BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

TESTING AND RELEASE INFRASTRUCTURE IN THE CONTAINER WORLD

INFRASTRUKTURA PRO TESTOVÁNÍ A NASAZOVÁNÍ V OBLASTI KONTEJNERŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. ADAM ORMANDY**
AUTOR PRÁCE

SUPERVISOR **Ing. LENKA TUROŇOVÁ**
VEDOUCÍ PRÁCE

BRNO 2018

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2018/2019

Master's Thesis Specification

Student: **Ormandy Adam, Bc.**
Programme: Information Technology Field of study: Information Technology Security
Title: **Infrastructure for Testing and Deployment in the Field of Containers**
Category: Algorithms and Data Structures
Assignment:

1. Study the theoretical background behind the container technology and OpenShift Container Platform. Study principles of continuous integration (CI) and continuous deployment (CD).
2. Propose a method for automatically checking best practices during the software development and testing.
3. Implement the proposed method in a prototype tool.
4. Compare effectiveness of using the implemented tool against manual checks of the best practices.
5. Discuss the obtained results and future work.
6. Write the thesis in English.

Recommended literature:

- DUVALL, Paul M, Steve MATYAS a Andrew GLOVER. *Continuous integration: improving software quality and reducing risk*. Upper Saddle River, NJ: Addison-Wesley, c2007. ISBN 978-0321336385.
- HUMBLE, Jez a David FARLEY. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 978-0321601919.
- *Docker Documentation* [online]. San Francisco: Docker [cit. 2018-10-05]. Dostupné z: https://docs.docker.com/
- *OpenShift Container Platform 3.9 Documentation* [online]. Raleigh (North Carolina): Red Hat [cit. 2018-10-05]. Dostupné z: https://docs.openshift.com/container-platform/3.9/welcome/index.html

Requirements for the semestral defence:

- Items 1 and 2.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor: **Turoňová Lenka, Ing.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2018
Submission deadline: May 22, 2019
Approval date: November 1, 2018

Abstract

Efficiency loss caused by repetitive manual tasks is a common problem throughout the IT sector. Developers often test, build, and deploy their software manually. That is not only time-consuming, but also dull and prone to errors and mistakes. This thesis tries to solve that in the context of one DevOps team, by unifying the development and testing tools, and by applying the principles of continuous integration and continuous deployment in the production environment. It is focused on Python, Jenkins, and container-based software and workflows. The main tools used in the thesis are GitLab CI, OpenShift and Tox. Thanks to work in described in the thesis, the number of projects with CI/CD pipelines increased from 7 to 50 percent, the amount of Python style violations started to decrease, containers have proper metadata, the container build process is automated, time and effort are saved by not doing repetitive tasks, and more.

Abstrakt

Znížená efektivita spôsobená robením repetitívnych a manuálnych prác je častým problémom v IT. Vývojári často testujú a nasadzujú svoj software manuálne, čo je nielen náročné na čas, nezáživné a náchylné k chybám. Táto práca sa snaží v rámci jedného DevOps tímu, vyriešiť tento problém pomocou zjednotenia vývojárskych a testovacích nástrojov, a pomocou aplikovania pricípov CI a CD do produkčného prostredia. Zároveň sa sústredí na software využívajúci Python, Jenkins a kontajnery. Hlavnými použitými nástrojmi sú GitLab CI, OpenShift a Tox. Vďaka tejto práci sa podarilo zvýšit počet projektov, ktoré používajú CI/CD zo 7 na 50 percent, zvrátiť rastúci trend v počte porušení štýlu v jazyku Python, opatriť kontajnery metadátami, zautomatizovať proces tvorby kontajnerov, ušetriť čas nerobením repetitívnych úloh a pod.

Keywords

testing, deployment, integration, container, containers, continuous integration, continuous deployment

Klíčová slova

testovanie, nasadzovanie, kontajner, kontainery, continuous integration, continuous deployment, priebežné integrovanie, priebežné nasadenie

Reference

ORMANDY, Adam. *Testing and Release Infrastructure in the Container World.* Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lenka Turoňová,

Rozšířený abstrakt

Strata efektivity kvôli repetitívnym manuálnym úlohám je bežným úkazom v IT. Vývojári a inžinieri často testujú, kompilujú a nasadzujú svoj software manuálne. To je nielen časovo náročné, ale aj nudné a náchylné na chyby. Manuálne postupy zároveň umožňujú a podporujú používanie zbytočne komplikovaných postupov a nechcenú koncentráciu vedomostí nutných pre vývoj, testovanie a nasadzovanie software.

Práca sa snaží riešiť tento problém v rámci jedného DevOps tímu vo firme Red Hat. Problém sa snaží vyriešiť pomocou unifikácie nástrojov a postupov použitých pri testovaní a vývoju kódu a pomocou princípov kontinuálnej integrácie (continouos integration - CI) a kontinuálneho nasadenia (continouos deployment - CD).

Kontinuálna integrácia je prístup v softwarovom inžinierstve a vývoji, kedy členovia vývojového tímu často integrujú svoje zmeny medzi sebou, minimálne raz za deň. V rámci integrácie je software automaticky preložený a otestovaný. Častá integrácia poskytuje vedomosť, že vyvíjaný software je integrovateľný a zabezpečuje že chyby sú odhalené čo najskôr.

Kontinuálne nasadenie je pokračovanie kontinuálnej integrácie. V rámci kontinuálneho nasadenia je kód nielen automaticky otestovaný a preložený, ale aj nasadený do produkčného prostredia. Tento prístup vyžaduje nielen možnosť automaticky nasadiť kód, ale aj existenciu automatických akceptačné testov a automatickú kontrolu novo nasadeného kódu. Dôležitý je aj princíp navrátenia (tzv. rollback), čo je nasadenie predchádzajúcej, pravdepodobne fungujúcej verzie software do produkčného prostredia. Navrátenie je dôležité v prípadoch, ak sa chyby zistia až po nasadení do prostredia.

Kontajnery sú forma virtualizácie na úrovni operačného systému, ktorá používa linuxové jadro na oddelenie procesov tak aby bežali nezávisle na sebe. Týmto procesom sa hovorí kontajnery. Kontajnery sú v mnohom podobné bežným virtuálnym strojom. Hlavným rozdielom je to, že všetky kontajnery zdielajú linuxové jadro hostiteľského systému, zatiaľ čo virtuálne stroje majú vlastný OS a hypervízora. To spôsobuje, že kontajnery efektívnejšie využívajú zdroje hostiteľského systému, ale zároveň sú menej bezpečné. Ďalším veľkým rozdielom je, že ak hlavný proces v kontajnery skončí, celý kontajner je vymazaný, a dáta a stav kontajneru sú stratené. Z toho vyplíva, že kontajner nemôže byť vypnutý a potom zapnutý tak, aby pokračoval v práci.

V rámci prvotnej analýzy som zisťoval, ako efektívny bol manuálny postup v prevencií technického dlhu a vynucovaní tímových politík, štandardov a všeobecne uznávaných dobrých praktík. Analýza ukázala že manuálne postupy boli neefektívne a časovo náročne. Napríklad, počet porušení štýlu jazyka Python mal stúpajúci trend a už prítomné chyby zostávali neopravené. Kontajnery nemali dostatočné metadáta, napríklad nebolo možné zistiť z čoho a kedy boli kontajnery vytvorené. Kontajnery boli vytvárané na strojoch vývojárov, a často sa stávalo že tajomstvá a iné veci sa omylom skopírovali do kontajnerov. Jenkins joby neboli konfigurované automaticky ale ručne, čo spôsobovalo, že najnovšia a nasadená konfigurácia nesedeli. Chýb a porušení politík bolo mnoho, a väčšina bola spôsobená nepozornosťou, časovou náročnosťou úkonu alebo jednoducho preto, lebo niekto na to zabudol. To znamená, že automatizácia týchto činností má veľký potenciál zlepšiť situáciu.

Po analýze existujúcich problémov a zúžení rozsahu práce na software, ktorý používa Python, kontajnery a Jenkins (väčšina software vlastnená a vyvíjaná tímom používa tieto technológie) sme sa zamerali na návrh CI/CD pipelina. Rozhodli sme sa použiť tradičnú kostru CI/CD, teda vytvor spustiteľné artefakty, integruj a otestuj, nasaď. Kvôli tomu, že náš kód nepoužíva kompilované jazyky a niektoré testy a vlastnosti kódu (napr. štýl) je možné overiť ešte pred vytvorením spustiteľných artefaktov, tak som pred fázu vytvorenia

artefaktov pridal fázu otestovania takýchto vlastností. Ak pipelina zhavaruje na týchto testoch, ušetríme okolo 15 minút a zdroje ktoré by sme strávili vytváraním spustiteľných artefaktov, vo väčšine prípadov vytváraním kontajneru.

Potom sme museli nájsť CI/CD framework, ktorý bude splňovať naše požiadavky. Požiadavky boli: nízka alebo žiadna údržba, jednoduché používanie, a možnosť pracovať v internej sieti. Nakoniec sme sa rozhodli použiť GitLab CI. GitLab je open source platforma určená na vyvíjanie software. Má v sebe zakomponovaný verzovací systém, systém sledovania problémov, úložisko kontajnerov a CI/CD systém nazývaný GitLab CI. GitLab je už interne nasadený a stará sa oňho dedikovaný tím, to znamená že nie je nutné ho vlastnoručne nasadzovať a ani sa oňho starať. Diskutovaná bola aj možnosť používať Jenkins, ale pre zlé skúsenosti nebola táto možnosť vybratá.

Potom som sa vrhol na unifikáciu spôsobu, akým sú spúštané testy, statické analyzátory a kontroly štýlu. Po diskusiách sme sa dohodli na nástroji Tox. Tox je nástroj, ktorý vytvára rozhranie, skrze ktoré sa spúšťajú testovacie a vývojárske nástroje, vďaka čomu je používanie týchto nástrojov jednoduchšie a znižuje nutné znalosti na používanie týchto nástrojov. Používanie Toxu na spúšťanie nástrojov má aj tú výhodu, že ich spúšťa v oddelených Pythonovských virtuálnych prostrediach. To má niekoľko výhod, napr. nástroje sa navzájom neovplyvňujú, a je možné testovať mnoho verzií Python naraz. Tox je pre tieto a ďalšie výhody považovaný za best-practise Python komunitou. Pre tento nástroj som vytvoril vzorovú konfiguráciu tak, aby spĺňala tímové štandardy, bola použitelná takmer pre všetky naše projekty a zároveň ľahko upraviteľná pre špecifické použitia.

Potom som sa vrhol na automatizáciu vytvárania a manažmentu kontajnerov. Bolo nutné vytvoriť integráciu s OpenShiftom, čo je cloudová platforma postavená na kontajneroch. Vývoj integrácie prebiehal vo viacerých fázach, ale vďaka tomu je nová verzia kontajnerov vytvorená automaticky. To šetrí čas vývojarov, lebo to nemusia robiť ručne, vytvára to tlak na to, aby vytváranie kontajneru bolo opakovateľné a zdukumentované, metadáta sú prítomné pri každom kontajneri, politiky sú automaticky vynucované a tak ďalej.

Ďalej som sa venoval automatickej konfigurácií Jenkins jobov, keďže väčšina našich projektov sú Jenkins joby. Jenkins job definuje úlohu vykonávanú v open source automatizačnom serveri Jenkins. Často sa stávalo, že nasadená konfigurácia a konfigurácia vo verzovacom systéme boli odlišné. To sa nám podarilo odstrániť vďaka kontinuálnemu nasadeniu zmien. CI/CD pipelina nahraje najnovšiu produkčnú verziu do produkcie vždy po zmene produkčnej verzie alebo aspoň raz za týždeň, aj keď sa nič nezmenilo.

Všetky poznatky, príklady a inštrukcie ako nastaviť a začať používať CI/CD pipeliny, Tox a ďalšie som zhrnul do ukážkového projektu. Tento projekt pomáha vývojárom ktorý pridávajú zmienené nástroje do svojich projektov a zároveň vytvára miesto, kde sa budú koncentrovať opravy a nové verzie.

Vďaka práci popísanej v tejto diplomovej práci sa podarilo zvýšiť počet projektov, ktoré používajú Python, Jenkins alebo kontajnery a zároveň utilizujú CI/CD zo 7 na 50 percent. Zároveň sa podarilo zvýšiť úroveň používanie Toxu zo 4 na 40 percent projektov v Pythone. Rastúci trend v počte štylistických chýb v Pythone sa podarilo obrátiť a tento počet teraz klesá. Kontajnery sú vytvárané automaticky a politiky týkajúce sa Pythonu, Jenkinsu a kontajnerov sú vynucované automaticky.

Práca na veciach popísaných v tejto práci sa neskončila odovzdaním práce. Počas leta 2019 sa plánuje ďalšia fáza vývoja, ktorá by mala zlepšiť už existujúce časti a zároveň pridať podporu pre viac technológií, napr. Ansible. Taktiež je záujem zverejniť časti ako open source. Kontajner pre nástroj Tox už je zverejnený a dostupný pre verejnosť, ale chceli by

sme zverejniť ešte viac, napr. kontajner pre OpenShift CLI a našu integráciu GitLab CI a OpenShiftu.

Testing and Release Infrastructure in the Container World

Declaration

Hereby I declare that this masters's thesis was prepared as an original author's work under the supervision of Ing. Lenka Turoňová. The supplementary information was provided by Mr. Stanislav Ochotnicky. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Adam Ormandy
May 20, 2019

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Ing. Lenka Turoňová for the support, guidance, and especially patience.

I would like to thank my technical advisor Mr. Stanislav Ochotnicky. Without his ideas and guidance, this thesis would not be possible. Thank you for your time and patience with me.

Big thank you also goes to my colleagues that helped me transform my ideas to the reality. Thank you for the interesting discussions and all the help you gave me.

Last but not least, my mom and dad, my brothers, my friends, my girlfriend, and everyone else who helped me to become who I am.

Contents

Chapter 1

Introduction

In today's increasingly competitive and demanding world, when systems are more complex and error-prone, effectivity is the key. Security issues are found and exploited quicker and their cost, both monetary and reputational, is increasing. Bugs are becoming more severe and harder to debug. And expectations of customers are getting higher.

That puts a lot of pressure on the developers and operations teams, which have to perform better and quicker. This poses an issue, how to handle an increased workload. One of the solutions is to hire more engineers, but with the current shortage of IT employees in the US and Europe, this is becoming increasingly expensive and less viable solution.

Another way how to tackle this problem is automation. A lot of tasks done by IT engineers are dull, repetitive and error-prone if done manually. So letting expensive and highly educated employees to do these tasks is waste of resources, especially if the machines can do better, faster and more consistently.

This thesis documents the journey of Red Hat's DevOps team from manual to the automatic testing and deployments. The thesis will describe the original state, problems encountered when moving to the automated pipelines and how these changes influence thd quality of the codebase and overall effectivity of the team.

The thesis is divided into 12 main parts. The first 8 give theoretical background and explanation to concepts and technologies used in the thesis. 9th chapter provides assesment of the situation before the implementation of the solution proposed by the thesis. 10th chapter describes the process of desing, decision-making and implementation of the proposed solution. In the 11th chapter, I compare the states before and after solution. And the last chapter is conclusion with brief summarization of the achieved goals and ideas for future improvment and development of the solution implemented by this thesis.

Chapter 2

Software Engineering

Software engineering is a discipline that is concerned with the application of engineering principles to all aspects of software production. That usually includes the process of analysis, design, implementation, documentation, testing, and maintenance of a software solutions [53]. Software engineering is not just concerned with the technical aspect of software development, but also with non-technical aspects, e.g., project management and best practices [50].

2.1 Best Practices

A best practice is a method or technique that has been generally accepted as superior to any alternatives because it produces results that are superior to those achieved by other means or because it has become a standard way of doing things, e.g., a standard way of complying with legal or ethical requirements [59]. There are best practices for almost every part of software development. Some are concerned with documentation, some with architecture and some with the way how the code is developed.

Interestingly, because some best-practices are more a matter of opinion and agreement than practises that create technically superior result, communities are often divided and a topic of best practices can spur a heated debate, for example usage of the tabs and spaces for indentation or naming convention. This usually results in several practices living next to each other.

The opposite of the best practice is a bad practice or anti-pattern. An anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive [11, 4]. Notable anti-patterns are Singleton [64] and ninety-ninety rule [60].

2.2 Technical Debt

Technical debt (also known as design debt [51] or code debt) is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer [55, 66].

Technical debt is a metaphor that equates software development to financial debt [55]. Technical debt is not bad on itself. To give an example, a developer has to choose between a quick solution (workaround) or the proper solution that takes more time. If he picks the quick solution, he pushes the costs of making it right into the future, thus creating debt.

If he repays the debt in a short time, there will not be much of problem. But similarly to the financial debt, technical debt compounds interrests [48], e.g., workaround leads to bad design decision, thus creating more debt.

Article Pragmatic Technical Debt Management provides following inside into technical debt [48]:

> In the case of non-payment of the accrued technical debt for a long time, the software increasingly becomes harder to change and in the extreme case, the software product becomes technically bankrupt (i.e., it is not feasible to introduce a change reliably in the prescribed time). Such a situation often leads to project closure. Even if the project does not become technically bankrupt, thanks to the technical debt the development of new feature will take longer but will take longer to fix, code reviews will take longer and much more.

There are multiple dimensions of technical debt, to name a few [48]:

- implementation debt: Code duplication, static tool rules violations, and code smells.
- design and architecture debt: Design smells, design rules violations, and architectural rules violations.
- test debt: Lack of tests, inadequate test coverage, and improper test design.
- documentation debt: No documentation for important concerns, poor documentation, outdated documentation.

Common causes of the technical debt are [58]:

- business pressure (we have to release first),
- lack of standards and guidelines,
- desing complexities,
- lack of foresight, and
- lack of knowledge (Junior developers).

In a real-life development, a certain amount of technical debt is inevitable. But that does not mean that precautions to mitigate it. To avoid technical debt you have to spread awareness about it, set clean and understandable coding standards, enforce those standards, give enough time to developers and have a dialog between various organization parts (i.e., business, developers, management).

To repay a technical debt already in place, you should [48]:

- Identify, document, and track the debt.
- Not all debt is created equal. Prioritize.
- Amortize small debt payment into each iteration.
- Motivate and reward people for maintaining quality.
- Look out for possible large-scale debt repayments.

- Repay debt horizontally and not vertically (e.g., change in the code could make testing and documentation easier, thus making the repayment of the testing debt much easier and effective).
- Do not repay the debt in certain cases. Sometimes it is more efficient to create a new product with better technology and better standards then trying to fix the current solution.

2.3 Software Testing

Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not [57]. Testing is executing a system to identify any gaps, errors, or missing requirements is contrary to the actual requirements [57].

The testing involves the execution of the software components to evaluate one or more properties of interest. These properties could be [65]:

- The program meets the requirements that guided its design and development.
- The program responds correctly to all kinds of inputs.
- The program performs its functions within an acceptable time.
- The program is sufficiently usable.
- The program can be installed and run in its intended environments.
- The program achieves the general result its stakeholders desire.

Developers should start testing because the sooner the bugs are discovered, the cheaper is the fix them [30]. That does not mean that the software will be bug-free because even for the simplest of programs have an infinite amount of the inputs.

2.3.1 Categories and Types of the Software Testing

The testing in software development could be divided into two categories:

1. functional testing, and
2. non-function testing.

Functional testing is primarily is used to verify that a piece of software is providing the same output as required by the end-user or business [54]. It is mainly used to check if the software does what is required from it by the end user. The functional testing includes unit-testing, integration testing, user acceptance testing and more [57].

Non-functional testing tests the non-functional properties of the software, e.g., security, performance, portability, and scallability [23].

Testing can also be put into categories based on the amount of the information the engineers have about the internal workings of the software. The categories are [57]:

1. black box testing (no knowledge),
2. white box testing (complete knowledge), and
3. gray box testing(limited knowledge).

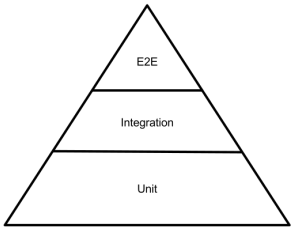

Figure 2.1: The testing pyramid is a concept in software testing describing the desirable relationship in quantity between End to End tests (E2E), Integration tests, and Unit tests [1].

2.3.2 Unit Testing

The unit-testing is a form of white box testing usually done in the first round of testing. During the unit-testing, the program is submitted to assessments that focus on specific units or components of the software to determine whether each one is fully functional [32].

Unit-testing tests each of the components (units) individually and in isolation from each other. Thanks to that, unit-testing could be done in the early parts of the development, increasing the change that the bugs will be discovered early. This also means that the unit-testing is usually done by the developers before the software is delivered to the testers for formal testing [32].

The definition of what the unit is shifts based on the source, but the unit usually refers to a function, individual program or even a procedure [32].

2.3.3 Integration Testing

Integration testing is the process of testing the units of the programs together as a group. It is designed to find defects in the interface between modules and units and to determine how efficiently the units run together [32]. It should be done after the initial unit-testing because even if units work as they should independently, the bugs in the integration of these parts could make the software useless.

2.3.4 Code Coverage

Code coverage (i.e., test coverage) is a term or a metric used in software testing to describe how much a testing plan covers program source code [52].

There are a number of coverage criteria, the main ones being [31]:

- function coverage – Has each function (or subroutine) in the program been called?
- statement coverage – Has each statement in the program been executed?
- edge coverage – has every edge in the Control flow graph been executed?

- condition coverage (or predicate coverage) – Has each Boolean sub-expression evaluated both to true and false?

Chapter 3

Continuous Integration

Continuous Integration (CI) is a software development practice where the team members integrate their work frequently, usually at least daily, thus resulting in multiple integrations per day. Each integration is verified by an automated build and tests to detect integration errors as quickly as possible.

The CI helps to identify and mitigate risks that are present in software development, such as the late discovery of defects, low visibility of the project (test coverage, code health, etc.), low-quality software, and inability to create deployable software.

The CI also provides a sort of quality net for developers, when they are developing, refactoring or maintaining the software. Especially when under time pressure, teams tend to abandon or forget about the standards and best practices, thus introducing technical debt into the code, and that is when the automated standard enforcer comes handy. It is important to emphasize that the quality of the CI analysis is only as good as good is the test suite.

The CI can provide substantial cost and time savings thanks to the automation of the error-prone, repetitive activities. The automation of these tasks than frees up the valuable time for more complex tasks, thus making the development faster.

The CI also has its costs. The first is time and effort that has to be put into the development of the CI and automated tests. The second one is the cost associated with the running of the CI servers and builds (electricity, computing power, etc.). But these costs are exceeded by the benefits the CI brings [19].

The trigger that starts the CI run is usually a developer who pushed his or her changes to the repository. But the run can also be triggered by the change of the dependencies, e.g., base container image has been updated. Or it can be triggered by the scheduled event, e.g., daily rebuilds started by the cron job.

3.1 Good Practices

To make the CI as effective as possible, the engineers should adhere to these practices [19]:

- Perform single command builds.
- Use automated builds.
- Automate testing and introspections.
- Separate build scripts from the IDE.

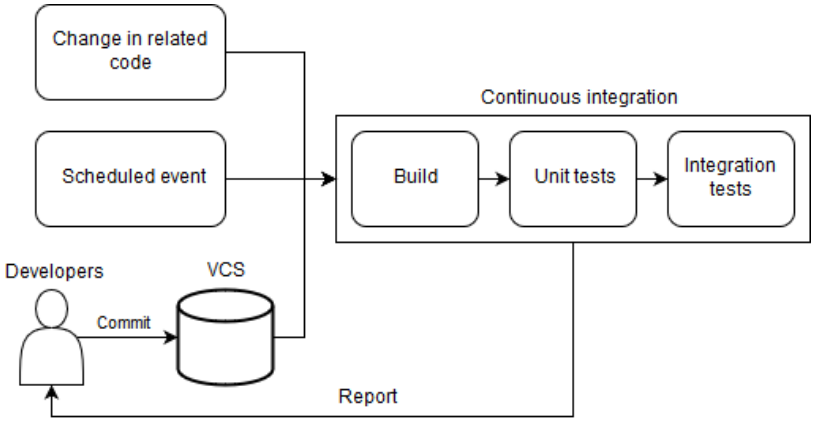

Figure 3.1: Diagram shows the phases of the CI run/build. The build can be triggered for example by the developer pushing his or her changed to the repository, or by a scheduled event (daily rebuild using cron job). The CI server fetched the changes from the SCM. Then the artifacts (binaries, container images, etc.) are built. After the build phase is a test phase. In the test phase the style checks, unit-tests, and integration tests are run. After the run/build is finished, the developers will receive feedback, i.e., if the run succeeded or failed and where it failed. The notification can be done using email, IRC or Slack bot.

- Create a consistent directory structure.
- Fail builds fast.
- Build for any environment.
- Use a dedicated integration build machine.
- Use a CI server.
- Run fast builds.
- Stage builds.

The CI is based on the idea that the integration of the software should be a non-event. Ideally, by pressing a single button or by running a single command [19], the software will be built, tested, inspected and deployed.

To achieve the one command integration the automated build and tests are required. The automation of these parts of the development saves not only time and effort but also severely decreases bus factor and helps to avoid false negatives that are caused by the mistakes in the integration, e.g., the developer forgot to download the right dependencies, so the build failed.

To gain the most significant benefit from the CI, the developers should build on the fresh machine dedicated to the builds and use CI server. The use of dedicated build machines should decrease the chance that the previous builds influence the current build. For example, the dependency was removed from the dependency list by mistake, but because the build is done in the environment where other builds were done, and the dependency is present thanks of that, the build will succeed, even if it should not.

The integration scripts have to be separated from the IDE to make the use of the dedicated CI servers possible. Also, the separation also removed the unnecessary dependency

that is IDE and makes it easier for other developers to pick a project and use the IDE they want.

One of the most important aspects that should be considered is that the CI should be as fast as possible because the sooner the feedback is received, the sooner the developers could start work on the fix. This also means that the build that will fail should fail as quickly as possible, i.e., the parts of the build that tend to fail should be run first, for example, the style checks.

To further decrease the time until the feedback results, the CI should use staged builds. That means that the CI should run the lightweight „commit“ builds that perform compile, unit-test execution and deployment followed by heavyweight „secondary“ builds that include component, system, and other slower-running tests and inspections [19].

3.2 Continuous Delivery and Deployment

The primary purpose of software development is to create working software. Without a successful deployment, it does not even really exists [19]. To avoid having undeployable software and „nightmare releases“ we can expand the principles of CI to the process of deployment. Continuous deployment is a culmination of practices and steps that enable to release working software any time, any place, with as little effort as possible [19].

Regardless of the platform of a platform, technology, or domain, deploying working software principally embodies six-high level steps [19].

- Label a repository's assets
- Produce a clean environment, free of assumptions.
- Generate and label a build directly from the repository and install it on the target machine.
- Successfully run tests at all levels in a clone of the production environment.
- Create build feedback reports
- If necessary, roll back the release by using labels in the version control repository.

To create a clean environment, one should remove all files, configuration changes, servers, and anything else from integration build machine and ensure that he can rebuild back to a state where the integration build is successful [19].

Proper CD always can roll back the release, in case that something goes wrong. That is closely related to asset labels because, without them, the identification of the state the software should rollback to would be difficult if not impossible.

If for some reason, automatic deployment is not possible or desired, one can add an intermediate step of manual approval between deployment to stage and deployment to production. This setup is called continuous delivery.

3.3 Continuous Integration and Developers

To achieve the greatest benefit from the use of the CI, the developers have to follow these practices and rules [19]:

- Developer has to commit frequently.

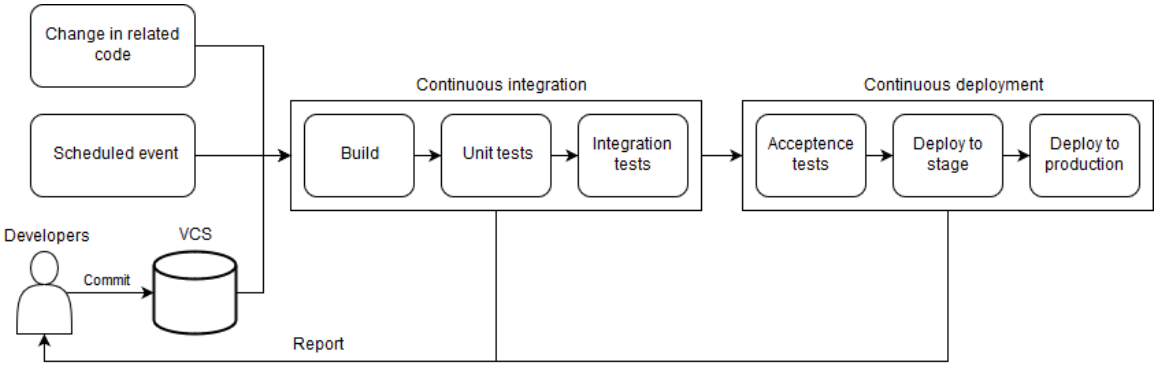

Figure 3.2: Diagram shows the process of continuous deployment. After the process of CI is finished, the acceptance testing begins. After the acceptance testing is successfully finished, software is deployed to a staging environment, and if nothing bad happens, software is deployed to production. Reports should be created from the whole process to ease troubleshooting or to make people aware of deployment taking place.

- The committed code cannot be broken.
- Fixing of the broken has the highest priority. The developer has to fix the broken code immediately.
- Developers have to create automated tests.
- All the tests and inspections have to pass.
- Developers should run private (test) builds.
- Avoid broken code.

By committing small incremental changes frequently, the CI can detect defect sooner, and the developer has an easier time finding discovered bugs. Small commits also decrease the chance of a situation, when multiple unrelated issues are uncovered when testing, which makes it hard to pinpoint the exact causes of the problems.

Chapter 4

Git

Version control systems are a category of software tools that help a software team manage changes to source code over time(VCSs) [5], creating a timeline of changes in a repository. One can later navigate through this timeline of changes, recalling specific versions of the files, making possible to reverse selected files or an entire project to a previous state, compare changes, found out who and when made the changes and when something that is causing issues was introduced. These possibilities make VCSs an essential part of the modern software team development practices.

Git is a free and open source distributed version control system, designed to handle everything from small to huge projects with speed and efficiency [15]. Git was designed by Linus Torvalds in the 2005 and has since become defacto standard of VCS [5].

As was mentioned above, Git is a distributed version control system (DVCSs). In a DVCSs, a client creates a clone of the repository, including its full history. All clients share their changes using a central remote server or servers, which makes possible to collaborate with multiple groups of people with one project [13].

4.1 Basic Git Operations

Git does not store data as a series of changesets or differences but instead as a series of snapshots [13], called commits. Commit contains a pointer to the repository snapshot, pointers to the parent commits and various metadata, like authors name, commit creation date and other.

Commits are organized into branches. Branches allow to diverge from the main line of development and continue to do work without messing with the mainline [13]. One commit can be in multiple branches at the same time. The main branch is usually called master and should contain only runnable code.

Branches can be merged. That comes handy when work in the feature branch is finished and push this feature into the master branch. When two branches are merged, a new commit is created that ties together the histories of both branches [5]. Merge creates a structure similar to 4.3. In the new commit changes from one branch are replayed onto another branch, which can create merge conflicts, for example when a file has been changed in both branches. Merge conflicts have to be resolved manually using merge-tools.

Branches can be also rebased onto each other. The functional result of the rebase is almost the same as with the merge. The main difference is that git rebase will replay commits of the branch onto another branch, creating the linear history of commits. Thanks

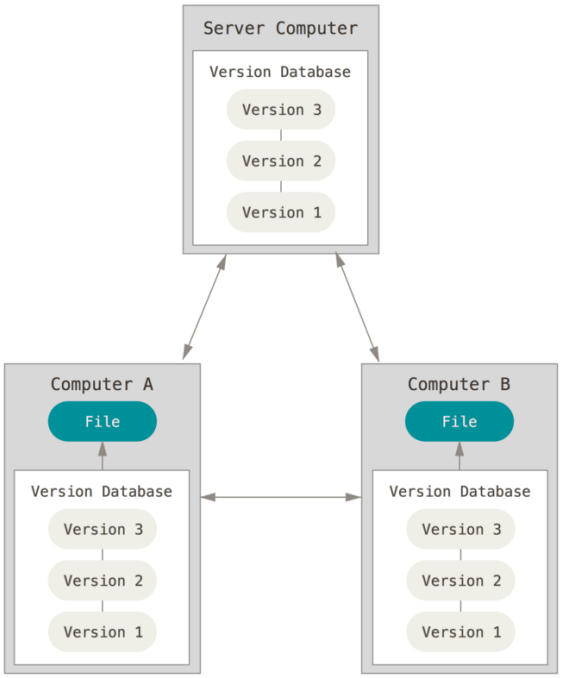

Figure 4.1: Diagram of a distributed version control system [13]

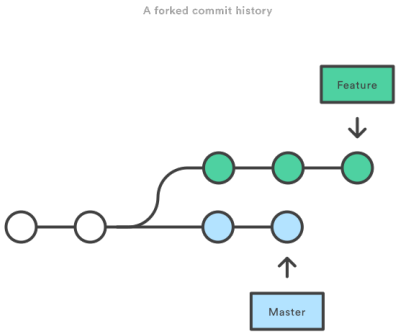

Figure 4.2: Feature and master branches [5].

to their similarity, they can be used interchangeably, and some use git rebase instead of git merge because linear commit history is easier to read.

Commit that branch pointer points to changes as new commits are added to the branch. This dynamic behavior can be problematic if a static pointer is needed, for example when the developers need to mark commit as a basis of a software release. For this and similar use-cases, one can use git tags, which will always point to one static commit, if not manually changed.

Git push and pull are used to synchronize local repository with remote repositories. Push will send changes like new commits and branches to the remote repository. If these changes break the history, the push will be rejected, and force push has to be used to save them in the remote repository. Pull will download changes from the remote repository and apply them to the local branches if local and remote history diverge, Git will try to merge them which can create merge conflicts.

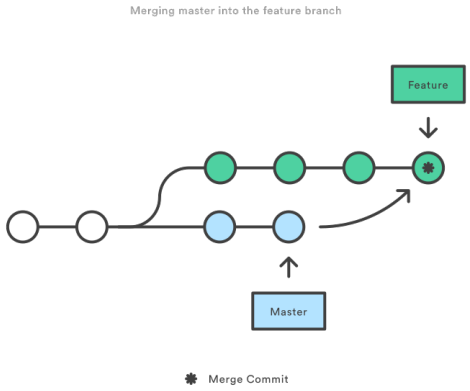

Figure 4.3: Branches after the merge [5].

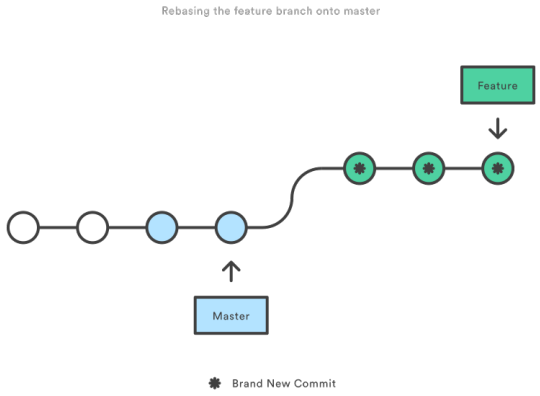

Figure 4.4: Branches after the rebase [5].

4.2 Workflows

A Git Workflow is a recipe or recommendation for how to use Git. Git workflows encourage users to leverage Git effectively and consistently [5]. Git is quite flexible and allows for an uncountable amount of workflows but in this thesis only the GitHub Flow [12] will be further described.

4.2.1 GitHub Flow

GitHub flow is primarily used for projects, that do not use traditional releases and has these essential attributes [12]:

- Anything in the master branch is deployable.
- To work on something new, a descriptively named branch is created off of master.
- developer should commit to that branch locally and regularly push the work to the same named branch on the server.
- When feedback or help is needed, or the branch is ready for merging, merge request is opened.
- After someone else has reviewed and signed off on the feature, the branch can be merged into the master.
- Once it is merged and pushed to 'master', changes should be deployed immediately.

A good practise when using GitHub flow is to have as few commits as possible in the master branch and developer should commit his changes to the feature branch as often as possible, which creates a problem when feature is merged into the master branch. This is usually solved by squashing the commits of feature branch into one commit before merging it into the master.

4.2.2 Forking Workflow

The Forking workflow is based around the idea of a central repository and private copies of the repository, i.e., forks [5]. That is a fundamental difference between other workflows is that developers develop changes on their fork and when they are finished, they ask for their changes to be merged to the central repository.

The most common workflow looks like this:

- A developer creates a fork of a central repository.
- The developer creates a new branch where you develop changes. The name of the branch should say what kind of changes are made in the branch.
- When finished, the developer creates merge or pull request to the central repository.
- If the merge request is successful, the changes are merged to the central repository.

The existence of central repository and private forks makes this workflow popular in cases where not everyone developing the code should have write access to the central repository. A most common example of that are public open source projects [5].

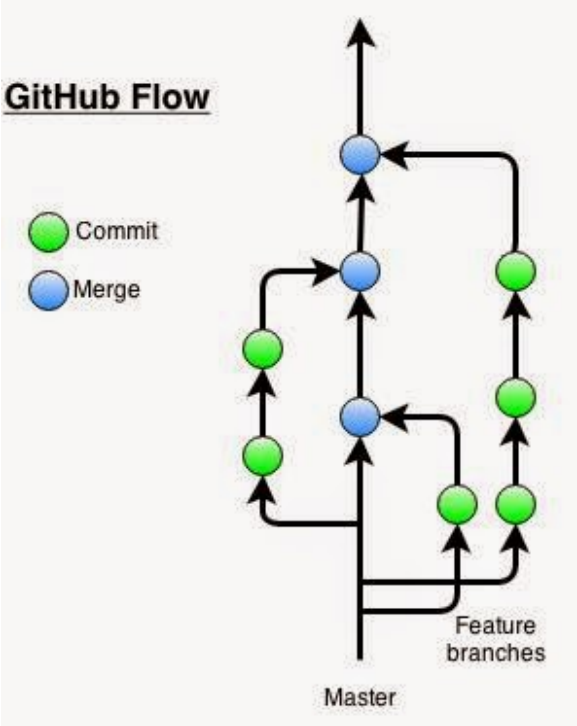

Figure 4.5: GitHub flow [14].

Chapter 5

Containers

The containers are operating-system-level virtualization technology that uses the Linux kernel and features of the kernel, like Cgroups and namespaces, to segregate processes so they can run independently [44]. These processes are called containers.

Containers are very similar to the standard Virtual Machines. The main difference is that the containers share the kernel of the host machine with other containers [18]. The Virtual Machine has to run the whole guest OS with a hypervisor. By contrast, the container uses the same amount of memory as any other process [18], which makes the container lightweight compared to the VM.

The differences between VMs and containers do not end there. Another big difference is that the containers are ephemeral [18]. If the main process inside containers stops, the whole container is removed, and data and the containers state are lost. This means that the container cannot be temporarily turned off and then started again to continue its work.

Similar to the VMs, containers can share directories with the host machine or with other containers. These shared directories are called volumes and can be used to pass data to the container or to get data from the container.

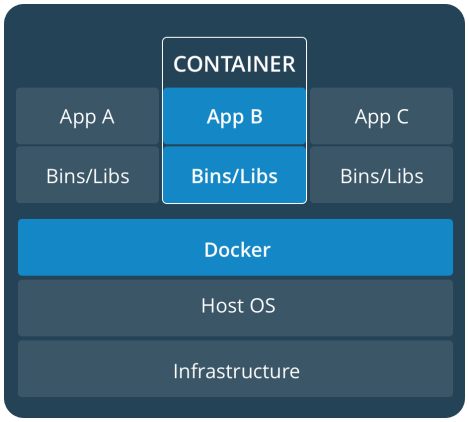

Figure 5.1: Multiple containers running on the host OS [18].

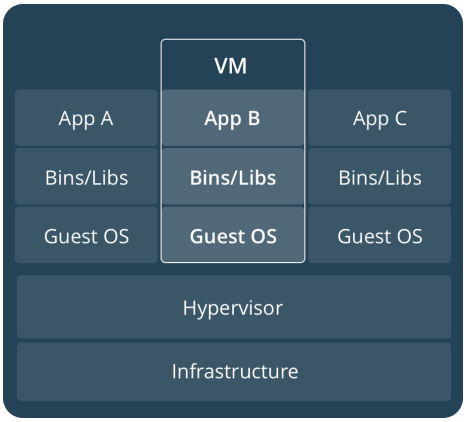

Figure 5.2: VMs running on the host OS [18].

A container is launched by running an image. An image is an executable package that includes everything needed to run an application–the code, a runtime, libraries, environment variables, and configuration files [18].

The images are built using recipes called Dockerfiles [18]. The Dockerfile usually defines the base image that the new image is based from (e.g., Fedora [20], Alpine [3]). These images are generally as small as possible and contain only shell and package manager. Then the image is created by executing the instructions defined in the Dockerfile, which could be shell commands (RUN) or instructions to copy files from the host machine (COPY and ADD). The labels can be added to the image during the build, and the creator of the Dockerfile can indicate which ports should be used and which directories could be shared with host machine or between containers using the PORT and VOLUME directives.

The ability to pack the whole application with its dependencies and then run it in a lightweight and isolated, reproducible environment made containers very attractive, especially for DevOps teams and testers.

5.1 Best Practices

There are some best practices and some good practices when it comes to containers and Dockerfiles. These practices are:

- Keep the images small [18].
- Do not run multiple processes inside one container [6].
- Image should have only one purpose [18].
- Do not store data inside container [6].
- Do not store credentials inside the containers [6].
- Create ephemeral containers [18].
- Use unprivilidged containers [8].
- Do not run as root inside the containers [6].
- Do not use only the “latest” tag [6].
- Add metadata.

The image producer should keep the size of the image as small as possible by not installing unnecessary packages inside containers, for example, the Python is not needed inside a database image. The minimalistic approach increases security because you cannot use ShellShock exploit if there is no shell inside the image.

Because the container is more similar to the process than to virtual machine, it is not recommended to run multiple processes inside the container. Multiple processes in one image make the startup more complex, images bigger, security worse and decrease scalability.

By including credentials in the image, the producer essentially hard-codes the credentials into the binary, which is bad practice. The preferred way is to add credentials at the runtime using the environment variables [6] or by sharing them using volumes, but both of these approaches have their cons and pros.

The use of unprivileged containers and using non-root users inside containers improves security by making limiting the power an attacker gains by compromising the container, which is especially important in the case of the privileged containers because there are ways

how to escape from the container if the privileged containers are used. This means that one defective container can compromise host system and other containers running on the host.

By producing only images with the latest tag, you create two problems. The first one is that users of your image are not able to rollback if there are some issues with your image. The second problem is that the builds based on the latest image are reproducible because the latest tag is dynamic. That means that the base image can change, creating the „it works on my machine“ type of situation.

Chapter 6

Python

Python [40] is an interpreted high-level, general-purpose programming language [63]. Python supports multiple programming paradigms (imperative, functional, object-oriented) and uses dynamic typing. Its design emphasizes code readability and simplicity, and it is very well suited for all kinds of projects, from small to big ones [63]. Python is frequently used in fields like automation, data science, and academia, thanks to its power, simplicity and rich environment of libraries.

Rest of this chapter talks about best practices for parts of software development in Python, such as testing and distribution. Some of these practices are official norms (PEP8 [46], PEP484 [45]), and some are community or company guidelines. Most of them are an object of debate.

6.1 Code Style

Unified code style makes software development easier because developers do not have to waste time trying to decipher the person's style, which is handy when one is reviewing or debugging code written by someone else. Because Python is quite flexible with its code style, a need for a universal coding standard arose. The purpose of the PEP8 [46] was to solve the need for the style unification. PEP8 covers things like indentation, naming conventions, the order of imports, amount and placement of black lines and much more.

Trying to check style is a tiring and error-prone endeavor. Luckily several automated tools can be used to check the PEP8 compliance, for example, Flake8 [67] and Pylint [56]. These tools can also check if the code is aligned with community recommendations like proper usage of the logging libraries or descriptiveness of names.

```
************* Module elastictools.elkinterface
elk.py:1:0: C0302: Too many lines in module (1025/1000) (too-many-lines)
elk.py:28:0: E0401: Unable to import 'dateutil.parser' (import-error)
elk.py:30:0: E0401: Unable to import 'elasticsearch' (import-error)
elk.py:119:47: W0613: Unused argument 'retry' (unused-argument)
-----------------------------------
Your code has been rated at 8.90/10
```

Figure 6.1: Result of an analysis done by PyLint.

```
"""Module with World class."""

class World:
    """Class for the multiplying."""

    def mult(self, x: int, y: int = 2) -> int:
        """
        Multiply two numbers.

        Args:
            x: Number x.
            y: Number y.

        Returns:
            A product of multiplication.
        """
        return x * y
```

Figure 6.2: Example of Python Google style docstrings with type hints.

6.2 Documentation

A great practice overall is to document and comment the code. In Python, developers can inline the documentation into the code using docstrings, looking similar to the Figure 6.2. The format of the docstrings has been proposed in the PEP257 [22]. Unfortunately, this format is not generally accepted, and many other formats exist, for example, Google [2] and reST format [9].

Some parameters of the docstrings can be checked using automatic tools, for example, Flake8-docstrings [17]. Some tools can generate full-blown technical documentation using information in docstrings. One of such programs is called Sphinx [10]. Sphinx is highly customizable and can create nice looking documentation out of the box.

Even though Python is a dynamically typed language, developers want to hint the types of the method's arguments. There was not an official way how to do this until the PEP484 [45] introduced type hints. Type hints are optional and do not add static typing to the Python (this can be done by decorators) but can be used for static analysis by Mypy [29].

6.3 Testing

Every code should be tested. Python has several frameworks and libraries that make testing easier. Two most used are Pytest [27] and Unittest [41]. Unittest is older and is based on the JUnit framework [41], which also means that it is not PEP8 compliant. Pytest is a more modern and pythonic framework and preferred by large parts of the community. These tests can be run by the frameworks themselves or by the runners like nose [33], which can do a more in-depth analysis of the test result or determine the code coverage.

Even though Python uses dynamic typing, it is possible to use static analysis. A tool used for static analysis is called Mypy [29]. Mypy uses type hints and specially format

```
[tox]
envlist = flake8,mypy
skipsdist = True

[testenv:flake8]
deps=flake8
     flake8-import-order>=0.9
commands= flake8

[flake8]
exclude = build
ignore = W503, D401, D413
import-order-style = google

[testenv:mypy]
deps=mypy
commands= mypy --ignore-missing-imports violationreport bin/get_repos.py
```

Figure 6.3: Example of a tox configuration file. Tox will run style checks using Flake8 and do static analysis using Mypy. Thanks to the setting in the section flake8, the Flake8 will ignore two directories and will not check for three errors.

comments to analyze code and can find out that wrong input types are passed or OOP principles are violated. Mypy is still under heavy development, and things can change, but it is already considered to be best-practice by some.

Testing should be as easy and as automated as possible. Some projects do this by creating shell scripts, which could lead to problems and is considered to be a bad practice. Another option is to use Makefiles [49], which will work but could create quite a boilerplate because Makefiles are not designed for Python development. The recommended option by the big part of the Python community is to use Tox [28]. Tox uses virtual environments [7] to separate test environments from each other, which can be used to test the code against various versions of Python interpreters. Tox does not run the tests and style checks himself but uses tools like flake8, Pylint, Mypy or Pytest.

6.4 Distribution and Dependencies

The Python is popular also thanks to the rich library environment. Most of the Linux systems have some Python libraries in their official repositories, but to get access to most libraries, a tool called PIP has to be used.

PIP is a package manager for Python packages or modules. The PIP can be used to install, update or remove Python libraries. It is the recommended tool for Python package management [36] and is part of the standard library from Python 3.4.

PIP can install packages from multiple sources, e.g., Git repositories. However, the most common source of the Python packages is PyPA. PyPA is an acronym for a Python Packaging Authority [37]. PyPA is a working group that maintains many of the relevant

```
flake8==3.5.0
gitdb2>=2.0.5
GitPython<=2.1.11
pyflakes!=1.6.0
```

Figure 6.4: Example of the requirements.txt file. This file is used by the PIP to install dependencies (Python packages). PIP tries to install the newest possible libraries.

projects in Python packaging [37]. Python developers can upload and download the libraries from ant to PyPA repositories for free.

If one wants to distribute his or her Python library or tool, he has to create a setup script. The setup script or setup.py is the center of all activity in the building, distributing, and installing modules. The primary purpose of the setup script is to describe module distribution [39], e.g., information like the name of the package, version, and dependencies, but can also contain thing like description, maintainers or link to the documentation. Setup.py is one of the requirements if one wants to publish the package using PyPA repositories [38].

If the developer wants to list the dependencies without creating a setup script, he or she can do so using the requirements files. Requirements file are a text file with a list of dependencies that looks similar to the Figure 6.4. Requirements are consumable by the PIP, which then installs the packages listed in the listed. Developers can use multiple requirements files for different parts of the roles (testing, production, documentation builds).

Chapter 7

Jenkins

Jenkins is a self-contained, open source automation server written in Java, which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software [25].

The capabilities of the Jenkins can be greatly increased by installing plugins. Jenkins plugin library is rich and plugins can be easily installed, updated and removed from the Jenkins UI. But the richness of the plugin library and ease of the installation can sometimes lead to misuse, which can result in the situation called „plugin hell“. Plugin hell is similar to the dependency hell, where plugins can require conflicting dependencies or one plugin interferences with another one.

The Jenkins uses the master-slave model, where the master is the Jenkins server used to orchestrate slaves, start jobs, check CI/CD triggers and more. The slaves are used to run the tasks. The slave can by any machine that can run Java, from a bare-metal server to containers.

The primary object of Jenkins is a job. The Jenkins job is a definition of a pipeline or a task, e.i., what, when should be done before, during and after the build. Thanks to the rich environment of the Jenkins plugins that job can be anything, from Maven build to a shell script.

The build is an instance of the job that will run on the slave. The build can be triggered manually, by a scheduled cron job or by version control tools, like CVS, Subversion, Git, Mercurial.

7.1 Jenkins Job Builder

Jenkins Job Builder is a tool written in Python that takes simple descriptions of Jenkins jobs in YAML or JSON format and uses them to configure Jenkins, by converting the YAML or JSON to XML job definition, which is consumable by the Jenkins API [24].

Jenkins Job Builder makes keeping the job descriptions in the readable text format in a version control system possible. Thanks to this, the changes and auditing are easier, because a user does not have to use cluttered Jenkins UI. The creation of the similarly configured jobs is also easier thanks to the flexible template system [24].

Jenkins Job Builder have other capabilities, e.g., it can remove jobs from the Jenkins, list jobs or get info about the installed plugins. One of the more interesting modes is the test mode that tries to convert job to XML, which can be used to sanity test JJB definitions.

7.2 Dynamic Jenkins Slaves

At first, the team used a pool of the VM based static slaves. That solution worked for a while, but then the jobs started to fail for unknown reasons. During the investigation of these fails, it was found that a form of dependency hell had caused these fails because every job installed its dependencies when it started.

The problems had several possible solutions. Either the team unifies the dependencies, which could be possible, but it would require a tremendous amount of human resources, or every job is going to run in its own Jenkins slave.

The team decided to go with the custom slave option. There are multiple ways how to achieve this, but the team used the Kubernetes plugin [47]. Kubernetes plugin uses container images to create dynamic Jenkins slaves in Kubernetes based clouds, e.g., OpenShift. Thanks to the Kubernetes plugin, every Jenkins job can have its container image with all the dependencies already installed.

Which container image is used to create Jenkins slave for the job can be set in the Jenkins UI (does not scale very well) or in the JJB definition itself, if the DSL [26] or pipeline [34] job definition is used. Other things can be also set for the dynamic slaves, e.g., if the slave should be reused, how many slaves can be active for a particular job or how long should slave be alive after the job has finished.

Chapter 8

Gitlab

GitLab is an open source end-to-end software development platform. It comes with built-in version control, issue tracking, container image registry, CI/CD, and more. GitLab has multiple editions, where Community edition is the core open-source and free edition. GitLab provides self-hosted version and SaaS version hosted on GitLab servers [21].

8.1 GitLab Continuous Integration

GitLab CI/CD pipelines are configured using a YAML file called `.gitlab-ci.yml` within the project. The `.gitlab-ci.yml` file defines the structure and order of the pipelines and determines what jobs to execute and what to do when specific conditions are met, e.g., the previous job failed, or a pipeline run is based on a specific branch.

YAML format used in `.gitlab-ci.yml` supports YAML anchors and expansion, which reduces the amount of replicated code.

GitLab CI configuration also supports variables. Variables can be defined in the trigger, at the group, project, pipeline, and job level. Some variables are pre-defined by the GitLab, e.g., CI_COMMIT_SHA which stored SHA hash of the commit the pipeline run is based on. Special kind of variables are secrets, which are set at the group or project level in the GitLab UI. These values of the secrets are censored from the logs. Variables are referenced in the YAML in bash format, i.e., $NAME_OF_THE_VARIABLE.

The most basic unit in the pipeline configuration is a job (shown in Figure 8.1). Job defines what, where, and when should be executed on a runner. The most crucial component of the job configuration is a script field. Field `script` defines what should be executed, e.g., a sequence of bash commands. It can be paired with `before_script` and `after_script`, which define what should be done before and after the `script`. If any of the scripts fails, the whole job fails. That usually results in failure of the whole pipeline, except when an

```
job1:
  before_script: "execute-before-script-for-job1"
  script: "execute-script-for-job1"
  after_script: "execute-after-script-for-job1"
```

Figure 8.1: Basic GitLab CI job with name job1.

```
job:
  # use regexp
  only:
    - /^issue-.*$/i
  # use special keyword
  except:
    - branches
```

Figure 8.2: Example of the `only` and `except` usage with pattern matching.

`allow_failure` field is set to True. The job can also be rerun if the desired retry count in the `retry` field.

Fields `only`, `when`, and `except` define conditions when and when not to run the job. Only defines the names of branches and tags for which the job will run. Except defines the names of branches and tags for which the job will not run. Option `when` has several specific setting, e.g., `manual` when the job has to be OK manually before starting or `on_failure` when the job starts only if at least one of the previous jobs failed. Fields `only`, `when`, and `except` can be used together, because they are inclusive (an example is shown in Figure 8.2) and work in the AND fashion. Options `only` and `except` also support regular expressions, which can be useful if you workflow names the branches in a specific way [21].

The job can create artifacts, which are files and directories which should be attached to the job after success. The artifacts can be then downloaded using GitLab UI or can be used by other pipeline jobs that start after the job that creates artifact is finished. `Artifact` field in job configuration sets the list of artifacts that are created, especially the names of the artifacts and the path in the job executors directory tree where the artifact is present. Other settings can also be set, e.g.,`expire_in`, which defines when the artifact expires, i.e., is going to be cleaned from the GitLab.

Stages group jobs into groups. The jobs in the stage run in parallel. Next stage can be started after all the jobs in the previous stage are finished. Field `stages` in the job configuration sets into which stage the job belongs. Default stages are `build`, `test`, and `deploy`.

8.2 Runners

GitLab Runner is the open source project that is used to run your jobs and send the results back to GitLab. GitLab Runners are used in conjunction with GitLab CI, the open-source continuous integration service included with GitLab that coordinates the jobs [21].

GitLab Runner is written in Go and can be run as a single binary. No language specific requirements are needed. It is designed to run on the GNU/Linux, macOS, and Windows operating systems [21].

GitLab Runner implements several executors that can be used to run your builds in different scenarios [21]. Executors range from simple shell executor, that runs the job on the same machine the runner runs on to the Kubernetes/OpenShift executor, that runs jobs inside a container cluster. The Kubernetes executor creates a pod inside a cluster, where the job can run [21]. Every job has its pod, which means that the jobs are isolated from

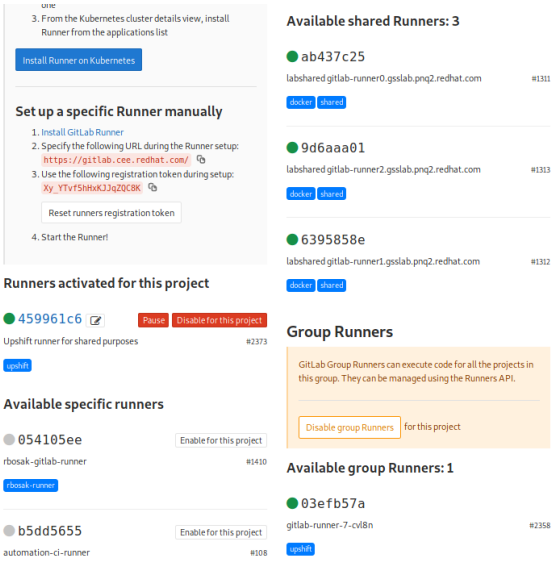

Figure 8.3: Runner settings in GitLab UI. You can see multiple shared runners, one group runner and few inactive specific runners. Every runner has few tags associated with it, e.g. docker or upshift.

```
build-image
  tags:
    - upshift
    - docker
  script:
   ...
```

Figure 8.4: GitLab CI job definition with the runner tags. The tags are allowing for job to run on runners that have both docker and upshift tags.

each other. The Kubernetes executor provides a wide range of configuration, from resource management settings to volume mounting.

The runners can be divided into two groups, based on repositories access to them. The first group are specific runners, which are runners specific to that particular repository and had to be allowed. The second group are shared runners, which can be shared within the whole GitLab instance or group. The shared runners are allowed automatically. Figure 8.3 shows GitLab UI runner settings with shared and specific runners.

Runner tags are also shown in Figure 8.3. Jobs can use tags to define a subset of runners where they can run. The runner tags are used if a job needs a special environment, e.g., runner on Windows machine or Kubernetes cloud.

8.3 Environments and Deployments

GitLab CI is not only capable of building and testing of the software but can be used to deploy software to various environments. These environments come in many flavors, but most prevalent is a stage, production and preview environments. To show what version of the

```
deploy_production:
  stage: deploy
  script:
    - echo "Deploy to production"
  environment:
    name: production
    url: http://prod.example.com
  only:
  - master
```

Figure 8.5: Example of GitLab CI job that deployes application to the production environment. The job starts only the pipeline is deploying master branch.

```
deploy_review:
  ...
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_ENVIRONMENT_SLUG.example.com
  only:
    - branches
  except:
    - master
```

Figure 8.6: GitLab CI enviroments section for dynamic enviroment [21]. Software will be deployed to this dynamic enviroment only if deployed from branch other than master.

software is deployed where it is deployed GitLab environments are used, i.e., environments allow control of the continuous deployment of software, all within GitLab [21].

Firstly, environments have to be defined in the `.gitlab-ci.yml` or to be more precise the job that deploys software to the environment has to be associated with the environments. This is done by adding an environment section to the job configuration as is shown in Figure 8.5. This setting creates a new environment in the environments and associates the job with environment deployment. That is important because in the case of redeployment or rollback only that job is rerun, not the entire pipeline.

The example in Figure 8.5 shows a static environment being deployed, but GitLab CI allows for dynamic environments to be deployed. That is useful for previews, where a new version is deployed to the dynamic environment during merge request, so the reviewers can see how the change looks and then the environment is scrapped after the merge request is accepted or closed. That is done by using variables in the environment. Figure 8.6 is an example of such job.

After the environment sees to be useful, stop job can be used to clean it up. Stop job is a job in .gitlab-ci.yml that cleans the environment, reclaims the resources, etc.

All the environments associated with the project and deployed with the project's GitLab CI pipeline can be seen in environments section of the GitLab UI. In the UI, we can not

only see a currently deployed version of the software, but we can trigger redeployment or rollback.

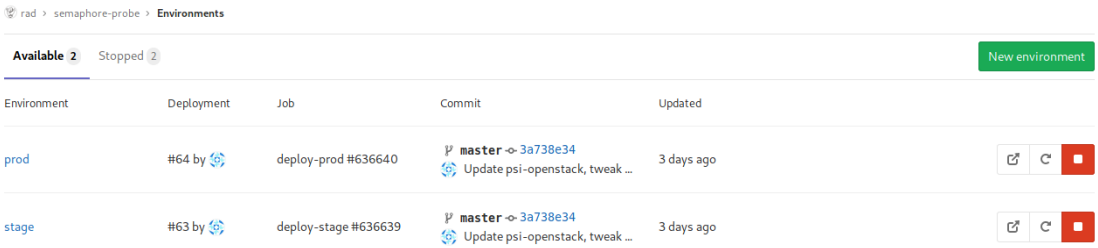

Figure 8.7: GitLab UI is showing all the environments deployed by the projects. You can see which version is currently deployed and when it was deployed. You can redeploy the software.

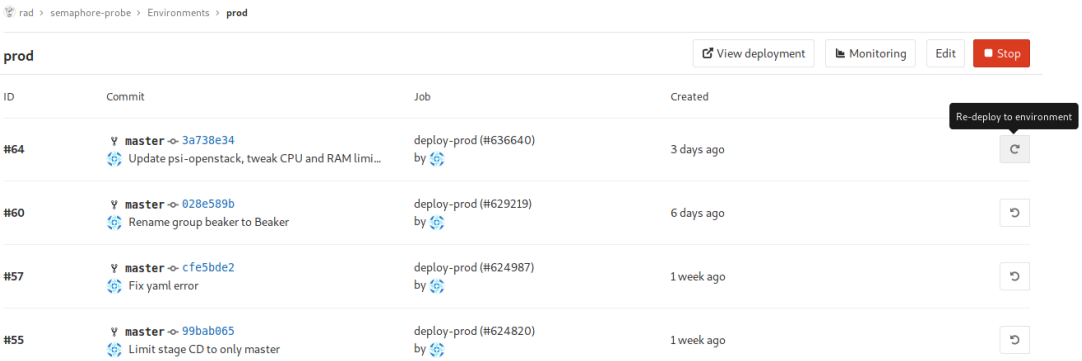

Figure 8.8: History of deployments to the environment. You can see which versions and when have been deployed to the environment. You can also redeploy or rollback to past version of the software.

Chapter 9

OpenShift

OpenShift is a family of containerization software developed by Red Hat. Its flagship product is the OpenShift Container Platform—an on-premises platform as a service built around Docker containers orchestrated and managed by Kubernetes on a foundation of Red Hat Enterprise Linux [61].

Platform as a Service (PaaS) or Application Platform as a Service (aPaaS) or platform-based service is a category of cloud computing services that provide a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app [62].

9.1 Pods, Services, Routes, and Deployments

OpenShift Container Platform is based around pods, which is a concept from the Kubernetes. The pod is one or more container deployed to together on one host, and it is the smallest compute unit that can be defined, deployed, and managed. Containers within the pod can share their storage and networking [42].

Pods run until their container(s) exit, or they are removed for some other reason. There are multiple policies that are concerned with the exited pods. Usually, the pod is cleared after exiting, but it can be retained to enable access to the container logs [42].

A Service is an internal load balancer. It proxies and balances the connection it receives to a set of pods. That enables for pods to be added and created without affecting the consistency of the service. The Services also add the ability to refer to it at a consistent address [42].

A route is used to expose Service to the external clients at a hostname. DNS resolution for a hostname is handled separately from routing [42].

A deployment ensures that a specific number of pod replicas are running at all times. That means that if pod is removed, the deployment controller makes sure that a new instance of the pod in instantiated. Similarly, if more replicas are running that desired, it deletes as many as necessary to match the defined amount. Deployment can be configured to auto-scale based on the load or traffic [42].

Each time a deployment is triggered, whether manually or automatically, a deployer pod manages the deployment (including scaling down the old one, scaling up the new one, and running hooks). The deployment pod remains for an indefinite amount of time after it completes the deployment to retain its logs of the deployment. When a deployment is

superseded by another, the previous replication controller is retained to enable easy rollback if needed [42].

9.2 Projects and Users

Interaction with OpenShift Container Platform is associated with a user. An OpenShift Container Platform user object represents an actor which may be granted permissions in the system by adding roles to them or their groups.

A project is a Kubernetes namespace with additional annotations and is the central vehicle by which access to resources for regular users is managed. A project allows a community of users to organize and manage their content in isolation from other communities. Users must be given access to projects by administrators, or if allowed to create projects, automatically have access to their projects.

There are 3 kinds of users [42]:

- Regular users, e.i., mainly humans.
- System users that are automatically when the infrastructure is defined. Mainly to enable the infrastructure to interact with the API securely. Example of the system user is the cluster administrator.
- Service accounts that are special system users associated with projects. Some are created automatically when projects are created, and some are created by the project administrators. They are often used to grand access and rights to bots, scripts and other non-human users.

9.3 Builds and Image Streams

A BuildConfig is a type of object that defines Builds. Builds are used to transform input into other objects, usually, source code into container image. OpenShift creates Docker-formated containers from build images and pushes them into a container image registry. Many aspects of the Builds can be defined in the BuildConfig, from the source code location to CPU and memory usage constraints.

The OpenShift Container Platform build system provides extensible support for build strategies that are based on selectable types specified in the build API. There are three primary build strategies available [42]:

- Docker build
- Source-to-Image (S2I) build
- Custom build

An image stream and its associated tags provide an abstraction for referencing container images from within OpenShift Container Platform. The image stream and its tags allow you to see what images are available and ensure that you are using the specific image you need even if the image in the repository changes. Image streams do not contain actual image data but present a single virtual view of related images, similar to an image repository.

Deployments and Builds can be set up in a way, that if an image in an image-stream has changed, the new version of Deployment or Build is created, which uses the new image [42].

9.4 Templates and Object Definitions

OpenShift user can export the objects to YAML of JSON files. These exports can be then used as a backup or during migration. A significant advantage of these exports is that users can store their OpenShift objects (e.g., Deployments and Services) in the version control systems, for example in Git.

A step up from the objects exported in the YAML or JSON files are templates. The template describes a set of objects that can be parameterized and processed to produce a list of objects. The objects defined by the templates can be anything the user can create, from Builds to Services [42].

```
kind: Template
apiVersion: v1
objects:
  - kind: BuildConfig
    apiVersion: v1
    metadata:
      name: cakephp-mysql-example
      annotations:
        description: Defines how to build the application
    spec:
      source:
        type: Git
        git:
          uri: "${SOURCE_REPOSITORY_URL}"
          ref: "${SOURCE_REPOSITORY_REF}"
        contextDir: "${CONTEXT_DIR}"
  - kind: DeploymentConfig
    apiVersion: v1
    metadata:
      name: frontend
    spec:
      replicas: 2
parameters:
  - name: SOURCE_REPOSITORY_URL
    displayName: Source Repository URL
    description: The URL of the repository with your application source code
    value: https://github.com/sclorg/cakephp-ex.git
    required: true
  - name: GITHUB_WEBHOOK_SECRET
    description: A secret string used to configure the GitHub webhook
    generate: expression
    from: "[a-zA-Z0-9]{40}"
message: "... The GitHub webhook secret is ${GITHUB_WEBHOOK_SECRET} ..."
```

Figure 9.1: OpenShift template in YAML format. The template creates a BuildConfig that is used to build container image of the example aplication and Deployment is then used to deploy 2 instances of the application.

Chapter 10

Situation Assessment

To make good decisions about our testing and release infrastructure, hard data about the current status of the codebase need to be gathered.

This chapter contains information about the technologies the team uses, how they use them and where are deficiencies, with examples of how these deficiencies cost time and human resources.

10.1 Codebase

The team uses multiple languages and technologies in a codebase. The codebase of our projects is distributed in approximately 100 Git repositories. As can be seen from Figure 10.1, most of the projects are Jenkins jobs, which use a combination of Python, JJB and containers. The team also has a few projects that use OpenShift and have OpenShift object definitions in their repositories. A small fraction of the projects also uses Javascript and Ansible, and there is a growing amount of projects that use Golang.

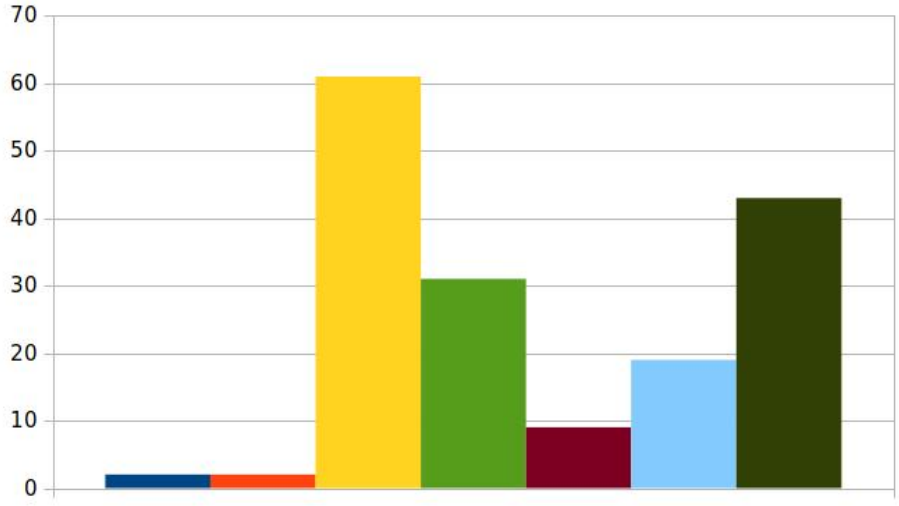

Figure 10.1: Technologies and program languages used in the codebase. These are Javascript (dark blue), Golang (orange), Python (Yellow), JJB (green), Ansible (dark red), OpenShift (light-blue) and Docker (dark green).

The combination of the JJB, Python, and containers is so prevalent because we use container images to create custom dynamic Jenkins slave for every job to avoid dependency hell (more info about it can be found in subsection 7.2).

10.2 Version Control and Directory Structure

All our projects use Git as the version control system, and most use Github workflow. Most of the codebase is hosted on the internal Gitlab instance, and parts that have been open-sourced and made available to the public are hosted on the GitHub. The team also uses a uniform directory structure, where tests are grouped under a directory called tests, scripts under directory scripts or bin and so on.

It is a good practice to keep resources of the project centralized, or at least keep the parts that are closely related (frontend and backend can be hosted in different repositories, etc.). We do not adhere to this practice, because we have all the JJB configuration files with the Jenkins slave Dockerfiles in one giant repository and the Python tool used by those jobs in a project repository.

The repository of repositories used to store JJB files should theoretically make a recovery after the loss of the Jenkins master easier, but this was not tested yet. This configuration makes CI/CD pipelines non-trivial if not impossible because the relationships between files are not easily apparent (i.e., if this file changes, which tests should be run) and testing 50 different things is not practical.

10.3 Code Style

The team uses multiple languages, like Ansible, Go, and JavaScript, but the team-wide style requirements are only defined for Python. The style requirements is that code is PEP8 compliant and was checked by the Flake8 or the Pylint.

Most of the projects that use Python enforce the standards semi-manually. That means that a developer or reviewer has to run the tools that check the PEP8 compliance manually.

To access how effective this manual process is, I have run the style checks using Flake8 on the current and past revisions of the code. The results of this analysis can be seen in Figure 10.3.

The results of the style violations are not very satisfactory. The manual enforcement was not very effective at preventing the Python style violations, and the amount of the violations has an increasing trend.

When I have taken a closer look at when these violations appeared, I found out that most of them happened at the end of the project, when developers had been under time pressure or when a large amount of the violations was already present in the project. The violations also have a habit of staying in the codebase after they are introduced.

The Python style violations cause all the issues that come with technical debt. The code smell, readability is decreased, the developers look bad, the code reviews also take longer, because you have to fix issues that were not caused by the current changes. All of these issues cost valuable time and can be a huge problem, especially shortly before release.

As mentioned in the first paragraph, the team has no policy about the style for the other languages and technologies. That can be another source of the hidden technical debt, especially when the style policies are adopted.

```
rad-jenkins/
├── jobs/ (Jenkins job definitions)
│   ├── prod/
│   ├── stage/
│   ├── auto-statusreports-stage.yaml
│   ├── e2e-container.yaml
│   └── ...
├── containers/ (Definition of container images used to for Jenkins slaves)
│   ├── rad-slave-3sn/
│   ├── rad-slave-amb/
│   └── ...
└── configs/ (JJB configuration files)
```

Figure 10.2: Directory structure of the rad-jenkins repository, which is used to store Jenkins job definitions and container images used for Jenkins slaves. As can be seen from the structure, the relationships between files are not apparent and parts of the Jenkins job codebase (i.e., Python code) is stored in another repository. That goes against principles of CI/CD and makes use of CI/CD pipeline difficult, inefficient (not able to tell what to rebuild), maybe even impossible.

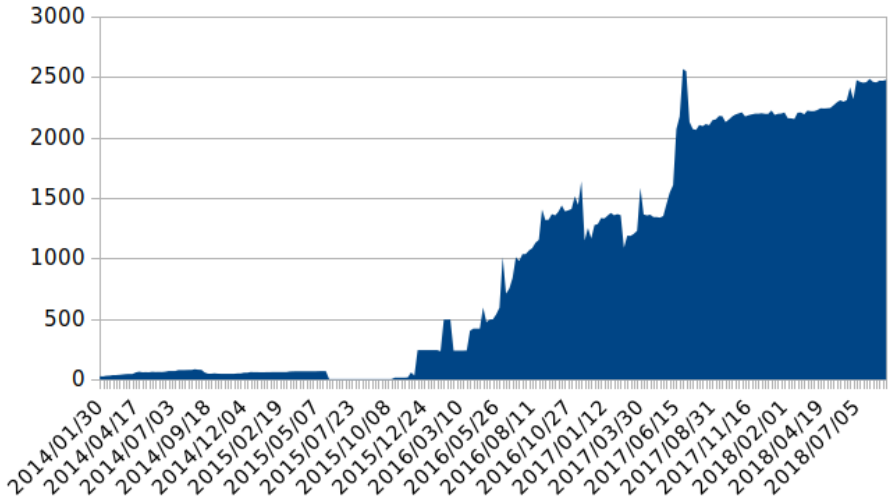

Figure 10.3: Total amount of PEP8 violations in the codebase. The code was anylized using Flake8.

10.4 Tests and Coverage

There is a team-wide policy that all the software developed by the team should have at least automated unit-tests. Most of our projects have a unit-test suite, but not higher levels of tests, i.e., integration, system tests. There is no policy concerning minimal test coverage needed.

The status of the tests is controlled manually by developers and reviewers, similarly to the style checks. The problems with this approach are similar to the problems with style enforcing, i.e., tests fail or are broken, but nobody noticed.

To give some real-life examples, a developer added a new feature to the tool that automatically imports virtual machine images to the internal cloud. With this feature, a bug was introduced, but nobody noticed because everybody forgot to check if the tests are passing. The defect broke metadata of the images and went unnoticed for two weeks. After the developer found about the bug, it was fixed in 20 minutes, but the metadata could not be recovered.

There was a case when the unit-tests of our project were broken for one year. That was caused by the rushed release, and then it was forgotten about. But then after a one-year new feature was requested. The developer started to implement this feature and noticed that the tests are failing. The developer spent several hours checking if the fails are real and then more time to fix the tests of the features he knew very little about.

10.5 Development Environment and Automation

The topic of a development environment is closely related to the testing and style enforcement. The development environments are not only used by the developers but also to the reviewers that have to create them to make sure that style is enforced and tests are passing.

Figure 10.4 and my manual examinations showed that most of the projects have instructions on how to create a development environment (e.g., which libraries need to be installed) in a README. Most of these instructions start with the list of dependencies that need to be installed. Listing the dependencies in the README is not a wrong way how to do it, but providing the list of dependencies using requirements files, setup.py or providing the whole development environment as s container image, would make the process of set up more straightforward, especially for the reviewers that will not be that familiar with the project and its dependencies.

The area where the documentation sometimes lacks is how to run the tests and style checks (shown in Figure 10.5). Few projects have this detailed in the README. However, this part of the development can also be automated using Makefiles or Tox. That would make testing easier, and the bus factor associated with testing would be reduced because the testing and style enforcement would be reduced to one command.

Another unfortunate consequence of the insufficient unification and automation that it is impossible to determine coverage for the project on mass. I tried to create a script which tries to determine test coverage through the repository history, but after several days I gave up because I was unable to determine the dependencies and which wrapper should be used to run tests.

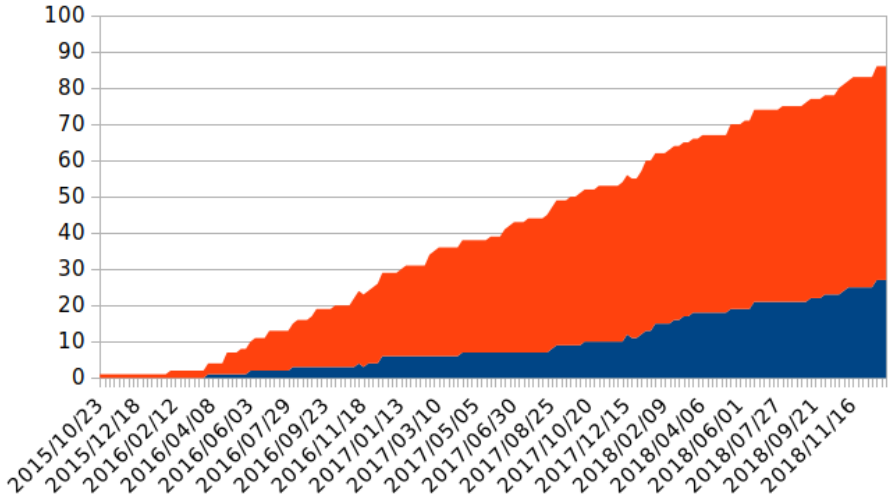

Figure 10.4: Presence of requirements.txt or setup.py files in the repositories that can be used to install dependencies with minimal human intervention. Blue means that the requirements.txt or setup.py was present in the repository, orange means that requirements.txt or setup.py was not present in the repository.

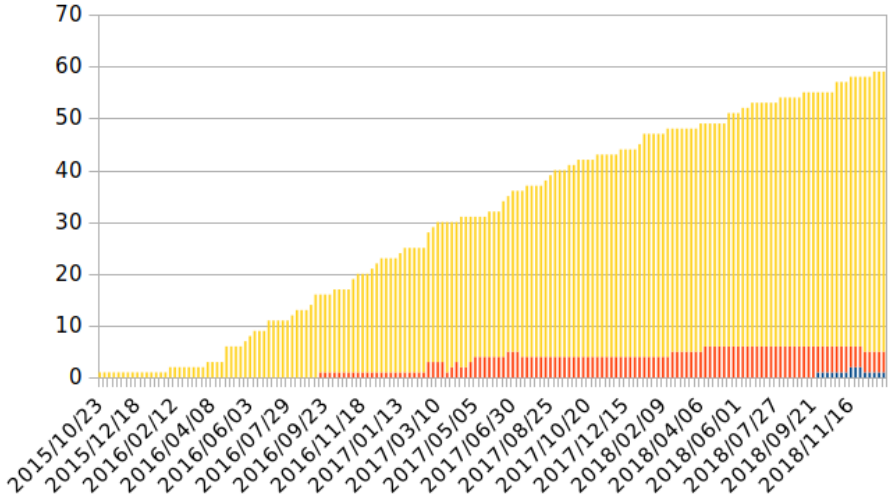

Figure 10.5: Types of the development automation in use. Tox (blue), Makefiles (orange) and none (yellow).

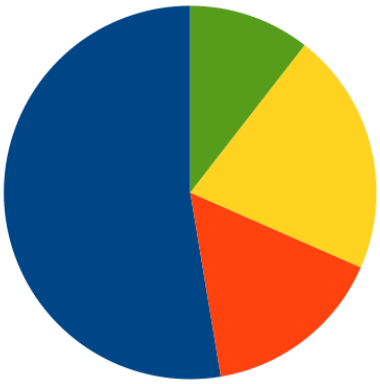

Figure 10.6: Places where the documentation is stored: Confluence or MOJO page(blue), Google doc (orange), README (yellow) and documentation buildable by Sphinx (Green).

10.6 Documentation

It is a team policy that all the code has to be commented and documented. The team has been successful in enforcing this policy, and except a few isolated instances, all the code has docstrings.

Most projects have technical documentation. The technical documentation has mostly a form of the Mojo page. This can create a problem because it is separated from code and engineers forget to update it sometimes.

Unfortunately, the team does not use buildable documentation to create API reference from the docstring and to keep technical documentation with the code. Only two projects use Sphinx to create its documentation.

10.7 Jenkins Jobs

The team has currently 168 Jenkins jobs in production. The team stores the configuration of these jobs as a YAML file that can be parsed by the Jenkins Job builder tool to create or update Jenkins job.

As mentioned in the Section 10.2, the team keeps all its JJB configuration YAML files in a single repository, which makes the documentation and CI/CD much harder than it would be if separated.

Our developers also do not create and update the Jenkins job automatically but manually. This already caused an issue, when a fix to the defect was merged with the master, but the fix was never deployed to the production.

There were also cases when somebody changed settings in the Jenkins UI and forgot to update the JJB configuration file. Then somebody updated the Jenkins job using the JJB file, and old issues appeared again, and because nobody remembered how to solve them, the engineers had to spend several hours looking for the fix.

10.8 Containers

The team has images in about 80 image streams. Most of these images are used to create slaves for our Jenkins master. The team stores all of these images in the container registry from which they can be downloaded by our tools, Jenkins master and OpenShift.

If a developer has to create a new image, he has to go through a process similar to this:

1. Start docker o service.
2. Start the build.
3. Wait for build to finish, which can take more than 15 minutes.
4. Do some testing.
5. Copy API token from the container registry.
6. Login into the registry with CLI.
7. Push image into the registry and wait for the push to finish.

As you can see, the process can be quite lengthy (5 - 20 minutes), contains at least three context switches, and is boring and error-prone. The process also does not prevent and sometimes even encourages these issues:

- increased bus factor,
- no automatic updates,
- missing or defective metadata,
- versioning and orphaned latest images, and
- developer's computer influences the image.

The biggest issue that the manual builds of our container image is increased bus factor because the developer is not forced to document and automate the process, because he does it on his machine. That is also closely related to missing metadata, especially metadata like build date and similar because people forget to include them when they are creating new images.

The average age of our images is more than two months because there are no automatic rebuilds and images are created by the developers when they are working on the project. This means that all the security issues or buggy libraries that have been fixed in the last two months or more are still present in the images.

One of the silent problems is that the state of the repository image is built from and the state of the developer's computer can influence the image. The most serious problem was when unbeknownst to the developer, the secrets like private keys were leaking into the image because the build process copies the whole project repository into the image.

The team also has around 20 images with the tag latest that have no versioned image with the same SHA hash. That means that when somebody uploaded a new latest image, he has not uploaded the same image under some version. That became a big problem when there was an issue in the image (bug in the system library), but developers were not able to rollback the image, because the latest image was rewritten.

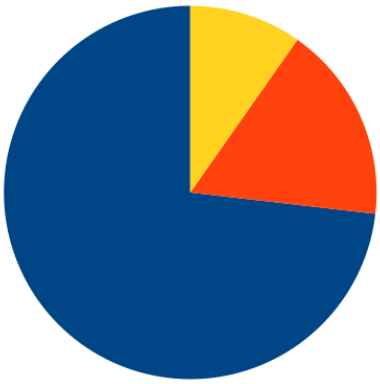

Figure 10.7: Purpose of the container images. Jenkins Slaves (blue), testing (orange) and other (yellow).

10.9 OpenShift

Few projects use the OpenShift directly and have definitions of the OpenShift objects, e.g., templates used to create infrastructure in OpenShift and configuration files used by the infrastructure.

Similarly to the container images and Jenkins jobs, these objects are uploaded to the OpenShift manually, and it causes similar problems. For example, the configuration of service is fixed manually in OpenShift, but the object definition in the Git repository is not fixed, which caused the issue to reappear.

10.10 Conclusion

The analysis found that most of your software is Jenkins jobs, and most used technologies are Python, JJB and containers.

The team has good policies that are up to industry standard. The real problem is in the enforcement of these policies and the general lack of automation. That can be surprising because the primary objective of the team is automation of the various workflows, but this is a common situation, where developers spent significant amounts of time automating costumer's workflow but forget to automate their workflows [19].

The reliance on manual workflows decreases of effectivity of the policies as shown with PEP8 violations (Figure 10.3), wastes time as shown with the case of container image builds, and increases bus factor associated with testing and deployment.

Chapter 11

Design and Development

As written in conclusion of the situation assessment, the main problem is not a lack of policies but their enforcement, the reliance on manual checks to be more precise.

To tackle this problem, we have to find a solution or solutions that are:

- automated,
- scalable,
- easy to use,
- easy to set up, and
- easy to maintain.

In this chapter, I describe how we solved the problem of automatic policy enforcement, why we decided to what we have done and what new challenges our solutions brought with them.

When I use we, our, us in this chapter, I mean the team because most of the decisions in this chapter are too essential and tasks are too big to be done by one human.

11.1 Continuous Integration and Continuous Deployment

We decided to use the principles of CI/CD as our main line of defense. CI/CD pipelines are by their definition automated, and if implemented right, they are also scalable, simple to use, setup and maintain.

To use CI/CD pipelines effectively, we also have to improve and unify the way of developing and testing our software, how we build various artifacts, and more. We also have to define new policies that solve various new problems that come with CI/CD, e.g., retention policies. These problems and the ways how we addressed them I describe in further sections.

11.1.1 CI/CD Framework

When deciding which CI/CD framework should be used to create our pipelines, we had to consider several aspects like reliability, maintainability, scalability, ability to use framework internally, generality, integration with GitLab and usability.

The first and one of the most crucial factor was that we did not want to deploy a new service that would handle the running of the CI/CD pipelines ourselves. The main reason is that the team does not have the human resources to maintain new service just for ourselves and persuading IT to maintain the new service for us was not an option. This factor limited the frameworks to frameworks that are already deployed and supported internally or have an online public version.

The second factor I had to consider was the ability to work within the internal environment. Majority of our codebase should not be publically available, mainly because it makes no sense. Our software has to be deployed behind the Red Hat's firewall, so deploying it from public CI/CD services could be difficult or impossible. This factor limited the option to Jenkins, Gitlab CI, and OpenShift CI because they are already deployed internally and are maintained by various Ops oriented teams.

The OpenShift CI option was considered for long time because OpenShift is Red Hat product, so help and support is more readily available. OpenShift CI uses Jenkins, which is created inside the OpenShift project itself, to run the pipelines. The main reason, why we did not use the OpenShift CI is that it shines mainly in OpenShift-centric and container-centric pipelines, and we wanted to use more general solution. I also did not like the lack of visibility, because results of the runs are stored in the OpenShift and are not out-of-box visible from the repository.

By removing the OpenShift CI from the list of considered frameworks, we are left with two last options, Jenkins and GitLab CI. Both of them are relatively similar. Both frameworks are capable of handling general pipelines; both of them are scalable and well established.

Jenkins has an advantage in its rich plugin library, but this advantage is reduced because internally only a reduced set of Jenkins plugins is available, and there is also a small possibility that we would end up in the plugin hell.

We also have a lot of bad experiences with Jenkins, from significant slowdowns caused by one misconfigured job to complete blackouts. It got so severe that we had to solve and debug these issues ourselves and for more than a half a year, we maintain the Jenkins masters ourselves. Because of the number of pipelines and jobs, the CI/CD would require its own Jenkins masters, which would mean that we would have to maintain more instances.

On the other hand, GitLab CI comes with the GitLab, new features are being added, and most importantly results of pipelines are visible in the web-UI. I also like that the pipeline settings are present in the repository with the code. The main disadvantage of GitLab CI is that in the version deployed internally, one cannot import anchors, templates and other things from other files or projects, which causes that certain parts of GitLab CI pipeline settings are repeated in all the project. This problem is solved in the new version of the GitLab CI, so it should not be a problem for more than a year.

In the end, we decided to go with GitLab CI. The first reason was that the results are visible inside the repository without additional work. Other and more important factors were stability and that we would not have to maintain service ourselves. That comes at the price that we cannot update service ourselves or leverage plugin library that Jenkins has, but the tradeoff should be worth it.

11.1.2 Continuous Integration

Every project has its CI/CD pipeline defined in .gitlab-ci.yml that is present in the repository. Most of the pipeline is based on the example pipeline from the Project Template

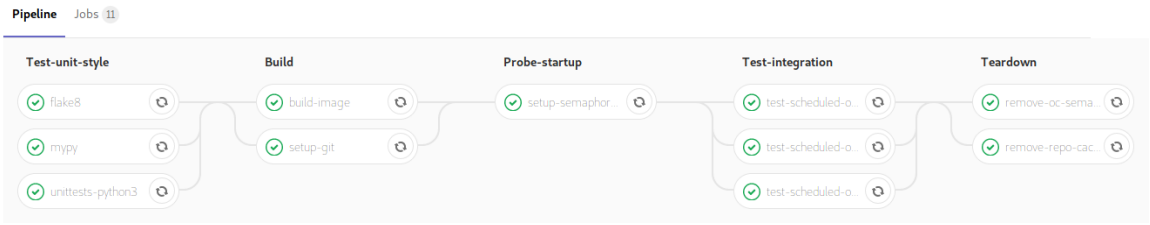

Figure 11.1: Example of GitLab CI pipeline. The pipeline is based of feature branch, so the CD part of the pipeline is missing. In the first stage, code is tested using Flake8, Mypy and Pytest. Then container image with the code is build in OpenShift and in the following stages, the integration tests are run. After the test phase, the test environment is tear down.

(more about that in the Project Template section), except parts which cannot be shared, e.g., integration tests.

I based our pipelines on a textbook CI pipeline with few tweaks. The first tweak is the inclusion of the pre-build phase. During this phase aspects of the codebase that do not require the code to be built are checked, e.g., style checks, JJB sanity checks, and Python unit-tests. More details about how this is done can be found in section 11.3.

After the pre-build checks are successfully finished, the build phase starts. In the vast majority of the cases, the container images are created during the build phase because Python, Ansible and most of the other technologies we use do not require compilation or building. Almost all of our images are built in OpenShift, and the exact intricacies of how the images are built in the OpenShift are described in section 11.6. For now, it is important to note that the image is stored in dev imagestream inside container registry.

Some projects also use the build phase to set up the testing environment for their integration and system tests. The phase is renamed to build-create-test-env or similar in that case to emphasize that build is not the only action done. The environment creation is done during the build phase to save time because the container image builds can take more than 6 minutes.

When the container image is built, the integration and system testing are started. In the best case scenario, the real integration and system tests are run, but for older projects or project that are in maintenance (i.e., no new features just bug fixes) sanity checks are used. That usually means that the test checks if the program inside the container can write help. If the program can write help, it usually means that syntax errors are not present and all the dependencies are in the container. These sanity checks are not ideal, but better than nothing.

If the branch the run is based on is not master, the pipeline ends with the teardown phase. During this phase, the testing environment is destroyed or at least cleaned.

If the branch the run is based on the master, the release phase starts after the teardown. During this phase, the container image is copied/moved from the dev imagestream to the prod imagestream. The tagging scheme is changed a little bit. In dev imagestream, images are tagged by the commit ID they have been created from In production imagestream we use incrementing number to tag images. The different tagging scheme is used to make pipelines simpler because commit ID is available in the entire pipeline.

11.1.3 Continuous Deployment

Some of the projects add continuous deployment to the pipeline. The deployment can vary a lot, but because most of our software are variations of Jenkins jobs, I have been able to come up with a unified way how to deploy them to the Jenkins master.

Deployment of the Jenkins job has two parts, which are done inside a special container. First one is the creation of the update of the Jenkins jobs by Jenkins Job Builder. The second part is to tag a container image that should be used as a basis for the dynamic Jenkins slave with the environment tag, e.g., prod and stage. That is done to be able to rollback the slave image too if something happens.

Since we use GitLab CI, developers and maintainers can see which version and commit were deployed to which environment (example shown in Figure 8.7) and can easily rollback using Web UI.

11.1.4 GitLab Runners

All of the GitLab runners used by us are container based. There is a possibility to use more static runners, VMs for example, but thanks to our reliance on containers, that was not yet needed.

The team responsible for our GitLab instance deployed 3 shared container runners. Each of them is capable of running multiple jobs in parallel. That made our adoption and experimenting with the pipelines easier, but during the process of the adoption, we noticed big spikes in the run durations and an overall increase in the lengths of the builds. After a short investigation, I found out that these shared runners were overloaded. The first sensible solution was to add some team runners. To do that, my colleague and I acquired a project in our OpenShift instance and created new runners there. He used Kubernetes runner because OpenShift is based on Kubernetes and OpenShift runner is not available.

The shared runners are VMs with Docker or Podman running on them. These causes that the runner in OpenShift and the shared runners are not equivalent. The main difference is that in OpenShift, the containers do not have root privileges, so I had to update images used to create environments for the pipeline jobs, so they do not require root access for their function. That mainly meant creating more specialized images and installing everything needed into them during the container image build.

11.1.5 Secrets, Variables and Forking Workflows

To decrease the secret maintenance and simplify the adoption process I chose to use team and group level secrets for secrets shared by the whole team, container build engine token and container registry tokens for example.

Unfortunately, I did not end there. In the first iteration of the pipelines, I used team level secrets to store values of the variables like name of our Tox container, name of OpenShift projects, and more. I thought that that would decrease maintenance when these values change. However, I have not taken into account that some of the projects use forking workflows, and I end up with more than 20 variables that were stored at the team level. When a fork is made, it is usually created in the developer's space, which means that the secrets are no longer available. That usually makes the pipeline unusable. 20 secrets are too much to be copied every time someone makes the fork. Because of that, I had to redesign the pipeline, so the team secrets are not used to store the value of the shared variables and then update already existing pipelines.

11.2 PEP8 and Linting

From the beginning, the team had a policy that all code has to be PEP8 compliant, but there was no policy on how to one should check the code and manual checks are tedious and ineffective. We had to decide if we are going to use Pylint, Flake8, and Autopep and how what should be the settings of the linter, e.g., not everybody in the team thinks that 80 characters per line are worth enforcing.

We decided to use Flake8 as the main linter. Pylint is the more popular linter, but it is sometimes too strict, and some of the things reported by Pylint does not add value (e.g., too long names). We are not using the Flake8 with out-of-box configuration. We ignore a few errors that are more a matter of opinion than actual good practice, and we changed the maximum length of the line to 100 characters instead of 80 characters. We also use a few modules that add additional checks, e.g., a module that checks if every function has a docstring and module that checks if naming conventions are adhered to.

11.3 Testing Frameworks and Coverage

We have a policy that all of the code has to be tested at least with unit-tests. Similarly to the PEP8, we did not have policies about which particular framework should be used to create and run those tests and what testes exactly means (i.e., coverage).

Firstly I looked at the testing frameworks. I found out that we use Unittests and Pytests frameworks. Most of the projects use Unittests because they were most prevalent when the engineers learned Python and Unittests are usually featured in beginners tutorials. Pytest is more Pythonic and is slowly becoming de facto standard. Even though I think that Pytest is better, the idea of rewriting our test base was not very appealing. We had many debates about it, and in the end, we decided to go with the Solomons choice. In documents, we recommend using Pytest for new and significantly refactored project, but we provide instructions on how to continue using Unittests.

We were more successful in deciding on the minimal coverage of our projects. One of the problems with coverage is that if it is too low, the code does not have to be adequately tested, and if it is too high, developers start creating useless tests to achieve it, thus doing useless work and possibly creating technical debt. In the end, we dedicated to set minimal coverage at 60% line coverage. Most projects if tested properly should achieve 60% coverage easily.

We created examples on how to set up the testing framework and CI/CD pipeline to check and report on the coverage. Thanks to this pipeline fail if minimal coverage is not achieved. Current coverage is also shown in the Merge request, so reviewers can see if the coverage has changed significantly. We also found a way how to add badges that show the current level of coverage on the main page of the repository in GitLab UI.

11.4 Tox

One of the first thing that we decided to solve was lack of the unification when it comes to the way how the code is tested and checked. One of the requirements was that the developers have to be able to run most of the tests and check on their machines. We also wanted to find a solution which is easy to set up both in the CI pipeline and on the developer's machine.

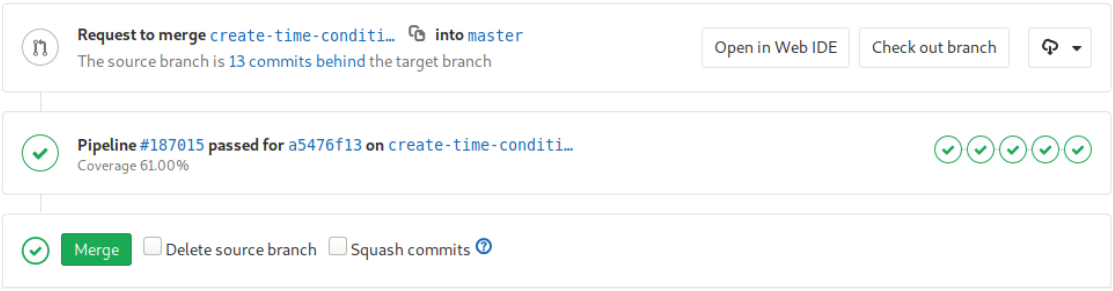

Figure 11.2: Result of the latest run of the pipeline that is based from branch in a merge request. If the pipeline fails, the merge buttons becomes red and the developer will be advised to fix problems that caused the failure of the pipeline run.

We decided to go with Tox because of several reasons. The first reason is that Tox is almost a standard in Python development. Thanks to that, there are many sources on the internet about how Tox should be used for different purposes.

The second reason was the virtual environments that are used by the Tox. That had two significant advantages for us. The first one is that the developer does not pollute his environment with dependencies, linters, testing frameworks. The second advantage that is maybe more important for us is the ability to install dependencies without the root or user permissions. That is very important for our Pipelinew, where the Tox runs inside a container with a random user ID, which makes installation of dependencies difficult (no root access).

After we decided that we are going to use Tox for most of our projects, I created a pre-made configuration file and added it to the Project Template, so the migration to the Tox is fast and straightforward, both for the already existing and new projects.

11.4.1 Tox Image

We use containers to create environments for the jobs. These containers do not have root privileges thanks to runners being in OpenShift, I had to create a Tox container that can be used to run Tox in our pipelines with everything (e.g., bindings for Python libraries) already installed.

Before I created a main container for the team, there already were 5 or 7 containers for the specific projects, which is not ideal. After a closer look, I discovered that most of the stuff that could not have been installed using Tox (e.g., C libraries used by Python and Pip) are installed in multiple containers. Because of that, it was relatively easy to create a Tox container that can be used by almost all the projects. If the project needs something very specific, the developers have to create special Tox container, but that has been needed only once.

The general Tox container is rebuilt every week and after some changes have been merged to master. I created sanity checks, where I test if Tox can install libraries and run tests on a code mockups to prevent the break of the Tox container after automatic update.

11.5 Project Template

The amount of effort I have needed to implement the pipelines was too big for one human. Taking into account the unified nature of our codebase, I have decided to create a Project Template that contains everything needed (instruction, snippets, examples) to use the CI/CD pipelines. It should be reasonably comfortable to add pipelines and be policy compliant thanks to the template.

11.5.1 Directory Structure

One of the first things we decided to unify was the directory structure of our projects. Most of the directories had a standard structure, but to make examples in the Project Template simpler, we decided to create a recommended structure that especially new projects should use. This new directory structure can be seen in Figure 11.3.

One significant change was that we decided to move JJB job definitions and Dockerfiles used to create Jenkins slave images to the repository and not store them in the rad-jenkins repository. We used the rad-jenkins repository to store all our Jenkins job definitions and Dockerfiles that are used to create Jenkins slave container images. The repository was created to make a recovery in case of the Jenkins master loss easier. However, the setup did not support any form of CI/CD, because figuring out what to test and rebuilt was non-trivial if not impossible. We did not remove the rad-jenkins repository entirely. I had an idea to use git submodules to link project repositories to the rad-jenkins repository so that we can keep the rad-jenkins for disaster recovery.

11.5.2 Instructions

The Project Template contains a lot of snippets, instructions, and examples that make set up and update of the pipelines and project as a whole more comfortable and more effort free. The Project Template is continuously updated to comply with the new policies, e.g., new policy about minimal code coverage is mirrored in the tox.ini example.

The content of the template can be divided into two parts, necessary and advanced (optional). The necessary part contains instructions and files needed for compliance with the team policies, e.g., tox.ini or basic GitLab CI pipeline with style and unit-test checks.

The advanced content contains nice to have features, e.g., part of the pipeline that implements continuous deployments or more thought style checks.

11.6 Containers

Even though almost all of our software runs in containers, we had no policies or processes that would solve the problems of container image metadata, the image builds and rebuilds, which tags should be used when and more. This state causes a lot of problems and inefficiencies described in the previous chapter and was one of the most important things to solve.

11.6.1 Container Metadata

Most of our container images used only the description, maintainer and summary metadata fields. That means that nobody had an idea from which commit the container image is based on, when the container was built and by whom.

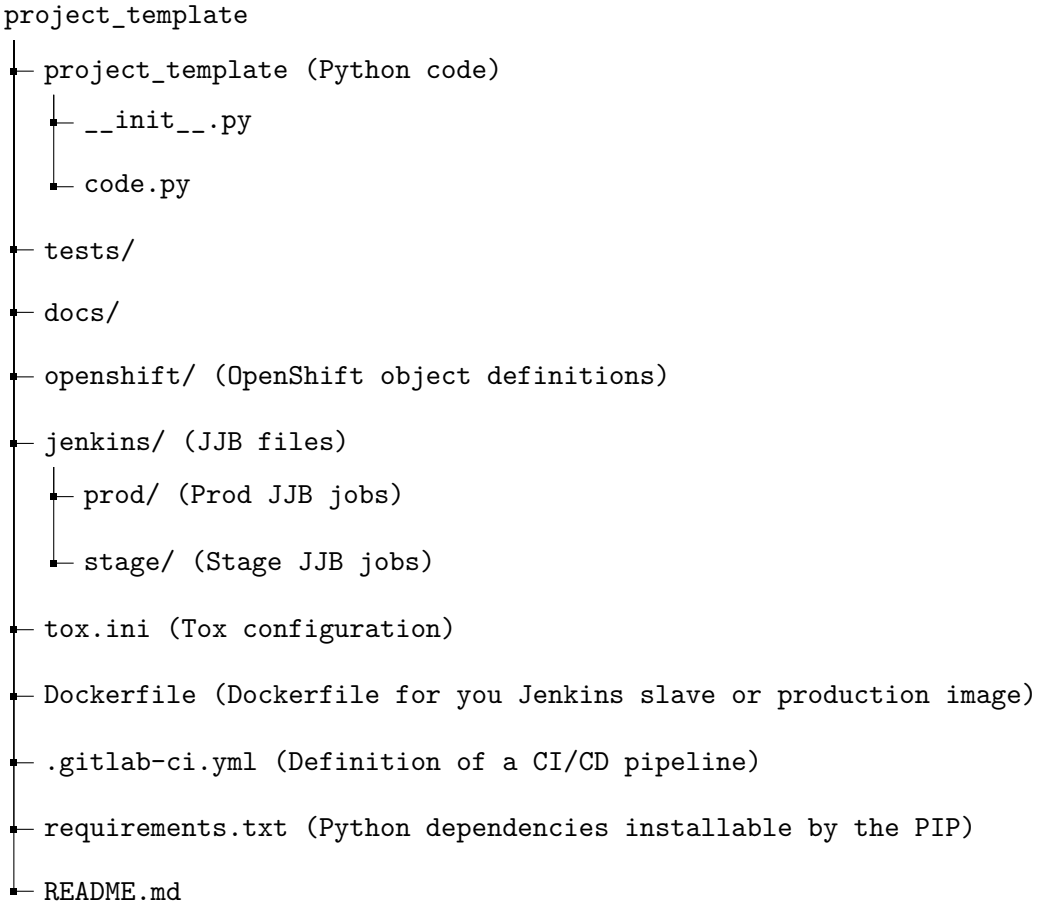

Figure 11.3: Recommended directory structure defined by the Project Template that should unify how the repository looks, which makes the setup of CI/CD pipeline, various checkers easier.

```
...
ARG GIT_COMMIT=unknown
ARG BUILD_DATE=unknown
...

LABEL maintainer="Adam Ormandy <aormandy@redhat.com>" \
      vcs-ref=$GIT_COMMIT \
      build-date=$BUILD_DATE \
      io.k8s.description="Probe of Semaphore service health monitoring
      ...
```

Figure 11.4: Snippet shows how ARGs can be used to set value of the image labels at the build.

I looked around for some guidance, however, I was not able to find a definitive answer on the container image metadata or some universal best practice. Therefore, I decided to go with the Project Atomic [35] metadata policy. Not all of the labels are of use to us (e.g., signatures), but most of the add great informational value (e.g., commit ID, source repository URL).

I had to solve a problem of how to be able to set a value of some metadata labels at build, e.g., commit ID and build time. My solution uses build ARGs to set these values of the metadata labels. If the value of the ARG is not set, the value defaults to unknown. That is done to make building container images on a developer's machine easier.

11.6.2 Container Image Build Engine

I had to find a way how to build our images and not use developers machines to do that. The solution had to:

- work behind Red Hat's firewall,
- require little or no maintenance, and
- be usable by a script.

The firewall limited our options quite significantly. We couldn't use public options like Quay.io [43] or public Docker cloud, because these solutions would not be able to fetch data from internal repositories and sources. That limited us to options that are already deployed internally or are easy to deploy internally, in our case OpenShift cloud [42] or machine with Docker [18] or Podman [16] installed on it.

If we choose the Docker/Podman solution, we have to find a place where these machines are hosted, that would be probably OpenStack cloud. Then we have to configure them and make them accessible for the GitLab runners. GitLab runner can fetch repository and job it is supposed to do by itself, but we would have to set up monitoring of some kind, so we know when we need to fix the runners.

On the other hand, if we choose OpenShift, we have to figure out how to set up build and start the build. We also have to figure out all the service users and their privileges, so everything works smoothly. The main advantages are that we can use API instead and

pre-made objects in YAML and we do not have to monitor and fix OpenShift ourselves, because that is the job of another team.

The OpenShift option involved less work (no machine configuration, monitoring) and the OpenShift cloud we could use has enterprise-level support, we chose to go with it as our main container image build engine.

Integration with CI/CD

I had to find a way, how the trigger OpenShift builds from GitLab CI. We could have set up OpenShift build in a way, that it would be triggered after every push and merge, but that would take control out of the GitLab CI pipeline and cause an unnecessary build (e.g., pipeline run fails before build phase because of PEP8 errors).

The other option and the option we have chosen is to use OpenShift CLI to the trigger job from the GitLab CI pipeline. I have created a template with a BuildConfig, which creates a parametrized build in an OpenShift that makes the creation of the builds in OpenShift easier. There is a copy of the BuildConfig template in every repository that has automatic container rebuilds. That increases the amount of the replicated code, but makes every repository an independent entity (because everything needed for pipeline runs is in the repository), which can be useful when the project is being open-sourced or handed over to other teams.

In the first iteration, engineers had to create a build in the OpenShift manually. The rationale behind that decision was that the builds do not have to be created often, the pipeline starts a build and that makes a pipeline configuration simpler.

The decision to manually create a build in OpenShift has proven to be a bad one after 1 or 2 months in several ways. The first problem was that engineers forgot to create the build or configurated it badly. The second issue was that sometimes people forgot to update build with new settings. The final and the main issue was that it did not work with the forking workflows. The pipeline in the engineer's fork started a build for a commit that was not in the main repository. Thanks to that the build failed.

Thanks to the reasons described above, the manual creation of the build in OpenShift did not scale and work for us. In the second iteration, I decided to update the pipeline to recreate build before starting a new build. That keeps the build setting consistent with the repository settings and adds support for the forking workflow because the repository URL can be changed before the build is started.

After those issues have been resolved, we noticed that the builds take too long if the project is under development. I found out that the main problem was that only one image could have been built at one time per one BuildConfig. To improve the situation, in the third iteration, I updated the pipeline and BuildConfig template to support concurrent builds. The concurrent build had not supported, because the OpenShift pushed the new images into container registry with the tag latest. That allowed a race condition, where if the build that started later but finished before another one could be overridden with another image, meaning that the rest of the pipeline would test the wrong image with the wrong code. Unfortunately, the OpenShift does not allow to set the tag of the resulting image pipeline starts (tag is set in the BuildConfig). To solve that, I used that the pipeline creates or updates BuildConfig before every run. I updated the pipeline, so during the BuildConfig update, the tag is changed to the commit ID of the commit that is going to be used as a source for the new image and the image is tagged as a latest after the image is built under the unique tag. There is a slim possibility that the BuildConfig can be updated between

```
script:
  # Use template to create BuildConfig with the right parameters
  - oc process -f openshift/templates/container-build.yaml
    -p BUILDCFG_NAME=$BUILD_BUILDCONFIG
       GIT_REPO=$CI_PROJECT_URL.git
       DOCKERFILE_PATH=$DOCKERFILE_PATH
       SOURCE_SECRET_NAME=$BUILD_SOURCE_SECRET_NAME
       PUSH_TARGET=$CONTAINER_PATH:$CI_COMMIT_SHA
       PUSH_SECRET_NAME=$BUILD_PUSH_SECRET_NAME > buildconfig
  # Update or create BuildConfig with proper configuration
  - oc replace -f buildconfig || oc create -f buildconfig
  # Start a new build
  - oc start-build $BUILD_BUILDCONFIG
    -o name
    --commit $CI_COMMIT_SHA
    --wait
    --build-arg GIT_COMMIT="$CI_COMMIT_SHA"
    --build-arg BUILD_DATE="$(date --rfc-3339=seconds)"
    --build-arg GIT_URL="$CI_PROJECT_URL"
    --build-arg AUTHORITATIVE_SOURCE="$REGISTRY_IMAGESTREAM_SERVER_NAME"
    --build-arg VERSION="$CI_COMMIT_TAG" > output
  - cat output | grep -oE '[0-9]+$' > build-id-$BUILD_BUILDCONFIG.txt
  # Create latest dev image
  - oc login --token="$REGISTRY_TOKEN" "$REGISTRY_SERVER"
  - oc project "$REGISTRY_PROJECT"
  - oc tag $REGISTRY_IMAGESTREAM_DEV:$CI_COMMIT_SHA
      $REGISTRY_IMAGESTREAM_DEV:latest
```

Figure 11.5: Snippet showing a build section used to build images in OpenShift. In the first part, the BuildConfig template stored in repository is used to create or update BuildConfig in the OpenShift. After that, the container image build itself is started and after the build is finished, the new image is tagged as a latest image.

the moment the BuildConfig has been updated and the build is started, but that takes less than 2 seconds, so the chances of that happening are slim. After the third iteration have been deployed to production, the run durations have been reduced, in some cases from 30 plus minutes to 20.

11.6.3 Retention Policy

One of the new problems caused by the existence of the pipeline is that we produce a significant amount of container images. Some of them have 1 GB or more, and if during a month of development you can create more than a hundred images, that could become a problem.

Even though we have not been contacted by the container registry maintainers yet, we decided to go with a proactive approach. We had long discussions about how to solve this

issue and if both development and production imagestreams are going to be affected by the retention policy.

In the end, we have agreed on this policy:

- Images from development imagestream has to be removed periodically (e.g., weekly).
- No image from the development imagestream can be more than two weeks old.
- Only exception to this rule is the image tagged as latest.

To achieve the goals of the retention policy, I created a Jenkins job that runs every week and removed old images.

11.6.4 Jenkins Slave Base Images

To use the environment as a Jenkins slave binary and other necessities are needed. Due to our heavy usage of containerized Jenkins slaves, we create base images with that already installed. Before the work described in this thesis, we used around 5 or 6 different images as a base image for Jenkins Slave containers. To keep these base images up to date and working is an important priority.

Before I started working on this, the base images were updated only when someone needed something new. That means that the base images could be several months old. That meant that engineers created these images on their machines and that nobody tested the images before the push. To change that, I have created a repository where sources for these images reside and pipeline that can rebuild and sanity check these images (pipeline checks if basic Jenkins job can run inside the new container). I also set up automatic weekly rebuilds, so the product images should not be more than a week old.

I also changed the way how we use Fedora to create Jenkins slave base images. Before, every major version of Fedora image had its imagestream and nobody deleted these imagestreams after the version ceased to be supported. That meant that our Fedora-based Jenkins slaves could have used 3 or 4 different versions of Fedora and then if somebody forgot to update Dockerfile, the project could use Fedora 24 base image. To stop this from happening and reduce the amount of maintenance work, I merged all the Fedora container images into one image.

Chapter 12

Evaluation

The most visible impact of my work was an increase in usage of CI/CD in our projects, as seen in Figure 12.2. Before I started my work, we used no or crude pipelines, and only 6 or 7 percent of our projects used GitLab CI/CD pipelines. Now around 50 percent of the projects that use Python, Jenkins, or containers, utilize my CI/CD pipelines. New projects and almost all projects that are actively developed utilize pipelines described by this thesis. Most of the projects use CI pipelines and deploy to production manually, but around seven projects use CD to deploy code into production. Around 80 percent of the projects that do not use CI/CD pipelines are archived or have not been updated in more than a year.

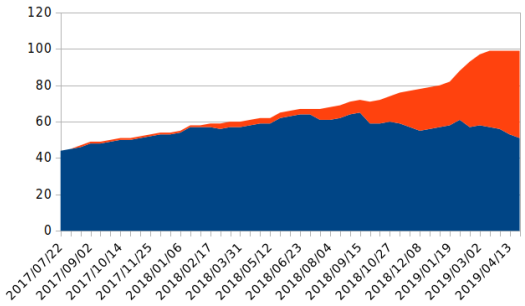

Figure 12.1: Usage of the GitLab CI pipelines by the projects that use Python, Jenkins, or containers. Orange are projects that use pipelines. Blue are projects that do not use the pipelines.

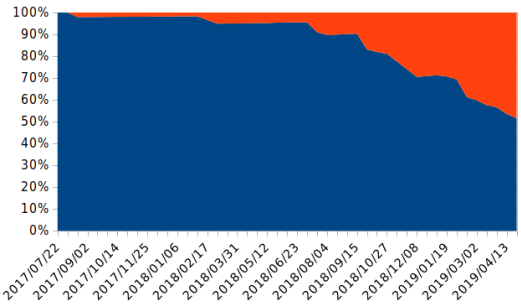

Figure 12.2: Percentage of projects that use the GitLab CI pipelines and Python, Jenkins, or containers. Orange are projects that use the pipelines. Blue are projects that do not use the pipelines.

One of the visible changes in the GitLab UI is the presence of the green arrow and coverage (can be seen in Figure 11.2) in almost every merge request. That not only makes sure that broken code is not merged by accident, but it also saves the reviewer's time, due to not having to run PEP8 checks and unit-tests manually. Projects also have badges in their description showing the status of the last pipeline that ran on master and the current coverage in the master branch. That helped to increase the visibility in our projects.

Something that I have not foreseen was increased removal of old projects. Due to the effort required to set up pipelines and because I reminded some developers that the projects exist, they had to think about, if the effort was worth it. That lead to the abandonment and subsequent cleanup of unused Jenkins jobs, container images, and whole projects.

Our Tox adoption rate has also risen quite a bit, from 5 percent to almost 40 percent, as shown in Figure 12.4. Engineers mostly used my Tox configuration from the Project Template, which makes future fixes and enhancements of the Tox configuration easier to apply, and it unifies the way the projects are tested. The reasons why most of the projects do not use Tox are similar to the reasons why most of them do not have pipelines. They are either archived or in legacy mode.

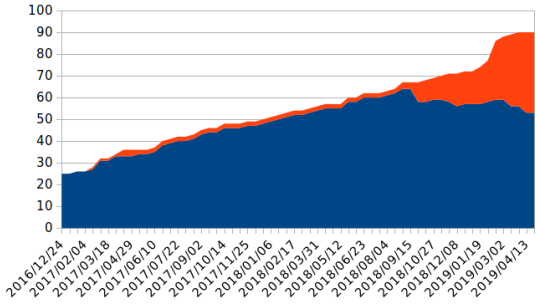

Figure 12.3: Python projects that use Tox. Orange are projects that use Tox. Blue are projects that do not use Tox.

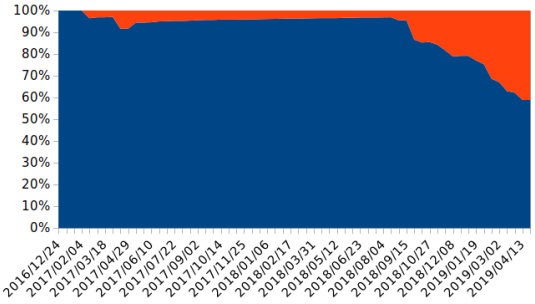

Figure 12.4: Percentage of Python projects that use Tox. Orange are projects that use Tox. Blue are projects that do not use Tox..

The increasing trend of PEP8 violations has been stopped. The amount of PEP8 violations is decreasing or is stable, as shown in Figure 12.5, even though the size of the codebase has increased. Amount of PEP8 violations in projects with Tox and CI/CD pipelines also has a decreasing trend. Legacy and archived projects account for most of the PEP8 violations, and there is little chance that they are going to be fixed soon.

I was surprised by the amount of PEP8 violations in the projects with Tox and pipelines. I expected peaks in the amount of PEP8 violations because people tend to fix these issues. After a closer look, I found out that the violations have been contained in files, which were excluded from the scan. For example, the Tox scanned for violations (directory src) in library and tests (directory tests) but not in the scripts (directory bin). That pointed out an issue that is going to be addressed in a future version of the pipelines and Tox configuration templates.

Currently, the pipelines rebuild around 37 container images, which is a significant increase from three automatically built images at the beginning of September 2019. The automatic rebuilds caused several interesting developments and improvements. The first change is a decrease in the image age. Before the pipelines were implemented, the images could be several months old. Now, the average image is around one and a half or two old. That means that our image has decreased the chance that they container old and dangerous versions of dependencies.

The second impact of the rebuilds is that we check every one or two weeks, that we can rebuild the image. That already showed to be a great thing, when one of the dependencies used by multiple projects has been updated. That update added a new requirement for a new Python C binding package. That meant that we were not able to rebuild the images from old Dockerfiles. Because multiple projects failed at once, the fix was quickly discovered by one developer and shared to other developers saving their time and the capability to rebuild the images has been restored. If the rebuilds have not been in place, the container image build failures would have manifested for several months, and everybody would have to find the solution by themself.

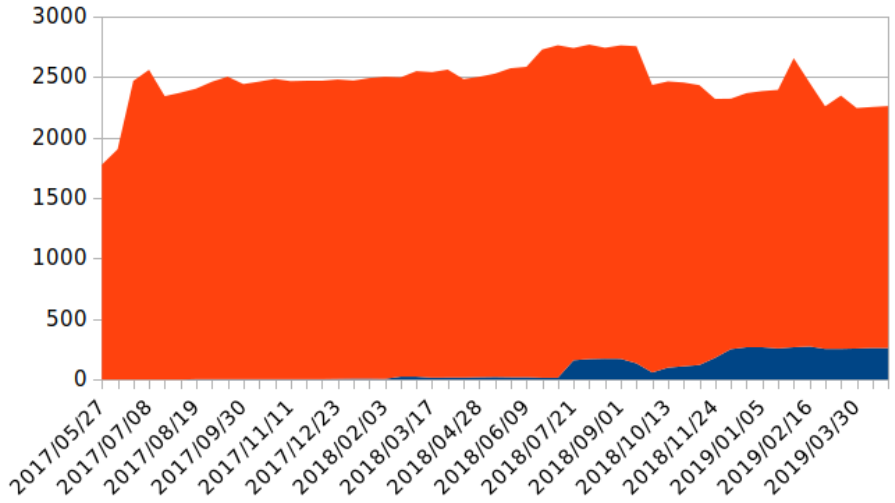

Figure 12.5: Amount of PEP8 violations in the projects. Violations in projects with CI/CD pipeline are orange. Violations in projects without pipelines or with pipelines not based on the ones described in this thesis are blue.

The third improvement is the presence of relevant metadata. Before automatic rebuilds, the images had only rudimentary metadata, such as summary and description. That meant that sometimes it was impossible to find out from which commit the image has been built from. Now, all the images rebuilt by the pipeline have metadata with information, from which commit, from which repository and when they have been built, along with some other metadata.

The fourth improvement is that imagestreams contain more than the latest image and the good practice of having multiple images to rollback to is enforced. Currently, only legacy projects do not comply with the policy of multiple images in the imagestream policy.

Another improvement caused by the automatic rebuilds is the move of the dependency lists from README to requirements file. That is caused by my recommendation to do so in the Project Template and because the snippets and examples are also showing that style of requirements lists handling as recommended.

The sixth and final change that was caused or enabled by automatic pipeline rebuilds is a change in how dependencies and code of Jenkins job container images are handled . Prior pipelines, most developers installed dependencies into container image during a build and then fetched the code from a repository when the job started, thus creating a dependency on the repository and not allowing for the assertion, that the code can run in the container (container image creation and code development are decoupled). Because the pipelines make the rebuilds effortless and easy, more Jenkins job-based projects started to put code into containers during the builds, removing the repository dependency and allowing for the assertion that the code works in the container. I know about seven projects that use this approach, which is six more than when I started working on the pipelines.

The final significant impact of the new approach described in this thesis is the slow destruction of `rad-jenkins` repository. My Project Template expected for the Jenkins job definitions and Dockerfiles to be stored in projects repository. That meant that the two primary use cases for `rad-jenkins` repository vanished. The amount of Jenkins job definitions in `rad-jenkins` has decreased. Maintainers removed some of them after they

found out that the jobs are no longer used and 18 projects moved their job definitions and Dockerfiles into their repositories and are present in `rad-jenkins` as Git submodules. Their migration of the jobs is still not finished, but the trend looks promising. I do not think that all jobs are going to be migrated, because a lot of the jobs left in the `rad-jenkins` repository are legacy and not worth the effort.

Chapter 13

Conclusion

I, with the help of the team, have been able to achieve the goals of the thesis, unification, and automation of testing and development processes. Around 50 percent of the projects that use Python, Jenkins, or containers, utilize my CI/CD pipelines. Around 40 percent of the Python-based projects use Tox. Policies are enforced automatically, which can be seen at the reduction in the amount of PEP8 violations. Containers are built automatically and rebuilt in weekly or biweekly fashion. Most importantly, the engineers redirected their time and energy from repetitive and mundane tasks to more creative and complex tasks.

My work has the most impact on new and under development projects. However, the impact on the projects that are in maintenance should not be understated. We are now sure that we can rebuild our container codebase in case of catastrophic failure and that they adhere to our standards. In some instances, problems that have been forgotten about have been brought to light thanks to the pipeline effort.

When I asked my colleagues for their opinions on my CI/CD work, their reactions have been positive. Even though the early adoption has not always been easy and without mistakes, they say that the pains have been worth it. On the other hand, in their opinion there is space for improvement. Two areas that need to be worked on according to them are forking workflows and new technologies (e.g., Ansible support). I want to work on these areas in the future iterations of the pipelines and Project Template.

One of the things I want to improve personally is optimization by the integration of new features from GitLab CI 11. GitLab CI version 11 was deployed internally in the early May of 2019, and I have not had time to include new features into the Project Template yet. The features have the potential to improve the performance of the pipelines, e.g., a pipeline rebuilds a container image only if there is a change in the source code.

There is already talk about the next phase of the CI/CD and Project Template development that could improve upon and fix some issues of the current versions. The exact scope and timeframe of this next phase are not yet decided, but the phase could start in the summer or September 2019. Even if the next phase is not going to happen, I and some of my colleagues want to opensource parts of our solution. Our Tox image is already available to the public, but we want to share even more.

Bibliography

[1] Abbe98: Testing Pyramid. Accessed: 2018-12-30.
Retrieved from:
https://commons.wikimedia.org/wiki/File:Testing_Pyramid.svg

[2] Alphabet Inc.: Google Python Style Guide. Accessed: 2019-1-14.
Retrieved from: https://google.github.io/styleguide/pyguide.html

[3] Alpine Linux Development Team: Alpine Linux. Accessed: 2019-1-16.
Retrieved from: https://alpinelinux.org/about/

[4] Ambler, S. W.: *Process patterns: building large-scale systems using object technology*. Cambridge University Press. c1998. ISBN 05-216-4568-9.

[5] Atlassian Corporation Plc: Attlasian Git tutorials. Accessed: 2018-11-19.
Retrieved from: https://www.atlassian.com/git/tutorials

[6] Benevides, R.: 10 things to avoid in docker containers. Accessed: 2018-12-31.
Retrieved from: https://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers/

[7] Bicking, I.: Virtualenv. Accessed: 2019-1-9.
Retrieved from: https://virtualenv.pypa.io/en/latest/

[8] Binnie, C.: Avoid Using Lazy, Privileged Docker Containers. Accessed: 2018-12-31.
Retrieved from: https://www.linux.com/blog/learn/sysadmin/2017/5/lazy-privileged-docker-containers

[9] Brandl, B. and the Sphinx team: reStructuredText Primer. Accessed: 2019-1-14.
Retrieved from:
http://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html

[10] Brandl, B. and the Sphinx team: Sphinx overview. Accessed: 2019-1-14.
Retrieved from: http://www.sphinx-doc.org/en/master/index.html

[11] Budgen, D.: *Software design*. Addison-Wesley. second edition. 2003. ISBN 02-017-2219-4.

[12] Chacon, S.: GitHub Flow. 2011.
Retrieved from: http://scottchacon.com/2011/08/31/github-flow.html

[13] Chacon, S.: *Pro Git: everything you need to know about the Git distibuted source control tool*. Apress. Second edition edition. 2014. ISBN 978-1484200773.

[14] Christensen, S.: Git Workflows That Work. Accessed: 2018-12-10.
Retrieved from:
https://www.endpoint.com/blog/2014/05/02/git-workflows-that-work

[15] git-scm.com community: Main page of Git project. Accessed: 2018-11-19.
Retrieved from: https://git-scm.com

[16] Containers organization: Podman. Accessed: 2019-5-5.
Retrieved from: https://podman.io/

[17] Cordasco, I. S.: flake8-docstrings - Pypi. Accessed: 2019-1-14.
Retrieved from: https://pypi.org/project/flake8-docstrings/

[18] Docker, I.: Docker Documentation. Accessed: 2018-10-05.
Retrieved from: https://docs.docker.com/

[19] Duvall, P. M.; Matyas, S.; Glover, A.: *Continuous integration*. Upper Saddle River, NJ: Addison-Wesley. first edition. c2007. ISBN 978-0321336385.

[20] Fedora Project: Fedora. Accessed: 2019-1-16.
Retrieved from: https://getfedora.org/en/

[21] GitLab Inc.: GitLab Documentation. Accessed: 2019-5-5.
Retrieved from: https://docs.gitlab.com

[22] Goodger, D.; van Rossum, G.: PEP 257: Docstring Conventions. PEP 257. Python Software Foundation. May 2001.
Retrieved from: https://github.com/python/peps/blob/master/pep-0257.txt

[23] Guru99: What is Non Functional Testing? Types with Example. Accessed: 2018-12-30.
Retrieved from: https://www.guru99.com/non-functional-testing.html

[24] Jenkins Job Builder maintainers: jenkins-job-builder 2.9.2.dev8 documentation ». Accessed: 2018-12-31.
Retrieved from: https://docs.openstack.org/infra/jenkins-job-builder/

[25] Jenkins maintainers: Jenkins User Documentation. Accessed: 2018-12-31.
Retrieved from: https://jenkins.io/doc

[26] Justin Ryan, D. S. M. S., Andrew Harmel-Law: Jenkins Job DSL / Plugin. Accessed: 2019-5-5.
Retrieved from: https://github.com/jenkinsci/job-dsl-plugin

[27] Krekel, H.; et al.: Full pytest documentation - pytest documentation. Accessed: 2019-1-14.
Retrieved from: https://docs.pytest.org/en/latest/contents.html

[28] Krekel, H.; et al.: Welcome to the tox automation project. Accessed: 2019-1-9.
Retrieved from: https://tox.readthedocs.io/en/latest/

[29] Lehtosalo, J.; et al.: mypy - Optional static testing for Python. Accessed: 2019-1-14.
Retrieved from: http://mypy-lang.org/

[30] McConnell, S.: *Code complete*. Microsoft Press. second edition. c2004. ISBN 07-356-1967-0.

[31] Myers, G. J.; Badgett, T.; Thomas, T. M.; et al.: *The art of software testing*. John Wiley. second edition. c2004. ISBN 04-714-6912-2.

[32] Pearson, L.: The Four Levels of Software Testing. Accessed: 2018-12-30.
Retrieved from:
https://www.seguetech.com/the-four-levels-of-software-testing/

[33] Pellerin, J.: nose. Accessed: 2019-1-8.
Retrieved from: https://nose.readthedocs.io/en/latest/index.html

[34] Pipeline plugin maintainers: jenkins-kubernetes-plugin. Accessed: 2019-5-5.
Retrieved from: https://github.com/jenkinsci/pipeline-plugin

[35] Project Atomic maintainers: Standard Container/Application Identifiers. Accessed: 2019-5-5.
Retrieved from:
https://github.com/projectatomic/ContainerApplicationGenericLabels

[36] PyPA: pip – pip 18.1 documentation. Accessed: 2019-1-10.
Retrieved from: https://pip.pypa.io/en/stable/

[37] PyPA: Python Packaging Authority - PyPA documentation. Accessed: 2019-1-10.
Retrieved from: https://www.pypa.io/en/latest/

[38] PyPA: Python Packaging User Guide. Accessed: 2019-1-8.
Retrieved from: https://packaging.python.org/#python-packaging-user-guide

[39] Python Software Foundation: distutils — Building and installing Python modules. Accessed: 2019-1-8.
Retrieved from: https://docs.python.org/3.8/library/distutils.html

[40] Python Software Foundation: Python. Accessed: 2018-11-25.
Retrieved from: https://www.python.org/

[41] Python Software Foundation: unittest - Unit testing framework. Accessed: 2019-1-8.
Retrieved from: https://docs.python.org/3.8/library/unittest.html

[42] Red Hat, Inc.: OpenShift Container Platform 3.9 Documentation. Accessed: 2018-11-19.
Retrieved from:
https://docs.openshift.com/container-platform/3.9/welcome/index.html

[43] Red Hat, Inc.: Quay. Accessed: 2019-5-5.
Retrieved from: https://quay.io/

[44] Red Hat, Inc.: What is Docker? Accessed: 2018-12-31.
Retrieved from: https://www.redhat.com/en/topics/containers/what-is-docker

[45] van Rossum, G.; Lehtosalo, J.; Langa, L.: PEP 484: Type Hints. PEP 484. Python Software Foundation. September 2014.
Retrieved from: https://github.com/python/peps/blob/master/pep-0484.txt

[46] van Rossum, G.; Warsaw, B.; Coghlan, N.: PEP 8: Style Guide for Python Code. PEP 8. Python Software Foundation. July 2001.
Retrieved from: https://github.com/python/peps/blob/master/pep-0008.txt

[47] Sanchez, C.: jenkins-kubernetes-plugin. Accessed: 2019-5-5.
Retrieved from: https://github.com/jenkinsci/kubernetes-plugin

[48] Sharma, T.; Samarthyam, G.; Suryanarayana, G.: Pragmatic Technical Debt Management. Accessed: 2018-12-28.
Retrieved from: https://www.infoq.com/articles/pragmatic-technical-debt

[49] Smith, P.: GNU Make. Accessed: 2019-1-9.
Retrieved from: https://www.gnu.org/software/make/

[50] Sommerville, I.: *Software engineering*. Addison-Wesley. 8 edition. 2007. ISBN 978-0-321-31379-9.

[51] Suryanarayana, G.; Samarthyam, G.; Sharma, T.: *Refactoring for software design smells: managing technical debt*. Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier. [2015]. ISBN 978-0128013977.

[52] Techopedia Staff: What is Code Coverage? - Definition from Techopedia. Accessed: 2018-12-30.
Retrieved from: https://www.techopedia.com/definition/22535/code-coverage

[53] Techopedia Staff: What is Functional Testing? - Definition from Techopedia. Accessed: 2019-05-16.
Retrieved from:
https://www.techopedia.com/definition/13296/software-engineering

[54] Techopedia Staff: What is Functional Testing? - Definition from Techopedia. Accessed: 2018-12-30.
Retrieved from:
https://www.techopedia.com/definition/19509/functional-testing

[55] Techopedia Staff: What is technical debt? - Definition from Techopedia. Accessed: 2018-12-30.
Retrieved from: https://www.techopedia.com/definition/27913/technical-debt

[56] Thénault, S.: Pylint. Accessed: 2018-11-24.
Retrieved from: https://github.com/PyCQA/pylint8

[57] Tutorials Point (I) Pvt. Ltd.: Software Testing - Quick Guide. Accessed: 2018-11-24.
Retrieved from: https://www.tutorialspoint.com/software_testing/software_testing_quick_guide.htm

[58] Watkins, S.: The Causes Of Technical Debt Do Not Exist In A Vacuum. Accessed: 2018-12-30.
Retrieved from: https://www.castsoftware.com/blog/the-causes-of-technical-debt-do-not-exist-in-a-vacuum/

[59] Wikipedia contributors: Best practice. 2001-. accessed: 2018-11-24.
Retrieved from: https://en.wikipedia.org/wiki/Best_practice

[60] Wikipedia contributors: Ninety-ninety rule. 2001-. accessed: 2018-11-24.
Retrieved from: https://en.wikipedia.org/wiki/Ninety-ninety_rule

[61] Wikipedia contributors: OpenShift. 2001-. accessed: 2018-12-30.
Retrieved from: https://en.wikipedia.org/wiki/OpenShift

[62] Wikipedia contributors: Platform as a service. 2001-. accessed: 2018-12-30.
Retrieved from: https://en.wikipedia.org/wiki/Platform_as_a_service

[63] Wikipedia contributors: Python. 2001-. accessed: 2018-11-24.
Retrieved from:
https://en.wikipedia.org/wiki/Python_%28programming_language%29

[64] Wikipedia contributors: Singleton. 2001-. accessed: 2018-11-24.
Retrieved from: https://en.wikipedia.org/wiki/Singleton_pattern

[65] Wikipedia contributors: Software testing. 2001-. accessed: 2018-12-30.
Retrieved from: https://en.wikipedia.org/wiki/Software_testing

[66] Wikipedia contributors: Technical debt. 2001-. accessed: 2018-12-30.
Retrieved from: https://en.wikipedia.org/wiki/Technical_debt

[67] Ziadé, T.; Cordasco, I.: Flake8. Accessed: 2018-11-24.
Retrieved from: https://gitlab.com/pycqa/flake8

Appendix A

Content of Attached Media

The attached medium consists of these folders:

container-first-pipeline-verifier - Example of the repository with CD pipelines, container images, Jenkins jobs, and Python.

jenkins-slave-base-container - Base container images for Jenkins job containers.

oc-container - Container with OpenShift CLI client that is used by the GitLab CI pipelines.

project-template - Project template repository.

tox-container - Public version of the container with Tox that is used by the GitLab CI pipelines.

upshift-gitlab-runner - OpenShift template used to create runners for GitLab CI in OpenShift. This repository was not created or developed by me.

jjb-container - Container with Jenkins Job Builder that is used by the GitLab CI pipelines.

semaphore-probe - Example of the project with CD pipelines, container images, and Python.

tox-container-internal - Container with Tox that is used by the GitLab CI pipelines.

violation-report - Python tool used to gather data about the codebase.