# NONFUNCTIONAL PRODUCTION REGRESSION TESTING

---

**implemented with Kubernetes**

. . .

# Contents

. . .

# 1 INTRODUCTION

## 1.1 Automation to Support Maintenance Work

DevOps [1] and microservices [2] 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 [3, 4].

However the responsibility of maintaining 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 non-functional 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 [5] is such a continuous automatable practice. But it covers the process only up to the deploy, whereas maintenance involves more than just the delivery.

Instead of employing more engineers, Google combats workforce problems with automation [6]. They recognized that automation does not only save valuable programmer time, it furthermore adds reliability to the process. Google explains how valuable automation is and successfully illustrates it with their site reliability engineering approach [7].

This thesis proposes with *Nonfunctional Production Regression Testing* (*NPRT*) a practice which addresses 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 non-functional changes in production. I implemented *NPRT* at two companies, Deutsches Institut für Normung (DIN) and GapFish, in order to make an evaluation.

## 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** The glossary explains the practices and techniques required in order to understand the problem in depth. In contrast to this, chapter 3 briefly explains the major Kubernetes resources on the example of a 3 tier web app and names the software dependencies.

**4 NPRT 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 Implementation of Deployer and Metric Comparison** Chapter 5 goes more into detail of the thesis contributions. It demonstrates the design of *Deployer* , the software I wrote in order 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* based on the experiences at DIN and GapFish. 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 Shift

The DevOps[1] approach and the microservices[2] culture create practices to solve the issues of the conflicting tension [7] 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" [3]. This quote 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, because 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 changed this. 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 still have the same functionality as 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[3] in comparison to bare metal infrastructure. And the easier it becomes, the more it enables developers to operate their software product themselves.

[1] see glossary: DevOps
[2] see glossary: microservices
[3] see glossary: dynamic infrastructure

Containerization[4] 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 developers 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 developers do not maintain the software products properly more issues arise, such as bugs or performance problems. On account of that, maintaining a software product comes with responsibility. If developers 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 is killing agility” [8]. Also Fitzgerald observes the issue of a ”misplaced focus on speed rather than continuity” [9]. 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 I 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 those invisible changes are more than 50% of the total changes [4].

[4] see glossary: container

Moreover, software products have a high amount of dependencies. This is very reasonable, because when programming a software product you do not want to do it from scratch. Firstly, the dependencies are different abstraction levels, such as a processor, a programming language and a programming framework. And secondly, the dependencies are modules which encapsulate a certain functionality. Those functionality can be 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.

On the web, there is rarely a software product which does not depend on open source software. 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 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 microservices is the flexibility to update only a small specific part of the whole software product [2]. With a microservice architecture, developers are able to update a single service without having to upgrade the whole product at once. Developers can choose a microservice for the update depending on its importance. Or developers can update microservices one by one without the major problem of upgrading the whole software product at once. Consequently this means developers need to update more often and the amount of work of updates itself is higher.

## 2.4 Disregard of Automation for Maintenance Work

There are several things companies can do to keep up with the amount of maintenance work. Facebook [4] for example hires more developers. This, however, is expensive. A much better solution is automation. Developers should automate some of the 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 [1]. 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 [4]. One example is A/B testing, where you compare two different versions in production. Another canary releasing[5], 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 [10].

Containerization reduces the risk of updating a service. Morris describes phoenix replacement [11] in immutable infrastructure[6]. Kubernetes[7] has the approach of rolling updates [8] which implements the same practices. A containerized ecosystem is typically immutable. Rather than changing a running container directly, the container image repository[9] would receive and store an updated container image including the changes to deploy. Then Kubernetes would create a new container based on that updated image. After Kubernetes started this new container, it can stop the old container. The container image repository still stores every container image version. Those saved images allow to easily rollback to a previous version. Automation enables developers to deploy more frequently which usually leads to deploys with few changes. Few changes per deploy and the ability to rollback makes updates less risky. In result, updates in an automated containerized immutable infrastructure are less risky compared to a manually operated infrastructure.

Murphy et al. illustrate the value of automation [6]. Software reliability engineers automate tasks to firstly save valuable human resources and secondly reduce the risk in those tasks because of human errors. This is a good approach and focuses on the quality of the software product.

Immutability reduces 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[10] 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.

[5] see glossary: canary releasing
[6] see glossary: immutable infrastructure
[7] see glossary: Kubernetes
[8] see glossary: rolling updates
[9] see glossary: container image and container image registry
[10] see glossary: continuous delivery

CD does not focus on the maintenance work and the DevOps approach does not focus on automating the maintenance work as previously examined. I 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 mater's thesis is a practice, which can fully be automated in order to maintain a software product. It extends the CD and continuous deployment pipeline[11] 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 techniques 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 practice itself via experiments as well as in the production environment of an existing software product.

[11] see glossary: continuous deployment

. . .

# 3 BACKGROUND

## 3.1 Typical 3 Tier Web App in Kubernetes

I'll explain the typical implementation of a 3 tier web app in Kubernetes as it helps a lot to understand the next chapters. The application tier is usually implemented with a **Deployment**[1] which monitors the presence of a specified number of stateless **Pods**[2]. A **Service**[3] acts as a load balancer and selects the Pods and forwards the request. The data tier is implemented with a **StatefulSet**[4] 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.

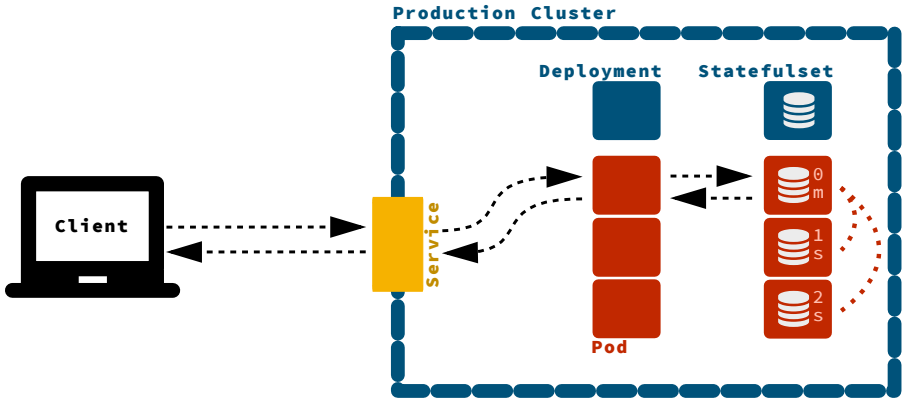



Figure 3.1: Typical 3 Tier Web App in Kubernetes

When 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 Service acts again as a reverse proxy.

[1] see glossary: Deployment and deploy
[2] see glossary: Pod
[3] see glossary: Service
[4] see glossary: StatefulSet

## 3.2 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[5]. it is a web server and Sinatra[6] is its framework. *Deployer* uses Git[7] and Subversion[8] to communicate with either Git or Subversion as the control version system. Moreover, *Deployer* has a plugin which sends messages to Bugsnag[9] and Slack[10] to inform the developer in case it identifies a problem. *Deployer* uses kubectl to interact with Kubernetes[11] master. *Deployer* itself is containerized with Docker[12] and its natural hosting solution would be Kubernetes. We published *Deployer* on Github[13] under GPL-3.0, a free software license.

The previously developed *GemUpdater* [14] creates pull request with dependency updates.

The monitoring system[15], which I used in the evaluation, is Datadog[16], a commercial service. Nevertheless Prometheus[17] is also a suitable open source solution as monitoring system. The Continuous Integration (CI)[18] systems we use for the evaluation are Codeship[19] and Jenkins[20]. As an image repository, we use Docker Hub[21]. Goreplay[22] served to make experiments with production traffic.

[5] `https://www.ruby-lang.org`
[6] `http://sinatrarb.com`
[7] `https://git-scm.com`
[8] `https://subversion.apache.org`
[9] `https://www.bugsnag.com`
[10] `https://slack.com`
[11] `https://kubernetes.io`, see also [12, 13]
[12] `https://www.docker.com`
[13] `https://github.com/gapfish/deployer`
[14] `https://github.com/schasse/gem_updater`
[15] see glossary: monitoring system
[16] `https://www.datadoghq.com`
[17] `https://prometheus.io`
[18] see glossary: continuous integration
[19] `https://codeship.com`
[20] `https://jenkins.io`
[21] `https://hub.docker.com`
[22] `https://github.com/buger/goreplay`

. . .

# 4 NPRT APPROACH

In *NPRT* the pipeline deploys a non-functional change in form of a canary to the production environment. The new version is compared with the stable version side by side in production. In case the new version shows a regression, the pipeline will automatically roll it back. The novelty of this approach is that it creates a feedback loop between development and production, which is fully automatable. The following elaborates on the term in more detail.

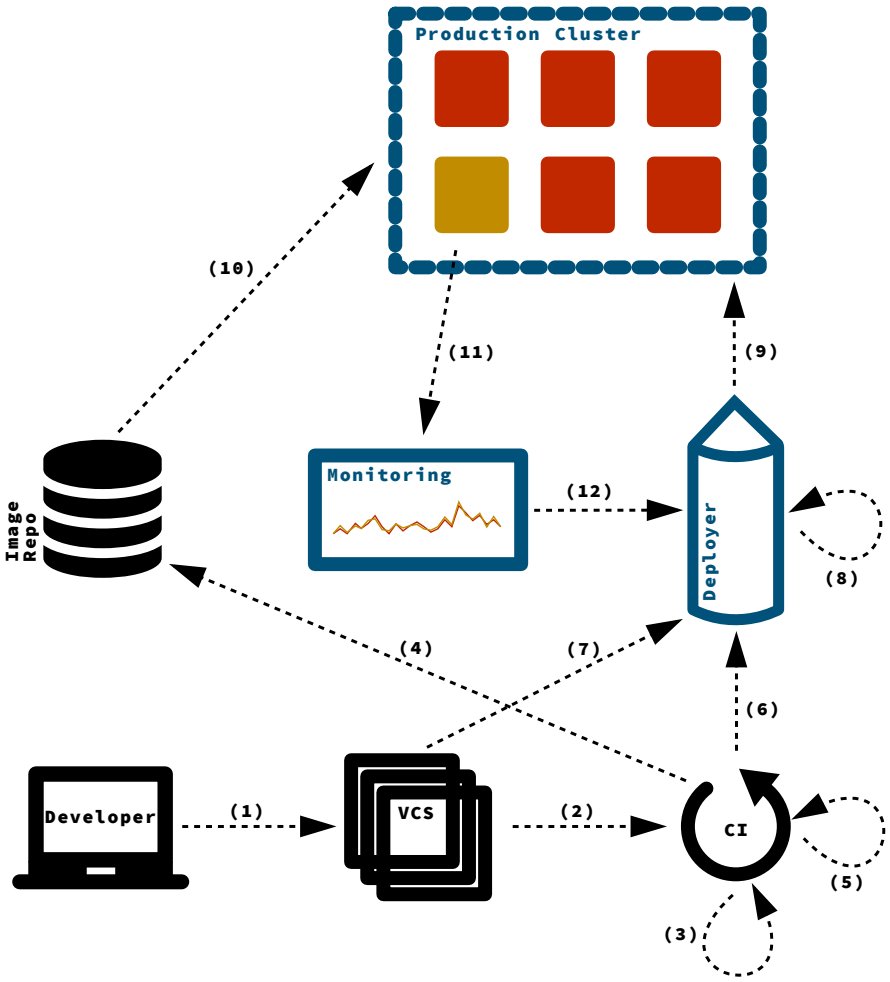
**Non-functional** refers to the change, which the monitoring system monitors. Usually maintenance changes are non-functional changes, for instance performance improvements or security updates. **Production** refers to the environment, in which the versions are monitored. The production traffic reaches the new version as well as the stable version. The term **regression** refers to the testing strategy. The monitoring system tests the metrics of the stable version and the new version for a regression.

*NPRT* provides some further features, which are not included in the term. Indeed the testing approach is completely automatable and developers are able to continuously apply it to new versions. I designed *NPRT* in respect to failing as fast as possible and to inform developers as early as possible. *NPRT* is most useful for maintenance changes. Even automated maintenance changes, for instance dependency updates, are possible.

*NPRT* naturally evolved from common practices such as CI, 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 deploy. In other words it supports developers to run and maintain applications in production, which formerly has been a business of operations teams.

## 4.1 Pipeline Overview

To understand the testing approach as a whole, it is necessary to show a complete overview of the whole pipeline and environment. Figure 4.1 depicts an overview. In the following I 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 resource definitions
(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 NPRT

## 4.2 Continuous Integration as Requirement

The first parts of the pipeline are commonly known and established practices: CI, CD and continuous deployment. It is necessary, though, to touch on them and integrate them in the whole picture. It will illustrate important design decisions for *NPRT* .

First the developer changes the code on his local machine and creates a new version. He then pushes the new version to the Version Control System (VCS) as shown in `(1)`.

After the push of the new version, the second step is a message `(2)` to the CI system. This message holds the reference of the new version and the CI server pulls the new version from the VCS. Now the CI 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 CI 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 CI 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 every built container image relates to a specific version. The version reference[1], which was created by the VCS, serves for this purpose. It is important to be able to trace the version through every step in the pipeline. With this thought in mind, the CI system tags the container image with the version reference and an extra name. Maybe it should be mentioned that the extra name is not absolutely necessary to definitely associate the container image with a version. Experience has shown that a human readable name is very helpful in recognizing a version and to know what the version is about at first sight. The CI 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 CI step is to run the tests. Figure 4.1 shows this in step `(5)`. The tests themselves can be split into multiple stages, such as unit, feature and smoke tests. At this point, though, I will not recall all the details of automated testing at this point.

The last step of the CI 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 CI system does not send a deploy message, the pipeline stops and the CI system may inform the responsible developer. If the tests are successful in all test stages, the CI system will send the deploy signal to *Deployer* .

It is reasonable to deploy only specific versions and not every commit. The practice which is pretty common, is that you develop new features on a

[1] see glossary: version reference

separate branch. For those versions you usually do not send a deploy signal even though the branch's build and tests are successful. Usually, after the tests, a review is done and the developers decide to deploy the changes to production. When the decision is made and one developer merges the change into a specified branch, for instance the master branch, the version will go to production.

Clarifyingly it has to be mentioned that the CI system sends each built image for every single version to the image repository. This is independent of successful tests or the decision to go to production. The reason why I want to have every built image in the repository is to be able to test it in the CI system, locally and maybe on a staging system.

The CI system gives the deploy signal when two requirements are fulfilled: the build and tests are successful and developers decided that the version will go to production.

Until this point, as already mentioned, this is CI practice which is commonly used in software development. I require it for *NPRT* . For *NPRT* it is especially important to associate and trace every step through the whole pipeline. For that purpose I need the CI system to tag, as the previous paragraphs explain, the container images with the exact version reference.

## 4.3 Customizations of Continuous Deployment

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

The next component in the pipeline is *Deployer* . *Deployer* is a service which realizes the customized parts of the continuous deployment practice. In the context of this thesis, I implemented *Deployer* . Chapter 5 describes *Deployer* in detail. 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 deploy process and as a consequence it was necessary to implement *Deployer* .

It would also be possible to implement the logic of the deploy in the CI system. But I had to decide against that, because the deploy needs full access to the production system and the CI system is in our case outsourced to a third party company. I did not want to give other companies full access to the production cluster. However, this meant that I had to implement some steps again, which a CI server already implements. *Deployer* pulls the version from the VCS as well as the CI. The difference to the CI system is that *Deployer* 's interest lies in the resource definitions and not the application code. The resource definitions[2] define the application infrastructure.

[2] see glossary: resource definitions

*Deployer* receives the deploy signal from the CI system(6). The deploy signal again includes the version reference. Now *Deployer* executes three major steps: firstly pull the resource definitions (7), secondly modify the resource definitions (8) and thirdly send the resource definitions to the production cluster (9).

In the first step, *Deployer* pulls the code from the VCS (7). The VCS also holds the resource definitions, which describe the application infrastructure. I wanted to version control[3] the resource definitions in order to be able to relate the version of the infrastructure, the version of the code and the version of the artifacts. In Kubernetes those definitions contain different resources, which chapter 3 explains with an example.

In the second step, *Deployer* modifies those resource definitions (8) in a way that the production system uses the correct container image which relates to the version to deploy. The modifications of *Deployer* also achieve that the running container is aware of its version. The latter is important to later tag the metrics with the version reference.

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). In result a deploy changes two things: application infrastructure 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 Pod instance of two versions in parallel are in the cluster. The production cluster fetches the container images in step (10) from the image repository. The images, which the CI system built in step (3). The production cluster is aware of the specific image identified by the tag.

## 4.4 Metric Collection and Comparison

Figure 4.1 illustrates the two differing versions with two 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 load balancer sends traffic to both versions and both versions respond to clients, on which section 5.4 elaborates.

The production cluster collects the metrics, which are necessary for the regression tests. I adapted those metrics from the four golden metrics of Google's Site Reliability Engineering (SRE) [14]. The metrics are throughput, latency, error rate and utilization.

The interest lies in the metrics of the two specific versions. Consequently the production cluster labels those metrics with the version references. This is important since the metrics of the differing versions will be compared with

[3] see glossary: version control

each other. The production cluster labels and then sends those metrics to the monitoring system (10).

The monitoring system stores all the metrics of the two versions in a time series database. The monitoring system evaluates those metrics by drawing graphs 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. In the case of latency, if the latency of the new version is much higher than the latency of the stable version, the monitoring system detects this as a regression. I explain the specifics of the metric comparison and in the section 5.5.2.

## 4.5 Rollouts and Rollbacks

Two possible scenarios arise. The first one would be that the new version runs in production for a certain amount of time and the monitoring system does not identify any regression. In this case the monitoring system does nothing. A deploy with the reference to the new version triggers the full rollout of the new version. A usual deploy message with the new version reference is sufficient to do so. *Deployer* receives the deploy message, deletes the canary and modifies resource definitions to have the new version as the new stable version and the production cluster proceeds, stops and starts the running instances accordingly.

The second scenario is that the new version turns out to be a regression compared to the stable version. In that case the new version in form of the canary should be rolled back. The monitoring system identifies the regression and sends a message to *Deployer* , as figure 4.1 pictures in step (12). Accordingly the test for regression has failed.

*Deployer* receives the rollback message, pulls the resource definitions again, modifies the resource definitions accordingly and applies the modified resource definitions to the production cluster. To be precise, it is actually a deploy message with the commit hash of the stable version, which the monitoring system is aware of because of the metrics it monitors. *Deployer* modifies the resource definitions similar to step (8). The deploy is idempotent, so that *Deployer* it can repeatedly receive the same deploys. This can happen in the case a developer as well as the monitoring system trigger the rollback. The production system itself takes care of deleting the canary instances in the new version. Since the instances of the stable version are already running in the production cluster, the production cluster does not modify the running stable instances.

. . .

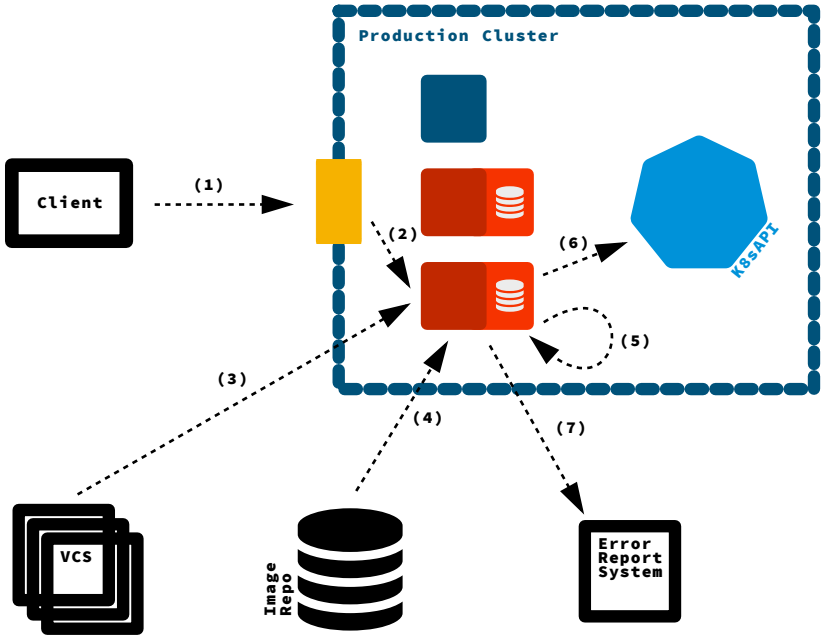
# 5 IMPLEMENTATION OF DEPLOYER AND METRIC COMPARISON

I implemented *Deployer* and setup metrics collection and comparison in order to realize *NPRT* with Kubernetes. *Deployer* receives a deploy message, modifies Kubernetes resource definitions and applies those modified resources to the production cluster. *Deployer* is able to automatically deploy a canary of a new version of an application which runs in parallel to instances of the stable version. The Kubernetes cluster is setup to collect the metrics from the instances of both versions. Datadog, the monitoring system compares the metrics and checks them for regressions.

## 5.1 Deployer Architecture

Similar to the previous chapter, I demonstrate the implementation with the help of figures. The following sections will explain all the steps of figure 5.1.

As mentioned earlier, *Deployer* is a web server which runs in Kubernetes itself and is stateless. The production cluster in figure 5.1 is the same Kubernetes cluster in which the application to test runs in. The yellow box is a Kubernetes Service. The Service is, as described in chapter 3, a load balancer. Clients can reach *Deployer* via an http interface. The figure shows that *Deployer* consists of multiple Pods. These Pods are replicated and are identical in their behaviour. The Pods are basically stateless, yet every Pod has its own caching layer which section 5.3.2 explains in more detail. *Deployer* uses approximately 50mb memory and few CPU, so computing resources are no problem. I designed *Deployer* to have synchronous communication with a client. *Deployer* waits for the deploy process to complete before it responds. This has the advantage of clients receiving feedback about the deploy. Furthermore it enables *Deployer* of being independent of other services such as a queuing or worker system. *Deployer* 's statelessness leads to easy horizontal scaling of *Deployer* . It is possible to add more Pods if needed in order to achieve more concurrent deploys.




(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

## 5.2 Deployer Interface

I designed requests to *Deployer* to be simple and usable. Curl or any other http client serves as a client. As a result, *Deployer* can easily integrate with other systems for instance with a CI system. Another way to interact with *Deployer* is to use the *depctl* command line interface.

The *depctl* command line interface wraps http calls and assists developers with automatic completion of the service 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 CI system. This makes development more fluent and developer friendly.

| 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 | |

Table 5.1: Deployer Interface

Examples:

```
curl --data commit=025838f \
  https://auth_token:secret@deployer.me.com/app/deploy

depctl deploy
```

The following will go through the interface of *Deployer* . *Deployer* provides different endpoints: ls, show, tags, deploy, canary and version. The following descriptions uses the term service in the meaning of a microservice.

The **ls** or index endpoint returns the configured services and returns the endpoints for the service, which the configuration defines.

The **tags** endpoint shows the available tags for that service. *Deployer* queries the Docker image registry for all available tags for all images for a service and returns them to the client.

**Deploy** and **canary** are the endpoints for either a regular or a canary deploy. The requests needs to provide either a version reference, a tag or both. The request updates the service, this is why the endpoint defines a http put.

The **version** endpoint simply returns the *Deployer* version.

## 5.3 Logic and Flow of a Deploy

### 5.3.1 Authorization

Next I want to go through the steps involved in the deploy process. Initially a client sends a http request to *Deployer* (1). The Service resource 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 I mentioned earlier, the CI system is a software as a service solution, and one of the reasons why I implemented *Deployer* is the concern about security. The CI system can not have full production cluster access. In the case of *Deployer* the auth token authorizes to only deploy a specific version from the repository. For us that means, that the CI system is only able to deploy versions from the VCS. The CI system is for instance neither able to deploy other code than ours nor read credentials from the production cluster.

### 5.3.2 Fetching the Code and Caching It

After *Deployer* authenticated a valid client, it then fetches the code from the VCS (3). If Git is the VCS, *Deployer* uses commit hashes and branches as version reference and to determine the code version. If Subversion is the VCS, *Deployer* uses revision numbers instead. *Deployer* implements Subversion, which is an older VCS, to be able to do the evaluation with the DIN system.

It is obvious that the process of fetching a repository includes persistence and disk interaction. *Deployer* uses the volume just as cache, though. The reason for the existence of the cache is performance. One of the bottlenecks of a deploy is downloading the repository. If a repository is large, for instance because of pictures or a long history of commits, it takes quite an amount of time to download it. If the download rate is additionally low the duration is even longer. This leads to long running deploys and a bad development flow experience. Therefore *Deployer* keeps the already downloaded repositories on its volume as a cache. The next time *Deployer* deploys the same repository in a different version and *Deployer* only fetches the changes.

As a simplification, every Pod has its own cache and lives as long as the Pod. Thus every Pod utilizes its own volume as a cache and the need for communication and an extra caching service does not exist. 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 application code as well as the resource definitions. In Kubernetes these resource definitions define the application infrastructure. Resources define for instance Deployments, StatefulSets and Services. These resource definitions should be stored in the '/ku-

bernetes' directory. This is a convention of *Deployer* and it assumes that the '/kubernetes' directory is the standard location. If application is unable to store the resource definitions in the '/kubernetes' directory, the configuration of *Deployer* provides an option to reconfigure the location.

The configuration of the Kubernetes resource directory enables two different methodologies of the microservice approach. The first one is the multiple repository methodology. In this methodology every single service or microservice receives a dedicated repository. The second methodology is the single monolithic repository. Such a repository contains multiple microservices.

### 5.3.3 Validations

After *Deployer* fetched the specific version reference from the repository, *Deployer* starts the deploy procedure. At first *Deployer* validates the arguments, which the client sent. The deploy request requires the service and the version to deploy. The client defines the version by passing either the commit hash, the Docker image tag or both. The tag includes the branch name, which makes the reference more readable.

*Deployer* initially validates the arguments of the request. *Deployer* checks if the service exists in the configuration. *Deployer* checks if the commit hash exists in the repository and *Deployer* checks if the tag exists for all the images, which are necessary in order to deploy the service in the new version. *Deployer* receives the available tags for a Docker images by communicating with the image repository (4).

For simplicity and usability the client sends usually only one argument, either the commit hash or the tag. One argument is completely sufficient to determine the version, so a client can provide either one of them. For instance, a developer has to pick only the commit hash for a manually triggered deploy. This creates a more efficient development flow. Regardless of the whether *Deployer* receives multiple or only one argument, *Deployer* validates if the deploy of the version is feasible.

The http client and *depctl* do not have any validations. Consequently, *Deployer* is the single point of defining validations.

### 5.3.4 Modifications of Resource Definitions

After *Deployer* validated the deploy request, *Deployer* takes the previously checked out resources and modifies them in the 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 **environment** and the image tag. The former is especially important for the metrics collection. *Deployer* modifies the environment in order to enable the instrumentation to

differentiate between the stable version and the canary version. The running Pods can later read the version information from the environment variables and are able to adjust the instrumentation configuration to add the version information.

The second important modification is the **image tag**. The image tag specifies the container image. The specific container image is import as it holds the specific application version. *Deployer* picks the correct image tag belonging to the version reference from the VCS.

*Deployer* modifies the resource definitions only when the image tag is unspecified. In case a Deployment resource already specifies an image tag, for instance somebody else maintains the image, then *Deployer* does not change the specified tag. The reason is that the CI does not build images, which are maintained by somebody else. Such images do not have a tag with the specific version reference of the VCS. One example is an additional statsd container running as a sidecar in the application Pod.

The other case would be that the Kubernetes resource defintion leaves the tag specification open. Then *Deployer* appends the tag to the container image according to reference which 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 happens during the deploy. For example *Deployer* does not find the tag to the corresponding commit hash, then the verification fails. Another example is that the Kubernetes API server returns an error. If this happens, *Deployer* will inform the developers. *Deployer* distinguishes between two different clients.

The first one is a CI system, which sent the deploy request. In this case the error the error reporting system[1] comes into play. *Deployer* sends the error to the reporting system (7) which collects the errors of multiple systems. The error reporting system informs the developers via sending notifications to a specified channel. GapFish has for example a Slack chat channel, to which the person on call subscribes and receives notifications from.

The other client is a developer sending a deploy request manually. In this case the developer typically uses *depctl* . *Deployer* differentiates usual curl requests from requests with *depctl* . Instead of utilizing the error reporting 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.

[1] Chapter 3 mentions Bugsnag. The application of GapFish uses it for error reporting.

As a result, developers always receive feedback as fast as possible in case a deploy fails. The information receives the developer in both cases, whether the CI system triggers a deploy or a developer manually triggers a deploy.

## 5.4 Testing Architecture

This section explains the testing architecture inside the production cluster. Few canary Pods in the new version run next to the stable version. These Pods are able to receive and respond to production traffic.

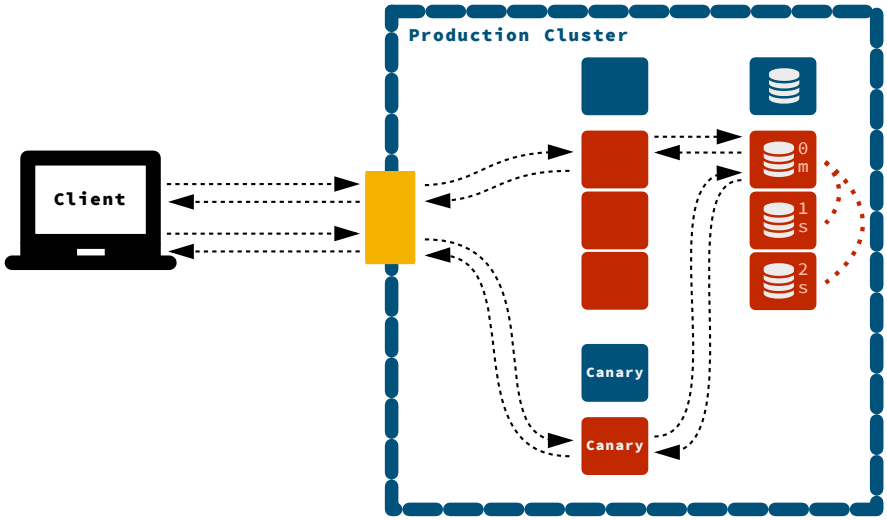



Figure 5.2: Canary Testing Architecture

To deploy a canary, the client sends the deploy canary request to a separate http path, which section 5.2 previously describes. The client needs to provide the same arguments as for the deploy request. Namely those arguments are the service and either a commit hash, a tag or both. When *Deployer* receives the canary deploy 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. Instead of deploying a new version, *Deployer* creates a additional new canary Deployment resource.

In the modification step *Deployer* does not only change the container image version, but it adds another canary Deployment. *Deployer* names the new Deployment resource with the name of the original Deployment resource and a canary suffix. Kubernetes differentiates the canary Deployment from the

original Deployment. Kubernetes creates Pods for the additional canary Deployment in the different version, which *Deployer* specified.

The Service though, which does the load balancing, selects both. It selects the Pods which belong to the original Deployment. And it selects the Pods, which belong to the canary Deployment. Hence the Service selects its Pods in account to the label. The original and the canary Deployment share a subset of their labels. A shared label would be for instance 'deploy=webserver'. And the two differentiating labels would be 'track=stable' and respectively 'track=canary'.

I designed the canary deploy to result in fewer canary Pods than stable Pods in order to reduce risk as the previous chapter describes. *Deployer* does this simply by scaling the canary Deployment to a single replica.

## 5.5 Metrics

### 5.5.1 Metric Collection

This section focuses on how metrics are being collected during when two versions are in production. There exist two separate collection mechanisms. Those are hosts, which collect metrics, and the application instrumentation, which sends metrics.

Firstly the group of metrics, which the host picks up. The host has information about the Pods' utilization of CPU and memory. Accordingly, on each Kubernetes host, a monitoring agent runs, which watches the '/proc' directory and the Docker daemon. The agent picks up the information frequently and then sends it to the monitoring system.

Secondly, the group of metrics, which the application sends. In order to collect the metrics, the application needs to be instrumented. Examples for those metrics are throughput, latency and error rate. In practice, I instrumented the applications to use the statsd protocol and statsd server for that purpose. Simple statsd libraries for most languages and frameworks exist even though the instrumentation could be better. In case there is a good instrumentation, developers can use it without much effort. The instrumented application then 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 the monitoring system.

It is important to correctly label the metrics in order to decide whether a canary or a stable version sends 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 *Deployer* has set to either stable or canary.

### 5.5.2 Metric Comparison

The comparison between the canary and the stable version is the part of *NPRT* , which creates the feedback loop.

The monitoring system basically consists out of a time series database, a graphing user interface and an alarm system. The time series database persists the metrics. And the user can define graphs from the metrics, which the user interface then displays. Moreover, the monitoring systems enables developers to define rules which monitor the metrics in the time series database. In case the metrics violate any rule, the monitoring system sends a notification.

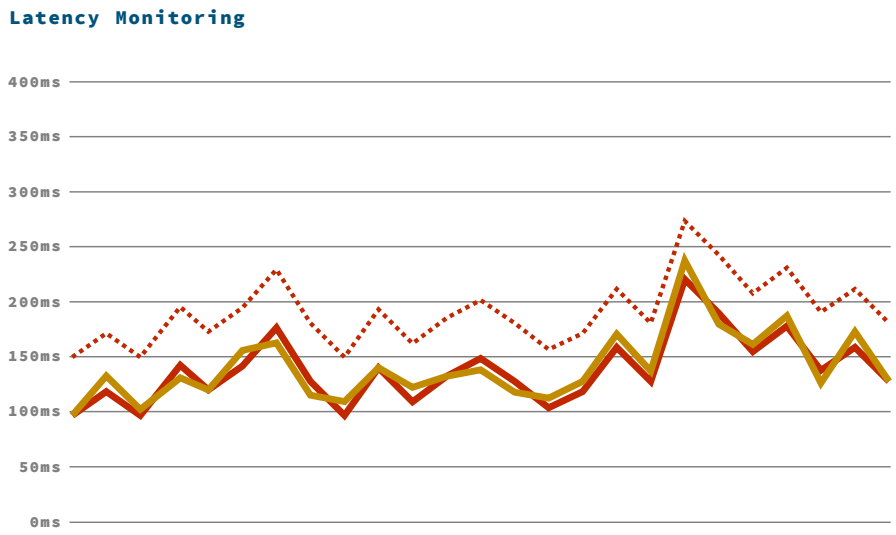



Figure 5.3: Comparing the Stable Version with the Canary Version

Figure 6.4 pictures an example of a rule on 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. The red dotted line depicts a threshold. The threshold depends on the latency of the stable version and is always 50ms higher than the stable version in the figure. In case the latency of the canary goes above the threshold, the canary violates the rule and triggers an alarm. The alarm is in fact a webhook, which would roll back the canary instance.

Next I discuss the metrics to monitor: Latency, utilization, errors and throughput.

Latency defines the duration of successful requests. Latency is a typical performance metric of a web server. Utilization mostly refers to metrics such

as CPU and memory consumption. The error rate is defined as counted errors in a time period. Applications sometimes have the problem of reporting too many errors without focus on important ones. Also throughput of successful requests can be an indicator of the health of an application.

Deviations between the stable and the canary always exist. The monitoring rules should identify problematic regressions. This is oftentimes hard to achieve. Section 6.1.4, in chapter evaluation, illustrates one such example of a metric comparison which serves as a test.

### 5.5.3 Metric Parameters

A comparison includes parameters such as a threshold, an aggregation window and a duration above the threshold. These parameters need adjustment in order to calibrate the rule and have a good recall and precision. The following gives insights on the factors, which affect those parameters.

The ratio of the stable and canary metrics as well as the throughput affect the size of the **aggregation window**. Let's assume the application has a very low throughput of only 1 request per minute and a ratio of 2 stable to 1 canary instance. In case the aggregation window is 1 minute large, then either the stable Pods would have 1 request or the canary Pod would have 1 request. This would be an improper aggregation window. Instead the aggregation window could be rather large with a size of 120 minutes. As result, the stable Pods would receive 80 requests and the canary Pods would receive 40 requests. This would be a much more appropriate aggregation window. Consequently the lower the throughput and the lower the stable canary ratio is, the larger the window should be.

The distribution of throughputs and latencies over the routes of the application affects which percentile to use. I make an example: If there would be a single route, which has a low throughput and a very high latency compared to other routes. The average would be an improper **aggregation**. The outliers of low latencies would affect the average. Instead a percentile or the median would represent the latency much better.

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

As a result, application with higher and the more evenly distributed throughputs have better metrics in order to define proper monitoring rules.

. . .

# 6 EVALUATION

This chapter evaluates the *NPRT* approach in three different aspects, experiments, a lessons learned section and a related work 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 the application use cases at DIN and GapFish. In the case of GapFish it was a application in production. The third aspect, related work, compares *NPRT* with other approaches in production and references similar tools like *Deployer* .

## 6.1 Experiments

I had the opportunity to do experiments with two different production applications. The first one is an Apache web server, which is the middleware in front of a tomcat application server. This application serves DIN's 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 web server to be able to run on Kubernetes. The Kubernetes resources for the Apache web server 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 Rails application to Kubernetes and utilized Codeship as CI, 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 applica-

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

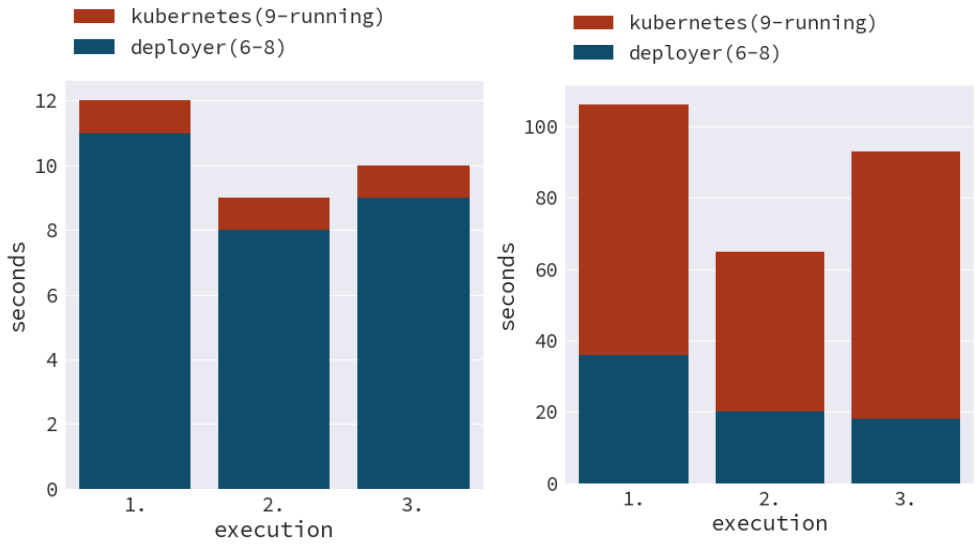
[2] p.e. `https://www.entscheiderclub.de` and `https://www.spiegel-panel.de`

tion were conducted on virtual machines hosted at Google. The appendix describes more about the specific details and the data used for the plots.

Each of the experiments has three consecutive executions with three different canary versions in the deploy and rollback experiments. The measurements had to be manually collected from all the different logs of the involved systems, that are a local http client, CI 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 Kubernetes 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 web server 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 CI 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 deploy. The makes a continuous deployment of new canaries absolutely feasible.

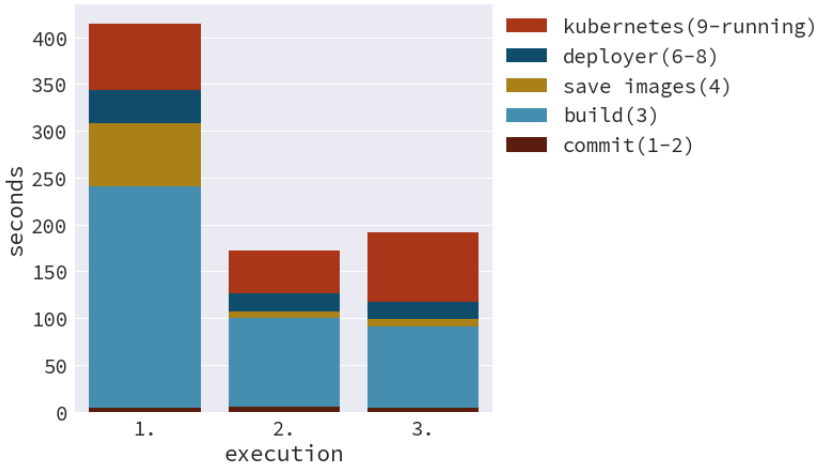



Figure 6.2: Deploy in Pipeline

### 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 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.

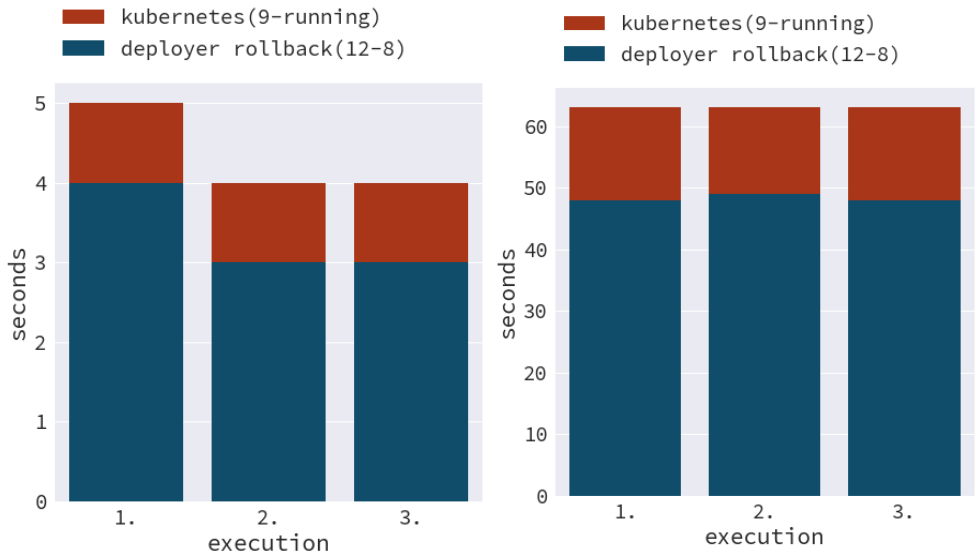
In the case of GapFish's Rails application, we have longer rollback times. All three executions of the rollback were 63 seconds. This is due to slow shutdowns of the worker Pods. The Kubernetes partly waits for the shutdown process of the Pods. That is why *Deployer* spends most of the 48-49 seconds 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[3]. So it is acceptable to have slightly 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,

[3]see glossary: zero downtime deploys




(a) DIN

(b) GapFish

Figure 6.3: Rollback

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.

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.

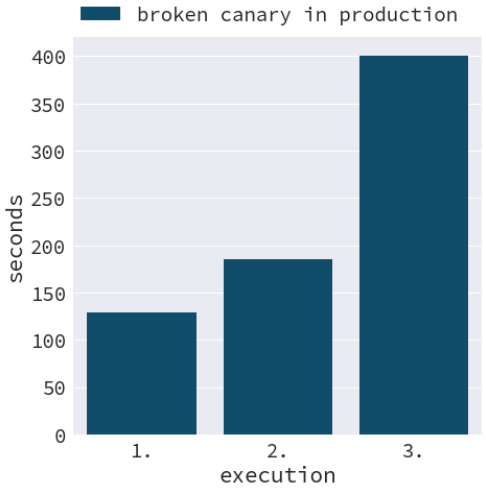



Figure 6.4: Metric Comparison

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

## 6.2 Lessons Learned

The GapFish 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 non-functional 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[4] 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.

[4]The users of the Rails application are only internal employees.

### 6.2.3 Continuous and Fully Automatable

*Deployer* connects the CI 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.

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 deploy 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 [15] 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 [16], which aims at functional tests being made in production. This enables testing microservices in complex environments. Also Tseitlin describes how Netflix [10] 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[5] has the ability to deploy applications to Kubernetes automatically. Also spinnaker[6] can deploy applications to cloud environments, with Kubernetes 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 [17], an open source tool similar to the previously mentioned gatekeeper at Facebook. It implements canary testing

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

and other testing strategies as well. The approach is a complete different, though, to *Deployer* . Bifrost is technically an automatically configurable middleware or load balancer. 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.

I used Datadog in the evaluation as monitoring system. However, there also exist other monitoring systems, such as the open source monitoring system Prometheus, which provides very similar features. Google has a similar monitoring system in there internal infrastructure [14].

Lastly, I am going to mention publications, which deal with metric analysis. Zimmerman motivates the need for data scientists software teams [18] and identifies the multiple roles [19]. Farshchi et al. evaluates a regression based approach in order to detect errors during rolling upgrades [20]. 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 [21]. 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 load balancer 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 non-functional 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 [22] 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 load balancer. The load balancer 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 load balancing.

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 deploys. 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 could simply clone the requests, send the original request to the stable version and send a cloned request to the canary. The load balancer 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 web servers.

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 [22].

**container** is a running instance of a container image [23, 24].

**container image** is an immutable snapshot of a container state [23, 24].

**container image registry** TODO.

**continuous delivery** is a practice to continuously deliver changes to production. Continuous delivery is related to DevOps [25].

**continuous deployment** is an extension to continuous delivery. An automated deploy process, which delivers most changes directly to production [22].

**continuous integration** is a practice of continuously merging code to a repository and automatically test it [26]. A continuous integration system is a server which automates the tests.

**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 [27].

**DevOps** is a made up of developers and operations. Devops practices aim for a short time between commit and deploy, still ensuring high quality [1].

**dynamic infrastructure** is software defined infrastructure [28, 29].

**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 [30, 31]. Beda gives a good overview of the architecture [13].

**microservices** TODO.

**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 [14, 32].

**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 [33].

**resource definitions** are files which define Kubernetes resources such as Service, Deployment, Pod etc..

**rolling updates** TODO.

**Service** is a Kubernetes resource which basically acts as a load balancer in front of Pods [34].

**StatefulSet** TODO.

**version control** is a practice, where developers keep track of every change of their code. A repository stores every version. Examples for version control systems are subversion and git. Humble explains what is important in version control [35].

**version reference** is either a commit hash (git), revision (subversion). In *NPRT* the container image tag includes a version reference.

**zero downtime deploys** is a way of deploying changes without downtime [22]. Rolling updates is an implementation.

. . .

# Acronyms

***NPRT*** *Nonfunctional Production Regression Testing* .

**CD** Continuous Delivery.

**CI** Continuous Integration.

**DIN** Deutsches Institut für Normung.

**SRE** Site Reliability Engineering.

**VCS** Version Control System.

. . .

# Bibliography


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

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

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

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

[5] 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.

[6] 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.

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

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

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

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

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

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

[13] 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.

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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).

[21] 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.

[22] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[35] 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.