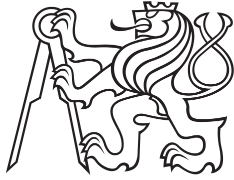
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computers

Continuous Integration of web therapeutical application

Vít Říha

Supervisor: doc. Ing. Daniel Novák, Ph.D.

May 2023

ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Říha** Jméno: **Vít** Osobní číslo: **465826**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Continuous Integration webové terapeutické aplikace

Název diplomové práce anglicky:

Continuous Integration of web therapeutical application

Pokyny pro vypracování:

The topic of the work is a existing web application that enables
management of therapeutical programme. The application runs on Django python web framework with PostgreSQL as database and Huey/Redis to providing asynchronous multithread processing and scheduled tasks.
1. Study and analyze existing therapeutical application.
2. Optimize scheduling algorithm in Rust enviroinment
3. Implement and configure the resulting optimization
4. Test the best framework on the real environment containing at least
5000 users

Seznam doporučené literatury:

[1]Erich, Floris & Amrit, Chintan & Daneva, Maya. (2017). A Qualitative Study of DevOps Usage in Practice. Journal of Software: Evolution and Process. 00.
10.1002/s
[2]M. Shahin, M. Ali Babar and L. Zhu, "Continuous Integration, Delivery and
Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," in IEEE Access, vol. 5, pp. 3909-3943, 2017
[3]Arachchi, S A I B & Perera, Indika. (2018). Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. 10.1109/MERCon.2018.8421965
[4]Sheyyab, Mahmoud. (2019). Managing Quality Assurance Challenges of
DevOps through Analytics.
[5] Khan, Muhammad & Jumani, Awais & Mahar, Farhan & Siddique, Waqas & Shaikh, Asad. (2020). Fast Delivery, Continuously Build, Testing and Deployment
with DevOps Pipeline Techniques on Cloud. Indian Journal of Science and
Technology. 13. 552-575. 10.17485/ijst/2020/v13i5/148983.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Daniel Novák, Ph.D. katedra teoretické informatiky FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **13.02.2023** Termín odevzdání diplomové práce: ______________

Platnost zadání diplomové práce: **22.09.2024**

doc. Ing. Daniel Novák, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání

Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

In the first place, I want to thank my supervisor, doc. Ing. Daniel Novák, Ph.D., for supporting me throughout the entire process of working on this thesis with an overwhelmingly positive attitude. Next, I want to thank Ing. Jindřich Prokop for helping me to get oriented in the project and providing basic technical support related to the original project. I also want to thank my closest family for their support and understanding when working on the thesis. I would like to dedicate this work to my beloved grandmother, who passed away on 11 March 2023.

V první řadě chci poděkovat mému vedoucímu práce, doc. Ing. Danielu Novákovi, Ph.D., za jeho podporu v průběhu celého procesu tvorby této diplomové práce s nesmírně pozitivním přístupem. Dále chci poděkovat Ing. Jindřichu Prokopovi za jeho pomoc zorientovat se v projektu a poskytování základní technické podpory vztahující se k původnímu projektu. Také bych chtěl poděkovat mé nejbližší rodině za jejich podporu a pochopení při tvorbě této práce. Tuto práci bych chtěl věnovat mé milované babičce, která nás opustila 11. března 2023.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 26, 2023

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 26. května 2023

Abstract

Automation is one of the most critical aspects of modern software development practices. CI/CD and DevOps are concepts that enable applying automation throughout the entire development process - from pushing the first commit to delivering a functional product. This thesis covers the processes of CI/CD, the technology that can be used to implement it, and the mentality behind DevOps, as well as technology that supports it. The second part of the paper is dedicated to analysis of an existing implementation of CI/CD on a real project, and depiction of first-hand experience of re-implementing a performance-critical Python module in Rust using PyO3, and integrating it into the existing CI/CD pipeline, with the goal of performance optimisation.

Keywords: CI/CD, DevOps, containerisation, Docker, pipeline, QA, testing, Python, Rust, PyO3

Supervisor: doc. Ing. Daniel Novák, Ph.D.
Na Zderaze 269/4,
120 00 Praha 2 - Nové Město

Abstrakt

Automatizace je jedním z nejkritičnějších aspektů moderního softwarového vývoje. CI/CD a DevOps jsou koncepty umožňující automatizaci napříč cepým procesem vývoje softwaru - od nahrání prvního commitu po dodávku funkčního produktu. Tato diplomová práce pokrývá procesy v CI/CD, technologie, které umožňunjí jeho implementaci, mentalitu za DevOps a technologie, které ho podporují. Druhá část této práce je věnována analýze existující implementace CI/CD na reálném projektu a vylíčení zkušeností z první ruky z reimplementace Pythonovského modulu, který je kritický z pohledu výkonu, v jazyce Rust za použití knihovny PyO3 a jeji integraci do existující CI/CD pipeliny, s cílem optimalizace výkonu aplikace.

Klíčová slova: CI/CD, DevOps, konteinerizace, Docker, pipeline, QA, testing, Python, Rust, PyO3

Překlad názvu: Continuous Integration webové terapeutické aplikace

Contents

1 Introduction **1**

2 Introduction to CI/CD **3**

2.1 Definition and core principles of CI/CD 3
2.2 Benefits of implementing CI/CD in software development 3
2.3 Overview of the CI/CD pipeline and its stages 5

3 CI/CD Tools, Technologies and Best Practices **7**

3.1 Popular CI/CD Tools 7
3.1.1 Jenkins 7
3.1.2 Circle CI 8
3.1.3 Travis CI 8
3.1.4 GitLab 8
3.1.5 Conclusion 8
3.2 Containerisation and Orchestration Technologies 9
3.2.1 Containerisation vs. Virtualisation 9
3.2.2 Docker 9
3.2.3 Orchestration 10
3.3 Infrastructure as Code 10
3.4 Automated testing strategies 11
3.4.1 API Testing 11
3.4.2 GUI Testing 11
3.4.3 Nonfunctional Testing 12
3.4.4 Security Testing 12
3.4.5 Regression Testing 12
3.5 Version control and branching strategies 13
3.5.1 Version control tools 14
3.5.2 Git branching strategies 15
3.6 Code quality checks and static code analysis 19
3.6.1 Code review 19
3.6.2 Static code analysis (SCA) 20

4 DevOps Culture and Technologies **23**

4.1 DevOps and its relationship with CI/CD 23
4.2 Communication and feedback loops in CI/CD processes 24
4.2.1 Reinforcing feedback loop 25
4.2.2 Balancing feedback loop 25
4.2.3 Feedback loops in CI/CD 25
4.3 Collaboration Tools 26
4.3.1 Issue tracking systems 26
4.3.2 Communication Tools 26
4.3.3 Monitoring and Logging Tools 27

5 Practical application of CI/CD on existing project **29**

5.1 Pipeline Configuration 29
5.1.1 Used tools and technologies . 29
5.1.2 Structure and stages of the CI/CD pipeline 31
5.2 Testing and QA 32
5.2.1 Employed testing strategies . 32
5.2.2 Use of code quality checks and static code analysis tools 33
5.2.3 Handling of test failures 33
5.3 Conclusion 33

6 Reworking a Core Module **35**

6.1 Implementation Process 35
6.1.1 Choosing the framework 35
6.1.2 Getting started 36
6.1.3 Common patterns between the original code and Rust implementation 39
6.1.4 Finishing up 40
6.2 Integration into CI/CD 40
6.3 Lessons learned 41

7 Conclusion **43**

Bibliography **45**

Figures

3.1 Virtualisation vs. Containerisation [12] 10
3.2 GitFlow and GitHub Flow strategies 17
3.3 GitLab Flow and Trunk-based strategies 18

4.1 DevOps infinity loop [31] 24

Tables

Introduction

This thesis aims to explore the domains of Continual Integration, Continual Deployment, Continual Delivery and DevOps, describing their interconnection, the tools available to software development teams to simplify internal processes that accompany the work they do every day. Subsequent part of the thesis focuses on an analysis of an implementation of CI/CD on a real project. The last part provides an insight into the experience of implementing a performance-critical Rust module that runs in a Python environment.

The first chapter will introduce the reader to CI/CD and its core principles, provide information of the benefits that developer teams get from taking advantage of CI/CD, and also give an overview of what is CI/CD pipeline and what are its stages.

In the next chapter, readers will be introduced to different popular tools teams can use to configure their CI/CD pipeline, giving perspective on their pros and cons, and what teams could benefit from each tool the most. The chapter will continue with a section focusing on containerisation and orchestration technology, presenting the reader for example with how is containerisation different from virtualisation. It will then focus on a particular containerisation tool - Docker, which is a leading containerisation tool in the industry. The chapter also touches upon orchestration technology, as well as the rise of an alternative approach to building infrastructure in the form of Infrastructure as Code. A big spotlight is given to automated testing strategies, because they are a critical component of a healthy CI/CD process. The next section of the chapter is dedicated to version control systems and several different branching strategies, which can have a big impact on the overall productivity of the team - both positive and negative, followed by a section focusing on internal code review process and static analysis tools.

Chapter number four will deal deal with DevOps culture and related technologies, describing the interconnection between CI/CD and DevOps, how communication is an important aspect of the DevOps mentality and feedback loops present in the CI/CD processes. The last section of the chapter introduces the readers to additional tools that aid teams of developers in achieving higher efficiency, like issue tracking systems, communication tools, and monitoring and logging tools.

The fifth chapter will be dedicated to an analysis of employment of CI/CD

on an existing software project, that will involve describing the pipeline configuration, what tools and technologies were used to implement the CI/CD, as well as an analysis of its pipeline. The subsequent section delves into one of the most important pats of a healthy CI/CD pipeline - test automation. Used testing strategies will be analysed, along with the code coverage of the various modules of the project and handling of test failures. At the end of the chapter, we will look into the possibilities of using static analysis tools in this particular project.

The last chapter will report my experience with rewriting a core, performance-critical module into Rust, an efficient and memory-safe programming language. We will delve into the process of implementation, from choosing the appropriate technology, to the final integration into the project's CI/CD pipeline.

Chapter 2
Introduction to CI/CD

CI/CD is a mechanism used in software development, which greatly simplifies the process of adding new features to the product. It introduces automation into stages of app development, from integration and testing to delivery and deployment.

2.1 Definition and core principles of CI/CD

"CI" in the acronym stands for Continuous Integration, which is a part of the process used by developers. When a new feature goes through CI successfully, it means that the project can be build, tested and merged to the main branch. Where CI ends, CD begins. "CD" can refer to two concepts, which are sometimes used interchangeably, but when used separately, they can ultimately highlight how much automation there actually can be.

One of the meanings is "Continuous Delivery". This denotes the automation of bug testing and uploading the developer's changes to a shared repository, from where the operations team can deploy the changes to the production environment. This provides a minimal-effort solution to new code deployment.

The other meaning "CD" can have is "Continuous Deployment". This usually means automatic deployment of the developer's changes to the production environment. Notice the difference, where in Continuous Delivery, the changes are pushed to production manually by operations team. Continuous deployment takes advantage of the automation of the previous stages and automates the next step in the pipeline as well. [1]

2.2 Benefits of implementing CI/CD in software development

Releasing software is generally a difficult and time-consuming task. Its automation allows development teams to release more frequently, enabling modern coding methodologies, such as agile software development.

The simplification of the process grants tech companies the ability to release changes to their software more frequently. This means that new features can reach the end user sooner, as well as in the event of an overlooked bug, that

might drive potential users away, the fix can be released in time to mitigate the damage. In this age, time is of the essence in creating any product, be it software, or any other type of technology. If competition gets to the market sooner, it might not matter that your product is better, just because users already have what they wanted.

Shorter time to market helps in more ways than only making the product more competitive. Being able to deploy the application more quickly enables a wide range of people to validate the approach and participate in the development process, allowing product managers and other key members of the project's team, as well as external test participants, uncover defects - both design flaws and bugs in the code - which can, in the end, save weeks or event months of moving in the wrong direction.

Faster integration allows for smaller increments on the codebase, which in turn makes reviewing the code much easier, as there are fewer changes and additions to review. Sharing one's changes with the rest of the team also ensures everyone is working on the same foundations, which prevents regression and makes merging the changes easier.

One of the most important parts of the CI/CD pipeline are automated tests. These make sure that changes made by a developer do not break a previous feature, or features, of the software. Each developer is expected to write their own exhaustive tests of any new feature, or more precisely new methods (these are called unit tests). These tests can be run manually to check that the code works as expected. During CI, these are some of the tests that are executed automatically to check that the developer did not break another one's work. This is much faster compared to manual testing, but also has greatly improved chances of detecting breaking changes, because they are much more consistent and reliable. The earlier a bug is discovered, the easier it is to fix the damage, because less of the functionality is dependent on that part of the system. More layers of automated tests include end-to-end or performance tests.

While we are on the subject of bug fixing, smaller releases also mean, that if it so happens that a bug slips through the tests into production, it will be much faster to identify the breaking change. This gives the team options to tackle the issue in several ways. They can for example rollback the change and go back to the last stable version, which is something that can be done almost immediately, while removing little functionality thanks to small increments in the code. Next step, whether the changes have been rolled back or not, is fixing the bug. Because there are fewer changes, the exact place where the code is broken can be found fast and the team can start working on the patch quickly, releasing the fix in the next iteration very soon.

Many tools used for CI/CD provide useful metrics, ranging from build times or defect rates, to test fix times. This information can help the team identify weak spots in the development process, or identify the need for higher capacity of the hardware. [2]

2.3 Overview of the CI/CD pipeline and its stages

A CI/CD pipeline is generally divided into several stages. Each of the stages completes a given task. The primary goal of the pipeline is reducing the risk in the process of deploying changes to the code, by automating the process, making it more consistent. A pipeline should take of compiling and testing the code, fabricating a deployable product and deploying the application to a server.

The first stage of a CI/CD pipeline should be the trigger. A pipeline should start automatically when a new commit is pushed to the main branch on the project repository. The purpose of this stage is making the pipeline truly automatic, in the sense that as soon as the developer is done with a feature and it is pushed to the main branch, the CI/CD process starts without the need for additional interaction, which removes the possibility for the developer to forget to start the process.

The next step is the code checkout. This is where the CI server retrieves the change that triggered the pipeline from the project's repository.

After the fresh code is pulled into the CI server, it typically needs to be compiled. Some programming languages do not need compilation, so they can skip this part of the step. Although, that does not mean they can skip this step entirely. Most projects require some external libraries that are not part of the project, or that need to be installed into the environment for the project to be able to run. Part of this step can also be providing a clean environment, so that there is no interference from other sources. This is where containerisation comes into place.

Now that the code is compiled and we have a fresh environment ready, it is time to run the tests. There is plethora of frameworks for individual languages that the pipeline can use. This stage should not only make sure that the tests pass, but also verify, that test code even exists in the first place, checking the code coverage across the project, ensuring that as the codebase grows, the tests add up as well.

When all the tests pass, the pipeline moves on to packaging the code. Depending on the project, the result of this stage can take different shapes. A modern approach is using Docker, a virtualisation software, that can turn the entire project using different technologies into a single "docker image", that can be deployed to the production server and run seamlessly.

We are nearing conclusion, entering the second last stage of our pipeline. The software is almost ready to be deployed, which means it can run on its own. This is the last opportunity to make sure that the latest changes have not broken a critical component of the software, by running acceptance tests. Again, there are many tools available to automate these tests. This time, compared to previous test run, we usually need to simulate user's interaction with the UI, as opposed to testing outputs of individual methods with specific inputs.

The final step is delivery, or deployment. Not all pipelines have this step automated, and there can be multiple reasons for that. For example, the

production server might not support the feature, or the team has simply agreed that for some reason, the step should be in control of a human, for example to perform further manual testing.[3]

Chapter 3

CI/CD Tools, Technologies and Best Practices

In this chapter, I will focus on different types of tools used for setting up CI/CD and production environment. Deciding for the tools that are right for the project is important and can affect how difficult it is to manage the pipeline, how much it costs and what features the tools should support and provide.

Further, I will target the best practices when it comes to CI/CD and software development. I will go over testing strategies, code quality checks, static code analysis and version control. Following best practices helps keep the project well arranged, secure, and maintainable. The quality of the software reaches higher quality and features are delivered faster, with reduced risk of introducing bugs and flaws, minimising downtime and disruptions. It also helps keeping the project scalable thanks to automation in both infrastructure and deployment processes. Onboarding new members becomes a smoother process thanks to CI/CD pipeline and how simple it is to spin up the developer environment compared to projects without CI/CD.

3.1 Popular CI/CD Tools

Choosing the right CI/CD tool can be difficult, especially with the range of options that are available on the market.

3.1.1 Jenkins

Jenkins is probably the first that should come to mind when considering a CI/CD tool. No licensing fees and it being open-source makes it one of the most popular tools in the industry. It supports all major operating system - Windows, Linux, MacOS, and other Unix-like OS. It has its own ecosystem of plugins, which allows it to "integrate with practically every tool in the continuous integration and continuous delivery toolchain"[4]. Nevertheless, its extensibility can turn into a hurdle, as the tool can become too complex to manage for an average developer, which creates a need for "Jenkins experts", forming bottlenecks and additional costs. [5]

3.1.2 Circle CI

Circle CI is another very popular tool. It supports every stage of the CI/CD pipeline. It offers a free tier for small projects, but if the team needs more than one user, subscription fees start at $15 per month. Although Circle CI offers self-hosted subscription, the default is cloud-hosted solution. The platform supports building applications for Docker, Linux, MacOS, Windows, Arm and GPU heavy computation. It also offers integration with over a hundred other services and platforms, such as Bitbucket, GitHub, GitLab, or Jira. They claim to be 70% faster than other CI/CD platforms, which is likely the source of its popularity. [6]

3.1.3 Travis CI

Travis CI is the most expensive service of this comparison, starting at $64 per month for the lowest tier [7] and free option only for open-source projects. They offer both self-hosted and cloud-based solutions. The platform supports variety of languages and operating systems. Teams have to manage their code in GitHub or Bitbucket, if they choose Travis CI as their CI/CD tool. The tool's main benefit is quick first setup, along with preinstalled build and test tools, and intuitive UI. [8] [9] [10]

3.1.4 GitLab

GitLab is best known for their git version control system (VCS) online repository. Some people might not know they also offer their own CI/CD ecosystem. This makes it easy to configure automated build, integration, testing and deployment on the very same platform where the codebase resides. These advantages come at the price of several disadvantages as well. The most obvious one is dependency the GitLab repository. It also does not support stages within phases and can not generate reports.[9]

3.1.5 Conclusion

The choice ultimately comes down to several considerations. Jenkins is probably the most versatile, configurable option, but at the price of difficulty to configure and the need for self-hosted server. On the other hand, if these are not an issue, the project will save money on licensing fees. All the other options depend heavily on what the rest of the project toolchain consists of. Once the team decides to use a host for their repository, the selection of available CI/CD solutions becomes thinner. Finally, the team should choose the platform that supports the features they will be able to use while considering their pricing.

3.2 Containerisation and Orchestration Technologies

Containerisation is a modern approach to building cloud-native applications. The technology combines all the necessary libraries and the application to create a "container", which can then run on different platforms thanks to virtualisation. In traditional approach, code is developed and initially tested on the developer's platform, which is usually different that the server the code will eventually be running on - even if the developer uses Linux-based OS, they most likely have a different CPU architecture, which means the code needs to be compiled differently, which can introduce inconsistent behaviour between the developer's system and the target server. Containerisation brings a solution to this problem by bundling the needed configuration files, libraries and source code into a single package, that can then run in an environment that is consistent across different platforms, isolated from other software running on the machine. [11]

3.2.1 Containerisation vs. Virtualisation

Virtualisation enables running multiple operating systems on a single machine, which allows for software developed for different systems to work simultaneously on a single physical machine. Typically an application gets bundled together with an operating system, needed libraries and other dependencies into a Virtual Machine (VM). Multiple VMs then run on a single computer. This can save the company a lot of money compared to needing to run a different physical server for each application just because it was made for a different OS, or maybe just a different version of the system. [11]

The key difference in Containerisation is that the container does not include a copy of the operating system, which saves a lot of disk space, as well as lowers the overhead of each running application. In place of the whole system, the container is made to run on a specialised runtime engine, which is installed in the server's operating system. This means, that this engine is assigned resources, which it then distributes between all the containers running in this virtual environment. [11]

3.2.2 Docker

By far the most popular containerisation tool is Docker, with over 80% market share. [13] It is an open source platform, that supports the whole containerisation lifecycle - from building a docker image (the bundle containing all the necessary files to run an app), to running and maintaining the containers. A docker image is defined by its DockerFile, which is a simple text file that specifies the steps necessary to create the docker image from the source files, provide all the dependencies, etc. When a docker image is run, it is referred to as a docker container - the running instance of the application. [14]

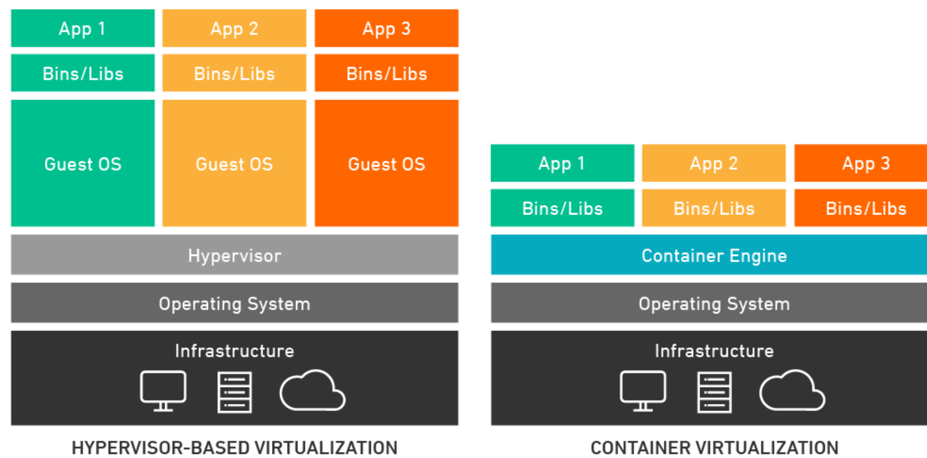


Figure 3.1: Virtualisation vs. Containerisation [12]

3.2.3 Orchestration

"Orchestration is the automated configuration, management, and coordination of computer systems, applications, and services." [15] When it comes to container orchestration, the most popular tool to do the job is Kubernetes, an open source platform developed originally by Google. It allows streamlining multiple dev ops automated processes, that would normally have to be launched manually, into a single, easy to manage process, or governing containers across multiple hosts.

3.3 Infrastructure as Code

Infrastructure as Code (IaC) is a method for programmably and automatically managing and providing infrastructure resources, such as virtual machines, networks, storage, and other components. Instead of manually setting up and configuring infrastructure resources, it uses code (usually in the form of declarative or imperative scripts) to define, configure, and deploy them. This brings the organisations plenty of advantages, such as using version control for tracking changes, automation - reducing manual effort, reproducibility - sharing the configuration across different environments, or scalability. [16]

The two typical approaches to creating IaC are:

- Declarative IaC: Describes the desired state of the infrastructure, without specifying the exacts steps required to reach that state. Popular declarative IaC tools include Terraform, or AWS CloudFormation.

- Imperative IaC: Takes the opposite approach and specifies the exacts steps required to configure the desired infrastructure. Common imperative IaC tools are Ansible, or Chef.

3.4 Automated testing strategies

Automated testing is a core concept in CI/CD. The idea is, that with every merge into the main branch of the repository, a battery o tests gets executed to make sure every part of the system works as expected, which is critical before any release.

3.4.1 API Testing

Modern applications often expose number of API endpoints. Verifying that these endpoints work as expected is vital, especially in modern architectures, such as microservices. APIs are typically exposed as REST-ful or SOAP-based endpoints. Consumers of the endpoint can request information from the producers based on a contract defined by the producer. The contract is expected to be consistent - in case it needs to be updated, developers typically introduce a new version of the endpoint and keep the old one functional, to allow backwards compatibility, at least for a limited time period. This allows consumers to work on updating their API calls for some time without the need to stop using the endpoint. [17]

There is a selection of tools that can be used as part of the CI/CD pipeline to verify that the APIs work correctly. The most popular tools include SoapUI, and Swagger. They enable executing API tests right from the pipeline and support generating reports on the results of the tests.

3.4.2 GUI Testing

It does not matter how efficient and productive your app is, if the user cannot interact with it because of a broken user interface. Performing GUI testing manually is tedious and very time consuming, and even prone to mistakes, where a QA engineer might overlook a missing component that was not included in the test scenario. Automating GUI tests can easily cover support for multiple browsers and uncover flaws by simulating real human interaction - mouse clicks, keyboard interaction with input fields, etc. - before they reach the production environment. [17]

Among the most popular GUI testing tools are Selenium, or Appium, both of which can be incorporated into CI/CD pipelines. QA needs to specify step-by-step user interaction with the GUI. The framework executes these steps and verifies, that the result is correct, not only at the end of the execution, but throughout the entire process. The frameworks offer different strategies to evaluate correctness of the result. The most basic one is verifying, that a specific HTML element (or element containing specific text) is present in the DOM (Document Object Model). A much more advanced manner is visual testing, where the tool keeps an array of images, each representing the expected state of the system at a certain point. These images are then compared to the actual state of the application during testing.

3.4.3 Nonfunctional Testing

Nonfunctional testing is a very wide class of tests. It focuses on the uncontrollable influences that can affect the application, such as load spikes, stress, volume, or network issues. These problems are often encountered for the first time in production, when it is too late to put the fire out. Depending on what specific area the team wants to focus on, there is plenty of tools they can use to incorporate the tests into their CI/CD pipeline.

3.4.4 Security Testing

Security in software has been a big topic for a long time. It is important to discover security flaws in the application as soon as possible, so that potential attackers do not even get a chance to exploit them. This is where the concept of "continuous security testing", or "CST", comes in. It is the notion of adding security tests into CI/CD pipeline, so that if there is a flaw found, at least it is found early, when it is still relatively easy and cheap to fix it. [18]

CST splits into several fields:

- Software composition analysis (SCA): Sometimes the flaw is not introduced into the application by the developer team directly. SCA makes sure, that external libraries used by the application are secure and do not contain exploits that could jeopardise the application's security.
- Static application security testing (SAST): Static code analysis detects bad practices in source code, including security vulnerabilities. It is an immensely helpful tool that should find its way into every software project, no matter its size.
- Dynamic application security testing (DAST): Typically implemented at the end of the CI/CD pipeline, DAST detects vulnerabilities of a running application. It is a black-box penetration testing tool, checking for the most common vulnerabilities, so that the application does not become an easy target right after release.[18]

Ultimately, the most surefire method of discovering vulnerabilities is hiring a third party to pentest the application. A seasoned tester has a big chance of finding a security flaw even with no prior knowledge of the application.

3.4.5 Regression Testing

Regression tests that verify that the latest changes have not introduced unintended side effects to functionality that previously worked as expected. These are typically end-to-end tests, which cover the largest portion of the application at once. They usually use the same tools as mentioned earlier - Selenium, Appium, or Cucumber, which means that the same tool can be reused with GUI tests and regression tests. Manual regression testing can be

time-consuming, so automating them as part of the CI/CD pipeline can save a lot of resources. [19]

The lifecycle of automated regression testing can be divided into three phases:

1. Test suite creation: Firstly, a QA specialist prepares the tests that check the critical components of the application. They need to identify what parts of the application should be tested, formulate the test cases, and write the scripts.

2. Test suite execution: The prepared test suite then gets integrated into the pipeline. Typically only a selection of the tests need to execute, based on the requirements, as running every test can take a lot of time. CI tools can have an enormous impact on how efficiently the tests are run.

3. Maintenance: Maintenance is a big part of automation in general. With every change in the source code, the tests need to be adapted as well. Especially in case of change requests where original requirements are modified, which the regression tests would flag as a bug. The bigger the codebase, the more complex maintenance becomes. However, even this part of the process can be automated with the tools providing self-healing processes.[19]

3.5 Version control and branching strategies

Version control is a critical component not only in CI/CD, but in software development generally. There are seemingly endless benefits to using version control systems (VCS).

- History and versioning: VCS enables tracking changes in the codebase over time, including who made the change and when. This also allows the team to revert the changes if something goes wrong.

- Collaboration: Multiple developers can work on the same project. Individual team members can create their own branches in the repository, work on them in parallel, and merge them together when they are done with their feature.

- Code integrity and backup: VCS ensures the integrity and safety of the codebase, acting as a backup in case of accidental deletions, overwrites, etc. Restoring a previous version of the code is very easy.

- Code review and QA: VCS integrates code review into the workflow, typically through process called pull request (PR). The developer submits a PR, and another developer reviews the code, points out imperfections, and then either accepts the PR, merging it into another branch, or rejects it to let the developer make additional changes.

- Traceability and Accountability: Every change to the codebase is tracked and recorded in detail, including information on who made the change, when, and even why (as long as the author specifies). This allows for enhanced accountability, auditing and simplifies the process of identifying the origin of bugs or other issues.
- Deployment and rollback: VCS plays a crucial role in the process of deployment. It allows for marking specific versions as release, which makes it easy to deploy stable code. In case an issue arises with the deployed version, it is very easy to roll back the changes to the previous stable version.
- Automation: VCS integrates seamlessly with CI/CD pipelines, enabling automated execution of batteries of tests, build and deployment whenever changes are pushed to a specific branch.

[20][21]

3.5.1 Version control tools

Git is by far the most popular VCS[22], to the point where it basically became synonymous with "source code management" [23]. Other familiar VCS tools are for example Subversion (often abbreviated to SVN), Mercurial, or CVS. Most modern integrated development environments (IDE) make working with VCS an extremely simple task. Traditionally, before IDEs became popular, git was typically used via its command line interface (CLI). Nevertheless, many developers still prefer the CLI over the IDE's graphical interface. The most basic commands include:

- init: Initialise a local git repository in the current directory.
- add: Mark files to be tracked by the VCS. For example, `git add .` adds all the files in the current directory for staging for the next commit.
- commit: Commits changes to the repository. Developer should commit fairly often, each time they make a logical change to the code that can be reasonably labelled. Commits should contain a message (using the `-m` argument) describing the changes in that commit. The team typically sets conventions about what the commit messages should look like. Commonly, the message is written in the imperative. Example usage of the command: `git commit -m "Fix Stack Overflow exception"`
- push: Uploads commits current or specified branch to a shared repository (called remote repository). After commits are pushed, other team members can ‘pull‘ the changes onto their machines. Example usage: `git push origin`, where `origin` is the name of the remote repository.
- pull: Downloads commits from the remote repository to the user's machine and merges it into the local branch. Example usage: `git pull origin`.

[25] Another aspect that teams need to consider, is choosing the git hosting solution. While most of the providers offer mainly the same features, the team needs to consider compatibility with the rest of the pipeline. Among the most popular hosting services are GitHub, Bitbucket, and GitLab, all of which support Git VCS, code review processes, and issue tracker, with only GitLab lacking support for SVN. [24] All of these platforms also support self-hosting the service, for the teams that have the infrastructure and want to keep their or their clients' data on-premise.

3.5.2 Git branching strategies

Branching is a fundamental concept in VCS that allows developers to work independently on separate features in the same codebase. Branching strategies play a crucial role in optimising collaboration, code stability and release management. By selecting and implementing an effective branching strategy, the team can efficiently manage modifications of the code, minimising conflicts, and supporting smoother integration of new features into the codebase. In this section, I will explore several common git branching strategies.

> A branching strategy is the strategy that software development teams adopt when writing, merging and deploying code when using a version control system. It is essentially a set of rules that developers can follow to stipulate how they interact with a shared codebase. [26]

GitFlow

One of the more complicated, but also the most flexible strategy, is called GitFlow. This strategy implements several types of branches:

- master (or main): This branch contains the stable releases of the product. These are typically the versions that run in the production environment. This branch "lives forever", meaning the it is never merged into another branch.
- develop: Another branch designed to be stable, containing the latest stable changes, that are not yet merged into the main branch. This branch also has infinite lifetime.
- feature: Developers create these branches off the develop branch and introduce the newest changes into the codebase. When the feature is finished, these branches get merged back into develop branch.
- release: These are help branches to assist in preparing a release to production. Typically, small bugs and merge conflicts are resolved in these branches. They are created from the develop branch and must be merged both into master and develop branches.

- hotfix: Similarly to release branches, these branches help in preparing for a release. Unlike the release branch, they are created from the main branch when a critical bug arises that needs to be solved as soon as possible - typically before the next planned release. It only serves to resolve the bug, after which it gets merged to both master and develop branches.

[26]
This strategy offers several advantages and disadvantages. One significant benefit is its support of parallel development, facilitating developers to work on separate branches while protecting the stability of the main branch for releases. It provides clear and distinct branches for specific purposes, making it easier to organise work. GitFlow excels when managing multiple versions of production code. On the other hand, with increasing number of branches, managing the merging of changes can become challenging. The complexity of creating release branches to finalise work and then merging it back into development and main branches, can make it difficult to identify the source of issues in case the tests fail, as the history of commits can become overwhelming. This subsequently can slow down the development process, making it less efficient for teams to implement CI/CD. In these cases, a simpler strategy, like GitHub Flow, might be a better fit.[26]

GitHub Flow

The GitHub Flow strategy lacks develop, release and hotfix branches, keeping only master and feature branches. The release branches are created directly from the master branch, and when finished, are simply merged back into it. This approach is suitable for smaller teams of developers that do not need to handle multiple versions of the production application. The main benefit is shorter time to release of each feature, which makes implementing CI/CD on top of this strategy fairly easy.

The main disadvantage of this approach is that it is more prone to introduce bugs into production, which can destabilise it more easily. As a result, bug fixes often happen directly in the main branch, which can make it cluttered and unorganised. When the team grows bigger, the number of branches grows with the team. That can make it difficult to resolve merge conflicts, which is amplified by the fact, that these conflicts need to be resolved on the production branch.[26]

GitLab Flow

GitLab Flow strategy uses the main branch the same way GitFlow uses its development branch. The main branch gets merged into a pre-production branch when the current changes are considered ready for deployment. There may be more than one of these branches between the production and main branches. When the time comes for the next deployment, the latest commit in the last pre-production branch gets merged into the production branch. This

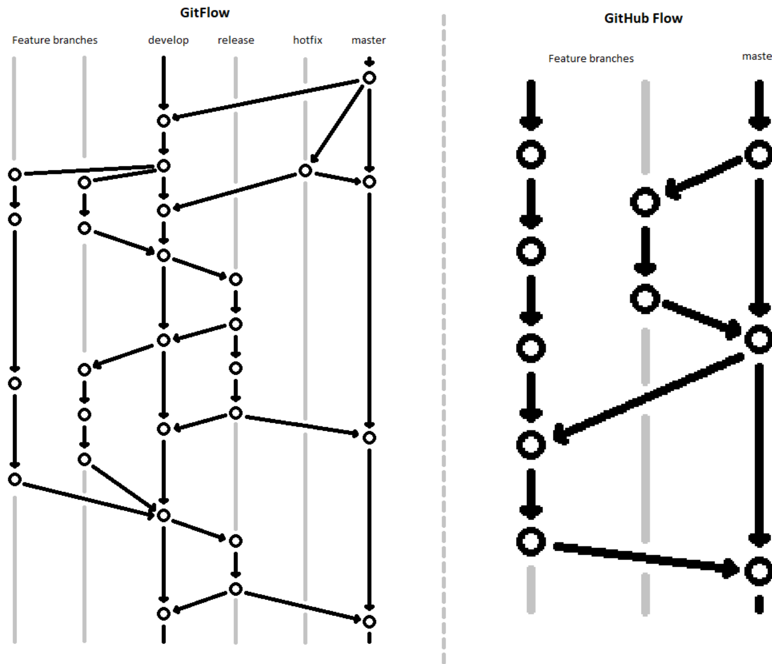


Figure 3.2: GitFlow and GitHub Flow strategies

approach introduces isolation between the different environments, with the possibility of maintaining the different versions separately. It is appropriate for use on projects with short release windows, or when the release timing is not under the team's control.[26]

Some of the disadvantages of the GitLab Flow strategy are the following: limited support for complex workflows, making it less suitable for large-scale projects with tricky branching needs, as well as its reliance on feature flags, which can introduce complexity. The lack of dedicated release branch can make coordination and release management more challenging.

Trunk-based strategy

Lastly, the trunk-based strategy omits all but the main and releases branches. The production environment is created from the main branch and the feature branches stem from the main branch. Developers are encouraged to merge their changes into the main branch as often as possible, which prevents merge conflicts. Developers also should merge the main branch into their feature branch right before merging it back into the main branch, which makes the merge into main much smoother, especially if done through pull requests with code review, because there is there will me no merge conflicts. This approach is extremely effective combined with CI/CD because of the frequent merges with master, meaning frequent deployments.

Keeping the trunk consistently updated not only helps with easy implementation of CI/CD, it also fosters collaboration and better visibility among developers. Unlike other methods, where changes are only visible after

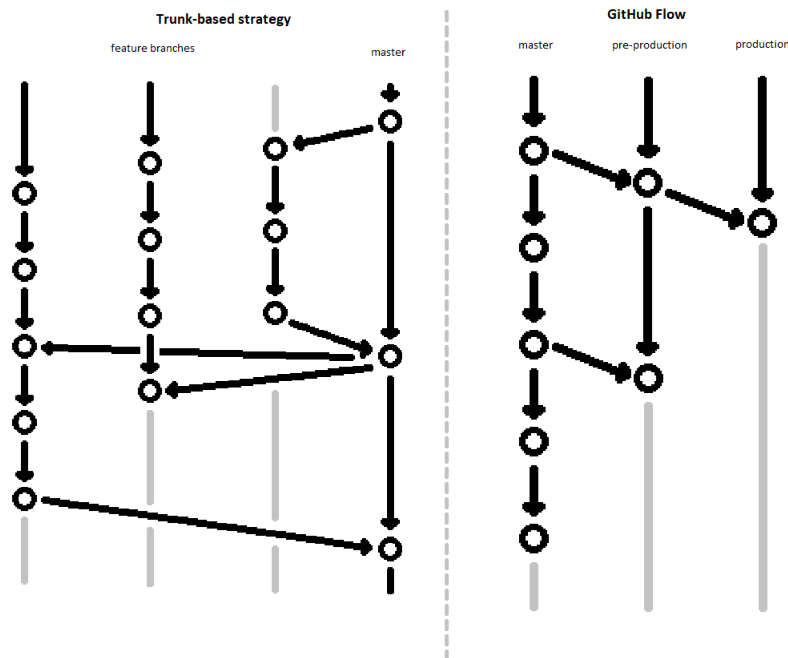


Figure 3.3: GitLab Flow and Trunk-based strategies

merging, direct commits into the trunk provide immediate visibility. With short-lived branches, this strategy eliminates the stress of merge-conflicts and facilitates easier conflict resolution. However, it suits more experienced developers due to the required independence, while junior teams often require closer monitoring, which makes this strategy a poor fit.[26]

Conclusion

Selecting the right branching strategy is crucial as it impacts various aspects of the project, including production environment quality, team comfort, and deployment difficulty.

A poor choice can lead to frequent merge conflicts, integration issues, and reduced software quality. Conversely, a well-suited strategy can promote a stable production environment.

Team members' comfort levels are influenced by the chosen strategy, with an appropriate one promoting productivity and collaboration. Conversely, an ill-fitting strategy can hinder teamwork.

Deployment complexity is also affected, with a suitable strategy streamlining the process and minimising downtime. Conversely, an inappropriate strategy might introduce more challenges.

Careful evaluation of project requirements and goals is essential to determine the most suitable branching strategy for success in all development aspects.

3.6 Code quality checks and static code analysis

Producing high quality code is top priority for developer teams. However, achieving the goal can be challenging. In the following section, I will delve into various methods and tools that can aid team in achieving and maintaining code quality.

Specifically, I will explore code review process and static code analysis. These tools and methods provide valuable insights, as well as automated analysis of the codebase, helping to identify potential issues, adherence to coding standards, and best practices.

By leveraging these tools, teams can streamline their code review process, proactively detect and address code quality issues, and ultimately enhance the overall quality and maintainability of their code.

3.6.1 Code review

Code review is a critical practice that plays a vital role in ensuring high-quality software development. By incorporating the expertise of experienced developers within the team, code review effectively reduces the occurrence of bugs and errors that would otherwise make their way into the production code, honing the skills of less experienced team members in the process. With code review, every change made in the codebase, be it a new feature, or a fix in existing code, undergoes examination and evaluation, allowing for constructive and insightful feedback and suggestions for improvement. This meticulous process leads to a significant improvement in the overall quality of the code, fostering cleaner and more maintainable codebase. [27]

By embracing code review as an integral part of the development workflow, teams can elevate their software development practices to a higher standard and deliver reliable, cleaner and robust products to users and clients.

I will now explore several approaches to code review:

- Pair programming: This approach involves two developers (typically one junior and one experienced) working together on the same code, allowing for real-time code review and mentorship. However, it can lack objectivity and requires more resources
- Over-the-shoulder: This informal method involves a qualified member of the team sitting down with their colleague to review their code, offering immediate feedback. This approach may lack tracking and documentation capabilities.
- Tool-assisted: Software-based code review tools offer a seamless and efficient way to review code, providing features such as comment tracking, asynchronous reviews, notifications, and usage statistics, enhancing the review process, providing review metrics and compliance reporting.

[28]

3.6.2 Static code analysis (SCA)

Static analysis is an automated debugging method that examines source code without its execution. It ensures compliance to coding standards, safety and security by identifying vulnerabilities. It complements manual code review process and is commonly used for satisfaction of coding guidelines like MISRA, a set of software development guidelines for the C programming language, or even more specialised standards, for example ISO 26262, which is an international standard for functional safety of electronic systems installed in automobiles. [29]

Static analysis can typically be performed twice in the development workflow.

Modern integrated development environments (or IDEs) often include basic SCA tools, with the option to install various extensions and tools, enabling more thorough analysis. These tools are typically configurable to meet the team's coding guidelines. A typical example is enforcing a semicolon at the end of statements in JavaScript, while it is not required in the language.

Static code analysis also has its place in the CI/CD pipeline, where it creates a feedback loop enforcing the set guidelines and standards, detecting security vulnerabilities, and even detect potential runtime errors, such as memory leaks.

Benefits

Static code analysis provides several benefits, particularly in compliance with industry standards. The tools offer speed, depth, and accuracy in code analysis.

In terms of speed, automated static code analysis is much faster than manual reviews. It detects and locates errors in code early, allowing for quicker resolution. Early error detection also reduces the cost of fixing coding mistakes.

When it comes to depth, static code analysers excel at examining code paths that testing might miss. They provide a comprehensive analysis of potential issues in the code based on applied rules, enhancing the overall quality of the code.

In regard to accuracy, automated tools eliminate the human error inherent in manual code reviews. They meticulously scan every line of code, identifying potential problems and ensuring high-quality code even before testing begins. This level of accuracy is crucial when adhering to coding standards. Common tools used as part of CI/CD are: SonarQube, Checkmarx, Synopsis Coverity, or Micro Focus Fortify Static Code Analyzer [30].

Challenges

While SCA is generally a highly valuable tool in software development, it does come with certain challenges and difficulties.

SCA may sometimes produce false positive results, marking code as problematic when it in fact works correctly. This can lead to wasted time and effort in investigating non-existent issues. The opposite case can also be true, where misses existing issues in the code, which can give the team a false sense of security.

Moreover, SCA tools might struggle to handle complex codebases, or languages with intricate syntax, which can result in incomplete or inaccurate analysis.

Additionally, high configurability is an aspect that can both be a positive and a negative, as fine-tuning the tools to suit specific project requirements and coding standards can be difficult. Also, integrating the tool into existing development workflows may require significant effort and expertise.

Lastly, Running SCA in large codebases can be time-consuming. It may slow down development workflow and CI/CD pipelines, affecting productivity and efficiency.

Chapter 4

DevOps Culture and Technologies

Many companies are embracing DevOps practices to make their software delivery faster, more efficient, and reliable. DevOps emphasises teamwork, automation, and continuous improvement, transforming how software is created, deployed, and maintained.

This chapter explores the concept of DevOps focusing on its relationship with Continuous Integration/Continuous Deployment (CI/CD). I will delve into the importance of effective communication and feedback loops within CI/CD processes. Additionally, I will examine the range of collaboration tools available that facilitate seamless teamwork and knowledge sharing among development and operations teams.

4.1 DevOps and its relationship with CI/CD

DevOps is a way of working that combines practices, tools, and a cultural philosophy to automate and bring together software development and IT teams. It emphasises the importance of teamwork, communication, and using technology to make things easier.

The idea of DevOps started around 2007, when people in the software development and IT operations communities started to worry about the traditional software development model, in which developers wrote code separately from the operations team, who handled deploying and supporting the code. DevOps, which combines "development" and "operations", was created to bring the two groups together into one continuous process. [31] DevOps primarily emphasises the importance of communication and collaboration between developers and testers. This was necessary because in traditional setups without DevOps, developers often had limited knowledge about the challenges faced by QA and Operations teams. Additionally, QA and Operations personnel often lacked a comprehensive understanding of the business requirements for the software since they worked on multiple projects simultaneously. [32]

In DevOps, the infinity loop is used to demonstrate the interconnection of phases in the DevOps lifecycle. Although it may seem like the phases follow a linear sequence, the loop represents the importance of ongoing collaboration and iterative enhancement throughout the entire lifecycle.

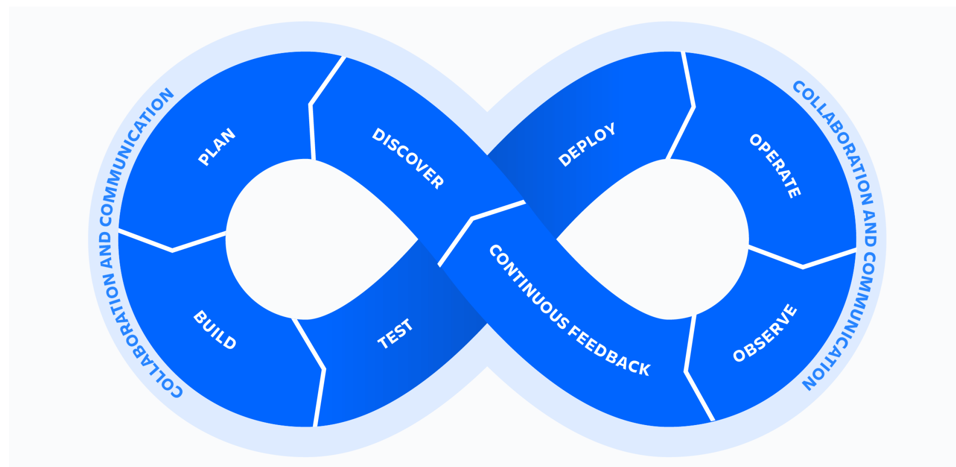


Figure 4.1: DevOps infinity loop [31]

The infinity loop signifies the continuous nature of DevOps practices. It highlights that the work doesn't end once a phase is completed, but rather it loops back to earlier stages to foster collaboration and make improvements. This iterative approach allows teams to continuously refine and optimise their processes, promoting efficiency and quality throughout the software development lifecycle.[31]

By using the infinity loop, DevOps practitioners emphasise the significance of constant communication and collaboration among team members. It encourages them to work together closely, share insights, and learn from each other's experiences. This iterative feedback loop enables teams to identify and address issues promptly, ensuring continuous improvement and delivering better outcomes for the project.

DevOps and CI/CD have a strong connection, as teams using DevOps also benefit from CI/CD. In DevOps, continuous testing is used to detect important bugs early on, which helps save money that would otherwise be spent on fixing bugs in later stages. [33]

In conclusion, the infinity loop in DevOps serves as a powerful symbol, reminding practitioners of the need for ongoing collaboration, learning, and improvement. It reinforces the core principles of DevOps and promotes a culture of continuous enhancement in software development practices.

4.2 Communication and feedback loops in CI/CD processes

A basic definition of feedback loop is the following: "Feedback loops are sets of relationships between entities whereas a change in one entity causes a change in another entity and that change eventually leads to a change in the first entity."[34]

4.2.1 Reinforcing feedback loop

A reinforcing loop is a feedback loop that creates accelerating change. In this loop, when a change occurs in one part, it causes a similar change in other entities, resulting in even more change in the initial entity.[34]

It is important to understand that this type of feedback loop can amplify either positive or negative change. In a positive amplifying loop, an increase leads to more increase, creating a compounding effect. On the other hand, in a negative amplifying loop, a decrease leads to further decreases, intensifying the negative impact.

To give an example of the reinforcing feedback loop, consider a business that experiences growth in its customer base. This leads to increased revenue and resources, allowing the business to invest more in marketing and product development. As a result, it attracts even more customers, leading to further growth and success. This reinforcing loop of amplifying change continues, driving the business's expansion.

4.2.2 Balancing feedback loop

A balancing feedback loop as opposed to an amplifying feedback loop leads to system stability without further change. In a balancing feedback loop, the system reaches a point of equilibrium where no further change occurs. [34]

A typical example of a balancing feedback loop is thermoregulation of the human body. Using different biological mechanisms, the human body of a healthy individual is able to maintain the temperature of approximately 37 °C, by constantly monitoring external temperature changes and applying the said mechanisms to increase or decrease the body temperature as needed, maintaining a stable internal environment.

4.2.3 Feedback loops in CI/CD

CI/CD plays a crucial role in establishing effective feedback loops within DevOps teams. Development teams work on small portions of code and regularly upload them to a shared repository for deployment. This approach enables continuous feedback, allowing the DevOps team to conduct thorough testing and quickly identify any bugs, which are then reported back to the development team, which then acts on this feedback, closing the loop.

Utilising appropriate feedback loop technologies can enhance pipeline speed and improve efficiency. CI/CD pipeline tools are a critical component of the DevOps feedback loop. Many DevOps tools provide various feedback stream options, allowing customisation to fit organisational needs. Another useful feature some DevOps tools provide is automatic generation of dashboards triggered by developers pushing their code to the shared repository.

4.3 Collaboration Tools

The market for collaboration is very broad. I have already covered some categories of tools used in DevOps, such as VCS (e.g. Git) or CI/CD tools (e.g. Jenkins). In this section, I will explore issue tracking systems, collaboration and communication tools, and monitoring and logging tools.

4.3.1 Issue tracking systems

DevOps and project agility require continuous integration, continuous delivery, and shorter release cycles. In this fast-paced business environment, it is no longer feasible for project managers and teams to address system issues on an ad hoc basis. Modern approach lies in adopting a capable issue tracking tool. Such tools aim to capture issues early, prioritise them based on their severity and impact, assign them to the right team members, and track their progress. By doing so, teams can effectively address all issues and defects in line with agile and DevOps practices, ensuring smooth software development and delivery. [35]

Ticket (or issue) is the centrepiece of an issue tracking system. Among the main characteristics of a ticket are: description, priority, assignee, reporter, status (e.g. "To do", "In progress", etc.), and comment section, where team members can share information concerning the task at hand. There are typically different varieties of a ticket, which in combination with the ability to create a tree structure by assigning parent-child relationships with one another, creates a transparent image of the project. Tickets can also be linked together outside of the tree hierarchy, signifying different relationships, for example indicating, that a certain ticket blocks another ticket from advancing.

A good integration between the issue tracking system and VCS allows developers to create a new branch directly from the interface of a ticket, which streamlines the process of starting working on a new issue. Another example of useful integration with another component of DevOps is integration with a CI tool. This can for example enable automatic ticket creation when a failed build or test occurs, allowing teams to quickly address and track the issue.

Among the most popular issue tracking systems are Jira, Backlog, Trac, or Redmine. Some ecosystems, like GitLab or Atlassian, offer their own tracking systems as part of a complex solution for project management, which lowers the number of integrations that need to be configured manually to really take advantage of CI/CD and DevOps.

4.3.2 Communication Tools

Communication tools play a crucial role in facilitating collaboration and effective communication within DevOps teams. These tools provide an environment for team members to connect, share information, and work together.

One such commonly used tool is Microsoft Teams, which is a tool many people know from a different background, like school, which makes for an amazing learning curve when onboarding new members, because they usually already have some experience with the tool. It allows team members to chat, make video calls, or calls with screen sharing, which is particularly useful for remote teams, as for instance QA engineer can demonstrate a bug to a developer. It also integrates with other Microsoft tools, such as Office and Azure, forming a unified platform for communication and collaboration.[36]

Another very popular communication tool commonly used in DevOps is Slack. Slack allows team members to create channels dedicated to specific projects, topics, or teams, enabling focused discussions and easy access to relevant information. It supports real-time messaging, file sharing and integrates with various other DevOps tools improving productivity and information sharing.[37]

By leveraging such communication tools, DevOps teams can effectively communicate, share knowledge, and collaborate seamlessly, leading to increased efficiency.

4.3.3 Monitoring and Logging Tools

Effective monitoring and logging are essential for maintaining the performance, availability, and reliability of modern software systems. These tools provide valuable insight into the health and behaviour of applications and infrastructure, enabling teams to identify and address issues proactively.

DevOps monitoring tools, such as Prometheus and Datadog offer real-time visibility into key metrics, allowing teams to monitor various metrics, such as resource utilisation, response times, and error rates. These tools facilitate proactive monitoring, alerting and performance analysis, helping teams detect and troubleshoot problems efficiently. [38]

Papertrail and Scalyr are examples of logging tools, that enable the collection, aggregation, and analysis of log data generated by applications. They help in understanding system behaviour, diagnosing errors, and investigating incidents by providing centralised log management and powerful search capabilities. [39]

These tools play a critical role in ensuring the smooth operation and improvement of software systems by giving teams valuable insights into their systems, enabling them to make data driven decisions and detect and resolve issues quickly.

Chapter 5
Practical application of CI/CD on existing project

This chapter deals with the practical implementation of CI/CD principles and application of DevOps on an existing project, namely a web therapeutical application for addiction recovery. It focuses on two essential aspects of applying CI/CD: pipeline configuration and test automation.

The project is built on top of Serafin - a logic-driven web content creation kit [40] based on Django, which is a Python framework for building web applications. Serafin creates flows that define user interactions. These flows are generated from a JSON-formatted file. The project also contains GUI for creating these flows, which means the sessions can be easily altered. I will focus on the back-end of the application, excluding the parts of the project that are inherent to Serafin.

5.1 Pipeline Configuration

First, I will describe the configuration of the CI/CD pipeline. I will talk about the tools that were used to build the pipeline, and discuss what additional CI/CD and DevOps tools could be used to further improve the configuration.

Then I will outline the structure and stages of the CI/CD pipeline. This will involve breaking down the pipeline into its key components and explaining the purpose and functionality of each stage. Then I will suggest what stages that are not yet implemented would benefit this project the most.

5.1.1 Used tools and technologies

VCS and VCS hosting service

Git has become the go-to option for modern software projects, and this particular application is no exception. Its strength as a VCS combined with its widespread adoption makes it an ideal choice, especially when working with a team the frequently changes its members, such as a team made primarily of students. Most developers are already familiar with git, which makes it easier for new members of the team to participate in the project quickly.

The team made the decision to use GitLab as the solution to hosting their shared Git repository. They opted for the Software-as-a-Service (SaaS) model instead of a self-hosted solution, mainly because a better availability, because a self-hosted solution typically requires use of a VPN which would make setting up new developers unnecessarily more difficult. The choice of GitLab has a big impact on their choice of the rest of the toolchain, as GitLab provides an ecosystem of DevOps tools, without making the team go through the hassle of configuring integrations with other tools.

CI/CD

As I hinted in the previous section, the team adopted GitLab CI/CD as the main CI/CD tool. As their pipeline is fairly short, it was not a significant decision to make, because they only use the basic functionality CI/CD pipelines can offer. The pipeline is defined in a text file, `.gitlab-ci.yml`, located in the project's root directory. It is a file in the YAML format, that defines the various stages that the pipeline consists of.

Containerisation

The team decided to go the way of containerisation, which is a modern approach, enabling the team to split the application into different modules, while still only needing a single server to run the application. You can read more about Docker in section 3.2.2.

Orchestration

Because the application is only accessed by low thousands of simultaneous users per day, the team chose Docker Swarm for their orchestration tool. As opposed to Kubernetes, Swarm only offers scaling on demand, which is not an issue because of the relatively small user base. It is simple to use, because it shares its CLI with Docker, which the team was already familiar with.

Because Swarm has no graphical user interface, the team decided to use Portainer, which is a lightweight container management tool running separately in its own Docker container.

Server hosting

The team chose Amazon Web Services (AWS) to host their application for its pricing, simplicity, performance, and history of stability and security. Setting up Continuous Deployment from GitLab to AWS was straightforward, as it is a popular combination of services and the process is well documented.

Issue tracking system

The team does not use an issue tracking system, because the stable team of developers consists of only two people. Students often contribute to the

project, but their assignments are typically long-term. Still, in my opinion, the team could benefit from an issue tracking system to document their advances in development, as well as maintain a transparent image of the project's current state, even if only permanent developers would use it.

Communication tool

As their communication tool, the team uses Slack, which turned out to be somewhat clumsy at times, especially as it lacks the option to make video calls, which means that for our weekly consultation, we had to use a different communicator, which in my eyes is slightly redundant. In my opinion, Microsoft Teams would be a better option, as it supports video calls, as well as every feature of Slack that the team uses. Taking advantage of the fact that the university also uses Teams as a communication tool, for example in classes, might also be an option, although I did not verify this information.

Monitoring and logging tools

The application only produces logs inside the respective docker containers, which might become problematic if the container happens to crash. If the logs are not backed up on a persistent storage volume, the logs will most likely be lost forever and the team will have a hard time figuring out what caused the crash. The logs can be accessed using the aforementioned Portainer GUI, which also serves as a faucet to monitoring basic performance metrics, such as CPU and memory usage.

5.1.2 Structure and stages of the CI/CD pipeline

The pipeline that the project uses is extremely simple, maybe to the point of it being wasted potential.

It currently only contains two stages - build and deploy. First, the build stage starts. It is defined by a script, which is the following command: `docker compose -f docker-compose.yml build`. This command builds the docker image, ensuring that the build succeeds.

Once the build stage has finished, the deploy stage starts. This stage is also defined by a script, this time it is a Swarm command (Docker and Swarm share the same CLI): `docker stack deploy -c docker-compose.yml ${STACK_NAME}`. This command is used to instruct Swarm to deploy a stack defined in the same docker-compose file as above.

A recurring topic in this thesis is, that the concept of CI/CD and the DevOps mentality emphasises testing as a crucial part of the entire process. Running battery of tests highly raises the chances of the code being stable, yet the test stage is completely missing in the pipeline despite the fact, that the codebase has test coverage. This is an area where I can see big potential for improvement.

5.2 Testing and QA

A big part of the codebase is represented by Django's Object-relational mapping representation. I decided to exclude them from the metrics, because testing this code is irrelevant to my intentions. Besides, in case these mappings do not work, the tests that I am concerned about will fail as well.

5.2.1 Employed testing strategies

Only a handful of the system's functionality has some sort of automated testing employed This is due to the fact that Serafin, the system this project is based on, has very little code coverage in itself. Considering this, I will focus on the modules developed as part of this particular implementation. It is namely Engine module, which is a critical component of the application, that handles all major processes, like determining sessions for individual clients, including sending email and SMS notifications. The other tested entity created as part of the project is the Expressions module, which takes care of parsing logical expressions written in plain text, which is used to build a decision tree, on which is based scheduling of individual users' sessions.

Engine module

The engine module only contains one method that could be considered public, called `run`, which traverses the decision tree and returns the first node that that passes certain conditions, which represents what the next action for the specified client is - for example what session comes next for them. This makes testing this module is fairly difficult, as its functionality requires communication with every other module present in the project, and its expected result depends on data that is present in the database as opposed to just method parameters. This is why I would consider these tests more of integration tests rather than unit tests. The test for the module has four test methods. They cover 95% of the code in the engine module, which is fairly impressive, but the test class contains a total of 33 assert statements. This means that the each test method contains many assertions, which is a bad practice, as it can make the debugging process more difficult, because the test will not get the chance to discover further errors after the first failing assertion. This leads to the tests reporting inaccurate findings in case the code contains multiple errors.

Expressions module

The tests for expressions module have similar issues as those for the engine module. It contains six test methods, that albeit logically divide the different types of expressions that are parsed into six categories, each method contains more than ten assertions, which can result in fewer reported error than the code realistically contains, and it can be difficult to identify the problems at first glance. The code coverage reaches 100%, as covering all the code paths

can be easily achieved just by using the right parameter input. These tests could therefore be improved by splitting the methods into many more, where each method should technically contain just a single assertion.

5.2.2 Use of code quality checks and static code analysis tools

Although GitLabs offers static code analysis even for free tier, this project does not have it enabled. This is an area of the project that could be easily improved. Implementing static code analysis can prevent security vulnerabilities, performance issues and generally speeds up the process of software development by detecting potential issues and bugs early. More information on static code analysis can be found in section 3.6.2.

5.2.3 Handling of test failures

As there is no continuous testing employed in the CI/CD pipeline, testing the code is fully in hands of the team of developers. It is each team member's responsibility to run tests and verify that their changes did not break existing functionality. When a test detects an error, it typically means one of two things:

1. Breaking change: Some of the modifications to the code has broken a part of the system and it needs to be fixed.
2. Intentional change: The original logic has been changed according to the requirements, but the tests have not modified to reflect it. The responsibility to keep the tests aligned with the application logic is on the developer that implements the change.

It is common practice that the developer who implements a new feature also writes unit tests to add the verification of its functionality and to keep code coverage high.

5.3 Conclusion

Generally, the CI/CD implementation in the project seems very basic, which in retrospect makes sense, as the application sees very little new development. As the application is data-driven, the changes to the application flows, such as client sessions, are not made by altering the codebase, but rather by changing the underlying data. The development that is done on the codebase are mostly performance optimisations, which is something that would definitely benefit from proper test automation, but as any of the development is not time critical, the main advantage of implementing it would be in the testing consistency rather than time efficiency.

Chapter 6

Reworking a Core Module

This chapter recounts my experience with rewriting a critical module of the application in Rust programming language. I will describe the implementation process - the difficulties and caveats. Next, I will outline the steps needed to integrate the module into the build stage of the CI/CD pipeline. Last, I will go over what I learned in the process and what could be further improved.

The reason for the decision to re-implement the module is fairly straightforward. The goal is performance optimisation, as Python is known for not being the most efficient programming language when in comes to performance. Oppositely, Rust is known for its exorbitant obsession with memory safety, as well as amazing performance.

6.1 Implementation Process

6.1.1 Choosing the framework

There are several frameworks and methods that enable running Rust code in Python environment available. I will go over them briefly:

- rust-cpython: Framework which allows for writing a native Python module in Rust, as well as the other way around - to use Python in a Rust library.
- PyO3: This framework originated as a rust-cpython fork in a period when rust-cpython was not actively maintained, so it has many features in common. The main difference is easier memory management.
- CFFI and ctypes: Python provides libraries that can load and call native C functions, and Rust is able to expose a C-compatible API using the Foreign Function Interface. This is basically a way of tricking Python into thinking that it is running a library written in C, while in reality it runs a Rust library.

As I had very limited experience with the Rust programming language, I wanted to choose the option that would be the easiest to get started with. I decided to go forth with PyO3, mainly because there is extensive documentation, and because it has the simplest memory management.

6.1.2 Getting started

I was considering adopting the test driven development approach, because the original module had some tests ready to run, but I soon realised, that I could not use these tests for this purpose, as they are not true unit tests. In the end I decided to go from the smallest prototype that I could test manually, for example just in Python console, and just try to replicate the code from the original module.

Learning PyO3

First, I needed to create the skeleton of a PyO3 project. This is a straightforward process. Using python package `maturin`, which can be installed using `pip`, one can initialise a PyO3 project in single step. The package generates both the necessary files. `Cargo.toml`, which is a Rust project definition file, that specifies project name, version, description, and dependencies, among other things. The other generated file is located in newly created `src` folder, and it is called `lib.src`. The name of the file is important, because later when building the module, this is the filename that the compiler expects.

The bare `lib.src` file contains a simple test code intended as an introduction to the PyO3 framework, showing how to build a very simple function in rust and expose it to the Python environment. This is where I decided to experiment a little bit before diving into the documentation. At this point, I was a bit sceptical about the memory management. My goal was to find out, whether the generated python module can accept an instance of a data structure and mutate - as part of that rust code. First I prepared a function that accepts an instance of a list as a parameter and inserts an item to the list.

Listing 6.1: First encounter with PyO3 - file src/lib.rs

```
use pyo3::prelude::*;
use pyo3::types::PyList;

#[pyfunction]
fn add_to_list(list: &PyList) -> PyResult<()> {
    list.append("new item")?;
    Ok(())
}

#[pymodule]
fn PyO3_test(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(add_to_list, m)?)?;
    Ok(())
}
```

The procedure to test the code was simple:

1. In Python, create an instance of an empty list.

2. Call the function on the Python module built from the PyO3 file.

3. An item is added to the list in the function of the module from the PyO3 file.

4. Use `print` to display the content of the list in the original Python script after the function call.

Listing 6.2: First encounter with PyO3 - file main.py

```python
import PyO3_test

if __name__ == '__main__':
    list_instance = []
    PyO3_test.add_to_list(list_instance)
    print(list_instance)
```

I used `maturin develop` command to build the PyO3 module and inject it into the Python virtual environment of a test project and ran the python script. To my great surprise and pleasure, the output of this test was `['new item']` in the console. This was something that I hoped would not be an issue, but I had to verify this fact as soon as possible. This meant a couple of things. Most importantly, when an instance of an object gets passed as a parameter to my Rust module, the original Python context keeps ownership of the object. Also, the Rust function does not create a copy of the object when it receives it as a parameter. We can deduce this from the fact that when I called `print(list_instance)`, we can see the `'new item'` inside the list, which was added in the Rust module. This all might sound trivial, or unimportant, but it ultimately means, that I could rewrite the original module without changing the signatures of the original functions, which means that adopting the new module would just mean changing the import statement wherever the Engine module is used, because all the objects passed to it can in fact be mutated inside the Rust code.

Let's go over the structure of the Rust code in listing 6.1. First lines contain `use` statements, which is a way of importing different crates (Rust naming for modules). Then, we can see a macro `#[pyfunction]`. This is a way of signifying that this is a function that we will likely want to expose to the Python code the module will run in. The function `add_to_list` accepts a single parameter - a reference to a `PyList`. PyO3 comes ready with interfaces for all the basic data types that Python uses. `PyList` is the equivalent to `list`, and `PyDict` is the equivalent to `dict` [41]. My `add_to_list` function's return type is `PyResult<()>`. That is another type that is specific to PyO3. It is a shorthand for Rust's `Result<T, E>` type, where `E` implements `From<E> for PyErr`. This will raise a Python exception if the `Err` variant is returned. The `()` in my function signature's return type just means that the function does not return a value, but is fallible. This return type allows me to use the `?` on line 6. The return type of the method `PyList.append` also returns a `PyResult<()>`. The question mark

operator unwraps valid values or returns erroneous values, propagating them to the calling function. The next function uses the macro `#[pymodule]` which carries out exporting the initialisation function of our module to Python.

Starting work on Engine module

After I tried out some of the basic concepts of using the PyO3 framework, I began work on the Engine module by creating the constructor, which meant first defining class variables. A class in PyO3 is defined using two macros: `#[pyclass]`, which is created using Rust's `struct` statement, where the class variables are located. The other part of a class is the implementation of its methods. This is achieved using the macro `#[pymethods]` in combination with Rust's `impl` statement.

The original Engine module has many dependencies in the rest of the project. I needed a way to call methods on these different modules in the context of my Rust module. PyO3 offers interface to achieve this. This is a method that I wrote to be able to access the database using Django. It is used in the constructor, where the reference to the user can either be passed directly, or just a user id is passed, in which case the user needs to be retrieved from database.

Listing 6.3: Calling methods of Python libraries

```
fn get_user_by_id(py: Python, user_id: i32)
                            -> PyResult<PyObject> {
    PyModule::import(py, "django.db.transaction")?
        .call_method1("set_autocommit", (false, ))?;
    Ok(PyModule::import(py, "django.contrib.auth")?
        .call_method0("get_user_model")?
        .getattr("objects")?
        .call_method(
            "get",
            (),
            Some([("id", user_id)].into_py_dict(py))
        )?.into_py(py))
}
```

In Python, getting a user instance from database using Django is achieved in two lines of code, one of which is the import statement. The original code were two lines, each calling a function or method in an imported module, so technically four lines. That means that I first needed to import these two module. PyO3 offers an interface that can call any function and access any attribute on any PyObject. This is because Python technically does the same, as it is an interpreted language, and type checks are performed in runtime. The painful part about this rewrite was the fact that every time I need to call a method on a non-standard object, I needed to insert an additional function call between the object and the function call or attribute. This can be seen in listing 6.3, where the original code called `transaction.set_autocommit(false)`,

I needed to import the module and use `call_method1` function to call the `set_autocommit` method. The other call is even worse, as the original code accessed an attribute on the imported module, and only after that it called a method on that attribute. The code expands fairly fast as a result of these extra calls. The original Python module contains about 640 lines of code, many of which are docstrings, while the finished PyO3 module reached about 840 with no docstrings.

6.1.3 Common patterns between the original code and Rust implementation

On the rare occasion that the Python module checked for a `None` value, I got the opportunity to take advantage of Rust's `Option` object. Rust has a mechanism that ensures that whenever the developer cannot be 100% certain that a variable contains a non-None value, then it is visible on first sight. `Option` is an `enum` that has two values: `None` and `Some(T)`, where `T` is the data type of the value the `Option` can contain. This way, an optional argument `bar` of type `i32`, a 32-bit integer, of a function `foo` would be noted as `fn foo(bar: Option<i32>)`. This makes it clear, that `bar` might not contain a value. To access the contained variable, one can call `bar.unwrap()`. This is considered unsafe, as in the case where `bar` is actually `None`, Rust panics (the terminology for unexpected termination of the program). A different way to handle `Options` is by calling `let bar = bar.unwrap_or(<default value>)`, which reinitialises the variable `bar` with type `T` and either the value contained in the `Option` in case it is `Some`, or the default value passed as a parameter in case the `Option` is `None`. Another two approaches to handling `Options` are an `if let Some(value) = bar` along with an `else` statement, and a `match` statement.

Listing 6.4: Example handling of Option.

```
while !special_edges.is_empty() {
    if let Some(edge) = self.traverse(py, edges)? {
        // trigger the node...
    } else {
        break;
    }
}
```

Listing 6.5: The original code in Python

```
while special_edges:
    edge = self.traverse(special_edges)
    if edge:
        # trigger the node...
    else:
        break
```

As an example of the if let Some, I present the code in listing 6.4, which is part of transition method in the Engine module. The return type of self.traverse is PyResult<Option<PyObject>>, which means that using the question mark operator, we effectively change the value to Option<PyObject> type. The if branch handles the case where Option is a Some value, and the else branch handles the None case.

6.1.4 Finishing up

When most of the functionality was done, I started regularly running the tests that were prepared by developers previously working on the project. This uncovered some bugs that were mostly fairly easy to fix. The process of building the module is following:

1. Use maturin build --release command to build the optimised binary .whl file.
2. Use pip install with the path to the .whl file to introduce the module to the Python environment.

After these steps, the Engine class can be imported into the project's modules using from engine_rs import EngineRS. Notice, that I named the module slightly differently than the original one, so that in case we needed to revert to the original implementation, we could just rename the references used in the code to the original class, while it is apparent at first glance which Engine class is in fact being used.

6.2 Integration into CI/CD

To incorporate the build process into the CI/CD pipeline, I decided to add the procedure to the Dockerfile. The Dockerfile is a text document, containing the instructions required to build a docker image. This process is part of the build stage of out GitLab CI/CD pipeline. The build is composed of several steps:

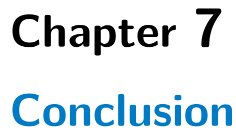
1. Download rustup, a tool used to install rust compiler.
2. Run the rustup tool
3. Add /root/.cargo/bin to the PATH environment variable. This step is required, so that maturin can access the rust compiler easily.
4. Copy the Rust source files onto the Docker container.
5. Run the maturin build --release command. This step builds the binary specific to the platform of the system the command is ran on. This is the reason why we do not build the binary locally - because this approach is more flexible, as it always builds the binary for the correct platform.

6. run `pip install <path to .whl file>`. This step is more complicated, because `maturin` does not support specifying the name of the output file. The name always contains the information about the platform the binary was built for, among other things. For example on my machine, the generated file's name is `engine_rs-0.1.0-cp38-none-win_amd64.whl`, where `win` in the name specifies Windows as the OS the binary is built for, and `amd64` the CPU architecture.

This finalises the process of installing the module on the Docker container. Unfortunately, despite the tests passing with no issues, and the pipeline finishing with no problem, the code does not run properly on the target system. It is a big disappointment to myself and the rest of the team, because I ran out of time to try and troubleshoot the issue before the deadline to hand this thesis in. This also means, that regrettably, I cannot run any reasonable performance tests while the code is not implemented correctly.

6.3 Lessons learned

The biggest lesson to me is definitely that no matter how many tests there are to validate the existing functionality, or how high the code coverage is, bugs can still easily find their way into the source. I learned how to use the PyO3 framework to build efficient Rust modules that can be used in Python scripts, which I can see as a valuable skill that can be extremely useful in other practical applications.



Chapter 7
Conclusion

This thesis explored the world of CI/CD and DevOps in detail, summarising the most important information regarding the topic. We compared the most common CI/CD tools, introduced reader to containerisation, explained the difference between containerisation and virtualisation, and provided an example of the most common containerisation tool that is very often used in modern applications - Docker. Moreover, we touched upon orchestration technology, as well as modern approaches to building virtual infrastructure using Infrastructure as Code. We described in detail different nontraditional categorisation of strategies that can be used in automated testing. We also characterised various git branching strategies, which is an important aspect of software development in teams of developers. We closed the chapter by portraying the code review process and illustrating the importance of static code analysis in software projects.

In the next chapter, we defined DevOps and expressed the relationship it has with CI/CD. Additionally, we detailed communication in DevOps and feedback loops present in the CI/CD process and their importance. Furthermore, we supplemented the technology we described so far with some more tools related to team collaboration and DevOps, such as issue tracking tools, communication tools, and monitoring and logging tools.

The fourth chapter delves into analysis of an existing implementation of CI/CD of a real project. We describe the configuration of the existing pipeline, point out its strengths, as well as spots where there is potential room for improvement. We focused on the tools and technology that are used in the project and CI/CD and we analysed the specific stages that the CI/CD pipeline contains. We also analysed the testing strategies that are employed in the project, analysed the code coverage of the project's codebase, how the tests are incorporated in the CI/CD pipeline and how their potential failure is handled.

The last chapter deals with my experience with re-iplementing the most performance critical module of the project in Rust programming language with the goal of improving the application efficiency to boost performance without the need of infrastructure scaling. I describe the process of choosing a suitable framework to do the job, and describe the overall progress of building the module. I described how to build the binary of the module from the Rust

source code and how to configure the project so that the Rust module can be imported and used in native Python. Then I described the way that I modified the CI/CD pipeline to build the module in the Docker container.

Unfortunately, I was unable to run the resulting module in the final docker image for a reason that is currently unknown to me, which results in me being unable to provide results of a performance test. Nevertheless, my work on this project is not over and I will continue the work to troubleshoot the implementation and provide a working product in the end.

Bibliography

[1] Red Hat. *What is CI/CD?*. [ONLINE] Available at: https://www.redhat.com/en/topics/devops/what-is-ci-cd. [Accessed 18 May 2023].

[2] TeamCity, Jetbrains. *What are the benefits of CI/CD?*. [ONLINE] Available at: https://www.jetbrains.com/teamcity/ci-cd-guide/benefits-of-ci-cd/. [Accessed 18 May 2023].

[3] Tom Donohue, tutorialworks. *The 7 essential stages of a CI/CD pipeline*. [ONLINE] Available at: https://www.tutorialworks.com/cicd-pipeline-stages/#acceptance. [Accessed 18 May 2023].

[4] Jenkins. *Homepage*. [ONLINE] Available at: https://www.jenkins.io/. [Accessed 18 May 2023].

[5] Iris Chubb, inedo blog. *5 Reasons People Hate Jenkins CICD*. [ONLINE] Available at: https://blog.inedo.com/jenkins/everybody-hates-jenkins/. [Accessed 18 May 2023].

[6] Circle CI. *Homepage*. [ONLINE] Available at: https://circleci.com/. [Accessed 18 May 2023].

[7] Travis CI. *Travis CI Cloud Pricing*. [ONLINE] Available at: https://www.travis-ci.com/pricing-cloud/. [Accessed 18 May 2023].

[8] Cuelogic. *Best Continuous Integration (CI) Tools In 2019: A Comparison*. [ONLINE] Available at: https://www.cuelogic.com/blog/best-continuous-integration-ci-tools. [Accessed 18 May 2023].

[9] Hitesh Jethva, Cloud Infrastructure Services. *Jenkins vs Gitlab – What's the Difference? (Pros and Cons)*. [ONLINE] Available at: https://cloudinfrastructureservices.co.uk/jenkins-vs-gitlab/. [Accessed 18 May 2023].

[10] Katalon. *Best 14 CI/CD Tools You Must Know | Updated For 2023*. [ONLINE] Available at: https://katalon.com/resources-center/blog/ci-cd-tools. [Accessed 18 May 2023].

[11] IBM. *What is containerization?*. [ONLINE] Available at: https://www.ibm.com/topics/containerization. [Accessed 18 May 2023].

[12] Armor. 15 March 2018. *Containerization: The Need to Know*. [ONLINE] Available at: https://res.armor.com/resources/blog/containerization-the-need-to-know/. [Accessed 23 May 2023].

[13] 6sense. 2023. *Top 5 Containerization technologies in 2023*. [ONLINE] Available at: https://6sense.com/tech/containerization. [Accessed 19 May 2023].

[14] IBM. *What is Docker?*. [ONLINE] Available at: https://www.ibm.com/topics/docker. [Accessed 19 May 2023].

[15] Red Hat. 15 October 2019. *What is orchestration?*. [ONLINE] Available at: https://www.redhat.com/en/topics/automation/what-is-orchestration. [Accessed 19 May 2023].

[16] Jack Roper, spacelift. 5 May 2023. *Infrastructure as Code : Best Practices, Benefits & Examples*. [ONLINE] Available at: https://spacelift.io/blog/infrastructure-as-code. [Accessed 19 May 2023].

[17] Brandon Cox, Red Hat. 9 February 2022. *5 software testing strategies to build into your CI/CD pipeline*. [ONLINE] Available at: https://www.redhat.com/architect/test-applications-cicd-pipeline. [Accessed 20 May 2023].

[18] Snyk. *Continuous security testing*. [ONLINE] Available at: https://snyk.io/learn/what-is-ci-cd-pipeline-and-tools-explained/continuous-security/testing/. [Accessed 20 May 2023].

[19] Rahul Padney, EuroSTAR. 3 January 2022. *Regression Testing in CI/CD. Can we Automate It Completely?*. [ONLINE] Available at: https://huddle.eurostarsoftwaretesting.com/what-is-regression-testing-in-ci-cd-can-we-automate-it-completely/. [Accessed 22 May 2023].

[20] Ishan Gaba, Simplilearn. 23 February 2023. *What is Version Control and What Are Its Benefits?*. [ONLINE] Available at: https://www.simplilearn.com/tutorials/devops-tutorial/version-control. [Accessed 22 May 2023].

[21] Atlassian. *What is version control?*. [ONLINE] Available at: https://www.atlassian.com/git/tutorials/what-is-version-control. [Accessed 22 May 2023].

[22] abdalslam. 7 February 2023. *Version Control Systems VCS Statistics, Trends 2023*. [ONLINE] Available at: https://abdalslam.com/version-control-systems-statistics. [Accessed 22 May 2023].

[23] Gitlab. *What is version control?*. [ONLINE] Available at: https://about.gitlab.com/topics/version-control/. [Accessed 22 May 2023].

[24] Tobias Günther, Git Tower. July 2020. *14 Git Hosting Services Compared*. [ONLINE] Available at: https://www.git-tower.com/blog/git-hosting-services-compared/. [Accessed 22 May 2023].

[25] Git. *Documentation*. [ONLINE] Available at: https://git-scm.com/doc. [Accessed 22 May 2023].

[26] Rowan Haddad, Flagship. 8 March 2023. *What Are the Best Git Branching Strategies*. [ONLINE] Available at: https://www.flagship.io/git-branching-strategies/. [Accessed 22 May 2023].

[27] Smartbear. *What is Code Review*. [ONLINE] Available at:1 https://smartbear.com/learn/code-review/what-is-code-review/. [Accessed 22 May 2023].

[28] Smartbear. *Common Code Review Approaches*. [ONLINE] Available at: https://smartbear.com/learn/code-review/what-is-code-review/. [Accessed 22 May 2023].

[29] Richard Bellairs, Perforce. 10 February 2020. *What Is Static Analysis? Static Code Analysis Overview*. [ONLINE] Available at: https://www.perforce.com/blog/sca/what-static-analysis. [Accessed 22 May 2023].

[30] Liku Zelleke, comparitech. 17 January 2023. *6 Best Static Code Analysis Tools*. [ONLINE] Available at: https://www.comparitech.com/net-admin/best-static-code-analysis-tools/. [Accessed 23 May 2023].

[31] Atlassian. *What is DevOps?*. [ONLINE] Available at: https://www.atlassian.com/devops. [Accessed 23 May 2023].

[32] Shreya Bose, BrowserStack. 4 January 2023. *What is DevOps?*. [ONLINE] Available at: https://www.browserstack.com/guide/ci-cd-vs-agile-vs-devops. [Accessed 23 May 2023].

[33] Shreya Bose, BrowserStack. 4 January 2023. *Difference between CI and CD, Agile and DevOps*. [ONLINE] Available at: https://www.browserstack.com/guide/ci-cd-vs-agile-vs-devops. [Accessed 23 May 2023].

[34] Ant Weiss, Medium. 30 July 2018. *Understanding Feedback Loops in DevOps*. [ONLINE] Available at: https://medium.com/antweiss/understanding-feedback-loops-in-devops-e93b92b74bd1. [Accessed 23 May 2023].

[35] Scott Andery, ArticleCube. *Issue Tracking System in DevOps*. [ONLINE] Available at: https://www.articlecube.com/issue-tracking-system-devops. [Accessed 23 May 2023].

[36] Community, Microsoft. 15 February 2023. *Welcome to Microsoft Teams*. [ONLINE] Available at: https://learn.microsoft.com/en-us/microsoftteams/teams-overview. [Accessed 24 May 2023].

[37] Slack. *Slack Features*. [ONLINE] Available at: https://slack.com/features. [Accessed 24 May 2023].

[38] Arfan Sharif, Crowdstrike. 6 February 2023. *What is DevOps Monitoring?*. [ONLINE] Available at: https://www.crowdstrike.com/cybersecurity-101/observability/devops-monitoring/. [Accessed 24 May 2023].

[39] Ravindra Savaram, MindMajix. *Logging DevOps Tools*. [ONLINE] Available at: https://mindmajix.com/top-11-logging-devops-tools. [Accessed 24 May 2023].

[40] Inonit AS. 21 August 2018 *Serafin*. [ONLINE] Available at: https://github.com/inonit/serafin. [Accessed 25 May 2023].

[41] PyO3. *Mapping of Rust types to Python types*. [ONLINE] Available at: https://pyo3.rs/v0.18.3/conversions/tables. [Accessed 25 May 2023].