NONFUNCTIONAL PRODUCTION REGRESSION TESTING

---

implemented with Kubernetes

. . .

Contents

. . .

1 INTRODUCTION

1.1 Automation to Support Maintenance Work

Devops [2] and microservices [17] are practices or even cultures, which break down the silos between the development and operation teams and their conflicting goals. For instance Amazon and Facebook successfully use those practices to build and run their software products [18, 13].

However the responsibility of the maintenance of a software product changed. A single team is responsible in two aspects: their own code and the dependencies they are using and relying on. The need for maintenance of own code exists, for instance with nonfunctional changes concerning performance, scalability or bug fixes. Code written by others needs to be maintained as well via adapting and updating to new versions.

DevOps practices interconnect operations including those maintenance necessities with development work. Nevertheless there is room for improvement in terms of continuous automatable processes regarding maintenance. The Continuous Delivery (CD) pipeline [20] is such a continuous automatable practice. But it supports only until the deploy whereas maintenance involves more than just the delivery.

Instead of employing more engineers, Google faces workforce problems with automation [28]. They recognized that automation does not only save valuable programmer time, furthermore it adds reliability to the process. Google explains how valuable automation is and successfully illustrates it with their site reliability engineering approach [30].

This thesis proposes with NONFUNCTIONAL PRODUCTION REGRESSION TESTING (NPRT) a practice, which faces the amount of maintenance work. The practice is designed for teams, which have the responsibility of developing and operating their own code as well as code dependencies. NPRT extends the CD pipeline and automates continuous testing of nonfunctional changes in production.

1.2 Thesis Outline

2 Problem Definition Chapter 2 elaborates on the problem of the amount of maintenance work and demonstrates how the thesis contributes to solve the issue.

3 Background Chapter 3 briefly summarizes the background and provides good references to the practices and technologies, which the thesis presumes. Furthermore, it shortly explains the major kubernetes resources on the example of a 3 tier webapp.

4 Approach Chapter 4 describes the conceptual macro view of NPRT. The text goes through all the steps of the pipeline and demonstrates the components and how they communicate with each other.

5 In Practice Chapter 5 goes more into detail of the thesis contributions. It demonstrates the design of Deployer, the software I wrote in account to enable NPRT in practice. The chapter also elaborates on the collection and comparison of the metrics.

6 Evaluation Chapter 6 evaluates NPRT in three aspects. Firstly, it discusses the performance with the help of experiments. Secondly, it provides a qualitative evaluation of NPRT. And lastly, it relates NPRT to other practices, tools and sophisticated metric analysis.

7 Conclusion Chapter 7 summarizes the thesis and suggests more automation opportunities and extensions to the implementation.

. . .

2 PROBLEM DEFINITION

2.1 Maintenance Responsibility Shifts

The DevOps [2] approach and the microservice [17] culture create practices to solve the issues of the conflicting tension [30] between a development and operation team. On the one hand the aim is to change the software product quickly and on the other hand to provide a stable and reliable software product of high quality. Vogels, the chief technology officer of amazon, once said in an interview "You build it, you run it" [18]. This cite represents the awareness of teams to not only be responsible for the development and deliver features. Moreover the team is also responsible of operating their product in production, becuase operation is an essential part of a successful software product. One team is consequently responsible of the development, operation and maintenance of their own code.

Furthermore the responsibility of software dependencies shifts. In the past, development teams delivered code and depended on the previously installed software dependencies. Operation teams were responsible of maintaining those installed dependencies. Virtualization made a change. Teams operating virtual infrastructure, operate and maintain the hardware and software of this infrastructure. They do not maintain the specific dependencies of the code, which the development team develops. The development team can choose which dependencies they want to use. Accordingly there is a clear isolation between the dependencies of the virtual machine infrastructure and the development team.

You could argue that virtual machines have still the same functionality like bare metal servers and need to be maintained in the same way by the same operation teams. Nevertheless it becomes easier to maintain dynamic virtual infrastructure [25] in comparison to bare metal infrastructure. And the easier it becomes, the more it enables developers to operate their software product themselves.

Containerization manifests the responsibility shift. It is a further step of isolation. The operating system is able to isolate cpu and memory for every process. Yet with containerization the operating system can also isolate the disk between processes, which means it can isolate the installed dependencies for every process. Containerization techniques even potentiate the ease

of managing dependencies. So the responsibility of the software dependencies shifts resultantly towards the team, which builds and runs the software product.

2.2 Lack of Focus on Maintenance

If we did not do maintenance properly, there is a risk of decreasing the quality of the software product. For instance there can be security vulnerabilities. Those can be very harmful to the company of the software product. If we do not maintain the software products properly more issues arise, such as bugs or performance problems. On account to that, maintaining a software product comes with responsibility. If we disregard maintenance and operations work, the quality of the software product suffers.

What does it mean to operate and maintain a software product in production? It is all the work related to running the software product. It includes the configuration of the servers. It includes monitoring of performance, errors and security issues. And it includes handling of incidents. For example if there are any issues, you would need to react to those issues and to make the software product for instance scale or apply security patches.

Agile and continuous practices focus primarily on feature development. Those practices altered software development to deliver features faster into production. Highsmith argues velocity kills agility [19]. Also Fitzgerald observes the issue of a "misplaced focus on speed rather than continuity" [14]. Focusing on development velocity only, has a major drawback. Operating software in production and maintaining it needs to be done properly to ensure the quality of a software product. Companies easily disregard the importance of operation and maintenance of the software product.

2.3 Underestimated Amount of Maintenance Work

The software written by the development team needs to be maintained as well. As we illustrated previously, different issues occur in production. For instance performance and scaling issues or simply bugs. The developers need to refactor the software without changing the actual functionality. Those changes are mostly invisible or do not alter any function visible to users. At Facebook [13] those invisible changes are more than 50% of the total changes.

Moreover Software products have a high amount of dependencies. This totally makes sense, because when programming a software you do not want to do it from scratch. Firstly the dependencies are different abstraction levels, such as a processor, a programming language and for instance a programming framework. And secondly the dependencies are modules which encapsulate a certain functionality. Those are very common functionalities for applications, for instance authentication of a user. The abstraction as well as the reusable software modules make it easier and faster for developers to reach their goals.

The open source communities change and release their software continuously. Sometimes those changes consist of new feature, often times just refactorings and every now and then security patches. This results in releases and new versions.

In the web, there is rarely a software product, which does not depend on open source software. The most web servers run linux and there are many popular open source frameworks, which make up the basis of the running software products.

Companies like RedHat or Ubuntu provide enterprise support to those open source projects. Those companies bundle those open source projects, care about the bugs and security issues and provide tools to upgrade from one version to another easily. As a result companies with a software product can outsource the maintenance work of the dependencies. There is still the need though, to integrate those changes to the own software product.

One problem is that it is not always easy to upgrade from one version to another. Refactorings result in breaking changes and the depending software needs to adapt to those changes. As a result software such as windows xp is still out in the wild. Due to the fact that companies are not able to adapt to those major changes at once. However the very same applies to open source frameworks such as django or ruby on rails.

One reason for the introduction of microservice is the flexibility to update only a small specific part of the whole software product [17]. With a microservice architecture we are able to update a single service without having to upgrade the whole product at once. We can choose the microservice for the update because it is more important. Or we can update one microservice by another without the major problem of upgrading the whole software product at once. Consequently this means we need to update more often and the amount work of updates itself is more.

There are several things you can do to keep up with the amount of work. Facebook [13] for example hires more developers. This is expensive though. A much cheaper solution is automation. We should automate some of the maintenance work.

2.4 Disregard of Automation for Maintenance Work

You could argue that DevOps practices already exist and connect development and operations including maintenance of a software product. However the focus lies mostly on testing the quality of a change in a lean fashion and to reduce risk of a change in production [2]. It is true that those practices help to deliver changes faster to production and help to detect issues at low risk. Nevertheless monitoring and reacting to errors requires manual work.

There exist practices, which already enable production testing [13]. For example A/B testing, where you compare two different versions in production. Or canary releasing, with which you change only a few replicated parts of the

production environment to reduce risk. There is also shadow releasing, where users are not affected by the change, which goes into production. Netflix tests its services with the simian army in production to detect unforeseen events in a controlled environment [32].

Containerization takes out the risk of updating a service. Morris describes phoenix replacement in immutable infrastructure [26]. Kubernetes has the approach of rolling updates, which implement the same practices. A containerized ecosystem is typically immutable. Rather than changing a running container, we would create a new container based on an updated image including the changes to deploy. After the new container was started, we can kill the old container. Usually an image repository holds every built container image version. Those saved images allow us to easily rollback to a previous version. Hence updates are less risky in a containerized immutable infrastructure.

Murphy et al. illustrate the value of automation [28]. Software reliability engineers automate tasks to firstly save the valuable human resources and secondly take risk out of the tasks because of human errors. This is a good approach and focuses the quality of the software product.

Immutability takes out some of the risks and automation brings value. Combining those two arguments, there is an obvious need of an automated process for updating services.

There is a lack of truly automatable practices for maintaining a software product. CD brought up good automatable practices to continuously deliver changes and features into production. It has the lack of good practices though to maintain a software product.

CD does not focus on the maintenance work and the DevOps approach does not focus on automating the maintenance work as previously examined. We conclude that there is a need for practices, which continuously support and automate maintenance work.

2.5 Contribution of this Thesis

The contribution of this masterthesis is a practice, which can fully be automated in order to maintain a software product. It extends the CD and continuous deployment pipeline as those go until release and no further. NPRT extends the pipeline to the production environment, where we need to operate and maintain a system.

The thesis suggests NPRT as a practice which consists of different techniques. Those technique cover three major steps. First, quickly deploy a change to production. Second, monitor the change, having it in production at low risk. And third in case of a regression automatically roll back.

I implemented the tool, Deployer, in order to connect essential components of the practice, NPRT. The thesis evaluates the performance of the tool as well as the the practice itself via experiments as well as in the production environment of an existing software product.

. . .

3 BACKGROUND

3.1 Prerequisite Practices and Technologies

Some background is required to understand the thesis. The glossary lists those most important practices and technologies. Moreover it provides well chosen references in order to enable the reader to glean more detailed descriptions of the prerequisite practices and technologies. Those practices are well known in software engineering as well as the concepts of the most technologies.

3.2 Typical 3 Tier Webapp in Kubernetes

I'll explain how you would implement a typical 3 tier webapp in kubernetes as it helps a lot to understand the next chapters. You would implement the application tier with a deployment, which monitors the presence of a specified number of stateless pods. A service acts as a loadbalancer and selects the pods and forwards the request. The data tier can be implemented with a statefulset, which monitors the presence of the stateful pods. Those stateful pods keep their host name and mount the same volumes when they are being restarted.

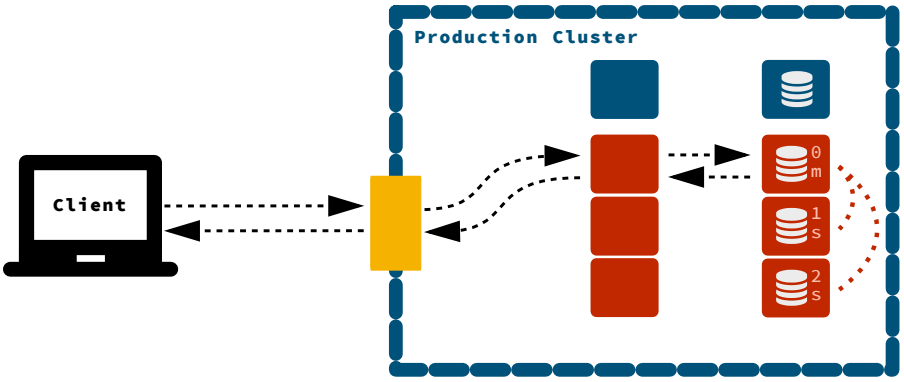


Figure 3.1: Typical 3 tier webapp in kuberentes.

So the service receives a request from the client. It 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 a proxy.

3.3 Software Dependencies

This section shortly mentions the dependencies, which I used to implement Deployer and the tools for other components of the pipeline.

Deployer is implemented in ruby[1] and since it is a webserver, sinatra[2] is its framework. Deployer uses git[3] and subversion[4] to communicate with either git or subversion[5] as the control version system. Moreover deployer has a plugin which sends messages to bugsnag[6] and slack[7], to inform the developer in case it identifies a problem. Deployer uses kubectl to interact with kubernetes[8] master. Deployer itself is containerized with docker[9] and its natural hosting solution would be kubernetes. We published deployer on github[10] under GPL-3.0, a free software license.

The previously developed GemUpdater[11] creates pull request with dependency updates.

The monitoring system, which I used in the evaluation, is datadog[12], a commercial service. Nevertheless prometheus[13] is also a suitable open source solution as monitoring system. The continuous integration systems we use for the evaluation are codeship[14] and jenkins[15]. As an image repo, we use docker hub[16]. Goreplay[17] served to make experiments with production traffic.

[1] https://www.ruby-lang.org
[2] http://sinatrarb.com
[3] https://git-scm.com
[4] https://git-scm.com
[5] https://subversion.apache.org
[6] https://www.bugsnag.com
[7] https://slack.com
[8] https://kubernetes.io, see also [24, 4]
[9] https://www.docker.com
[10] https://github.com/gapfish/deployer
[11] https://github.com/schasse/gem_updater
[12] https://www.datadoghq.com
[13] https://prometheus.io
[14] https://codeship.com
[15] https://codeship.com
[16] https://hub.docker.com
[17] https://github.com/buger/goreplay

. . .

4 APPROACH

In NPRT we automatically deploy a new version in the form of a canary. A monitoring system compares the metrics of the canary with the metrics of the current stable version side by side in production. The in production test verifies, if the nonfunctional metrics of the canary show a regression and eventually automatically rolls back the canary. This outlines NPRT in few sentences. Nevertheless the following text will go through the term in more detail. The approach is a novelty and we name the approach in the context of this thesis.

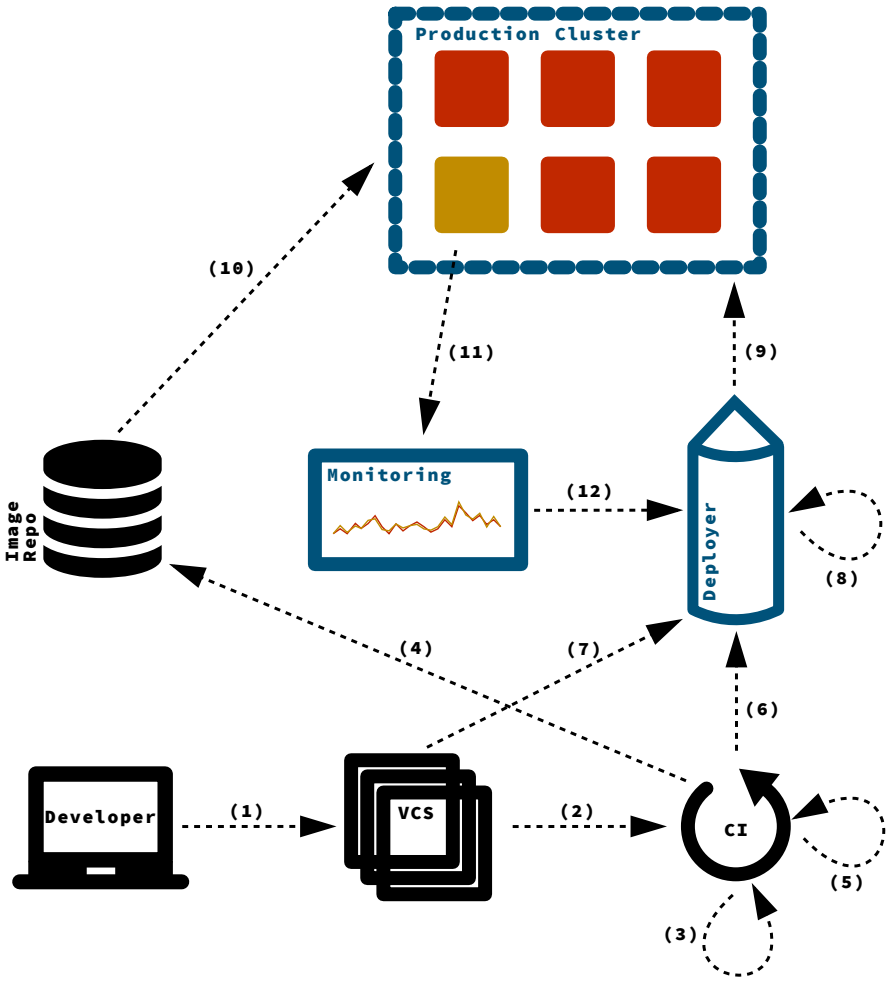
Nonfunctional refers to the change, which the tests evaluate. Nonfunctional changes are usually maintenance changes, for instance performance improvements or security updates. Production refers to the environment on account to the monitoring of the new version being in production. The term regression refers to the testing strategy. The test compares the metrics of the two differing versions, the stable version and the new version. The new version is instantiated in form of a canary. We test the canary version for a regression, concretely a decline of the monitored metrics. If the test identifies a regression, we roll back to the stable version.

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

This testing approach naturally evolves from common practices such as continuous integration, CD and continuous deployment and extends those practices. The already established practices support developers before and until the software deployment. In contrast to that, NPRT, supports developers during and after the deployment. In other words it supports the developers to run applications in production, which formerly has been a business of operations teams.

4.1 Pipeline Overview

To understand the testing approach in a whole, it is necessary to show a complete overview of the whole pipeline and environment. Figure 4.1 pictures this overview. In the following we will go through the steps of the pipeline and discuss them.



(1) push new version
(2) ci pulls code
(3) build
(4) save container image(s)
(5) run tests
(6) deploy
(7) pull infrastructure code
(8) modify resource definitions
(9) apply to production cluster
(10) pull container image(s)
(11) send metrics
(12) trigger rollback

Figure 4.1: Overview of the NPRT flow.

4.1.1 Continuous Integration is a Requirement

The first parts of the pipeline are commonly known and established practices: continuous integration, delivery and deployment. It is necessary, though, to touch on them and integrate them in the whole picture. It will illustrate important design characteristics for the new testing approach.

First the developer changes some code on his local machine and creates a new version. He then pushes the new version to the version control system as shown in (1).

After the push of the new version, the second step is a message (2) to the continuous integration system. This message holds the reference of the new version and the continuous integration server pulls the new version from the version control system. Now the continuous integration system has three major jobs. Firstly it starts a build process (3)(4), secondly it runs the tests (5) and the thirdly gives the deploy signal (6).

In step (3), namely the build, the continuous integration system typically compiles binaries, renders assets and may create further artifacts. For our purposes it is especially necessary to build at least one or multiple container images. The continuous integration system then pushes the ready built container image to an image repository in (4). Later the image repository serves the built images.

One import thing is that we need to associate every built container image with a specific version. Therefore we use the version reference, which was created by the version control system. It is important to be able to trace the version through every step in the pipeline. With this thought in mind, the continuous integration system tags the container image with the version reference and an extra name. Just to mention that the extra name is not absolutely necessary to definitely associate the container image with a version. But it turned out that a human readable name is very helpful to recognize a version and to know what the version is about at first sight. The continuous integration system derives that extra name from the branch name. This tag, consisting out of the version and extra name, will follow us through the whole pipeline as readable and unique reference.

The second continuous integration step is to run the tests. This is shown as step (5) in figure 4.1. The tests themselves can be split into multiple stages, such as unit, feature and smoke tests. Yet we do net need to recall all the details of automated testing at this point.

The last step of the continuous integration system is to send a deploy signal, step (6) in the figure. But it depends on the results of the tests, whether to send that deploy signal or not. The tests can be successful or fail. If the tests fail, the continuous integration system does not send a deploy message, the pipeline stops and the continuous integration system may inform the responsible developer. If the tests are successful in all test stages, the continuous integration system will send the deploy signal to the deployer.

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 versions you usually don 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, the continuous integration system sends each built image for every single version to the image repository. This is independent of successful tests or the intention to go to production. The reason why we want to have every built image in the repository is to be able to test it in the continuous integration system, locally and a staging system. 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 we already mentioned, it is continuous integration practice, which is commonly used in software development. We require it for our approach and again, it is important to NPRT to associate and trace every step through the whole pipeline. For that purpose we need to especially tag the container images.

4.1.2 Customizations of Continuous Deployment

NPRT is an approach, which we designed to be completely automatable. It is crucial to not only have continuous integration, but to have a customized continuous deployment process as well.

The next component in the pipeline is the deployer. Deployer is a service, which realizes the customized parts of the continuous deployment practice. In the context of this thesis, we implemented the software. We describe deployer in detail in the next chapter. This chapter demonstrates how deployer integrates in the pipeline and in the environment among all the other tools. 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 do 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 continuous integration system pulls the version from the version control system as well as the deployer. (2 different things: one ap-

plication code, 2 infrastructure code. But good to have them both in a single repository, to have the relation.

Deployer receives the deploy signal from the continuous integration system (6). It again includes the version reference. Now deployer executes three major steps: firstly pull infrastructure code (7), secondly modify the infrastructure code (8) and thirdly send the infrastructure code to the production cluster (9).

So in the first step deployer pulls the code from the version control system (7). The version control system also holds the files, which describe our infrastructure. We want to version control the infrastructure definitions of in order to be able to relate the version of the infrastructure to the version of the code and the version of the artifacts. In kubernetes those infrastructure definitions are made up of different resources, which were already mentioned in the background chapter.

In the second step deployer modifies those infrastructure (8) defintions in a way, that the production system uses the related container image. The running container needs to be aware of the its version. The modifications of deployer achieve that as well. The latter is important to later tag the metrics with running version.

The third step of deployer is to apply the modified infrastructure definitions to the production cluster, which is shown in the figure as step (9). We note that two things are changed: infrastructure changes as well as application code changes.

The production cluster receives the modified infrastructure definitions. The production changes the cluster state according to the definitions. Most important is that two versions in parallel are in the cluster. The production cluster fetches the container image in step (10) from the image repository. The image, which the continuous integration system built in step (3). The production cluster is aware of the specific image identified by the tag, which we described before.

4.1.3 Metrics Collection and Comparison

The figure illustrates the two different versions with two differnt colors. Most of the running instances are in the stable version (red) and only one instance or in practice few instances are in the new version (orange). The loadbalancer sends traffic to both versions and both versions respond to clients, which is not shown in the figure.

The production cluster collects the data, in which we are interested for the regression tests. The collected metrics are nonfunctional metrics. We adapted those metrics from the four golden metrics of google's sre. The metrics are throughput, latency, errorrate and utilization.

We are interested in the monitoring data of specific versions. Consequently the production cluster label the metrics with the specific version.

This is important, since we want to compare the metrics of the different version. The production cluster sends those labeled metrics to the monitoring system (10).

The monitoring system stores all the metrics of the two different versions in a timeseriesdatabase. The monitoring system evaluates those metrics by drawing different graphs and diagrams and comparing those graphs with each other. One example would be that it draws two graphs for the latency in one diagram. The first latency graph is the one of the stable version. The second latency graph is the one of the new version. The monitoring system monitors now those two graphs for a regression. That means in the case of latency, that if the latency of the new version is much bigger than the latency of the stable version, the monitoring system detects this as a regression. In other words we would have a deviation of a monitored metric, which is above a certain threshold. We discuss the specific implementations of the different metrics and their comparisons in the following chapter.

4.1.4 Rollouts and Rollbacks

Now there are two different scenarios. The first one would be, that the new version runs in production for a certain amount time and the monitoring system does not identify any regression. In this case the monitoring system does nothing. A scheduled job triggers the full rollout of the new version. This job sends a usual deploy message of the new version to deployer in case the new version still exists in production and was not rolled back beforehand. Deployer receives this deploy message, deletes the canary and modifies infrastructure definitions to have the new version as the new stable version and the production system proceeds and stops and starts the running instances accordingly.

The other scenario is that the new version turns out to be a regression compared to the stable version. In that case we want to rollback the new version. Our monitoring system identifies the regression and it sends a message to deployer, as shown in step (12) in 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 is aware about because of the metrics it monitors. Deployer now modifies the infrastructure definitions similar to step (8). The important thing is that there must not be instances of the new version anymore. The production system itself takes care of deleting instances in the new version. Since instances of the stable version are already running in the production cluster, the production cluster does not touch the other instances.

. . .

5 IN PRACTICE

This chapter discusses deployer. We implemented deployer in order to realize NPRT with kubernetes. Deployer receives deploy messages, modifies kubernetes resources and applies those modified resources to the production cluster. With deployer you can automatically deploy another version of an application, which runs in parallel.

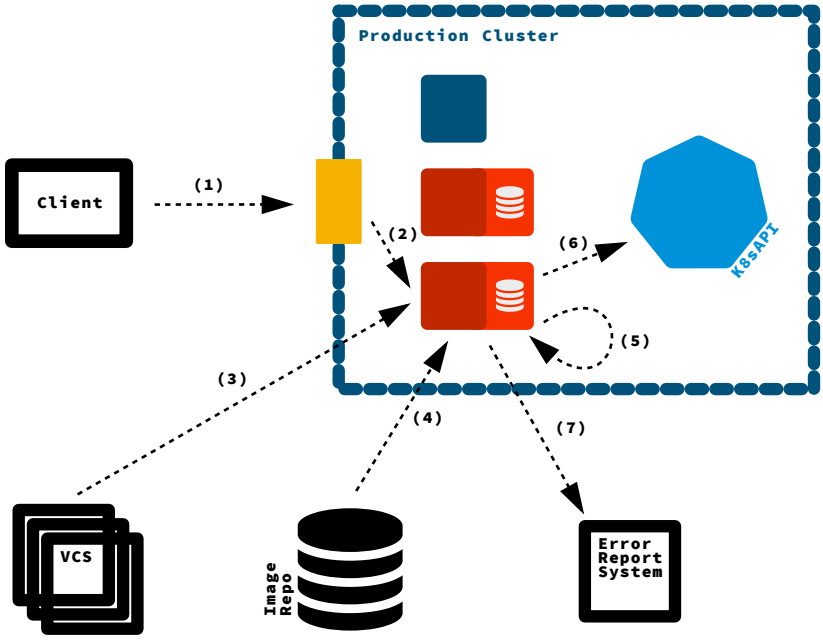
5.1 Deployer Architecture

Similar to the previous chapter, we demonstrate the implementation with the help of some figures. And we are going through all the parts of the figure.

As mentioned earlier, deployer is a webserver, which runs in kubernetes itself and is stateless. The production cluster in in the figure is the same kubernetes cluster the app runs in, which we want to test via NPRT. The yellow box is a kubernetes service. The service is, as we already know, a loadbalancer and clients can reach deployer via an http interface. The figure shows us that deployer consists 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. The statelessness leads to easy horizontal scaling of deployer. We can just add more pods if needed. Along the way, deployer uses approximately 50mb ram and very few cpu, so computing resources should not be a problem. A deploy takes about 10 seconds. This is just an orientation. Nevertheless this mostly depends on the size of the repository and the download rate from the version control system. A request to deployer 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.

5.2 Interface

Requests to deployer are very simple. We can use curl request or any other http client and execute it for instance from our continuous integration system. Another way to interact with deployer is to use the depctl command line interface.



(1) deploy request
(2) loadbalance request to pod and authenticate
(3) fetch resource definitions from version control system
(4) validate availability of container image(s)
(5) modify resource definitions (image version and environment)
(6) apply resource definitions to kubernetes API
(7) eventually report errors

Figure 5.1: Detailed Deployer Architecture and Steps.

Depctl command line interface wraps http calls and assists developers with automatic completion of the repository to deploy and completion of the version to deploy. With depctl developers can easily skip steps of the whole NPRT flow. For instance they can skip the tests, which would run on the continuous integration system. This makes development more fluently and developer friendly.

In the following we will go through the interface of deployer. Deployer provides different endpoints: ls, show, tags, deploy, canary and version.

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 shows 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.

You need to call *deploy* and *canary* on a specific repository and you need to provide either a version reference, a tag or both. The request updates a service, so this is why we are using a http put call. An example curl request would be

The *version* endpoint simply returns the deployer version.

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=025838f \
   https://auth_token:secret@deployer.company.com/gapfish/deploy

depctl deploy
```

5.3 Logic and Flow of a Deploy

5.3.1 Authorization

Next we want to go through the steps involved in the deployment process. Initially a client sends a http request to deployer (1). The service proxies the request to deployer (2). After that deployer needs to authenticate the client via http basic authentication. In other words, the client authenticates itself 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. The continuous integration system cannot have full production cluster access. In this case the token authorizes to only deploy a specific version from the repository. For us that means, that the client is not able to do everything to the kubernetes api consequently not everything to our whole production cluster. It is for instance not able to deploy other code than ours or read credentials from the production cluster.

5.3.2 Fetching the Code and Caching It

After deployer authenticated a valid user, it then fetches the code from the version control system (3). When there is git as the version control system, deployer uses commit hashes and branches as version reference and to determine the code version. When there is subversion, deployer uses revision numbers instead. We implemented subversion, which is more of an ancient technology, to be able to do the evaluation with the DIN system.

As we know, the process of fetching a repository includes persistence and disk interaction. That is true. Deployer uses its volume just as a cache, though. The reason why there is a cache at all is of course performance. One of the bottlenecks of a 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. If the download rate is additionally very low the duration is even longer. This leads to long running deployments and a bad development flow experience. Therefore deployer keeps the already downloaded repositories on its volume as a cache. The next time it deploys the same repository in a different version deployer only fetches the changes.

As a simplification, every pod has its own cache and lives as long as the pod. So every pod utilizes its own volume as a cache and there is no need for communication to an extra caching service. Since docker containers are immutable, every time kubernetes recreates a pod, kubernetes destroys the volume. Thus the cached 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 for instance 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 as the location. If an application is not able to locate it in that directory 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 the kubernetes resource path enables two different methodologies in the microservice approach. The first one is the multiple repository methodology. In that methodology for every single service or microservice we would have a dedicated repository. The second methodology is the single monolithic repository, which contains multiple microservices.

5.3.3 Validations

After having the specific version reference of the repository fetched and checked out, deployer starts the deploy procedure. At first deployer validates the arguments given 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 makes the reference more readable.

Deployer initially validates the arguments of 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 if a the tag, which references the version, 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 (4). The client does in contrast not have any validations[1]. The reason for that is that we want to have a central definition for the validations.

For simplicity and usability the client has the options to either

- exclusively give a version reference
- or exclusively give a tag

because it makes things a lot simpler when a developer needs to deploy manually. This makes the development flow more efficient. The version reference is totally sufficient to determine the version, so the clients usually provides the version reference only.

5.3.4 Modifications of Resource Definitions

After deployer validated the deploy request, deployer takes the previously checked out resources and modifies them in a next step (5). Especially deployments and statefulsets are relevant to those modifications. In contrast to those resources, deployer does not modify all other resources, such as services.

Deployer applies two main modifications: the image version and the environment. The latter is especially important for the metrics collection. Deployer modifies the environment, so that the stable version and the canary version know about themselves as such. The running service can later read

[1] We can use any http client such as simple curl or the more comfortable depctl.

the version information from environment variables. The pods receive an according label, so that other applications, which may consume the kubernetes api, can also distuinguish between the stable and the canary version.

The other important modification is the image version. The image tag specifies the version of the docker image. Deployer modifies the resource only when the image tag is unspecified. In case the deployment already specifies a tag, such as 'debian:wheezy', deployer does not change that tag. The reason is there may be containers, which the continuous integration system does not build and tag with the specific version reference. Commonly another party maintains these docker images. One example is an additional statsd container running as a sidecar in the application pod.

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 reference the client provided.

The next step is to communicate the modifications to the kubernetes api (6). Deployer sends the modified resources to the kubernetes api. The kubernetes master manages the rollout of the changes. It swaps out one pod by another and replaces the old version with the new version. The procedure is called rolling update.

5.3.5 Errorhandling

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 different clients. One is, a continuous integration system requests the deploy. In that case our error report system[2] come to play. Deployer raises an error and the error is sends it to the error report system (7), which collects all the errors of any system. The error report system informs the developers via sending notifications to a specified channel. We have for example a slack chat channel, on which the person on call subscribes and receives notifications from.

The other client is a developer sending a deploy request manually. In that case the developer typically uses depctl. Deployer discriminates usual curl requests from requests with depctl. Instead of utilizing the error report system, deployer answers the http request with an errorcode and a message in the http body. As mentioned earlier the communication with deployer is synchronous.

[2] In technologies section, we mention Bugsnag. This is the error report system we use.

5.4 Testing Architecture

In this section we look at the testing architecture inside the production cluster. A few canary pods in the new version run next to the stable version. These pods are able to receive and respond to production traffic.

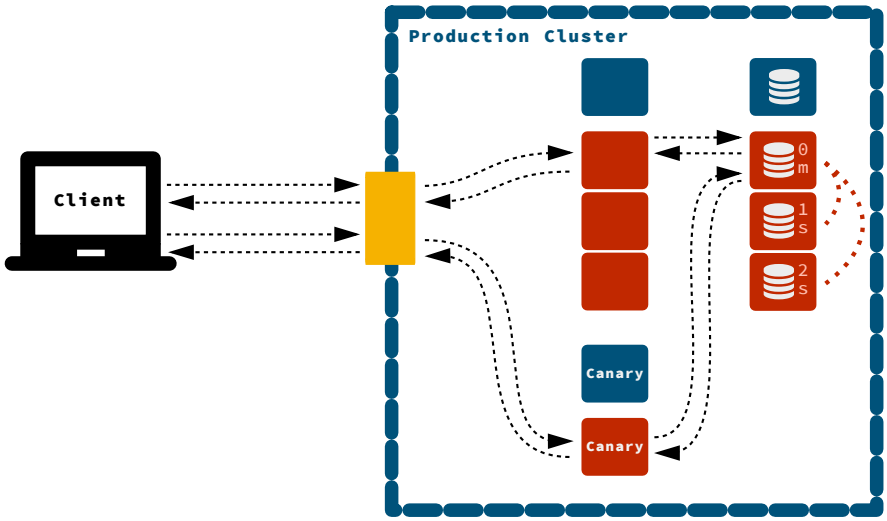


Figure 5.2: Testing Flow Implementation.

To deploy a 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 receives the canary request, it proceeds almost identical to the deploy request. The steps are verification, fetching the version, resource modification and application to the kubernetes api. The difference to the deploy lies in the modification step. Deployer creates 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 treads 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 the original deployment created. And it selects the pods, which the canary deployment created. Hence the service selects its pods on

account of the label and the original and the canary deployment share some of their labels. An example would be a shared label like 'deploy=webserver' and tow different labels like '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.

5.5 Metrics

5.5.1 Metrics Collection

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. Metrics collected from hosts and metrics from applications.

Firstly the group of metrics, which are picked up from 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. The agent picks up the information frequently and then sends it to our monitoring system.

Secondly the group of 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 you get the instrumentation without much effort. The instrumented application sends the data to the statsd server after 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 as either stable or canary.

5.5.2 Metrics Comparison

Another part of NPRT is the comparison within the monitoring system.

The monitoring system consists basically out of a timeseriesdatabase, a graphing user interface and an alarm system. The timeseriesdatabase persists the metrics. And the user can define graphs from those metrics, which the user interface displays. Moreover you can define rules in the alarm sys-

tem, which monitor the metrics in the timeseriesdatabase. In case the metrics violate any rule, the monitoring system sends a notification.

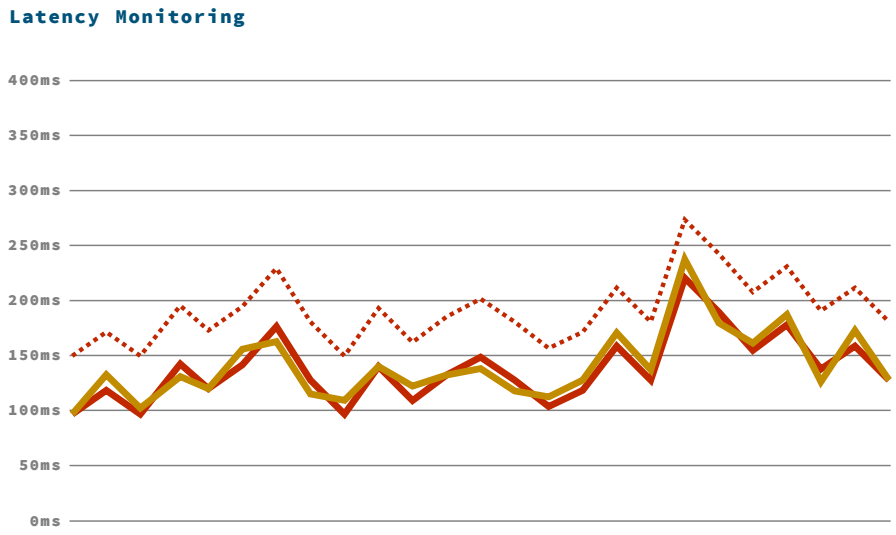


Figure 5.3: Comparing the stable version with the canary version.

Let's have a closer look on those rules. The figure shows the example of a rule on the latency. The red continuous line is the latency of the stable version. The yellow continuous line is the latency of the canary version. The graph compares them side by side. Then we have a threshold, which is the red dotted line. The threshold depends on the latency of the stable version and is in our figure always 50ms higher. In case the latency of the canary goes above the threshold, the graph would violate the rule and trigger an alarm. The alarm would trigger a webhook, which would roll back the canary instance.

Next we have a look at the metrics, we want to monitor:

Latency of successful requests. This is a performance metric. And the monitoring rule should identify serious regressions. To achieve that, we use median or other percentiles so that outliers do not trigger a rollback, but only serious regressions do.

Utilization refers to the metrics CPU and Memory consumption. We can have the same aggregation window and aggregation method like we have it with Latency.

Errors We simply count errors. Best is an application, where no errors occur. Then we can trigger a rollback of the canary instance in case it raises an error. Nevertheless most applications are not free of errors. On account to this, we rollback in case the canary raises a new error, which the application never has raised.

Throughput is not a metric we define a rule on. The version of the canary does not have any influence on throughput. The production environment determines throughput. Throughput is relevant for the parameters of the other metrics, though.

5.5.3 Metric Parameters for Latency and Utilization Metrics

The ratio of stable and canary instances as well as the throughput affect the size of the *aggregation window*. What would we do in case we had a very low throughput of only 1 request per minute and a ratio of 2 stable instances and 1 canary instance? If we had chosen a small window of 1 minute, we would have had either 1 request on the stable or 1 request on the canary instance. That would not be a good comparison. Instead we choose a rather big window of 120 minutes. Then we would have 80 requests on the stable instance and 40 on the canary instance. This is a definitely better comparison. Consequently the lower the throughput and the lower the stable canary ratio is, the bigger needs the window to be.

The distribution of different throughputs and different latencies over all the routes of the application affects which *percentile* to use. I make an example: If we had one single route which has a very high throughput and a very low latency compared to all other routes, then we would not choose the median. Instead we would choose the 95th percentile to well represent the routes with a high latency.

The range of the different throughputs and the different latencies also affects the *threshold*. If we had a bigger range of throughputs and latencies, we would need to choose a higher threshold. However we can affect the volatility of the metrics by choosing a bigger aggregation window. This allows as to lower the throughput.

We see that the higher the throughput is and the distribution is more evenly, the better will the regression testing work.

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.

. . .

6 EVALUATION

This chapter evaluates the NPRT approach in two different aspects, experiments and a lessons learned section. Firstly the experiments aim to present an evaluation the approach's performance in the context of a whole delivery pipeline and how a sample metric comparison performs. The second part is a qualitative discussion of the lessons learned of an application use case, which was used in production.

6.1 Experiments

I had the opportunity to do experiments with two different production applications. The first one is an apache webserver, which is the middleware in front of a tomcat application server. This application serves DIN's the website[1]. In the case of DIN, I setup a version control system, a kubernetes cluster, with deployer, instrumentation and monitoring setup. Software engineers working for DIN helped to setup a container image registry. I also migrated the apache webserver to be able to run on kubernetes. The kubernetes resources for the apache webserver consisted of a service and a deployment with two pods.

The second application is a ruby on rails application, which employees of GapFish utilize as backoffice tools for several panel platforms[2]. As an employee of GapFish I previously migrated the ruby on rails application to kubernetes and utilized codeship as continuous integration, docker hub as container image registry and integrated datadog as monitoring system. The application consists of a service and nine deployments. Eight of the deployments are different background workers each with one pod. And one deployment is for the ruby on rails application server with two pods.

The experiments for the DIN apache application were conducted on virtual machines rented from Atos. The experiments for the GapFish application were conducted on virtual machines hosted at Google. You can read more about the specific details and the data used for the plots in the appendix.

Each of the experiments has three consecutive executions with three different canary versions in the deploy and rollback experiments. The measure-

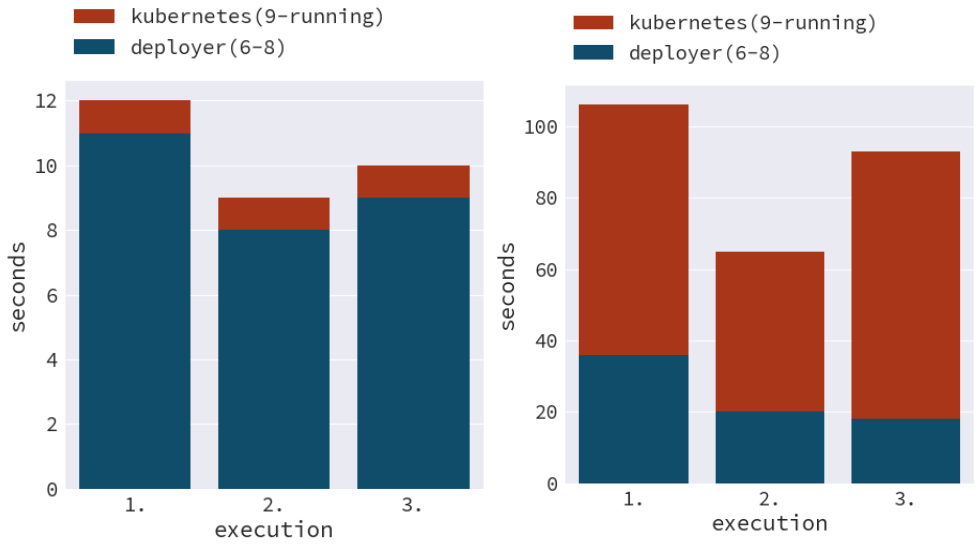
[1] `https://www.beuth.de`

[2] p.e. `https://www.entscheiderclub.de` and `https://www.spiegel-panel.de`

ments had to be manually collected from all the different logs of the involved systems, that are a local http client, continuous integration system, deployer and kubernetes.

6.1.1 Deploy

The first experiment examines the performance of a deploy of a canaries. In account to that an http client triggered a deploy as described in previous chapters. Figures 6.1a and 6.1b show the duration between the deploy request (6) and the running canary. The blue part of the bar indicates the time of the deployer pulling the infrastructure code from the version control system, check of the version reference and image availability, modification of the resource definitions and application to the kuberentes cluster (6-8). The red part is the duration of kubernetes starting the canary (9-running).



(a) DIN Apache Middleware

(b) GapFish Rails Application Server

Figure 6.1: Deploy

Figure 6.1a shows that DIN's apache middleware takes between 9 and 12 seconds with a median of 10 seconds. Deployer took in the third execution 9 seconds and the start of the canary was in every execution at 1 second.

Figure 6.1b shows that GapFish's rails application server takes between 65 and 106 seconds with a median of 93 seconds. The time deployer took was 18 seconds in median.

In both experiments the second and third execution are faster than the first one. That is because deployer caches the code from the version control system and does only need to download the difference in the consecutive executions. The impact of the cache on DIN's apache webserver was not that big, because

the repository size is comparably small. Whereas the cache could save almost half of the time at the GapFish rails application.

The start of DIN's apache pods is with 1 second quite fast. This demonstrates the quick scheduling of kubernetes and quick startup of docker containers compared to virtual machines. The start of GapFish's application is between 45 and 75 seconds with median of 70 seconds.

Accordingly DIN's apache application deploy significantly faster. So deployer is able to handle multiple deployments as in the case of GapFish. The multiple requests to the kubernetes api slightly reduce the deploy duration though. An improvement would be a batched request. Deployer's cache works well for big repository sizes.

It is important to pay attention with slow application startup durations, when looking for fast deployments. In the case of GapFish's rails applicatoin server, this makes the biggest impact.

The variation of the second and third executions come from the nature of the cloud environment and internet services. In particular the requests to the image container registry differ in the duration.

6.1.2 Deploy in Pipeline

This section shows a deploy in the full context of the CD pipeline. The steps start with the commit and push of a new version in the version control system, the build of the docker image(s), saving the recently built images in the image registry, time spend by deployer and the remaining startup of the canary pods. The testsuite runtime is not included in this diagram, so that focus lies on the systems this thesis integrated.

At GapFish I could access all the involved system. This was not the case at DIN. That is why the experiment was conducted for the GapFish rails application only.

The total time of the CD pipeline was between 172 and 414 seconds with a median of 192 seconds. Again the caching saves time in the second and the third execution. This time the continuous integration system and the image registry partly cache the layers of the images. They only have to build and send the differing parts of the container image.

As a result we see the overall pipeline duration for a typical canary change at round about 3 minutes, including a build and deployment. The makes a continuous deployment of new canaries absolutely feasible.

6.1.3 Rollback

In this section we take a the look at the rollback performance of deployer. The blue part of the bar again represents the time deployer is involved, that is from the rollback request including steps (7) and (8) until the kubernetes api

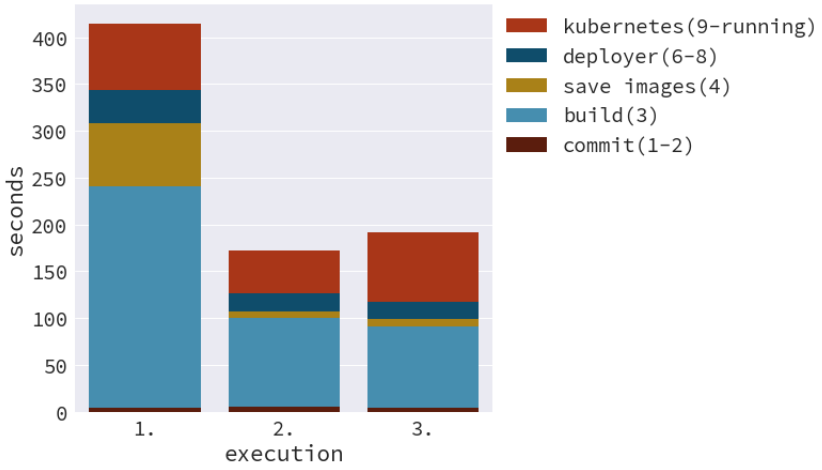


Figure 6.2: Whole Pipeline

responded. The red bar represent the remaining time the pods take to shut down.

Figure 6.3a shows the rollback duration of a DIN apache canary is between 4 and 5 seconds. The duration send in deployer is less than in the deploy case. That is mostly because it the delete request to the kubernetes api is faster.

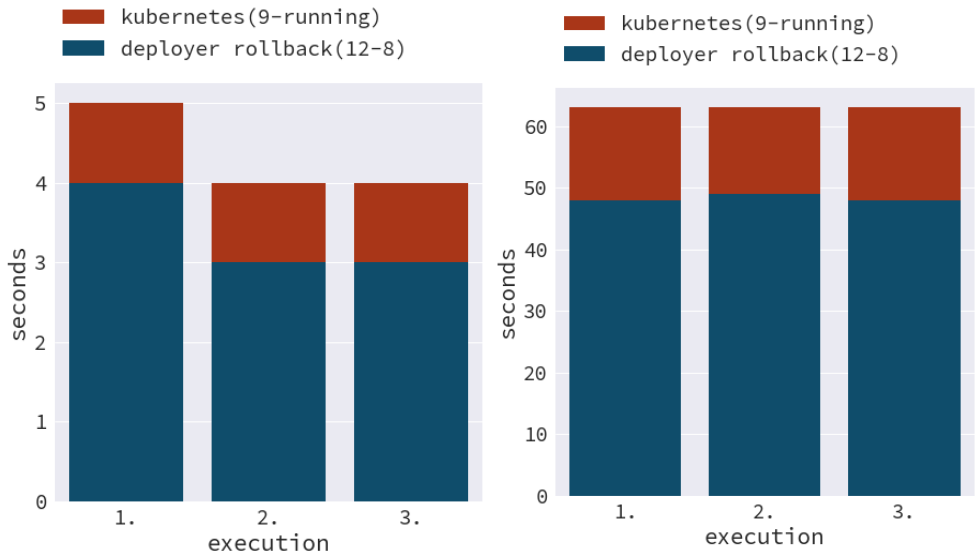
In the case of GapFish's rails application, we have longer rollback times. All three executions of the rollback were 63 seconds. This is due to slow shutdowns of the worker pods. The kubernetes partly waits for the shutdown process of the pods. That is why deployer spends most of the 48-49 seconds waiting for kubernetes api responses.

Also for the rollback case, deployer performs better with DIN's apache system, because of less deployment resources and the fast apache application. Nevertheless the deploys as well as the rollbacks involve zero downtime. So it's acceptable to have a bit longer durations.

6.1.4 Metric Comparison

This last experiment aims at a first evaluation of the feasibility and a first assessment of a metric comparison and the time of a broken canary being in production. DIN provided one hour of production traffic of their website, which a goreplay process recorded on production and replayed with the test setup.

Due to the fact the test setup database state differs to the production database, the throughput of the requests with http 200 responses was at the low rate of 25 requests per minute. Requests of in production logged in users mostly had http 404 as responses. Those successful requests basically consisted of the publicly visible pages of the website.



(a) DIN

(b) GapFish

Figure 6.3: Rollback

At first I deployed a canary version with no changes compared to the stable version. The two version side by side calibrated the metric comparison. The first half of the production traffic determined the size of the aggregation window as well as the threshold of the comparison.

After the calibration was done, the actual experiment starts. The experiment measures the time between a defect canary being started and the canary being rolled back. Each of the three executions had its own unique part of the second half hour production traffic.

```
ratio =
  apache_GET_200_count.rollup(sum, 30) /
  (2 * apache_canary_GET_200.rollup(sum, 30))
threshold = 3
abs(1 - ratio) > 3
```

The test compares throughput of successful responses. The canary responses are weighted by the factor 2, because there was one canary pod and two stable pods. An aggregation window of 30 seconds and a threshold 3, which means the canary throughput had to differ 3 times as much as the one of the canary.

Figure 6.4 shows that in each of the three executions the test has successfully detected the broken canary. The duration of the defect canary in production was between 129 and 400 seconds with a median of 185 seconds.

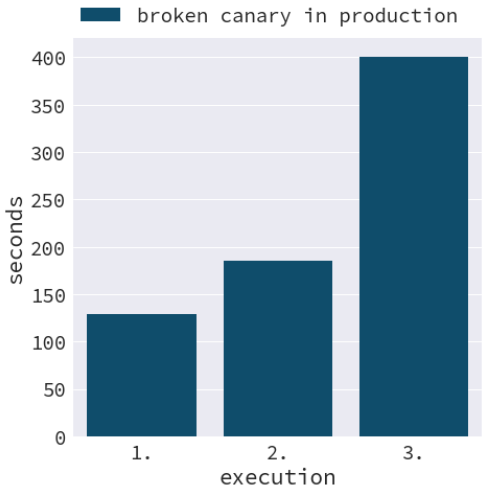


Figure 6.4: Metric Comparison

The experiment demonstrates that the automation of a regression test in production is feasible and completely automatable.

6.2 Lessons learned

The GapFish ruby on rails application of the previous experiments as well as two other GapFish ruby services utilize the NPRT approach in production. This section presents qualitative feedback, which comes mainly from use of NPRT in production.

On the one hand nonfunctional regression testing was used to test maintenance of own code, such as refactorings and performance improvements. On the other hand NPRT was also used to maintain dependencies. The previously developed GemUpdater regularly and automatically opened pull requests with dependency updates. The canary was tested using error metrics, which I explain later on in more detail.

6.2.1 Quick Debuggable Deploys

Deploys of a round about 1.5 minutes conform the need of developers of sufficiently quick deploys. This is good compared to oftentimes long running test suites as it is the case with GapFish's rails application, which has a current median duration of 12.5 minutes. The time of a deploy is compared to that much faster and engineers do not get distracted and when they deploy manually.

One benefit is the ability to track the version reference from commit to production the test in production throughout the whole pipeline. This makes it easy to debug versions, which run or used to be running in production. It

is also possible to track the code change of stable and a canary version which used to be running side by side.

It has been a good decision to version the kubernetes resources, which are basically infrastructure definitions as well. This made it possible to track also track the infrastructure changes.

Due to the nature of containerized infrastructure and its immutability, rollbacks to specific versions are simple. The rolling update mechanism provides deploys and rollbacks without downtime. This fact lead to a certain confidence of the engineering team to deploy and rollback and the amount of deploys increased.

6.2.2 Engineers know the Product

The experiments already showed that a simple side by side comparison of the throughput can have good results for a test metric with production traffic. You could argue that production metrics can be hard to evaluate and in a test environment it is rather easy to achieve good results. The problem of overfitting in statistics is well known.

When choosing the wrong metrics for the production regression tests of the ruby applications of GapFish, it would not have been feasible to deliver as good results as the first test showed. The problem lies in the granularity of the metrics. For example a total throughput or total error rate of a comprehensive application is misleading. GapFish's rails application serves not only humans, but provides also interfaces for other services. This leads on the one hand sparse usage[3] on routes for users and on the other hand machines machines utilized some routes extensively. The overall throughput was a poor fit.

The much better choice for a metric was using the error aggregation of bugsnag. Bugsnag aggregates errors, that it shows specifically newly introduced errors. This has been the best metric for the regression tests.

In conclusion as a developer and engineer of the already have a good intuition on what is working and what not. This is another evidence for the benefits of the same team building and running a software product.

6.2.3 Continuous and Fully Automatable

Deployer connects the continuous integration system, kubernetes and the monitoring system. These systems create the ability to create a fully automatable delivery pipeline and a system for fully automated rollbacks. Those are the step stones for more possibilities.

NPRT reduced the efforts of maintenance work on own code. The Engineers were able to test changes in production quickly and have a small feedback loop.

[3]The users of the rails application are only internal employees.

The engineers working for DIN are still sceptical to have unsupervised deploys. But in the example of GapFish, the engineers became confident about the automation of the deployment and regression test in production and start to use unsupervised deploys and tests more often.

The possibility of dependency update automation with GemUpdater turned out to save effort at low risk as well. The combination of GemUpdater and NPRT lead to a continuous fully automated approach for dependency updates, which used to be an important, but tedious maintenance work.

6.3 Related Work

There are multiple publications, which use methods to test software production in production. Tang et. al. demonstrates how Facebook does gradual rollouts with gatekeeper [31] and tests with high amounts of traffic. But not only canary or A/B tests are done in production. For example Falé proposes synthetic monitoring [15], which aims at functional tests being made in production. This enables testing microservices in complex environments. Also Tseitlin describes how Netflix [32] tests their systems with the simian army, which produce random defects in a controlled production environment. This practice lays open unknown issues, which can be fixed in order to gain better reliability.

There are also countless open source tools, which help to implement CD similar. Vamp[4] has the ability to deploy applications to kubernetes automatically. Also spinnaker[5] can deploy applications to cloud environments, with kuberentes being one of them. Spinnaker very recently released a feature to analyse canaries, just like for NPRT needed. The difference to deployer is that vamp and spinnaker are very complex and exhaustive. They also require other databases. Whereas deployer is a very simple and lightweight tool with focus on a single purpose and consists of only one docker image. Nevertheless deployer can be integrated to several monitoring solutions by the simple http interface.

Schermann et. al. developed bifrost [29], an open source tool similar to the previously mentioned gatekeeper at Facebook. It implements canary testing and other testing strategies as well. The approach is a complete different, though, to deployer. Bifrost is technically an automatically configurable middleware or loadbalancer. It does not provide any automation for the deploy itself. Deployer implements the deploy automation. It could be interesting to combine the two tools to have more testing strategies available.

Lastly I am going to mention publications, which deal with metric analysis. Zimmerman motivates the need for data scientists software teams [35] and identifies the multiple roles [23]. Farshchi et al. evaluates a regression based

[4] `https://vamp.io`

[5] `https://www.spinnaker.io/`

approach in order to detect errors during rolling upgrades [12]. They demonstrate how to identify the set of metrics, which have the highest chance to detect the errors. Their approach has a high precision and recall. In comparison to that Bakshy and Frachtenberg developed a statistical model to analyse errors of a distributed service in an A/B experiment [1]. They provide guidelines for the design of error analysis models.

. . .

7 CONCLUSION

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 it 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 [21] 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 continous 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 of 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.

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 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.

7.1 Summary

We illustrated that we are able to automate the whole testing procedure, which is the advantage of NPRT. We can extend the CD pipeline in a natural way and support developers not only until the deploy, furthermore we automated a part of the developer's job during run time. The pipeline is now advanced in a way, that we can change application code and test it without the supervision of the developers at a very low risk. The change goes through the whole

pipeline including tests in a testing environment, an automated deploy and tests in the production environment. If the NPRT and automated deploy is successful, developers will be completely free of work. Potential changes of bots would be robust enough to be able to act fully automated and unsupervised. However we still test the changes they make and can be sure to not have a regression in our production system.

Finally we want to summarize, what we examined so far: The whole process of NPRT is fully automatable, every step is determined and traceable throughout and even reproducible until the deploy on the basis of the version reference. We 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.

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 modificactions 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 NPRT to deployer.

7.2 Outlook and Future Work

NPRT is good approach to continuously automate the maintenance of a software product. Nevertheless there are possible improvements. A automatic approach such as GemUpdater works well. More automatic service of such kind are needed. One obvious use case is the automation of updating docker base images. A docker base image updater would help to maintain continuously update the operating systems and software, on which the applications docker images are based. Furthermore there is a similar need for update mechanism for other languages such as java or python, which come with their own dependency managers.

I image a completely unsupervised update mechanism for updates. An engineer would resultantly only have to do manual work in case of a regression. Engineers would need to have more confidence in unsupervised deploys, though.

Another aspect is the lack of good open source instrumentation. On the one hand good instrumentation exists. On the other hand the instrumentation is proprietary and it cannot be utilized outside of the proprietary ecosystem. We should invest more in more fine grained open source instrumentation for popular application frameworks and webservers.

In order to be able to better maintain the metric comparisons, these would need to go into version control as well. With datadog it was not possible to version those tests, though. With other tools for instance prometheus this may be possible.

. . .

APPENDIX

Experiment Servers

This section describes the servers, on which I cunducted the experiments.

DIN The kubernetes cluster at DIN was in version 1.7.1 and was installed on 4 VM instances provided by Atos. Each instance had 1 vCPU and 4GB memory. The installed operating system on each VM was Red Hat Enterprise Linux Server release 7.3.

GapFish The staging kubernetes cluster at GapFish was in version 1.8.4 and had 8 worker instances. The instances were VMs hosted at Google. The machine types were n1-standard-2 with 2 vCPUs and 7.5 GB memory. The operating system was Google's Container OS in version 1.8.1-gke.1.

Experiment

This section provides the raw data, which was collected in the experiments and used for the plots in the evaluation chapter.

plot	exec	steps	timestamp	secs
deploy	1.	6-8	2017-12-29 10:45:47	11
din		9	2017-12-29 10:45:58	1
		running	2017-12-29 10:45:59	
	2.	6-8	2017-12-29 10:46:30	8
		9	2017-12-29 10:46:38	1
		running	2017-12-29 10:46:39	
	3.	6-8	2017-12-29 10:47:02	9
		9	2017-12-29 10:47:11	1
		running	2017-12-29 10:47:12	
deploy	1.	6-8	2018-01-06 15:11:02	36
gapfish		9	2018-01-06 15:11:38	70

		running	2018-01-06 15:12:48	
	2.	6-8	2018-01-06 15:14:50	20
		9	2018-01-06 15:15:10	45
		running	2018-01-06 15:15:55	
	3.	6-8	2018-01-06 15:18:29	18
		9	2018-01-06 15:18:47	75
		running	2018-01-06 15:20:02	
rollback din	1.	12,7-8	2018-01-04 15:44:29	4
		9	2018-01-04 15:44:33	1
		running	2018-01-04 15:44:34	
	2.	12,7-8	2018-01-04 16:10:24	3
		9	2018-01-04 16:10:27	1
		running	2018-01-04 16:10:28	
	3.	12,7-8	2018-01-04 16:19:39	3
		9	2018-01-04 16:19:42	1
		running	2018-01-04 16:19:43	
rollback gapfish	1.	12,7-8	2018-01-06 15:24:19	48
		9	2018-01-06 15:25:07	15
		running	2018-01-06 15:25:22	
	2.	12,7-8	2018-01-06 15:27:46	49
		9	2018-01-06 15:28:35	14
		running	2018-01-06 15:28:49	
	3.	12,7-8	2018-01-06 15:30:52	48
		9	2018-01-06 15:31:40	15
		running	2018-01-06 15:31:55	
metric compar- ison din	1.	11	2018-01-04 15:52:20	129
		12	2018-01-04 15:54:29	
	2.	11	2018-01-04 16:07:19	185
		12	2018-01-04 16:10:24	
	3.	11	2018-01-04 16:12:59	400
		12	2018-01-04 16:19:39	
whole pipeline gapfish (build step (5) skipped)	1.	1-2	2018-01-06 15:05:54	4
		3	2018-01-06 15:05:58	237
		4	2018-01-06 15:09:55	67
		6-8	2018-01-06 15:11:02	36
		9	2018-01-06 15:11:38	70
		running	2018-01-06 15:12:48	
	2.	1-2	2018-01-06 15:13:03	6
		3	2018-01-06 15:13:09	94
		4	2018-01-06 15:14:43	7

```
|           |      |     6-8 | 2018-01-06 15:14:50 |   20 |
|           |      |       9 | 2018-01-06 15:15:10 |   45 |
|           |      | running | 2018-01-06 15:15:55 |      |
|           |   3. |     1-2 | 2018-01-06 15:16:50 |    4 |
|           |      |       3 | 2018-01-06 15:16:54 |   87 |
|           |      |       4 | 2018-01-06 15:18:21 |    8 |
|           |      |     6-8 | 2018-01-06 15:18:29 |   18 |
|           |      |       9 | 2018-01-06 15:18:47 |   75 |
|           |      | running | 2018-01-06 15:20:02 |      |
```

```
| nr | step                         |
|----+------------------------------|
|  1 | push new version             |
|  2 | ci pulls code                |
|  3 | build                        |
|  4 | save container image(s)      |
|  5 | run tests                    |
|  6 | deploy                       |
|  7 | pull infrastructure code     |
|  8 | modify resource definitions  |
|  9 | apply to production cluster  |
| 10 | pull container image(s)      |
| 11 | send metrics                 |
| 12 | trigger rollback             |
```

. . .

Glossary

Canary Releasing is a releasing practice to reduce risk. A canary serves as test instance for a change in production [21].

Container is a running instance of a container image [9, 10].

Container Image is an immutable snapshot of a container state [9, 10].

Continuous Delivery is a practice to continuously deliver changes to production. Continuous delivery is related to DevOps [34].

Continuous Deployment is an extension to continuous delivery. An automated deployment process delivers most changes directly to production [21].

Continuous Integration is a practice of continuously merging code to a repository and automatically test it [16].

Deploy I use the term deploy in the thesis in order to differentiate the procedure from deployment, the kubernetes resource..

Deployment is a kubernetes resource, which ensures the existence of replicated pods. Deployments are an evolution to replicationcontrollers, which have similar features [7].

DevOps is a made up of developers and operations. DevOps practices aim for a short time between commit and deploy, still ensuring high quality [2].

Dynamic Infrastructure is software defined infrastructure [25, 27].

Immutable Infrastructure In such an infrastructure nodes are destroyed and created in order to change them.

Kubernetes is a cluster management system, which Google's borg heavily influenced [33, 5]. Beda gives a good overview of the architecture [4].

Monitoring System collects information about a software product and monitors it for defects. I refer in the thesis to modern monitoring system design as Ewaschuk and Bass et. al describe [11, 3].

Pod is a kubernetes resource. It is a unit of multiple containers, which are located on the same host to be able to share resources [6].

Service is a kubernetes resource, which basically acts as a loadbalancer in front of pods [8].

Version Control System versions code repositories. Examples are subversion or git. Humble explains what is important in version control [22].

Zero Downtime Deploys is a way of deploying changes without downtime [21]. Rolling updates is an implementation.

. . .

Acronyms

CD Continuous Delivery. 1, 6, 9, 27, 32, 36, 39

NPRT NONFUNCTIONAL PRODUCTION REGRESSION TESTING. 1, 2, 6, 9, 12, 15, 17, 22, 25, 30–32, 36, 37, 39

. . .

Bibliography

[1] Eytan Bakshy and Eitan Frachtenberg. "Design and analysis of benchmarking experiments for distributed internet services". In: *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2015, pp. 108–118.

[2] Len Bass, Ingo Weber, and Liming Zhu. "DevOps: a software architect's perspective". In: Addison-Wesley, 2015. Chap. What Is DevOps?, pp. 3–26.

[3] Len Bass, Ingo Weber, and Liming Zhu. "DevOps: a software architect's perspective". In: Addison-Wesley, 2015. Chap. Monitoring, pp. 127–154.

[4] Joe Beda. *Core Kubernetes: Jazz Improv over Orchestration*. `https://blog.heptio.com/core-kubernetes-jazz-improv-over-orchestration-a7903ea92ca`. last visited 16.11.2017. May 2017.

[5] Brendan Burns et al. "Borg, Omega, and Kubernetes". In: *ACM Queue* 14 (2016), pp. 70–93.

[6] CoreOS. *Overview of a Pod*. `https://coreos.com/kubernetes/docs/latest/pods.html`. last visited 14.11.2017.

[7] CoreOS. *Overview of a Replication Controller*. `https://coreos.com/kubernetes/docs/latest/replication-controller.html`. last visited 14.11.2017.

[8] CoreOS. *Overview of a Service*. `https://coreos.com/kubernetes/docs/latest/services.html`. last visited 14.11.2017.

[9] Docker Docs. *Get Started Part 1: Orientation and setup*. `https://docs.docker.com/get-started/`. version 17.09, last visited 14.11.2017.

[10] Docker Docs. *Get Started, Part 2: Containers*. `https://docs.docker.com/get-started/part2/`. version 17.09, last visited 14.11.2017.

[11] Rob Ewaschuk. "Site Reliability Engineering: how google runs production systems". In: ed. by Betsy Beyer. O'Reilly, 2016. Chap. Monitoring Distributed Systems, pp. 55–66.

[12] Mostafa Farshchi et al. "Metric selection and anomaly detection for cloud operations using log and metric correlation analysis". In: *Journal of Systems and Software* (2017).

[13] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. "Development and deployment at facebook". In: *IEEE Internet Computing* 17.4 (2013), pp. 8–17.

[14] Brian Fitzgerald and Klaas-Jan Stol. "Continuous software engineering: A roadmap and agenda". In: *Journal of Systems and Software* 123 (2017), pp. 176–189.

[15] Serge Gebhardt Flávia Falé. *Synthetic Monitoring*. `https://martinfowler.com/bliki/SyntheticMonitoring.html`. last visited 14.11.2017. Jan. 2017.

[16] Martin Fowler. *Continuous Integration*. `https://martinfowler.com/articles/continuousIntegration.html`. last visited 14.11.2017. May 2006.

[17] Martin Fowler. *Microservices*. `https://martinfowler.com/articles/microservices.html`. last visited 27.12.2017. Mar. 2014.

[18] Jim Gray. "A conversation with Werner Vogels". In: *ACM Queue* 4.4 (2006), pp. 14–22.

[19] Jim Highsmith. *Velocity is Killing Agility!* `http://jimhighsmith.com/velocity-is-killing-agility`. last visited 27.11.2017. Nov. 2011.

[20] Jez Humble and David Farley. "Continuous Delivery: reliable software releases through build, test, and deployment automation". In: Addison-Wesley, 2010. Chap. The Deployment Pipeline, pp. 103–140.

[21] Jez Humble and David Farley. "Continuous Delivery: reliable software releases through build, test, and deployment automation". In: Addison-Wesley, 2010. Chap. Deploying and Releasing Applications, pp. 249–274.

[22] Jez Humble and David Farley. "Continuous Delivery: reliable software releases through build, test, and deployment automation". In: Addison-Wesley, 2010. Chap. Configuration Management, pp. 31–54.

[23] Miryung Kim et al. "The Emerging Role of Data Scientists on Software Development Teams". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 96–107.

[24] Kubernetes. *Kubernetes architecture*. `https://github.com/kubernetes/kubernetes/blob/release-1.5/docs/design/architecture.md`. last visited 14.11.2017. Oct. 2016.

[25] Kief Morris. "Infrastructure as code : managing servers in the cloud". In: O'Reilly, 2016. Chap. Dynamic Infrastructure Platforms, TODO.

[26] Kief Morris. "Infrastructure as code : managing servers in the cloud". In: O'Reilly, 2016. Chap. TODO, TODO.

[27] Kief Morris. "Infrastructure as code : managing servers in the cloud". In: O'Reilly, 2016. Chap. Software Engineering Practices for Infrastructure, pp. 179–194.

[28] Niall Murphy, John Looney, and Michael Kacirek. “Site Reliability Engineering: how google runs production systems”. In: ed. by Betsy Beyer. O’Reilly, 2016. Chap. The Evolution of Automation at Google, pp. 67–84.

[29] Gerald Schermann et al. “Bifrost: Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies”. In: *Proceedings of the 17th International Middleware Conference*. Middleware ’16. Trento, Italy: ACM, Dec. 2016, 12:1–12:14.

[30] Benjamin Treynor Sloss. “Site Reliability Engineering: how google runs production systems”. In: ed. by Betsy Beyer. O’Reilly, 2016. Chap. Introduction, pp. 3–12.

[31] Chunqiang Tang et al. “Holistic configuration management at Facebook”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM. 2015, pp. 328–343.

[32] Ariel Tseitlin. “The antifragile organization”. In: *Communications of the ACM* 56.8 (2013), pp. 40–44.

[33] Abhishek Verma et al. “Large-scale cluster management at Google with Borg”. In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. 2015.

[34] Eberhard Wolff. “Continuous Delivery: der pragmatische Einstieg”. In: 2. dpunkt.verlag, 2016. Chap. Continuous Delivery und DevOps, pp. 235–246.

[35] Thomas Zimmermann. “Software Productivity Decoded: How Data Science Helps to Achieve More (Keynote)”. In: *Proceedings of the 2017 International Conference on Software and System Process*. ICSSP 2017. Paris, France: ACM, 2017, pp. 1–2. ISBN: 978-1-4503-5270-3.