NONFUNCTIONAL PRODUCTION REGRESSION TESTING

---

implemented with Kubernetes

. . .

Contents

. . .

1 INTRODUCTION

1.1 Introduction

Many people talk about DevOps [1] as well as there are multiple definitions and interpretations of the term DevOps. DevOps is referred as a philosophy, a culture, practices and specific tools. For my research, I will focus on two different aspects of the term DevOps:

The first one is the perspective of operation teams. Operation teams traditionally modeled infrastructure by installing physical hardware and by manually installing software components. With the rise of virtual machines and the cloud, it became possible to model infrastructure in software [24]. Modelling via software enables operation teams to use tools and practices as seen in software engineering. Infrastructure code is version controlled, tested and can be automatically deployed.

The other aspect of DevOps is the perspective of developer teams. Previously developer teams were only responsible for developing new features. Software engineering practices got established and proven. One of those practices is the continuous delivery pipeline [19]. The last step of the continuous delivery pipeline is the deployment. Formerly operation teams were responsible for deploying new features. The deployment as last step of the continuous delivery pipeline shifts a responsibility from operation to development. This shows that developer teams are becoming more and more responsible for running the software, they built.

1.2 Thesis outline

In the following outlines the structure of the thesis. We discuss every chapter briefly and talk about its contents.

In the first chapter we will go through the foundations. The chapter provides references the practices and technologies, which are crucial to understand the thesis. The references are properly selected to understand the details in case those are not known. In summary those practices and technologies are kubernetes, continuous delivery, continuous deployment and techniques from infrastructure as code and site reliability engineering.

The second chapter is a conceptual macro view to the method nonfunc-

tional production regression testing. The text goes through components of the whole environment and discusses the most important concepts and how they communicate with each other. The explanation of the communication between the components clarifies how the pipeline embeds the methodologies of nonfunctional production regression testing and how those extend the pipeline.

In the third chapter we will get to the concrete design of nonfunctional production regression testing. To enable nonfunctional production regression testing, I wrote the software deployer. The chapter explains the details of the design concept and design decisions. After reading this chapter it is clear how to use the software deployer.

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

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

. . .

2 PROBLEM DEFINITION

2.1 The Continuous Delivery Pipeline consists of commitment, continuous integration and deployment

2.2 Software Deployment approaches evolved from manual to automated

2.2.1 Blue-Green Deployment allows Zero Downtime releases

The approach of blue green deployment involves two environments. One is the environment which serves production traffic, the other environment is in standby. You deploy to the other environment the new version. You then switch routing from the environment, which runs the old version, to the environment with the new version.

This has the major disadvantage that it is a waste of resources. Just one environment doing work with serving production traffic as the other environment is just idling. Even if you use the idling environment as a staging it would be oversized and still wasting resources.

2.2.2 Automation leads to resource saving Phoenix Deployment and Rolling Deployments

In times of dynamic resource allocation, automation and virtual machines, it became easy to automatically spawn new servers. In for a deploy of a new version you would not change the servers, but automatically create new ones with the version to deploy, then switch the traffic from the old servers to the new servers, and in the end just destroy the old servers. This procedure is called phoenix replacement.

2.2.3 Canaries test releases with a small amount of traffic

Canary releasing is a way to test new versions of the application in production. But testing in production is risky, because when there is an error which leads to a defect or failure the users will get affected. And this costs money in any way.

There comes canary releasing into play. A single canary can't do much harm and it's not a big catastrophe if the small canary is malicious. Let me explain it at the example of a typically scaled web application. Usually you do not have just a single web server, but multiple servers. So when releasing in the canary style, you change just a small proportion of the twenty webservers to the new version. Now two versions of the webservers are running at the same time.

Usually you want exactly the same version of webservers in production with the exact same configuration. This makes systems easier to debug in case of an error. The maximum count of different versions during canary releasing is two. But why go for two different versions in production with the canary releasing technique, when it makes debugging in general harder.

It's because with canary releasing you can achieve different goals. The first one is truly when an error occurs. Just because less users are affected by the error. The majority of webservers is still in the old version, so just a small portion of all the users have a poor experience with the erroneous small canary version fraction.

In case of success, when everything works as expected, it is proven that there is less risk of errors, even under real production conditions.

2.2.4 continuous deployment is not continuous delivery

2.3 Operators turned into Site Reliability Engineers

Companies, which run software on their servers, usually have a specialized operations team. A team of developers programs the software and after the team is finished, the software is handed to another team, which runs the software and keeps it running. Those teams are usually referred as operations teams. So what are their duties and responsibilities exactly?

At first the already developed software needs to be deployed on an infrastructure. Therefore the operations team must provide an infrastructure. Infrastructure generally consists of hardware like racks, servers and networking devices. But in most cases, software depends on other software components and services, which are for example programming languages, dns or databases. Software usually changes. Most importantly there are security patches, but also software updates with new features are mandatory to install. Hence the operations team is responsible for installing, configuring and updating hardware and software components.

To provide a certain quality of service, infrastructure as well as the software must be maintained. That means identifying any problems and moreover fixing the identified problems. To identify problems in the first place, there must be any kind of monitoring. A good monitoring system[2] finds failures and resulting faults, recognizes performance problems, usual workloads and it detects intruders. More obvious problems can be detected early,

even before the problem leads to failures. For example a disk will soon be full or utilization of resources like cpu, ram, network or disk i/o is too high. Other failures can not be detected in advance and lead to a defective system.

In case monitoring identifies a problem, the operations team needs to take care of the problem. Operations teams need to be on call and manage incidents. A new Incident must be rated how severe it is. In advance detected problems potentially leading to failures like a disk going to be full in few days, must not be handled immediately. Other incidents like an unavailable database service can be business critical and must be fixed as rapidly as possible. But often the cause of a failure is not obvious and the system must be debugged with specific tools. Critical failures mostly affect users, so another part of incident management is to inform the affected parties.

Google states that the traditional approach of an operations team[1] is not efficient. It is expensive, it does not scale very well and it creates tension in interests[31].

A growing system means there will be more hardware, more software components and a more complex configuration. More people are needed to install, monitor and maintain the infrastructure and services. The traditional operations team approach has the disadvantage, that it does not scale very well and produces more costs, the bigger the system becomes. Another disadvantage is conflicting interests of the developer team and operation team. Operations teams are payed to create stability and provide a certain quality of service. But change is the biggest source of failure[31, p.10]. That's why operations teams work under the motto of 'Never change a running system'. On the other hand, a good service changes to the needs of its users. Developers get paid to deliver as much features as possible, so they want frequent change. Those are conflicting interests and they harm the whole product. For Google it's clear, that the different assumptions and tension in interests of operations and developer teams produce a lot of hidden costs.

2.4 Site Reliability Engineers maintain systems like software engineers

Operations teams developed a lot of practices to both integrate the fundamentally conflicting interests of change and stability, but also scale the operations team approach. This reduces obvious and hidden costs and makes the whole work more efficient. A lot of those practices became possible, because the infrastructure turned from pure hardware into a dynamic software defined infrastructure. Google refers to this approach as the site reliability engineering approach. It is operations how an software engineer would do it.

New technologies made it possible to define all infrastructure in code. One of the first who provided an dynamic infrastructure is amazon with ec2.

[1] Google actually calls it the sysadmins approach

The technology of virtualization made provisioning and configuration to software tasks instead of hardware tasks. This is what it made automate-able. Also Docker made it easier to package software and to deploy and configure the services[24].

With the possibility to automate the tasks an operations team has to do, there will come many benefits. The obvious value is that is scalabillity. When a task is automated and can be done by a machine, it is very cheap and easy to execute the task a lot more often. But there is a much bigger advance, which automation gives. When a task is automated it is a well defined process, which will consistently performed, where a human can easily make a mistake by manually executing the task. It can also be extended, measured and done at inconvenient times for humans[26].

The before one off tasks of provisioning, deploying and configuring have now well defined practices and processes. Those tasks are now reproducible because the definition or configuration of the infrastructure can now be stored in a version control system. And with automate-able deployment processes like zero downtime releases, phoenix deployment or rolling updates, those processes can be done continuously.

Operations teams have collected great knowledge on monitoring. What are the key indicators for monitoring a system. What does a good monitoring infrastructure provide. Those questions I am going to examine in detail in the upcoming sections. But very important is that monitoring can notify a human or even better trigger automated tasks, that humans do not need to be involved at all, when a problem occurs.

Also incident management can partly be automated. Distributed systems like databases have nowadays automatic failovers. Issues which have been serious, major incidents before and a humans had to manually intervene can, are now automatically triggered and the systems heal themselves. Disaster recovery is now easily done, because you can automatically reprovision the whole infrastructure as mentioned before.

With those practices it means that in the end this means that the conflicting interests of change and stability can nowadays be more and more integrated.

2.5 Post Release Testing extends the Continuous Delivery Pipeline to support maintaining a system

The three main steps in the continuous delivery pipeline are: commitment to version control, continuous integration and release. After releasing, the new version will be operated. This includes monitoring, logging, security aspects and incident management[16]. To enable developers to take more responsible in running the software, it is necessary to extend the practices of the continuous delivery pipeline.

In my masterthesis I want to optimize the software engineering practices in order to empower developers in their bigger responsibilities. I want to focus on the process of deployment and enhance the continuous delivery pipeline. To achieve this, I want to examine operation practices like monitoring and incident management. And then extend the deployment process by three consecutive steps.

The first step is to completely automate deployments of software features and achieve continuous deployment. The next step is to sensibly monitor new releases to give automatically feedback. Third and lastly, identify incidents to automatically trigger rollbacks to self-heal the software system. I'm going to call these last two steps continuous post release testing.

. . .

3 BACKGROUND

Some background is required to understand the masterthesis. In the following I will not explain all the practices and technologies. Instead I will just list them and provide well chosen references to be able to glean description of the prerequisite practices and technologies. The practices are at least well known in software engineering as well as the concepts of most of the technologies.

3.1 Prerequisite Practices and Technologies

3.1.1 Practices

- At first common practices from software engineering. Initially version control with a *version control system*. Humble explains what is important in version control [20].
- Next the reader should know about continuous integration as Fowler propagates [17] and what a *continuous integration system* is.
- I refer to *DevOps*, which Bass et. al defines [1].
- The reader should know what *continuous delivery* is and how continuous delivery relates to DevOps [34].
- Important practices from continuous delivery are prerequisite: *canary releasing*, *zero downtime deploys* and *continuous deployments*, which Humble illustrates [21].
- I require the reader to have an idea of monitoring in a distributed system as Ewaschuk describes in "Site Reliability Engineering" [16] and how Bass et. al illustrates the design of a *monitoring system* [2].
- I presume the reader to know the concepts of dynamic infrastructures and infrastructures defined as code[24]. Morris explains how to common software engineering practices for infrastructure [25].
- The concepts of cluster management and how Google manages its clusters are important [33]. Hence those concepts heavily influenced the design of *kubernetes* [5].

3.1.2 Technologies

- The next references are practical introductions to the concepts of the technologies, which I extensively use in this masterthesis. It is necessary to know what docker *image* and *containers* are [14][15].
- It helps to have an idea of the architecture of kubernetes. Beda gives a good overview [3].
- Absolutely relevant for the next chapters are the kubernetes resources. Before reading chapter 4 and 5 the reader should know what *pods* [7], *deployments* (the new replicationcontroller) [8] and *services* [9] are.

3.2 Software Dependencies

Here I would like to mention in short the dependencies, which I used to implement deployer. Deployer is implemented in ruby[28] and since it is a webserver, sinatra[29] is it's framework. Deployer uses git[18] and subversion[32] to communicate with the either git or subversion as the control version system. Moreover deployer has a plugin which sends messages to bugsnag[4] and slack[30], to inform the developer in case it identifies a problem. Deployer uses kubectl to interact with kubernetes[23] master. Deployer itself is containerized with docker[13] and its natural hosting solution would be kubernetes. We published deployer on github[11] under GPL-3.0, a free software license.

The monitoring system, which we used in our evaluation, is datadog[10], a commercial 2service. Nevertheless prometheus[27] is also a suitable open source solution as monitoring system. The continuous integration systems we use for the evaluation are codeship[6] and jenkins[22]. As an image repo, we use docker hub[12].

3.3 Typical 3 Tier Webapp in Kubernetes

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

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

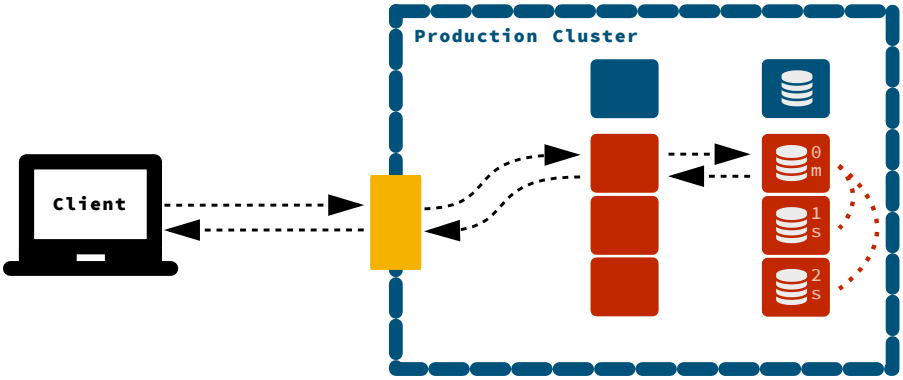

Figure 3.1: Typical 3 tier webapp in kuberentes.

. . .

4 APPROACH

4.1 In Short

In nonfunctional production regression testing we compare a new version with the current stable version side by side in production. We test in production if the nonfunctional metrics show a regression. This is nonfunctional production regression testing in two sentences. Nevertheless we will go through the term in more detail in the following. The approach is a novelty and we name the approach in the context of this thesis.

Nonfunctional refers to the metrics, which we evaluate in a test. They are nonfunctional and as a consequence generically applicable to multiple applications. The next word, production, refers to the environment, because we monitor the production application and collect the metrics from it. Finally the word regression refers to the testing strategy. The test compares the metrics of the two different versions, a stable version and a new version. We are create a canary from the new version in the production environment. We test the canary version for a regression, concretely a decline of the monitored metrics. If the test identifies a regression, we roll back to the stable version.

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

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

4.2 Pipeline Overview

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

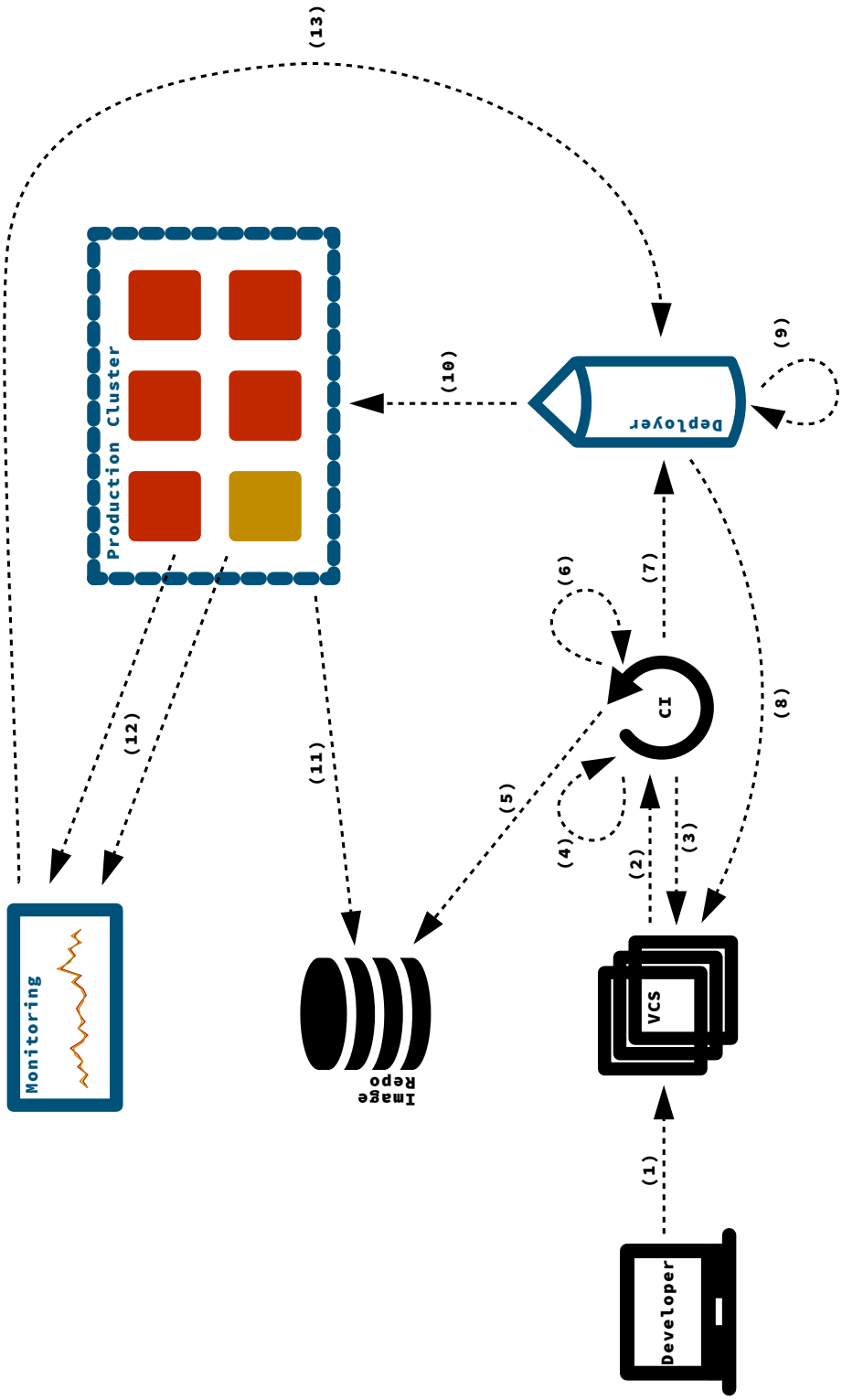

Figure 4.1: Overview of the NPRT flow.

and discuss them.

```
(1) push new version
(2) inform about new version
(3) fetch code
(4) build
(5) push container image(s)
(6) run tests
(7) deploy message
(8) fetch code
(9) modify resource definitions
(10) apply to production cluster
(11) pull container image(s)
(12) send metrics
(13) trigger full rollout or rollback
```

4.2.1 Continuous Integration is a Requirement

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

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

After the push of the new version, the second step is a message (2) to the continuous integration system. This message holds the reference of the new version and the continuous integration server pulls the new version from the version control system (3). Now the continuous integration system has three major jobs. Firstly it starts a build process (4)(5), secondly it runs the tests (6) and the thirdly gives the deploy signal (7).

In step (4), namely the build, the continuous integration system typically compiles binaries, renders assets and may create further artifacts. For our purposes it is especially necessary to build at least one or multiple container images. The continuous integration system then pushes the ready built container image to an image repository in (5). Later the image repository serves the built images.

One import thing is that we need to associate every built container image with a specific version. Therefore we use the version reference, which was created by the version control system. It is important to be able to trace the version through every step in the pipeline. With this thought in mind, the continuous integration system tags the container image with the version reference and an extra name. Just to mention that the extra name is not absolutely necessary to definitely associate the container image with a version.

But it turned out that a human readable name is very helpful to recognize a version and to know what the version is about at first sight. The continuous integration system derives that extra name from the branch name. This tag, consisting out of the version and extra name, will follow us through the whole pipeline as readable and unique reference.

The second continuous integration step is to run the tests. This is shown as step (6) in figure 4.1. The tests themselves can be split into multiple stages, such as unit, feature and smoke tests. Yet we do net need to recall all the details of automated testing at this point.

The last step of the continuous integration system is to send a deploy signal, step (7) in the figure. But it depends on the results of the tests, whether to send that deploy signal or not. The tests can be successful or fail. If the tests fail, the continuous integration system does not send a deploy message, the pipeline stops and the continuous integration system may inform the responsible developer. If the tests are successful in all test stages, the continuous integration system will send the deploy signal to the deployer.

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

However, just to clarify, the continuous integration system sends each built image for every single version to the image repository. This is independent of successful tests or the intention to go to production. The reason why we want to have every built image in the repository is to be able to test it in the continuous integration system, locally and a staging system. But this is just a side note.

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

Until this point, as we already mentioned, it is continuous integration practice, which is commonly used in software development. We require it for our approach and again, it is important to nonfunctional production regression testing to associate and trace every step through the whole pipeline. For that purpose we need to especially tag the container images.

4.2.2 Customizations of Continuous Deployment

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

The next component in the pipeline is the deployer. Deployer is a service,

which realizes the customized parts of the continuous deployment practice. In the context of this thesis, we implemented the software. We describe deployer in detail in the next chapter. This chapter demonstrates how deployer integrates in the pipeline and in the environment among all the other tools. It is crucial to have full control over the whole deployment process and as a consequence it was necessary to implement the software and have it customizable.

We could also implement the logic of the deploy deploy in the continuous integration system. But we had to decide against that, because the deploy needs full access to the production system and the continuous integration system is in our case outsourced to a third party company. We do not want to give other companies full access to another company. However this meant, that we had to implement some steps again, which a continuous integration server already implements. the continuous integration system pulls the version from the version control system as well as the deployer. (2 different things: one application code, 2 infrastructure code. But good to have them both in a single repository, to have the relation.

Deployer receives the deploy signal from the continuous integration system (7). It again includes the version reference. Now deployer executes three major steps: firstly pull infrastructure code (8), secondly modify the infrastructure code (9) and thirdly send the infrastructure code to the production cluster (10).

So in the first step deployer pulls the code from the version control system (8). The version control system also holds the files, which describe our infrastructure. We want to version control the infrastructure definitions of in order to be able to relate the version of the infrastructure to the version of the code and the version of the artifacts. In kubernetes those infrastructure definitions are made up of different resources, which were already mentioned in the background chapter.

In the second step deployer modifies those infrastructure (9) defintions in a way, that the production system uses the related container image. The running container needs to be aware of the its version. The modifications of deployer achieve that as well. The latter is important to later tag the metrics with running version.

The third step of deployer is to apply the modified infrastructure definitions to the production cluster, which is shown in the figure as step (10). We note that two things are changed: infrastructure changes as well as application code changes.

The production cluster receives the modified infrastructure definitions. The production changes the cluster state according to the definitions. Most important is that two versions in parallel are in the cluster. The production cluster fetches the container image in step (11) from the image repository. The image, which the continuous integration system built in step (4). The production cluster is aware of the specific image identified by the tag, which

we described before.

4.2.3 Metrics Collection and Comparison

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

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

We are interested in the monitoring data of specific versions. Consequently the production cluster label the metrics with the specific version. This is important, since we want to compare the metrics of the different version. The production cluster sends those labeled metrics to the monitoring system (11).

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

4.2.4 Rollouts and Rollbacks

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

The other scenario is that the new version turns out to be a regression

compared to the stable version. In that case we want to rollback the new version. Our monitoring system identifies the regression and it sends a message to deployer, as shown in step (13) in the figure. Accordingly our test for regression is failed.

Deployer receives the rollback message and sends a deploy to the production system. Just to be precise, it is actually a deploy message with the commit hash of the stable version, which the monitoring system is aware about because of the metrics it monitors. Deployer now modifies the infrastructure definitions similar to step (9). The important thing is that there must not be instances of the new version anymore. The production system itself takes care of deleting instances in the new version. Since instances of the stable version are already running in the production cluster, the production cluster does not touch the other instances.

. . .

5 IN PRACTICE

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

5.1 Deployer Architecture

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

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

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

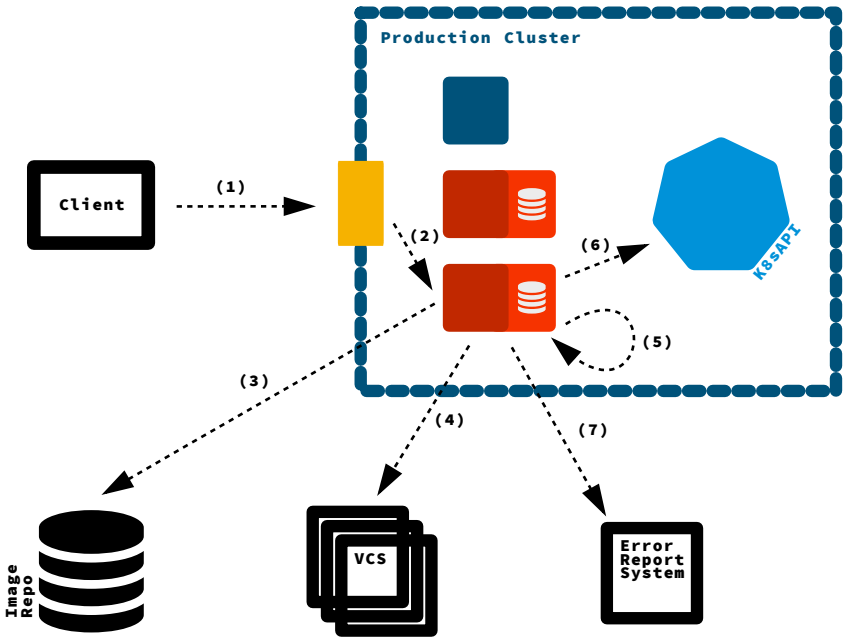

Figure 5.1: Detailed Deployer Architecture and Steps.

and waits for the deployment process to complete. Certainly we do not need a queuing system here and can scale deployer to the concurrent deploys we need.

5.2 interface

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

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

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

The *ls* or index endpoint returns the configured services and the show endpoint for a specific service shows the current configuration for that service.

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

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

The *version* endpoint simply returns the deployer version.

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

Examples:

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

depctl deploy
```

5.3 Logic and Flow of a Deploy

5.3.1 Authorization

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

5.3.2 Validations

After deployer authenticated a valid user, deployer starts the deploy procedure. At first deployer validates the arguments given by the client. The deploy request requires the service to deploy and a version of the service. The client needs to specify either the commit hash or the tag name or both. The tag would additionally include the branch name, which makes the reference more readable.

Deployer initially validates the arguments of the request. To be specific, it checks if the service exists in the configuration. Furthermore deployer checks if the commit exists in the repository and deployer checks if a the tag, which references the version, exists for all the images, which are necessary in order to deploy the service. Deployer executes the latter validation by communicating with the docker registry (3). The client does in contrast not have any validations[1]. The reason for that is that we want to have a central definition for the validations.

For simplicity and usability the client has the options to either

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

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

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

5.3.3 Fetching the Code and Caching it

After deployer validated the deploy request, it then fetches the code from the version control system (4). When there is git as the version control system, deployer uses commit hashes and branches as version reference and to determine the code version. When there is subversion, deployer uses revision numbers instead. We implemented subversion, which is more of an ancient technology, to be able to do the evaluation with the DIN system.

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

As a simplification, every pod has its own cache and lives as long as the pod. So every pod utilizes its own volume as a cache and there is no need for communication to an extra caching service. Since docker containers are immutable, every time kubernetes recreates a pod, kubernetes destroys the volume. Thus the cached data is not available anymore and in other words the cache is empty again.

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

The configuration of the kubernetes resource path enables two different methodologies in the microservice approach. The first one is the multiple repository methodology. In that methodology for every single service or microservice we would have a dedicated repository. The second methodology is the single monolithic repository, which contains multiple microservices.

5.3.4 Modifications of Resource Definitions

After having the specific version reference of the repository fetched and checked out, deployer takes the resources and modifies them in a next step (5). Especially deployments and statefulsets are relevant to those modifications. In

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

Deployer applies two main modifications: the image version and the environment. The latter is especially important for the metrics collection. Deployer modifies the environment, so that the stable version and the canary version know about themselves as such. The running service can later read the version information from environment variables. The pods receive an according label, so that other applications, which may consume the kubernetes api, can also distuinguish between the stable and the canary version.

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

The other case would be that the tag is not specified in the kubernetes resource. Then deployer appends the tag to the container image according to reference the client provided.

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

5.3.5 Errorhandling

Sometimes something unexpected is happening during the deployment. This can be for example that a verification of deployer fails, such as deployer does not find the tag corresponding to the commit. Or maybe kubernetes api server returns an error for any reason. If something like the described happens, deployer will inform the developers. We distinguish between two different clients. One is, a continuous integration system requests the deploy. In that case our error report system[2] come to play. Deployer raises an error and the error is sends it to the error report system (7), which collects all the errors of any system. The error report system informs the developers via sending notifications to a specified channel. We have for example a slack chat channel, on which the person on call subscribes and receives notifications from.

The other client is a developer sending a deploy request manually. In that case the developer typically uses depctl. Deployer discriminates usual curl requests from requests with depctl. Instead of utilizing the error report system,

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

deployer answers the http request with an errorcode and a message in the http body. As mentioned earlier the communication with deployer is synchronous.

5.4 Testing Architecture

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

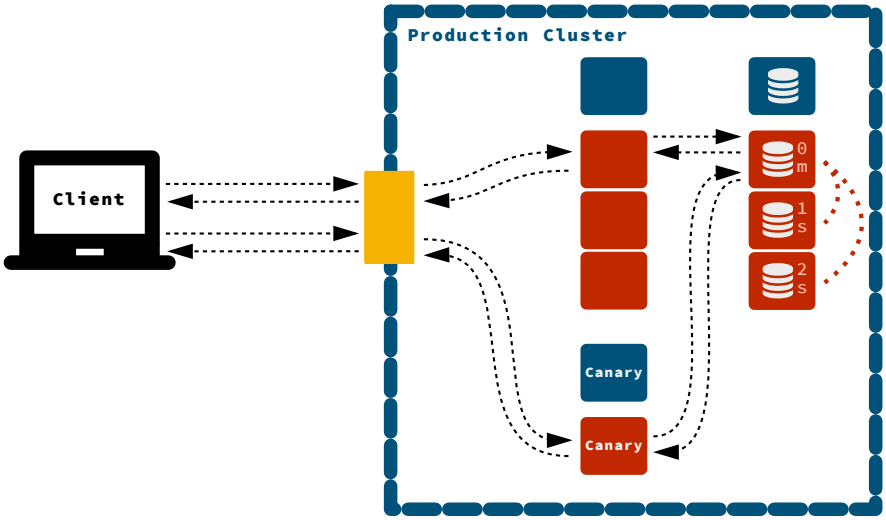

Figure 5.2: Testing Flow Implementation.

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

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

that deployment in the different version, which deployer specified.

The service though, which does our loadbalancing, selects both. It selects the pods which the original deployment created. And it selects the pods, which the canary deployment created. Hence the service selects its pods on account of the label and the original and the canary deployment share some of their labels. An example would be a shared label like 'deploy=webserver' and tow different labels like 'track=stable' respectively 'track=canary'.

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

5.5 Metrics

5.5.1 Metrics Collection

Now we want to focus on how we collect the metrics during the two versions are in production. We differentiate between two different collection mechanisms. Metrics collected from hosts and metrics from applications.

Firstly the group of metrics, which are picked up from the host. The host has information about the pods' utilization of cpu and memory. So on each kubernetes host, there is a monitoring agent running, which watches the `/proc` directory and the docker daemon. The agent picks up the information frequently and then sends it to our monitoring system.

Secondly the group of metrics, which the application sends. We need to have the application instrumented in order to collect metrics like throughput, latency and errorrate. In practice we use the statsd protocol and statsd server for that purpose. There are statsd libraries for the most languages and frameworks and you get the instrumentation without much effort. The instrumented application sends the data to the statsd server after each request and the statsd server aggregates the data. From there the statsd server forwards the aggregated metrics data to our monitoring system.

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

5.5.2 Metrics Comparison

Another part of nonfunctional production regression testing is the comparison within the monitoring system.

The monitoring system consists basically out of a timeseriesdatabase, a graphing user interface and an alarm system. The timeseriesdatabase persists the metrics. And the user can define graphs from those metrics, which the user interface displays. Moreover you can define rules in the alarm system, which monitor the metrics in the timeseriesdatabase. In case the metrics violate any rule, the monitoring system sends a notification.

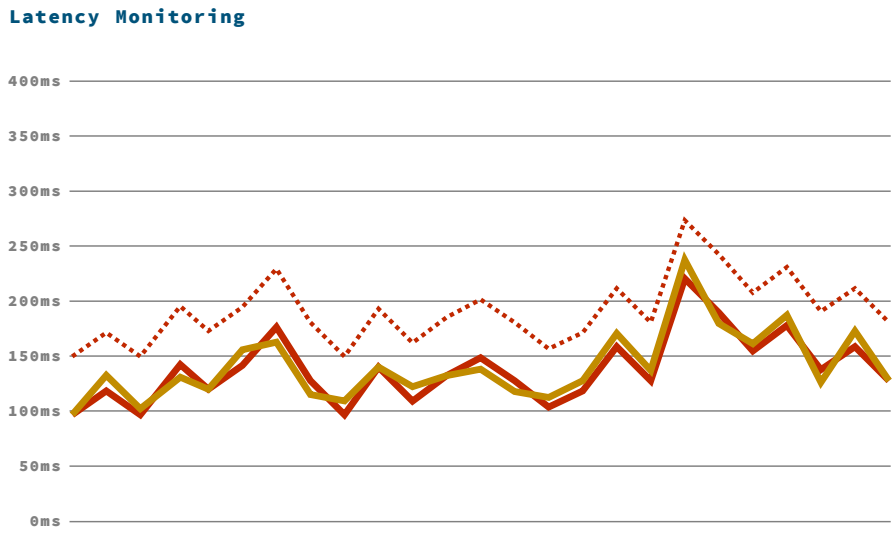

Figure 5.3: Comparing the stable version with the canary version.

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

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

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

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

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

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

5.5.3 Metric Parameters for Latency and Utilization Metrics

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

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

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

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

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

. . .

6 EVALUATION

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

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

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

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

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

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

version.

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

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

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

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

Especially to test the latter, security updates, is absolutely appealing, since we can fully automate the procedure of updating the dependencies of our application in a fully automated and in a way, which would have a very low risk. We could have a job, which checks frequently for any new version, pushes the updates to the version control system, the continous integration system runs the pre deploy tests, deployer deploys the update and even in production we check the update for an regression. We could save a lot of developer time, who would usually need to take care of the whole updating procedure.

And even if there is a degradation in production, a small amount of requests is affected, because we send only a reasonable amount of traffic, which arrives at the same time, to the potentially degraded version. Further more we limit the time the degraded version is in production, because we automate detection of the degradation and the rollback to the old stable version.

We let this running for a specified time in production. We need to decide on how long we want to compare the versions. That depends on how much traffic is in production, because when we would few traffic in production, we wanted to compare for a longer time. We suggest to have a well balanced test scenario in terms of load. The time depends on how much traffic there is in production and how often a team wants to deploy its application. A team which is working with a monolithic application has the disadvantage, that every change in every part of the software causes a deploy and deploys are more frequent. This limits the time in production for the canary. Instead if we have a microservice environment, the deploy affects only a specific service, hence little part of the whole application. As a result deploys are less frequent and we have more time for the canary in production.

We do not need to generate the test traffic, we do not need to weight traffic and we do not need to think about edge cases. These are all advantages, that we get for free from the production traffic. We save time and work, because the users generate the test data, instead of us.

The users create more requests and with that test data for parts of the application, which are more important. Consequently the users reasonably weight the test data. And lastly the longer we run the comparison in production, users will produce more of those edge cases, which would be hard to make up.

We are aware of that the two compared versions do not receive the very same requests. Hence the comparison is not perfect. In future work we could extend the technique to achieve that.

We could simply clone the requests, send the original request to the stable version and send a cloned request to the canary. The loadbalancer could then differentiate between the two responses of the two versions. We would reject the response of the canary. And we would forward the response of the stable version.

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

. . .

7 CONCLUSION

7.1 Resume

We illustrated that we are able to automate the whole testing procedure, which is the advantage of nonfunctional production regression testing. We can extend the continuous delivery pipeline in a natural way and support developers not only until the deploy, furthermore we automated a part of the developer's job during run time. The pipeline is now advanced in a way, that we can change application code and test it without the supervision of the developers at a very low risk. The change goes through the whole pipeline including tests in a testing environment, an automated deploy and tests in the production environment. If the nonfunctional production regression test and automated deploy is successful, developers will be completely free of work. Potential changes of bots would be robust enough to be able to act fully automated and unsupervised. However we still test the changes they make and can be sure to not have a regression in our production system.

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

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

Another nice thing to mention about deployer, is that it deploys itself,

which fits to the declarative model and recursion. That means we develop deployer itself with the continuous deployment flow and can apply nonfunctional production regression testing to deployer.

7.2 Outlook and future work

. . .

Bibliography

Background

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

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

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

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

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

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

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

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

[16] 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.

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

[19] 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.

[20] 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.

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

[24] Kief Morris. “Infrastructure as code : managing servers in the cloud”. In: O’Reilly, 2016. Chap. Dynamic Infrastructure Platforms, TODO.

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

[26] 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.

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

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

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

Software

[4] *Bugsnag*. `https://www.bugsnag.com/`. last visited 19.11.2017.

[6] *Codeship*. `https://codeship.com/`. last visited 19.11.2017.

[10] *DataDog*. `https://www.datadoghq.com/`. last visited 19.11.2017.

[11] *Deployer*. `https://github.com/gapfish/deployer`. last visited 19.11.2017.

[12] *Docker Hub*. `https://hub.docker.com/`. last visited 19.11.2017.

[13] *Docker*. `https://www.docker.com/`. last visited 14.11.2017.

[18] *Git*. `https://git-scm.com/`. last visited 19.11.2017.

[22] *Jenkins*. `https://codeship.com/`. last visited 19.11.2017.

[23] *Kubernetes*. `https://kubernetes.io/`. last visited 14.11.2017.

[27] *Prometheus*. https://prometheus.io/. last visited 19.11.2017.

[28] *Ruby*. https://www.ruby-lang.org/en/. last visited 14.11.2017.

[29] *Sinatra*. http://sinatrarb.com/. last visited 19.11.2017.

[30] *Slack*. https://slack.com/. last visited 19.11.2017.

[32] *Subversion*. https://subversion.apache.org/. last visited 19.11.2017.