NONFUNCTIONAL PRODUCTION REGRESSION TESTING

implemented with Kubernetes

Masterarbeit
Computer Science
Fachgebiet Komplexe und Verteilte IT-Systeme
Fakultät IV - Elektrotechnik und Informatik
Technische Universität Berlin

vorgelegt von **Sebastian Schasse**, Matrikelnr. 318569

Aufgabensteller: Prof. Odej Kao
Mitberichter: Prof. Sahin Albayrak

Berlin, 29. Januar 2018

Erklärung der Urheberschaft

Die selbstständige und eigenhändige Anfertigung versichere ich an Eides statt.

Ort, Datum Unterschrift

Acknowledgment

First, I would like to thank Odej Kao and Sahin Albayrak for reviewing my master's thesis .

I would like to express my deepest appreciation to all the employees of GapFish, especially Frank Schütz, Thomas Peikert, Robert Aschenbrenner, Gebhard Wöstemeyer, Christiane Okamoto and Ferdinand Bleschke. Without their continuous support and reviews, this thesis would not have been possible. They all supported me, and we had many constructive discussions that influenced this thesis. Special thanks to Oliver Weyergraf for making it possible to carry out this work at GapFish.

I would also like to show my gratitude to Thomas Böhm. He made it possible to evaluate my master's thesis at DIN. I thank Maik Tirock for helping me with the DIN environment and servers.

Finally, I thank my family for the support they gave me during my studies at TU Berlin.

Abstract

Many companies successfully use DevOps practices and microservices to eliminate the conflicts between developers and operations teams. Initially, the thesis explains one of the impacts of those practices: a shift of responsibility of maintenance work from operations to development. Developers have to maintain their own code as well as many dependencies. Containerization technologies consolidate this shift. A problem is that the continuous delivery pipeline supports developers only up until the deploy and not further. Nonfunctional Production Regression Testing (NPRT) extends the continuous delivery pipeline to the production environment. NPRT addresses the issue of maintenance in a continuous and fully automatable fashion. In NPRT, a new version in the form of a canary will be automatically deployed to production and compared to the stable version. In case of a regression, the canary will be automatically rolled back. In order to enable NPRT with Kubernetes, I implemented Deployer, which connects the continuous integration system with a monitoring system and Kubernetes. An Apache middleware, which is used at Deutsches Institut für Normung (DIN), and a Rails application, which GapFish develops and runs in production, served in multiple experiments in order to evaluate NPRT for real-life use cases.

Viele Unternehmen setzen DevOps Praktiken und Microservices ein, um Konflikte zwischen den Entwicklern und dem Operationsteam zu beseitigen. Zunächst erklärt die Masterarbeit eine der Auswirkungen dieser Praktiken: eine Verlagerung der Verantwortung der Wartungsarbeiten von Betrieb nach Entwicklung. Entwickler müssen sowohl ihren eigenen Code als auch viele Abhängigkeiten warten. Containerizationtechnologien verfestigen diese Verlagerung. Ein Problem ist, dass die Continuous Delivery Pipeline Entwickler lediglich bis zum Deploy unterstützt und nicht weiter. NPRT erweitert die Continuous Delivery Pipeline bis in die Produktionsumgebung. Es ist ein kontinuierlich anwendbarer und voll automatisierter Ansatz, der das Problem der Wartungsarbeiten addressiert. Bei NPRT wird eine neue Version in Form eines Canarys automatisch in Produktion deployt und mit der stabilen Version verglichen. Im Fall einer Regression wird der Canary automatisch zurückgerollt. Um NPRT mit Kubernetes zu ermöglichen, habe ich Deployerimplementiert, der das Continuous Integration System mit dem Monitoring System und Kubernetes verbindet. Ein Apache Middlewareserver, der bei DIN verwendet wird, und eine Railsanwendung, die von GapFish entwickelt und in Produktion betrieben wird, dienen in mehreren Experimenten, um NPRT praxisnah zu evaluieren.

. . .

Contents

. . .

1 INTRODUCTION

1.1 Automation to Support Maintenance Work

DevOps [1] and microservices [2] are practices, or even cultures, that break down the silos between the development and operations 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 for maintaining a software product has changed. A single team is responsible for two aspects: their own code and the dependencies they are using and relying on. Teams must maintain their own with nonfunctional changes concerning performance, scalability or bug fixes. Code written by others also needs to be maintained by adapting it 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 a continuous automatable practice to fill this need. But it covers the process only up to the deploy[1], whereas maintenance involves more than just the delivery.

Instead of employing more engineers, Google combats workforce problems with automation [6]. They recognized that automation not only saves valuable programmer time but also 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 a practice that addresses the amount of maintenance work with NPRT. The practice is designed for teams that have the responsibility of developing and operating their own code as well as code dependencies. NPRT extends the CD pipeline and automates continuous testing of nonfunctional changes in production. I implemented NPRT at two companies, DIN and GapFish, in order to evaluate the process.

[1] The term deploy serves in order to differentiate the procedure from Deployment, the Kubernetes resource.

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 solving 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 three-tier web application and names the software dependencies.

4 NPRT Approach Chapter 4 describes the conceptual macro view of NPRT. The text explains 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 further examines 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. First, it discusses the performance with the help of experiments. Second, it provides a qualitative evaluation of NPRT based on the experiences at DIN and GapFish. And last, it relates NPRT to other practices, tools and sophisticated metric analyses.

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 the development and operations teams. The aim is to change the software product quickly while providing a stable and reliable software product of high quality. Vogels, the chief technology officer at Amazon, once said in an interview "You build it, you run it" [3]. This quote represents the awareness of teams to be responsible not only for the development and deliver features but also for operating their product in production, because operation is an essential part of a successful software product. One team is consequently responsible for the development, operation and maintenance of their own code.

Furthermore, the responsibility of software dependencies is shifting. In the past, development teams delivered code and depended on the previously installed software dependencies. Operations teams were responsible for maintaining those installed dependencies. Virtualization changed this. Teams operating virtual infrastructure operate and maintain the hardware and software for 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.

One 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 operations teams. Nevertheless, it is becoming 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.

Containerization[4] manifests the responsibility shift. It is a further step of isolation. The operating system is able to isolate Central Processing Unit

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

(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 toward the team that 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, developers would need to react to those issues and for instance scale the software product 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 functions visible to users. At Facebook those invisible changes are more than 50% of the total changes [4].

Moreover, software products have a high amount of dependencies. This is very reasonable, because when programming a software product, one does not want to build it from scratch. First, the dependencies are different abstraction levels, such as a processor, a programming language and a programming framework. And second, the dependencies are modules, which encapsulate

a certain functionality. Those functionalities can be common functionalities for applications, for instance authentication of a user. Abstractions 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.

One 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 one compares two different versions in production. Another practice is canary releasing[5], which changes 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 that 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] that 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, this 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 save valuable human resources and 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.

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.

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

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 not further. NPRT extends the pipeline to the production environment, where developers 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 Three-Tier Web Application in Kubernetes

I will explain the typical implementation of a three-tier web application in Kubernetes as it helps 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.

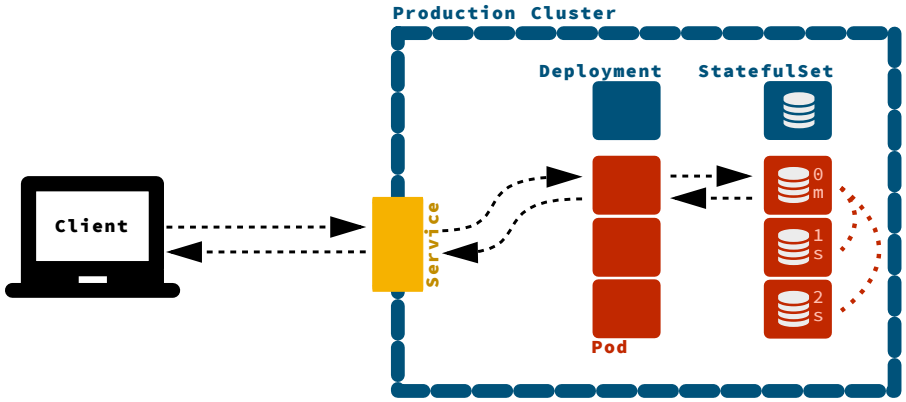


Figure 3.1: Typical Three-Tier Web Application in Kubernetes

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
[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]. Deployer 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] in order to inform a developer. Deployer uses kubectl to interact with the 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 requests with dependency updates.

The monitoring system[15], which was used in the evaluation, is Datadog[16], a commercial service. Nevertheless Prometheus[17] is also a suitable open source solution as monitoring system. Statsd[18] is a protocol and server for application instrumentation. The Continuous Integration (CI)[19] system, which was used in the evaluation, are Codeship[20] and Jenkins[21]. As an image repository, GapFish uses Docker Hub[22]. Goreplay[23] 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] https://github.com/etsy/statsd
[19] see glossary: continuous integration
[20] https://codeship.com
[21] https://jenkins.io
[22] https://hub.docker.com
[23] https://github.com/buger/goreplay

. . .

4 NPRT APPROACH

In NPRT, the pipeline deploys a nonfunctional 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 the development team and production environment that is fully automatable. The following elaborates on the term in more detail.

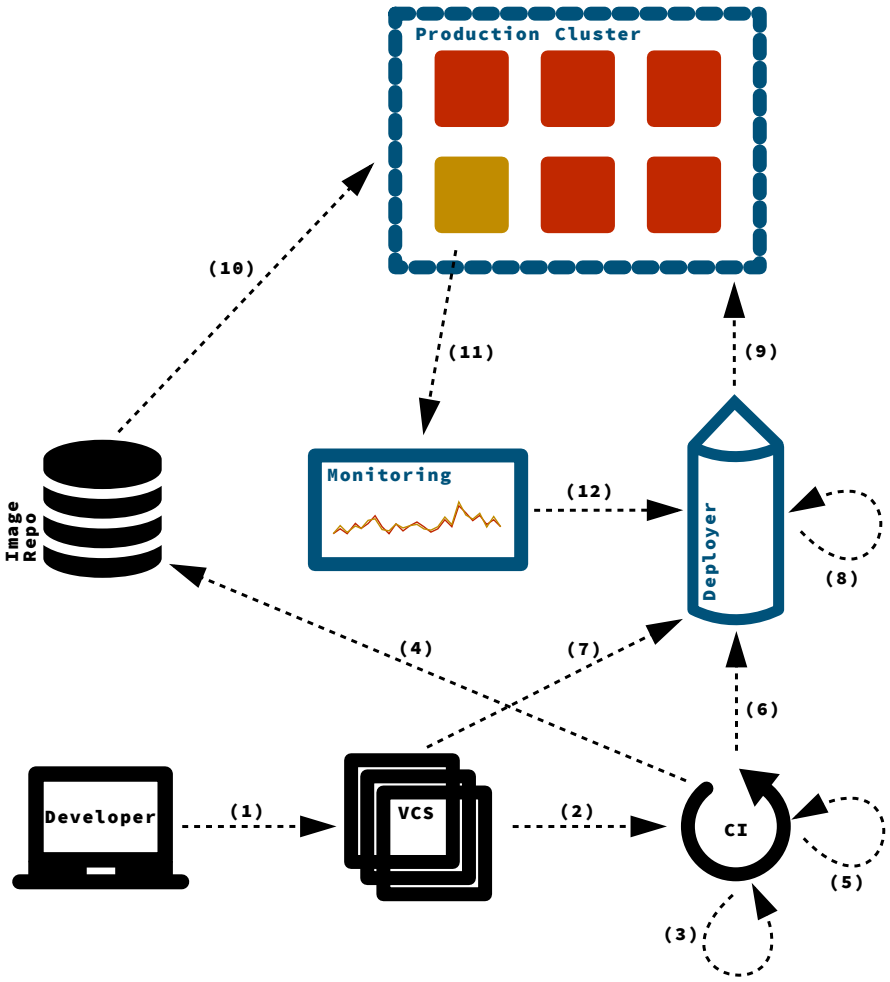
Nonfunctional refers to the change, which the monitoring system monitors. Usually maintenance changes are nonfunctional changes, for instance performance improvements or security updates. Production refers to the environment, in which the versions are monitored. The production traffic reaches the new 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 that 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 this overview. In the following, I will illustrate 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. This 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)[1] 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. First, it starts a build process (3–4), second, it runs the tests (5) and third, 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[2], 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. It should be mentioned that the extra name is not absolutely necessary to definitely associate the container image with a version. However, experience has shown that a human readable name is very helpful to 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. The tag, consisting of the version and extra name, will tag every step of 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. Developers commonly develop new features on a separate branch. For those

[1] see glossary: version control
[2] see glossary: version reference

versions one usually would 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. NPRT require it though. For NPRT it is especially important to associate and trace every step through the whole pipeline. For that purpose NPRT needs the CI system to tag the container images with the exact version reference.

4.3 Customizations of Continuous Deployment

NPRT is an approach that 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 the case of GapFish outsourced to a third party company. GapFish does 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[3] define the application infrastructure.

[3] 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: first, pull the resource definitions (7), second, modify the resource definitions (8) and third, 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. The resource definitions are version controlled 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, that Chapter 3 explained 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 that 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. Then, the production cluster changes the cluster state according to the definitions. Most important is that Pod instances of two versions are in parallel in the cluster. The production cluster fetches the container images, which the CI system built, in step (10) from the image repository. The production cluster is aware of the specific image identified by the tag.

Running multiple versions in production might not sound optimal. Kubernetes approaches zero downtime deploys via rolling updates. Rolling updates require it to be able to run two successive versions side-by-side. Accordingly, the proposed technique optimally fits to the model of rolling updates.

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. These few instances are called canaries. The load balancer sends traffic to both versions and both versions respond to clients, on which section 5.4 elaborates.

After Deployer deployed the canary, a regression might arise. On account to that, the regression should affect the least users as possible. This is why there is only few canaries running in the production cluster. This means for an exemplary ratio of three instances in the stable versions to one canary in the new version, only 25% of the traffic would hit the canary with the regression. Even if the request of a specific single user hits the degraded instance, the next

request of the same user has the probability of 75% to hit the stable version. This certainly lowers the risk of failure for a single user.

A limitation to the approach is that the new version needs to be able to run side-by-side with the stable version. This is why the new version should not have any functional changes compared to stable version, but only nonfunctional changes. However, that means NPRT cannot test new feature like in an A/B test. Instead, NPRT is appropriate to test performance improvements, refactoring or dependency updates.

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 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 now monitors those two graphs for a regression. 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 in 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 in order 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 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.

. . .

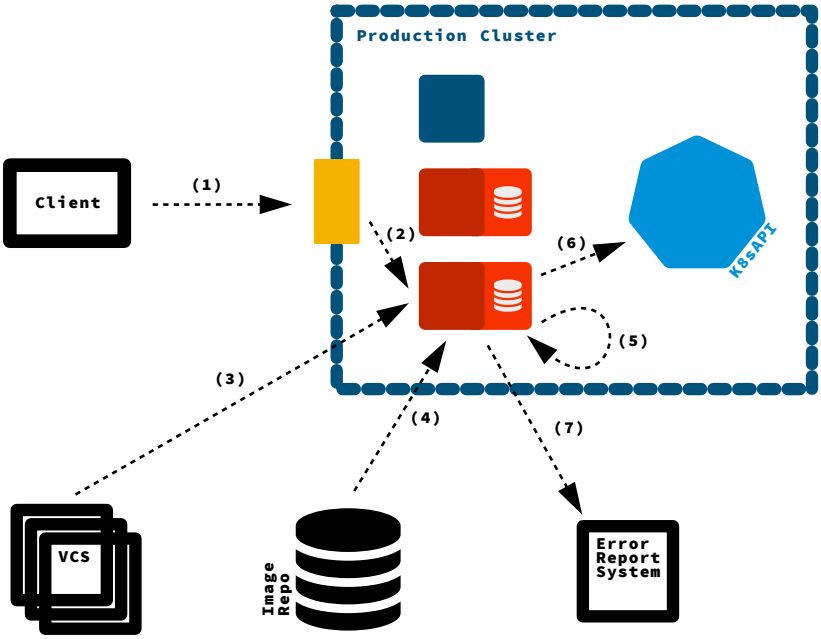
5 IMPLEMENTATION OF DEPLOYER AND METRIC COMPARISON

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, it is stateless and can even deploy itself. 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 a Hyper Text Transfer Protocol (HTTP) interface. The figure shows that Deployer consists of multiple Pods. These Pods are replicated and identical in their behaviour. The Pods are 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 immidiately. 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 explain the interface of Deployer. Deployer provides different endpoints: ls, show, tags, deploy, canary and version. The descriptions use 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

This section explains the steps involved in the deploy process. Initially, a client sends an 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, written by somebody else, nor read credentials from the production cluster.

5.3.2 Fetching the Code and Caching It

After Deployer authenticated a valid client, Deployer then fetches the code from the VCS (3). If Git is the VCS, Deployer uses commit hashes as version references 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 fetches only the changes.

As a simplification, every Pod has its own cache that lives as long as the Pod. Thus, every Pod utilizes its own volume as a cache, so that 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, which stores the caching data . Thus, the cached data is not available anymore or in other words the cache is empty again.

The VCS contains application code as well as the resource definitions. In Kubernetes, these resource definitions define the application infrastructure. These resources define Deployments, StatefulSets and Services. The resource definitions should be stored in the '/kubernetes' directory. This is a convention of Deployer and it assumes that the '/kubernetes' directory is the standard

location. If the 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 that 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 image by communicating with the image repository (4).

For simplicity and usability the client usually sends only one argument, either the commit hash or the tag. One argument is completely sufficient to determine the version. 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 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 the reference which the client provided.

The next step is to communicate the modifications to the Kubernetes Application Programming Interface (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 reporting system[1] comes into play. Deployer sends the error to the reporting system (7) that collects the errors of multiple applications. The error reporting system informs the developers via sending notifications to a specified channel. The person on call, which subscribes to that channel[2], receives the notification and is able to react.

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] The applications of GapFish use Bugsnag for error reporting.
[2] GapFish uses Slack, a team chat, as such a channel.

As a result, developers always receive feedback as fast as possible in case a deploy fails. The developer receives the information 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.

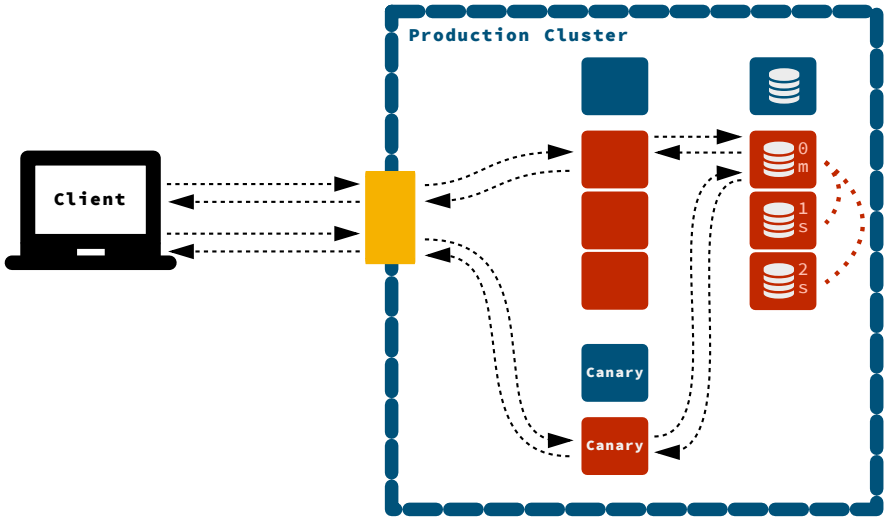


Figure 5.2: Canary Testing Architecture

To deploy a canary, the client sends the deploy canary request to a separate HTTP path, which section 5.2 previously described. The client has 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 an 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. Furthermore, 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. 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 the time of two versions being in production. There exist two separate collection mechanisms. Those are agents on the hosts, which collect metrics, and the application instrumentation, which sends metrics.

First, the group of metrics, which the agent on 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.

Second, 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 monitoring system consists 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 system 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.

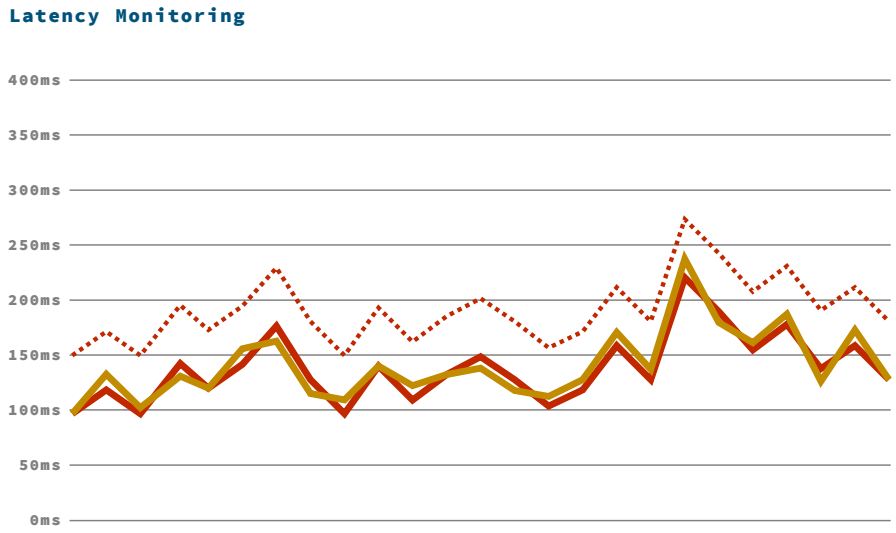


Figure 5.3: Comparing the Stable Version with the Canary Version

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.

The following discusses 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 the 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 that 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 in order to 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 one request per minute and a ratio of two stable to one canary instance. In case the aggregation window is one minute large, then either the stable Pods would have one request or the canary Pod would have one request. This would be an improper aggregation window. Instead, the aggregation window could be rather large with a size of 120 minutes. As a 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. Here is 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 be 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, applications 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. First, the experiments aim to present an evaluation of the approach's performance 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 an application in production. The third aspect, which is 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 one of DIN's website[1]. In the case of DIN, I set up a version control system, a Kubernetes cluster with Deployer, instrumentation and the monitoring system. Developers working for DIN helped to set up 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 back-office 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. One Deployment is for the Ruby on Rails application server with two Pods.

The experiments for the DIN Apache application were conducted on virtual machines rented from Atos. The experiments for the GapFish application were conducted on virtual machines hosted at Google. The appendix describes more about the specific details.

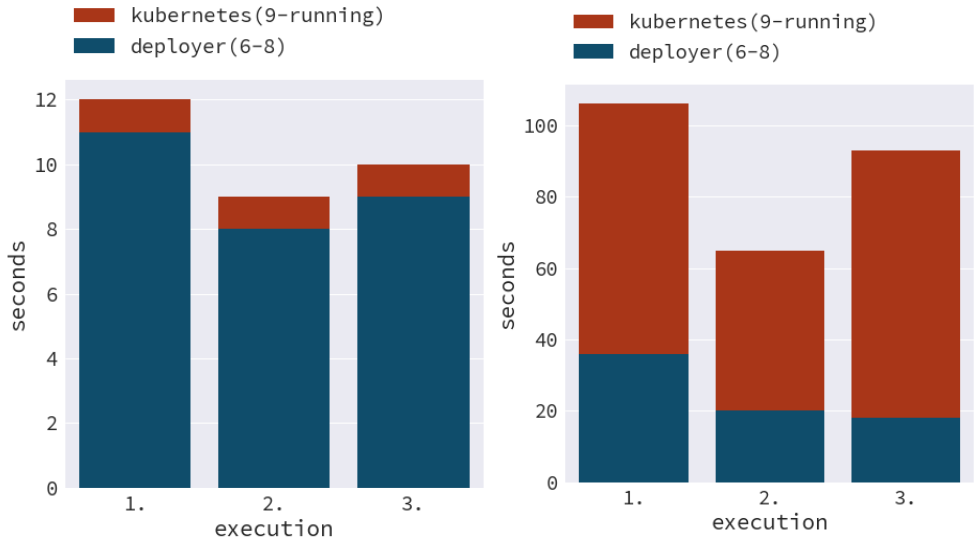
[1] `https://www.beuth.de`

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

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. Those systems were the local HTTP client, the CI system, Deployer and Kubernetes.

6.1.1 Deploy

The first experiment examines the performance of a deploy of canaries. On 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 VCS, the time to check of the version reference and image availability, the time of the resource definitions modification and the time of the 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. In the third execution, Deployer took 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 were faster than the first one. That is because Deployer caches the code from the VCS and only needs to download the difference in the consecutive executions. The impact of the cache on DIN's Apache web server was not that large, because the repository size is comparably small. 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 was significantly faster. Deployer is able to handle multiple Deployments as in the case of GapFish. However, the multiple requests to the Kubernetes API slightly reduce the deploy duration. Deployer's cache works well for larger repository sizes.

It is important to pay attention with slow application startup durations when looking for fast deploys. In the case of GapFish's Rails application server, this makes the greatest 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 Docker Hub, 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 to the VCS, 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. The focus also lies on the integrated tools.

At GapFish, I could access all the involved systems. 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, the diagram shows an overall pipeline duration for a typical canary change at around 3 minutes, including a build and deploy.

6.1.3 Rollback

This section takes a look at the rollback performance of Deployer. The blue part of the bar again represents the time Deployer is involved that is from

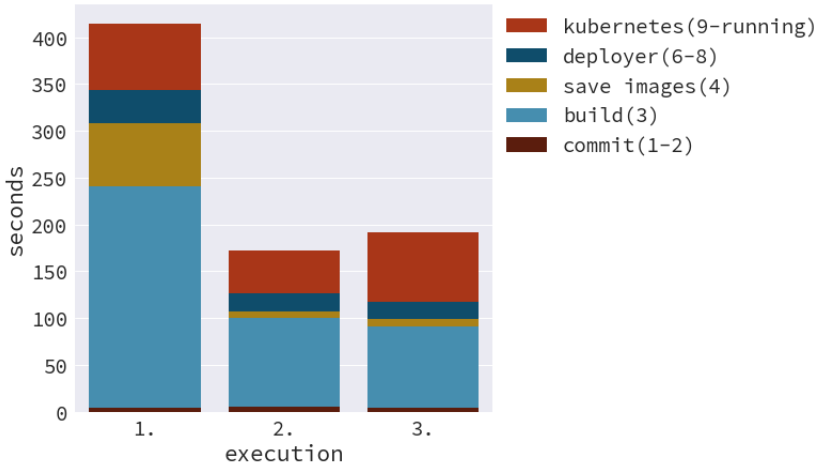


Figure 6.2: Deploy in Pipeline

the rollback request including steps `(7)` and `(8)` until the Kubernetes API responded. The red bars represent the remaining time the Pods take to shut down.

Figure 6.3a shows the rollback duration at DIN is between 4 and 5 seconds. The duration, which Deployer takes for a rollback, is less than for a deploy. That is mostly because a delete request to the Kubernetes API is faster.

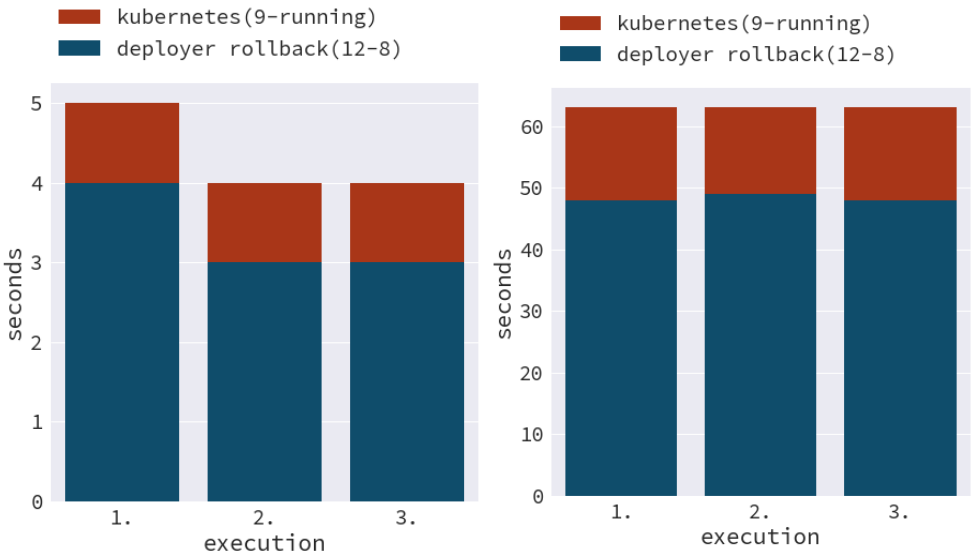
In the case of GapFish's Rails application, the rollback times are longer. All three executions of the rollback were 63 seconds. This is due to slow shutdowns of the worker Pods. Kubernetes partly waits for the shutdown process of the Pods. That is why Deployer spends most of the 48-49 seconds waiting for the Kubernetes API to respond.

Also for the rollback case, Deployer's performance is better with DIN's Apache system, because of less Deployment resources and the fast Apache application. Nevertheless, deploys as well as rollbacks involve zero downtime.

6.1.4 Metric Comparison

This last experiment aims at an evaluation of the feasibility of a metric comparison and the duration of a broken canary being in production. DIN provided one hour of production traffic of their website that a Goreplay process recorded in production. During the experiments, this traffic dump was replayed with the test setup.

Due to the fact the test database state differs to the state of 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. The successful 200 responses consisted of the publicly visible pages of the website.



(a) DIN

(b) GapFish

Figure 6.3: Rollback

At first, a canary version with no changes was running in comparison to the stable version. The two versions, 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 started. 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. Here is the definition of the monitoring rule:

```
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 the throughput of successful responses. The canary responses are weighted by the factor 2, because there were one canary Pod and two stable Pods. An aggregation window of 30 seconds and a threshold of 3, which means the canary throughput had to differ 3 times as much as the throughput of the stable version.

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.

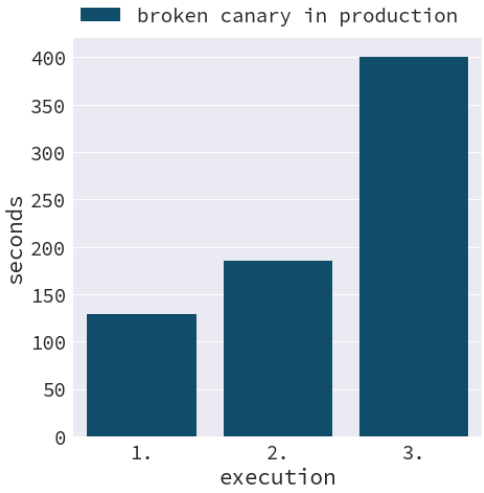


Figure 6.4: Metric Comparison

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. This section presents qualitative feedback which mainly comes from use of NPRT in production.

On the one hand, the developers used NPRT to test maintenance of own code, such as refactorings and performance improvements. On the other hand, the developers also used NPRT to maintain dependencies. The previously developed GemUpdater regularly and automatically opened pull requests with dependency updates. The canary version was tested against the stable version using error metrics that the following sections explain in more detail.

6.2.1 Quick Debuggable Deploys

Deploys of around 1.5 minutes conform to 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 that has a current median duration of 12.5 minutes.

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

sible to track the code change of a stable and a canary version which used to be running side-by-side.

It has been a good decision to version the Kubernetes resources. This made it also possible to 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 developer team to deploy and rollback. As a result, 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. However, The problem of overfitting in statistics is well known. Critics could argue that production metrics can be hard to evaluate and in a test environment it is rather easy to achieve good results.

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 experiment 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 also provides interfaces to other services. This leads on the one hand to sparse usage[3] on routes for users and on the other hand to machines utilizing 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. It shows newly introduced errors only. This has been the best metric for the regression tests.

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

6.2.3 Continuous and Fully Automatable

Deployer connects the 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 developers were able to test changes in production quickly and have a small feedback loop.

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

The developers working for DIN are still sceptical to have unsupervised deploys. But in the example of GapFish, the developers 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 that used to be an important but tedious maintenance work.

6.3 Related Work

There are multiple publications which use methods to test software 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] that 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 that produce random defects in a controlled production environment. The Simian Army lays open unknown issues in order to gain better reliability.

There are also countless open source tools which help to implement continuous deployment similar to Deployer. Vamp[4] has the ability to deploy applications to Kubernetes automatically. Also Spinnaker[5] can deploy applications to cloud environments with Kubernetes being one of them. Very recently, Spinnaker released a feature to analyze canaries, just what NPRT needs. The difference to Deployer is that Vamp and Spinnaker are very complex and exhaustive. They also require additional databases. Whereas Deployer is a very simple and lightweight tool with focus on a single purpose and consisting of only one Docker image. Nevertheless, Deployer can be integrated to several monitoring solutions via 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 and other testing strategies as well. The approach is a complete different though compared 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.

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

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

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

This last paragraph mentions publications which deal with metric analyses. Zimmerman motivates the need for data scientists in 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 has 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 analyze errors of a distributed service in an A/B experiment [21]. They provide guidelines for the design of error analysis models.

. . .

7 CONCLUSION

The thesis illustrated that NPRT extends the CD pipeline and supports developers not only until the deploy but well into the operation of their software product. The monitoring system evaluates the test without the supervision of the developers at a very low risk. The change goes through the whole pipeline, including tests in the production environment. If the test is successful, developers will have no further work to perform. Potential changes, implemented by bots, would be robust enough to be able to act fully automated and unsupervised.

7.1 Summary

The thesis proposed NPRT to support maintenance work. Chapter 2 examined the problem of maintenance work. It illustrated that maintenance responsibility shifts from the operations teams to developers due to DevOps and microservice practices. The simplicity of containerization enables and manifests this shift as well. The maintenance work consists of two main aspects: maintenance of the development team's own code and maintenance of dependencies. Many flexible microservices and many dependencies result in a high amount of maintenance work. To address the issue of maintenance work, the chapter suggests the value of automation, which is time saving and reliable. The chapter proposed NPRT in order to extend the continuous delivery pipeline and to automate nonfunctional regression tests in production.

Chapter 4 demonstrated a macro view of NPRT and how it integrates with all the involved systems. The chapter explained all the steps of the pipeline: First, a developer commits a new version to the VCS. Second, the CI system builds and tests the new version. Third, Deployer deploys the new version to production in the form of a canary. Fourth, the monitoring system collects and compares the metrics of the side-by-side running versions and in case of a regression. And lastly, Deployer rolls back the canary. It is important to tag each step including the metrics with the version reference in order to be able to debug a specific version. With those steps, NPRT creates an automated feedback loop between the development team and production environment.

I implemented Deployer in order to enable NPRT with Kubernetes and used Datadog for the metric comparison. Chapter 5 demonstrated the sim-

plicity and horizontal scalability of Deployer's architecture. Deployer is implemented in order to run in Kubernetes and to deploy itself. The chapter explained Deployer's HTTP interface and its endpoints ls, show, tags, deploy, canary and version. Simple HTTP clients or depctl can interact with Deployer. Moreover, the chapter explained a deploy of a canary. Deployer authenticates the client, validates the given version references, fetches and modifies the Kubernetes resource definitions and applies them to the Kubernetes API. In case of an error, Deployer informs the developers as quickly as possible by sending an appropriate return message to the client and by sending the error to a specified channel. The chapter described how I designed the architecture of two versions running in Kubernetes and how Kubernetes does the load balancing between them. The metrics of the application's instances are collected in two different ways. A monitoring agent picks some of the metrics from the host, and the application's instrumentation sends the other metrics to the monitoring system. Finally, Chapter 5 elaborated on the metric comparison itself and important parameters: the aggregation window, the aggregation and the threshold.

Chapter 6 evaluated NPRT in three different aspects: experiments, lessons learned and related work. For the experiments, I tested Deployer and the metric comparison with two different production applications. The first one was an Apache web server that DIN uses as a middleware. The second application is a Rails web server that serves as the back-office tool for GapFish. The experiments showed that Deployer's duration of a deploy varies between 9 and 106 seconds, and the duration of a rollback was between 4 and 63 seconds. The long deploy of 106 and the long rollback of 63 seconds were determined by the long start and stop times of GapFish's Rails application. In contrast to the long start and stop, Deployer's performance was quite fast. The experiments proved that a metric comparison in production is fully automatable. Also, the usage of NPRT at GapFish in production proved that the metric comparison is feasible because developers usually have a good understanding of their software product and are able to write suitable tests. The deploy durations were efficient to use Deployer in production as well. The chapter also demonstrated that GemUpdater, a tool to automatically update dependencies, successfully committed version changes which were then automatically tested in production via NPRT.

7.2 Outlook and Future Work

NPRT is a good approach to continuously automating the maintenance of a software product. Nevertheless, there are possible improvements. Goreplay has proven to be a good tool for the evaluation. In order to reduce even more risk, Goreplay could serve to clone the traffic and send it to the canary. In this case, it would be possible to compare 100% of the production traffic and make the metric comparison much simpler and more precise. Furthermore,

the users would not receive the responses of the cloned traffic. This would result in a riskless testing approach. But there is one open problem with this approach: there must be a mechanism to imitate the interaction with the database.

Another aspect to consider for improvement is the lack of good open source instrumentation. Though good instrumentation exists, it is proprietary and cannot be utilized outside of the proprietary ecosystem. There should be more investment 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, however, it was not possible to version those tests. With other tools, like Prometheus, this may be possible.

A realistic extension to NPRT would be completely unsupervised updates. An automatic approach such as GemUpdater works well. As a result, developers only have to do manual work in case of a regression. More automatic services of this kind are needed. One obvious use case is the automation of updating Docker base images. A Docker base image updater would help to continuously update the operating systems and software on which the applications' Docker images are based. Furthermore, there is a similar need for update mechanisms for other languages, such as Java or Python, which come with their own dependency managers.

. . .

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 the 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 is a repository that stores container images..

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

continuous deployment is an extension of continuous delivery. An automated deploy process delivers the changes directly to production [22].

continuous integration is a practice of continuously merging code into a repository and automatically testing it [26]. A continuous integration system is a server that automates tests.

Deployment is a Kubernetes resource that ensures the existence of replicated Pods. Deployments enable the implementation of stateless instances such as web servers and provide rolling updates. Deployments are an evolution to ReplicationControllers, which had similar features [27].

DevOps is made up of developers and operations. DevOps practices aim for a short time between committing code and deployment while still ensuring high quality [1].

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

immutable infrastructure in order to change instances in such an infrastructure, nodes are destroyed and created.

Kubernetes is a cluster management system which was heavily influenced by Google's Borg [30, 31]. Beda gives a good overview of the architecture [13].

microservices are a software architecture in which several small services create a modularized software system, each with a specific function. These services communicate with each other via a network [2].

monitoring system collects information about a software product and monitors it for defects. I refer in the thesis to a monitoring system with a modern design as Ewaschuk and Bass et. al describe [14, 32].

Pod is a Kubernetes resource. A Pod consists of multiple containers located on the same host that are able to share resources [33].

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

rolling updates is a mechanism to update multiple instances of a system without downtime. One instance will be created in the new version, and an instance of the old version will be shut down. The mechanism proceeds until every instance is in the new version.

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

StatefulSet is a Kubernetes resource similar to Deployments. However, the StatefulSet creates Pods with fixed hostnames and eventually binds them to fixed volumes. The StatefulSet enables the realization of distributed, stateful applications such as databases .

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) or 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.

API Application Programming Interface.

CD Continuous Delivery.

CI Continuous Integration.

CPU Central Processing Unit.

DIN Deutsches Institut für Normung.

HTTP Hyper Text Transfer Protocol.

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. Continuity with Dynamic Infrastructure, pp. 275–306.

[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, pp. 21–40.

[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.