlier you need to call deploy and canary on a specific repository and you need to provide either a commit hash, a tag or both. This call updates the service, so this is why we are using a http put call. An example curl request would be

Endpoint	depctl command	Parameters
GET /	ls	
GET / SERVICE	show	
GET / SERVICE /tags	tags	
PUT / SERVICE /deploy	deploy	commit, tag
PUT / SERVICE /canary	canary	commit, tag
GET / version	version	

Examples:

```
curl --data commit=025838f23a70f5d1df5851c31fa6e36dc0d58b9b \
      https://auth_token:secret@deployer.company.com/gapfish/deploy
```

```
depctl deploy
```

The ls or index endpoint returns the configured services and the show endpoint for a specific service shows the current configuration for that service. The tags endpoint show the available tags for that service. Deployer queries the docker image registry for all available tags for all images in a service and

returns them to the client. The version endpoint simply returns the deployer version.

The monitoring system provides a tool, which is simply called monitor. And those monitors continuously evaluate metrics by given rules. So when we have the metrics in the timeseriesdatabase, the monitors draw continuously graphs from those timeseriesdatabase metrics. As an example we have a graph for our errorrate. In the example we have first have a graph for the stable version's errorrate. The errorrate is very low as you expect it. The other graph is the graph of the canary version and this graph has some higher errorrates.

Next we define a composite graph. That would be the graph of the canary version and then subtract the graph of the stable version. So if the resulting graph goes over the threshold of 10 errors per second, a alarm is triggered. The monitoring sends the an alert to deployer, which is a deploy command with the commit hash of the stable version.

The monitoring system we were using in our evaluation, was datadog. But there is also the opens source monitoring system prometheus, which provides very similar features. Also google has a similar monitoring system in there internal infrastructure.

Finally in this chapter, we summarize what we have discussed so far. We went through the infrastructure of deployer, we saw, that deployer scales horizontally. We have seen that deployer has simple interface, with the two most important commands deploy and canary. We went through a canary deploy and saw that deployer validates the given arguments and how it notifies developers if an error occurs. We discussed how and what modifications deployer does to the kubernetes resources. We demonstrated the different test metrics and how the production system sends those metrics to the test system. We went briefly through the deployer interface and which commands are all provided. We had a look on the monitoring system and the tests itself and how we rollback the canary in case the test fails.

Another nice thing to mention about deployer, is that it deploys itself, which fits to the declarative model and recursion. That means we develop deployer itself with the continuous deployment flow and can apply nonfunctional production regression testing to deployer.

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.