Implementation of a DevOps environment with pre-commits for a Java web app

Dmitry Kochetov
BS17-SE-01 student
Innopolis University
Innopolis, Russia
d.kochetov@innopolis.ru

Ramil Askarov
BS17-SNE student
Innopolis University
Innopolis, Russia
r.askarov@innopolis.ru

Roberto Chavez
BS17-SE-02 student
Innopolis University
Innopolis, Russia
r.chavez@innopolis.ru

***Abstract*—In nowadays software engineering, DevOps has taken notorious importance, since with it developers and their teams can work consistently and simultaneously on software projects. In order to do so, many factors must be checked: how the commit process will be executed, tests that must be run to test changes in software, deploying the application in a specific environment, etc. DevOps is a set of techniques that addresses these tasks by performing them all in an effective manner by organizing all these steps with a highly structural implementation. DevOps environment must have a deployment pipeline which defines all the steps from its creation to the hands of its end user and ensures the product satisfies the expectations. The following work describes the implementation of a DevOps environment which can be automatically deployed on any machine. It automatically builds, tests and deploys a Java web app using the following open source tools: GitLab(SCV), Jenkins(CI/CD), Artifactory(artifact repo) and Docker/Ansible for creating different app environments such as development, stage and production.**

***Index Terms*—Open source software, Software design, Software reusability, Software architecture, Software testing, Pipeline processing, Application virtualization.**

I. Introduction

The project is a deployable DevOps environment implementation which contains CI/CD tools for automating building, testing and deploying for a basic Java web app which can perform basic math operations, i.e. it can add, subtract, multiply and divide numbers. Although this paper assumes that the reader already has understanding of Development and Operations principles, we will describe what a deployment pipeline is in order to briefly familiarize the reader with this field. Also we will describe our general objectives, and then in the following sections we will explain what our design consists of, including unique provided features and the tools that we decided to choose for the completion of this project.

A. Pipeline in Computer Science

We would like to start by mentioning what a pipeline is. A pipeline in a Software Engineering team is a set of automated processes that allows developers and DevOps professionals to reliably and efficiently compile, build and deploy their code to their production compute platforms [1]. There is no hard and fast rule stating what a pipeline should look like and the tools it must utilise, however the most common components of a pipeline are: build automation, continuous integration, test automation and deployment automation.

A DevOps environment generally consists of a set of tools which are normally broken down into the following categories:

- Source Control
- Build tools
- Containerisation
- Configuration Management
- Monitoring

The key objective of a software delivery pipeline is automation with no manual steps or changes required in or between any steps of the pipeline. Human error can and does occur when carrying out boring and repetitive tasks manually and ultimately does affect the ability to meet requirements and produce quality software.

B. Deployment pipeline

A Deployment pipeline is the process of taking code from version control and making it readily available to users of your application in an automated fashion. When a team of developers are working on projects or features they need a reliable and efficient way to build, test and deploy their work. Historically, this would have been a manual process involving lots of communication and a lot of human error. The stages of a typical deployment pipeline are shown in Fig.1 are as follows [2].

- Version Control: Software developers working on their code generally commit their changes into source control (e.g. github). When a commit to source control is made a the first stage of the deployment pipeline is started which triggers the code compilation, unit tests and code analysis. If all of these steps complete successfully the executable files are assembled into binaries and stored into an artefact repository for later use.
- Acceptance Tests: Acceptance testing is a process of running a series of tests over compiled code to test against the predefined acceptance criteria set by the business.
- Independent Deployment: An independent deployment is the process of deploying the compiled and tested artefacts onto development environments. Development environments should be a carbon copy of production

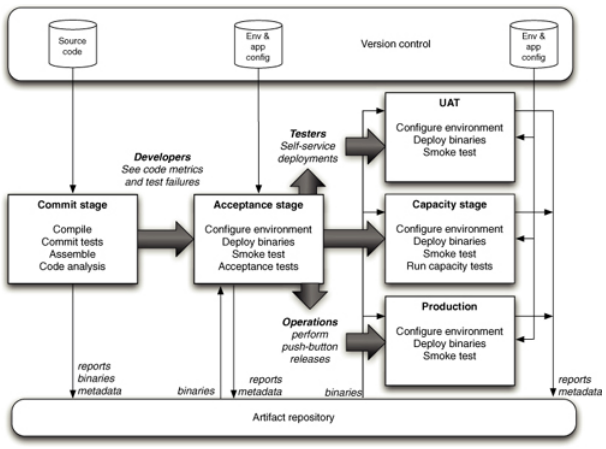

Fig. 1. General Operations diagram in a deployment pipeline.

environments or very similar at worst. This allows the software to be functionally tested on production like infrastructure ready for any further automated or manual testing.

- Production Deployment: Also called releasing. This process is normally handed by the Operations or DevOps team. This should be a very similar process to independent deployments and should deliver the code to live production servers. Typically this process would involve either deployments or releases to allow for zero down time deployments and easy version roll backs in the event of unpredicted issues.

C. Project Objectives

So far we have seen what a Pipeline in Computer Science is and how we can use it to perform the deployment of a software product. ***But how can we actually apply this theory to a real software?*** This methodology has been broadly implemented in software products development, and the purpose of this paper is to answer this question by implementing this deployment technique so that it can be of contribution for those embarking in the branch of Development and Operations. There are several ways to implement the pipeline of a deployment, some of them using an extensive set of tools and working with vast amounts of developers at the same time. However, most of them follow the general structure which was previously presented in this document. An example of such implementations can be found in the Tech Beacon technologies site, where the authors present a clear explanation of how to apply a deployment pipeline; and moreover, they indicate what tools can be used to carry out this process [3]. Our design will be based partially on the indications stated by Tech Beacon and also on the general structure of a deployment pipeline.

D. Project Considerations

Once that we understand how the general solution of our problem will be, we establish some considerations and limitations for the project implementation:

- The Integration server installation must be fully automated
- Split the pipeline into several stages each of them responsible for one task, following the general structure
- One limitation to consider is the computational power and requirements that the machines running this pipeline must have in order not to experience problems. Equipment and environment specifications are discussed in detail in the section ”IV. Discussion - Equipment and Environment specifications” of this document.
- The deployment will incorporate some unique features, such as precommits [4] and multiple concurrent stage servers, which will also be explained in the section ”IV. Discussion - Unique features of this implementation”.

Having established the general goals for this project and some particular considerations, we will describe our methods to achieve them.

II. Methods

In this section we will discuss about the tools used to carry out the solution of the project. We will also be explaining the procedures that were taken in order to implement the deployment pipeline and how the tools were used.

A. Project tools

There are quite a few DevOps tools that can execute different parts of the pipeline. For example, for the first stage of the pipeline, the commit stage, where the code must be fetched from a Version Control System (VCS), tools like GitHub or GitLab can be used. For further processes there are other technologies that can be used according the Operating system requirements and specific features of the pipeline. For example, Jenkins is an open-source automation server in which the central build and continuous integration process take place. It is a self-contained Java-based program with packages for Windows, macOS, and other Unix-like operating systems. With hundreds of plugins available, Jenkins supports building, deploying, and automating for software development projects [5]. Kubernetes is another example of a powerful tool in DevOps, since it is an open-source system for automating deployment, scaling, and management of containerized applications [6]. In fact, there are lots of tools that the development engineers can make use of. We leave to the reader’s curiosity and learning to look further into the properties and characteristics of this wide range of DevOps tools. Here we will describe only those tools that were used to implement this project.

1) Version Control System: For the first stage of our project we decided to work with **GitLab**, an open source code repository and collaborative development platform [7]. GitLab offers a location for online code storage and collaborative development of massive software projects. The repository includes version control to enable hosting different development chains and versions, allowing users to inspect previous code and roll back to it in the event of unforeseen problems. The

platform can be considered as competitor to GitHub, and as GitLab is developed on the same Git basis of version control, it functions very similarly. Some of GitLab's features include:

- Lightweight Directory Access Protocol integration [8].
- Community edition is open source (unlike GitHub).
- Pretty UI.
- Built in Web IDE.
- Commits can be tagged, it will be used for marking released versions.

Particularly, we work with GitLab because this version control system is the most suitable for our purposes, since it has support for many DevOps development tools, such as webhooks, integration with error tracking tools such as Sentry (an application Monitoring and Error Tracking Software), also logging, monitoring, and much more. Moreover, this system has even an auto DevOps function and its own CI/CD support but we preferred Jenkins to use in the project as CI/CD to get more experience of different tools.

2) Continuous Integration and continuous Delivery (CI/CD): Continuous integration is a coding philosophy and set of practices that drive development teams to implement small changes and check in code to version control repositories frequently. Because most modern applications require developing code in different platforms and tools, the team needs a mechanism to integrate and validate its changes. On the other hand, Continuous Delivery automates the delivery of applications to selected infrastructure environments. Most teams work with multiple environments other than the production, such as development and testing environments, and CD ensures there is an automated way to push code changes to them [9].

Therefore, we chose to work with **Jenkins** to implement this part of the project. Earlier in this document we already mentioned in short what Jenkins is. Here we would like to add that Jenkins has managed to establish well in the field of CI/CD since 2011 [10]. The assembly of the project is quite simple and intuitively configured via the web interface, but since it was required to make the deploying of the Integration server automated, it was spent a lot of hours to configure Jenkins through command line. A lot of settings allow developers to set up the pipeline for the needs of each project. We would also like to note that the assembly can be launched in various ways, for example, by the event of committing changes in the version control system, by schedule, by request to a specific URL, after completing another assembly in the queue, which allows developers to organize the deployment of even a very large project. Moreover, the advantages of this tool include a large number of available plug-ins, which very well expand its functionality.

3) Configuration Management (CM): Configuration management is a process for maintaining computer systems, servers, and software in a desired, consistent state. It's a way to make sure that a system performs as it's expected to as changes are made over time [11].

Configuration management also prevents small or large changes in a project from going undocumented. Without configuration management, we could end up not knowing what is on a server or what software updates were executed, which would make it hard to manage and maintain.

That is why implement CM in this project, and for this part we will use **Ansible**. Ansible is a technology that allows us to describe the environments we need by using simple yaml files in their structure [12]. Also, Ansible differs from other tools with the same functionality in its simplicity and in that it does not need to install an agent/client on target systems. Using this tool avoids the huge amount of written bash scripts and concentrates on the goal by writing human readable commands in YAML format. Moreover, this will allow us to deploy a necessary environment on any system with just a couple of commands, as well as facilitate the scaling of the current system. We use Ansible for both provisioning the Integration server itself and for provisioning Docker containers with tools for testing or running our app.

4) Artifact repository: We would like to start by defining what an Artifat repository is. An artifact repository can be thought of as a manager that controls end-to-end artifact lifecycle and supports different software package management systems while providing consistency to the CI/CD workflow [13]. An artifact repository is both, a source for artifacts needed for a build, and a target to deploy artifacts generated in the build process. Since we must build binaries only once to avoid related errors, the artifact repository is the best place to store and retrieve them. The DevOps artifact repository is crucial for the software development process, and that is why we also use it in our project. **JFrog Artifactory**'s universal artifact repository manager is one of the most popular at the moment and has managed to establish itself as an excellent tool. This repository has many integrations with CI/CD tools including Jenkins. Moreover, this repository has support for most binaries of different languages, which makes this tool universal. There is also support for many plugins that extend the functionality [14].

5) Virtualization: Service virtualization is the process of simulating the behavior of select components within an application to enable end-to-end testing of the application as a whole. Application development teams can use virtual services for emulating production or real services to conduct integration testing earlier in the development process [15].

For virtualization of the DevOps environment in this project we use **Vagrant** with **VirtualBox** as a provider. Vagrant is a tool for building and managing virtual machine environments in a single workflow. With an easy-to-use workflow and focus on automation, Vagrant lowers development environment setup time, increases production parity, and makes the "works on my machine" excuse a relic of the past [16].

Vagrant allows to very clearly and intuitively run images of virtual machines using just one file. In our project we used it for launching the *integration server*. Another tool that we utilize for the virtualization process in our project

is **Docker**. Docker is software for automating the deployment and management of applications in environments that support containerization. Allows you to ”pack” the application with all its surroundings and dependencies in a container that can be ported to any Linux-system [17].

In our system, each individual component is wrapped in a docker container, which allows us to isolate each application well from each other. Initially we wanted to use Virtualisation for our application servers but since these servers are inside the DevOps environment and VirtualBox does not support nested Virtualization, we moved to Containerization with Docker which even benefited us because of flexibility and decreased time of launching new containers.

6) Testing: Software testing in the DevOps world requires an automated test process that provides feedback at every checkpoint. Unit and Integration testing is needed on the build server to make sure the basic functionality works before proceeding to more time consuming tests. Functional testing, regression testing and more are needed on the test server [18].

- For the Unit and Integration testing we use **JUnit**, a library for testing JAVA applications [19]. In our case this tool will be necessary to test the backend part of our deploying project. With integration of **Selenium**, which is a tool for the automation of web browser actions [20], JUnit makes tests of frontend part ensuring that all the buttons execute what they are supposed to do.
- One of the important metrics of good code is test coverage. It measures how many lines of our code are executed during automated tests. We use **JaCoCo** to create code coverage reports for the project and if the coverage is less that a certain limit the pipeline fails.
- The last testing tool is **PIT** which provides mutation testing reports that can tell about quality of Unit tests. The tool creates copies of the code with slight changes like negated conditions that must force to fail the Unit test which cover this part of code. If it does not fail, it’s a useful indication for developers that the automated test must be reworked.

B. Project Architecture

1) Implementation of the Integration Server:

- Gitlab: An instance of Gitlab is deployed, with a pre-created user (which parameters can be configured in the configuration files) and the first repository is, in fact, our project. We have access to this instance with the development environment on the developer’s device and the development environment inside the integration server using the http protocol. Gitlab itself is also able to access the Jenkins services via set up webhooks.
- Artifactory: It is used to store candidates for the release. Jenkins deploys a specified artifact on the production environment.
- Jenkins: A Jenkins instance that receives webhooks from Gitlab and then takes the Jenkins file to launch the pipeline. Jenkins also creates a runner for each branch of the repository with the ”ready/*” mask, which contributes to the scalability of the integration server because there can be multiple such runners simultaneously. In addition to these runners, Jenkins has another, for the master branch, to deploy production environment.
- Ansible and Vagrant allow you to automate the deployment of the integration server so that it can be run on any machine without unnecessary actions

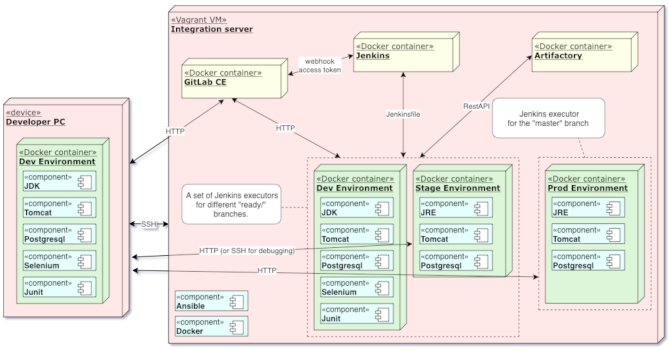

Fig. 2. High level architecture and tools for the Integration Server

2) Implementation of the Environments: We explain now how we develop the workflow of this process.

- Development Environment: The development environment is used for the active development of new features or fixes, it is deployed on the device of the developer himself, if necessary, have access to each other using ssh or http, for debugging. In Integration server this environment is used on ”ready/*” branches for building and unit testing, this process starts when a push to the branch with the ready mask occurs.
- Staging Environment: In this environment, a candidate undergoes acceptance and manual tests. If everything is successful the developer can move on and submit the changes to the master branch which is done in automated way in Jenkins, also the artifact will be pushed to Artifactory. Otherwise the developer will add new commits to the same branch to fix the issue.
- Production Environment: If the previous step was successful, it means the master master branch was updated with quality and tested code. With the help of Jenkins, the production environment is configured, and will pull the last artifact from the repository updating the version in production.

Fig.2 shows the design of architecture with used tools and how they are connected inside the Integration Server. Fig.3-5 show in what stages our tools are precisely used.

Up to this point, we have described the implementation of our pipeline and the tools that we used to deploy it. In the next sections we now will present the results that we obtained when deploying the system.

III. Results

Now we will discuss about what we have achieved when implementing our pipeline with the architecture that we de-

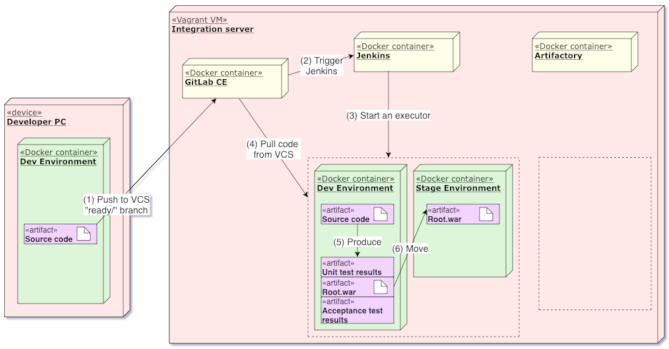

Fig. 3. Pipeline stages (1)

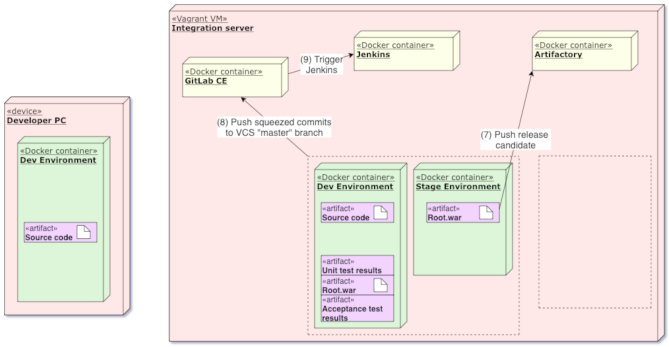

Fig. 4. Pipeline stages (2)

signed.

A. Functional Requirements

Functional requirements refer to the technical side of a software project. After our implementation, we accomplished the goals described below.

- The Code is the same during different stages.
- Different environments serve different purposes: The Development environment is for building and testing, stage for manual testing and production will be accessible by the end user.
- Code produced by one team can be integrated with code produced by other teams: Developers must pull changes from master otherwise their branch wont be accepted because it cannot be merged into master in a fast-forward way.
- Team members can work on different versions of the project and have build/stage environments for each version concurrently. Our architecture allows multiple ready/some-name branches where Devs can push and test their changes in a special remote environment assigned for their branch.
- Rollback option: If it appears that a bug squeezed through tests somehow, we can deploy an old working version.

B. Non-Functional Requirements

Non-functional requirements refer to the criteria that can be used to judge the operation of a system. In our project, we

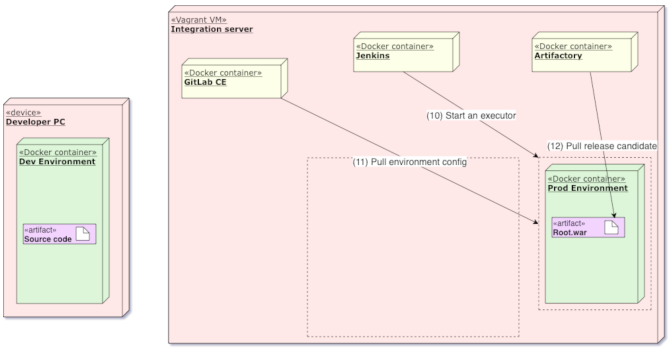

Fig. 5. Pipeline stages (3)

accomplished the next requirements:

- Achieved reliability in our project. This means that only tested code of a release candidate can be pushed to master by Jenkins. Thus, master branch is never broken
- Achieved good performance of deploying process. Using automated build scripts and test cases significantly reduces the cycle time. Project can be deployed on big mainframes
- Achieved efficiency by parallel running of Jenkins executors.
- Achieved maintainability - using configuration file settings changes easily
- Achieved scalability - Number of Jenkins executors can be increased easily

C. Processes

1) Build Process: It starts only after pushing to "ready/*" branches. It is completely automated with Maven, JUnit and other testing tools. The Build process also builds the .war candidate only once and takes less than 4 minutes for launching environments, building and testing.

2) Deployment Process: After a successful build, it deploys to the stage environment for manual testing. If the branch is accepted by user, the candidate is pushed to the Artifactory. This process takes less than 1.5 minutes for launching the stage environment.

3) Release Process: It starts only after pushing to "master" branch. It fetches the candidate from Artifactory. Overall, this process takes less than 1.5 minutes for launching production environment.

4) Rollback: It starts only after reloading a "master" branch pipeline with the corresponding tag. It fetches the candidate with the corresponding tag from Artifactory. The process takes less than 1.5 minutes for launching production environment again.

IV. Discussion

Now, we would like to discuss some factors that we experience during our work and also mention some technical considerations that must be taken into account to deploy the pipeline.

A. Initial Architecture and its posterior update

Initially the first architecture was relying on paid cloud services and deployment of the Integration server was not automated. After discussing with our supervisor, these problems were eliminated.

B. Problems during Development

As in any other project, teams experience difficulties, and it was not different in our case. First of all, at the beginning we had little to none previous experience with the tools. But this scenario was also a great opportunity to learn how to use them.

Secondly, as a group of 3 people it was hard to quickly make decisions and distribute tasks. It is necessary for a team to keep continuous communication but more importantly to be able to come to agreements in the shortest possible time.

Also, some things cannot be automated with and API or configuration files, therefore we had to use Selenium, and Jenkins plugins sometimes are not customizable enough.

C. Equipment and Environment specifications

1) Equipment Requirements: The computer for the Integration server must have at least 14GB of RAM and 40GB of hard drive memory in order to operate effectively.

2) Environment Requirements: The machine running Integration server must have installed only Python, Vagrant, Ansible and VirtualBox of versions specified in readme.txt file.

D. Unique features of this implementation

- The integration server can be deployed with one command.
- Because the system is modular, changing particular tool or adding new ones is simple.
- Precommit technique allows us to keep master branch always green, enforcing CI principles and not allowing the code which has not passed the tests to be merged by mistake.
- The architecture allows a few developers to build and test their versions of the app on the Integration server simultaneously.

E. Limitations in our Design

- Since the dev/stage environments are built each time from scratch, it takes time up to a few minutes to see the results of a pipeline.
- Everything including production server is inside Integration server, it introduces a Single point of failure. The system design must be distributed and replicated.

F. Further research

- Adding more different test suits to cover different cases of bug occurrence.
- Adding a caching layer for the Integration server to reduce time of dependency downloading.

V. Conclusions

The whole process of implementing a pipeline deployment is a journey which can be unknown and even terrifying for some fresh developers who are embarking in the world of DevOps. And that was in fact the case for us, the authors of this project. Some of us did not even know what DevOps means, but we can certainly assure that our experience and learning throughout the design and production of this pipeline has improved and grown in such a way that we will be able to perform well in the industry where DevOps engineering is required. And so we hope for the reader to experience the same results after reading and studying this paper. Finally, we would like to establish some conclusions on the technical and non-technical sides that we observed during and after the execution of this project.

A. Conclusions on the design

- According to the articles and books we read, precommit technique is not so common but we are convinced it helps developers who want to test their code by doing it outside of their computer, at the same time not disturbing other developers by ruining a common remote branch.
- Multiple dev/stage servers running in parallel eliminate time waiting in the queue, thus accelerating development speed.

B. Conclusions on the results

- Automating processes is not fun but the result can save a ton of time, we experienced it ourselves.
- DevOps techniques must be applied in every team developing software, even it's costly to change and get used to them, in a long run they benefit significantly in terms of speed and quality.

Acknowledgments

We would like to thank professor Alfredo Capozucca from Innopolis University, for teaching us the fundamentals of DevOps and review our work so that it can be well implemented. Also special thanks to Dmitry Kochetov, co-author of this document, who took on the leading role for the successful implementation and completion of the project.

References

[1] B. Son, "A beginner's guide to building devops pipelines with open source tools." [Online]. Available: https://opensource.com/article/19/4/devops-pipeline

[2] J. Humble and D. Farley, "Continuous delivery: Anatomy of the deployment pipeline." [Online]. Available: https://www.informit.com/articles/article.aspx?p=1621865

[3] R. Wilsenach, "How to set up your first deployment pipeline," Jan 2019. [Online]. Available: https://techbeacon.com/app-dev-testing/running-gauntlet-setting-your-first-deployment-pipeline

[4] L. Kruse, "To branch or not to branch..." May 2015. [Online]. Available: http://www.josra.org/blog/to-branch-or-not-to-branch.html

[5] Saurabh, "What is jenkins?: Jenkins for continuous integration," May 2019. [Online]. Available: https://www.edureka.co/blog/what-is-jenkins/

[6] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 167–167.

[7] G. Fedoseev, A. Degtyarev, O. Iakushkina, and V. Korkhov, "A continuous integration system for mpd root: Deployment and setup in gitlab," *Saint-Petersburg State University*, pp. 525–529, 2016.

[8] M. Rouse, “What is ldap (lightweight directory access protocol)?” Nov 2019. [Online]. Available: https://searchmobilecomputing.techtarget.com/definition/LDAP
[9] I. Sacolick, “What is ci/cd? continuous integration and continuous delivery explained,” Jan 2020. [Online]. Available: https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html
[10] “Jenkins official website.” [Online]. Available: https://www.jenkins.io/
[11] “Red hat - we make open source technologies for the enterprise. what is configuration management?” [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-configuration-management
[12] M. Mohaan and R. Raithatha, *Learning Ansible*. Packt Publishing Ltd, 2014.
[13] Mohamad and M. Masarwa, “What is an artifact repository?” Apr 2019. [Online]. Available: https://jfrog.com/knowledge-base/what-is-an-artifact-repository/
[14] “Universal artifact management for devops acceleration. official jfrog website.” [Online]. Available: https://jfrog.com/
[15] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, “Devops,” *Ieee Software*, vol. 33, no. 3, pp. 94–100, 2016.
[16] M. Hashimoto, *Vagrant: up and running: create and manage virtualized development environments*. ” O’Reilly Media, Inc.”, 2013.
[17] C. Anderson, “Docker [software engineering],” *IEEE Software*, vol. 32, no. 3, pp. 102–c3, 2015.
[18] “Devops and software testing. test.io website.” [Online]. Available: https://test.io/devops/
[19] E. Gamma and K. Beck, “Junit,” 2006.
[20] P. Watson, “Advanced selenium in java: With examples,” 2016.

Supplementary Material

As supplementary Material, we place the links of the project implementation for the reader’s research and learning.

- DevOps environment (GitHub) http://tiny.cc/jsw5oz
- The Java web app (GitHub) http://tiny.cc/00w5oz
- Presentation slides (GoogleSlides) http://tiny.cc/9pw5oz