

**UNIVERSITY OF VAASA**

**SCHOOL OF TECHNOLOGY AND INNOVATIONS**

**SOFTWARE ENGINEERING**

Jori Kankaanpää

**ON REDUCING RELEASE COST OF EMBEDDED SOFTWARE**

Master's thesis for the degree of Master of Science in Technology submitted for inspection, Vaasa, 11 November 2018.

Supervisor

Prof. Jouni Lampinen

Instructor

M.Sc. (Tech.) Lassi Niemistö

## PREFACE

Huge thanks for my supervisor Professor Jouni Lampinen and instructor M.Sc. (Tech.) Lassi Niemistö for all the assistance and constructive suggestions I've received for this study. Thanks also for everybody else who have given tips or otherwise helped me with the study.

I would also like to thank my family, colleagues and friends for all the support and patience during my studies in the past 5 years.

Vaasa, 18.10.2018.

*Jori Kankaanpää*

## TABLE OF CONTENTS

PREFACE	1
ABBREVIATIONS	4
ABSTRACT	5
TIIVISTELMÄ	6
1 INTRODUCTION	7
2 SOFTWARE DEVELOPMENT LIFE CYCLES	11
2.1 Waterfall	11
2.2 Agile	12
2.3 Release management	13
3 CONTINUOUS DELIVERY	16
3.1 Background	16
3.2 Continuous Integration	17
3.3 Continuous Delivery	21
3.4 Continuous Delivery in embedded domain	23
4 BUILD ENVIRONMENT ISOLATION	26
4.1 Containers as build environment providers	26
4.2 Docker for Continuous Integration	29
5 PLANNING	32
5.1 Current situation	32
5.2 Reducing work	40

5.2.1	Automate branch creation and version file updates	41
5.2.2	Updating configurations and building test packages	42
5.3	Controlling the pipeline	43
6	IMPLEMENTATION	46
6.1	Creating the branch creator for automatic branching and file updates	46
6.2	Creating the system exporter for providing test packages from CI	49
6.2.1	Isolating the build environment	50
6.2.2	Building test packages	54
6.3	Managing the pipeline with TeamCity	57
7	RESULTS	62
7.1	Results	62
7.2	Suggested next steps	64
8	CONCLUSIONS	66
	REFERENCES	68

## ABBREVIATIONS

<i>API</i>	Application Programming Interface, a definition according to which applications can communicate with each other.
<i>CD</i>	Continuous Delivery, a term with the meaning that product is ready to be deployed to the production anytime without long planning.
<i>CI</i>	Continuous Integration, a term with the meaning that each change is tested as soon as it is pushed to the repository.
<i>QA</i>	Quality Assurance, a process of verifying that the product fulfills the quality requirements.
<i>REST</i>	Representational State Transfer, an architectural style for implementing web services.
<i>SDLC</i>	Software Development Life Cycle, a process describing the whole software development process.
<i>TTM</i>	Time-to-market, time from a product idea to the finished product which is available on the market.
<i>VCS</i>	Version Control System, a system to store source files along with the versioned history.
<i>VM</i>	Virtual Machine, software which imitates physical hardware making it possible to install multiple machines inside a single physical machine.

---

**UNIVERSITY OF VAASA****School of Technology and Innovations**

**Author:** Jori Kankaanpää  
**Topic of the Thesis:** On Reducing Release Cost of Embedded Software  
**Supervisor:** Prof. Jouni Lampinen  
**Instructor:** M.Sc. (Tech.) Lassi Niemistö  
**Degree:** Master of Science in Technology  
**Major of Subject:** Software Engineering  
**Year of Entering the University:** 2013  
**Year of Completing the Thesis:** 2018

**Pages:** 73

---

**ABSTRACT**

This study focuses on lowering release cost for an embedded software project by improving the continuous integration pipeline and by moving towards continuous delivery. The study is made as an assignment for a Finnish software company. The case project is embedded software project written with C/C++ programming languages. Additionally, the project consists of a desktop tool for managing the embedded systems, but no special focus is given to this tool. The goal of the study is to reduce both the total time of the deployment pipeline and the amount of active manual working in the pipeline. This is achieved by automating tedious steps of the release and by constructing an automated pipeline which produces all the needed files for the release.

The work began by exploring the previous release process and by identifying the complicated or time-consuming parts of it. Based on the findings, three main focus areas were selected for development: work related to branching and file updates, work related to updating test systems configuration and work related to building the test binaries. After this, each of these three focus areas were improved one at a time by building tools to automate the steps with Python and Kotlin programming languages. Additionally, the continuous integration pipeline was further developed by taking Docker containerization technology into use, which provided better build environment isolation giving a possibility to better utilize binaries produced by the continuous integration server.

As a result of the study, a proposal for the improved release process was created focusing on the automation of the tedious steps. With the new process total deployment time went down to about 4 hours from previous 7 hours and 40 minutes, and the active manual work went down to a bit less than 1 hour from previous 4.5 hours. Additionally, some of the steps might be repeated multiple times during a release. On the other side, it was found out that the process also had some steps which were not feasible to automate such as steps which currently require manual consideration from release engineer. Due to this, the resulting pipeline is not yet fully automatic. This would be a good candidate for a further study since overcoming this issue would make the pipeline fully automatic after the code freeze which would further increase the benefits.

---

**KEYWORDS:** software engineering, software release process, continuous delivery

---

**VAASAN YLIOPISTO****Tekniikan ja innovaatiojohtamisen yksikkö**

<b>Tekijä:</b>	Jori Kankaanpää
<b>Diplomityön nimi:</b>	Sulautetun ohjelmistoprojektin julkaisukustannusten alentaminen
<b>Valvojan nimi:</b>	Professori Jouni Lampinen
<b>Ohjaajan nimi:</b>	DI Lassi Niemistö
<b>Tutkinto:</b>	Diplomi-insinööri
<b>Oppiaine:</b>	Ohjelmistotekniikka
<b>Opintojen aloitusvuosi:</b>	2013
<b>Diplomityön valmistumisvuosi:</b>	2018

---

**Sivumäärä: 73****TIIVISTELMÄ**

Tämän diplomityön aiheena on sulautetun ohjelmiston julkaisukustannusten alentaminen pyrkimällä lähemmäksi jatkuvan toimituksen prosessia. Työ toteutetaan suomalaiselle ohjelmistoyritykselle. Työ liittyy C/C++-pohjaiseen sulautetun järjestelmän ohjelmistoprojektiin, jonka asetusten säätäminen ja monitorointi tapahtuu erillisellä työpöytäsovelluksella. Tavoitteena on vähentää ohjelmiston julkaisuun liittyvien manuaalisten työvaiheiden määrää sekä niiden vaatimaa aikaa rakentamalla automatisoitu julkaisuputki, jonka lopputuloksena saadaan tarvittavat tiedostot ohjelmiston julkaisemiseen. Työpöytäsovelluksen julkaisuprosessiin työssä ei kiinnitetä erityistä huomiota.

Työ alkoi selvittämällä entisen prosessin kulku ja eri vaiheisiin kuluva aika, sekä se paljonko vaihe sisältää aktiivista manuaalista työtä. Selvityksen perusteella valittiin prosessin osat, joiden parantamisesta saavutettaisiin suurin hyöty. Prosessin kuvauksen perusteella havaittiin, että prosessissa on kolme osaa, joiden parantamiseen tulisi kiinnittää huomiota: julkaisuhaarojen luonti ja siihen liittyvät tiedostojen päivitykset, sulautetun järjestelmän asetusten päivitykset ja testaamista varten luotujen sulautettujen testiohjelmien kääntäminen ja paketoiminen. Myöhemmin näitä vaiheita kehitettiin muun muassa rakentamalla Python- ja Kotlin-ohjelmointikielillä apuohjelmia, jotka automatisoivat vaiheiden suoritusta. Lisäksi käännösprosessia kehitettiin ottamalla käyttöön Docker-konttitekniologia, mikä mahdollisti ympäristön paremman suojaamisen virhetilanteilta. Muutos mahdollisti jatkuvan integraation palvelimen luomien testiohjelmien laajemman käytön.

Työn tuloksena syntyi ehdotus uudeksi julkaisuprosessiksi, jossa automaation määrää on lisätty. Ehdotuksessa manuaalisten vaiheiden määrä väheni ja virheiden mahdollisuus prosessin aikana pieneni. Vaiheisiin kuluva kokonaisaika pieneni noin puoleen alkuperäisestä. Aktiivisen manuaalisen työn määrä väheni noin 80 prosentilla. Toisaalta todettiin, että prosessissa on sellaisia vaiheita, joiden automatisointi ei vielä tässä vaiheessa ollut mahdollista ilman lisäpanostusta niiden vaatiman tapauskohtaisen harkinnan vuoksi. Tämän vuoksi systeemin asetusten päivityksen automatisointia ei saatu täysin toteutettua. Julkaisuprosessin sujuvoittamiseksi se olisi kuitenkin hyvä jatkotutkimuksen kohde.

---

**AVAINSANAT:** ohjelmistotuotanto, ohjelmiston julkaisuprosessi, jatkuva toimitus

## 1 INTRODUCTION

The functional requirements of the software development process are increasing. At the same time, software should be developed faster with fewer resources while also keeping the number of software defects low. The market demands that the software release times are reduced and that the customers start seeing the added value from the software as soon as possible. These different demands conflict with each other and improvements to the whole software development process are needed in order to stay relevant in the competing field.

Back in the 1990s, the most commonly used software development life cycle (SDLC) model was the waterfall model (Isaias & Issa 2015). Over the time it has been observed that the given model is not often the optimal due to issues it has, especially regarding the requirement change management during the process (Rajlich 2006). As a result, various new models have emerged such as many different agile methods. Nowadays using one of the agile methods in one form or another is more of a norm than an exception. In a study conducted by Rodríguez, Markkula, Oivo and Turula (2012) 58% of 200 participated Finnish companies reported using agile or lean methods.

In order to adequately support an agile software development process, various practices and tools have emerged. Continuous integration (CI) and continuous deployment (CD) are practices that have recently gained a lot of attention in the companies. Using the agile software development model along with the continuous integration is supposed to help releasing software in faster cycles while keeping the quality of the software high (Fowler 2006).

Having a short release cycle is beneficial for a software project since that allows customers to start gaining value from their investment early on and the feedback cycle also gets shorter which benefits the requirement management and overall efficiency. However, achieving full continuous deployment might be a troubled task for a complex software project which has not been built with the continuous deployment in mind. Often there



might be for example some manual steps which require human intervention. Some process for handling situations like this then needs to be created.

This study is made for a Finnish software company Wapice Ltd where the author of the study has been working since 2013. The background to the research is that there was a request from a customer in 2017 to reduce the costs related to releasing a new version of the software developed by Wapice. To fulfill the request, multiple projects were launched. One of those was related to reducing manual work that needs to be done every time a new version of the software is released to the customer. This is the part this study will attempt to cover.

This study will focus on the matter of reducing software release costs with the help of continuous delivery in a complex software project. The goal of the study is to reduce software release costs by automatizing steps in the software release process. After the single tasks are automated, the goal is to build an automatic pipeline where time-consuming tasks are done automatically after the user has given the needed inputs.

The actual project consists of two main parts: embedded software which is run on the customer's embedded hardware and the desktop software used for configuring and monitoring the embedded systems consisting of the said embedded devices. The embedded software is packaged into Debian Linux package and distributed as such to customer. The customer further uses the Debian package to build customized packages for the different installations. Debian package is used since the main development environment is currently based on Ubuntu Linux. The desktop software is created using Qt-framework and it currently supports only Windows environment. Desktop tool is distributed as a single installer capable of installing the tool to the user's machine.

The pipeline for the desktop application is currently in better shape than the one for releasing the embedded software. There is also a separate project for reducing the release cost for the desktop application. Thus, this research will mostly focus on automatizing the release steps of the embedded software part of the software project. At the beginning of a study, it is known that there are currently many manual steps involved in releasing

an updated version of the embedded software. The goal of the study is to minimize this manual work and handle the entire process in a more organized way.

This study is limited to using continuous delivery to reduce the software release time and cost. For example, automatic testing is known to be a valuable tool for reducing release costs, but this study will not focus on the matter unless it is strictly related to continuous delivery. Continuous delivery is considered mainly for an embedded software development process. Thus, the solutions used might be different from the ones that would be used for a more typical web-based application. For the continuous integration server, the focus is limited in the study to a continuous integration server produced by JetBrains called “TeamCity”. As part of building an automated pipeline, some improvements to the existing build system are also to be done. To increase the robustness of build environment containerization technology is to be used. There, the study will limit the focus to Docker container technology which is supported by TeamCity out of the box. The study will not put much focus to alternative container technologies.

The goal is to start initial work for the study on the last quarter of 2017. The practical part of the study is to be finished in the second quarter of 2018 and documenting work will shortly follow practical part. Work will be finished at the latest by the autumn of 2018.

The study will begin with a literature review. In the literature review, the first chapter goes briefly through the advantages and issues of the agile software development life cycle models compared with the more traditional waterfall model. Ways to manage software releases are also shortly introduced in this part. After that, the focus is moved to continuous delivery, what it means, what are the benefits of it, what issues there often are when implementing it, specifically on embedded environments and which tools are available for helping to accomplish that. As the last part of the literature review, Docker container technology is to be explained and discussed how it can be utilized to improve the robustness of the build environment.

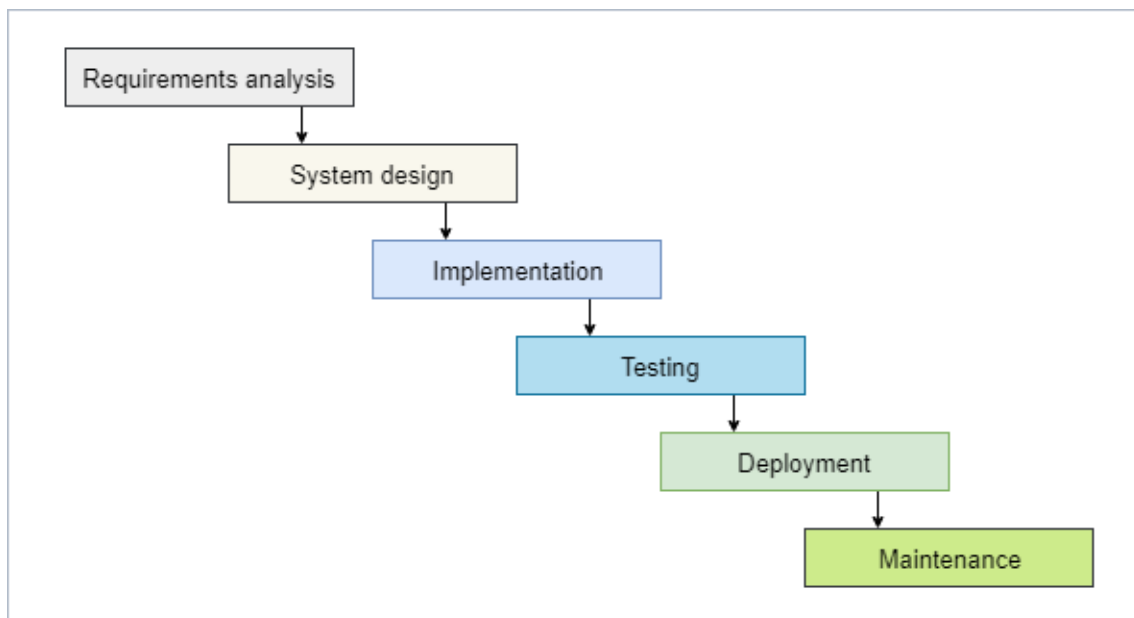
After the literature review is done, there will be a practical study for reducing the software release cost by striving towards the continuous delivery process for the case project. The

practical part will begin by introducing the current situation followed by the plan where the main points of focus are selected. Then the practical work is conducted after which the findings and results are analyzed, and recommendations for the future steps are given. Finally, in the conclusions chapter the progress of the whole study is evaluated.

## 2 SOFTWARE DEVELOPMENT LIFE CYCLES

### 2.1 Waterfall

The waterfall model is a traditional software development life cycle model that has been in use for a long time. It often consists of six stages although this depends slightly on how the stages are count. The six stages are: requirements analysis, system design, implementation, testing, deployment and maintenance. The process begins with the requirements analysis phase where the requirements of the application are collected and often a requirements document is also created. In the system design phase system to be created is analyzed and business logic is decided. In the implementation phase, the actual program is written and integrated and its functionality is verified in the testing phase. Once testing is done, the program is deployed to the production in the deployment phase which is followed by the possibly long-lasting maintenance phase. (Pfleeger & Atlee 2010: 75.) These stages are visible in Figure 1.



**Figure 1.** The traditional waterfall model with six stages. The model moves sequentially from top to bottom one stage at a time.

The waterfall model is a very structured model and it flows from the first phase to the last phase sequentially starting a new phase only after the previous phase has finished (Pfleeger & Atlee 2010: 54). This has a benefit that it forces a project to work in a structured manner which is often necessary for a large software project (Powell-Morse 2016). On the other hand, the waterfall model is considered inadequate in various ways. It has been criticized for not reflecting the usual software development process where software is developed iteratively (Pfleeger & Atlee 2010: 76). Also, due to structural, linear nature, testing begins very late in the life cycle leading to the late discovery of issues. It is also suboptimal for changing requirements during the software life cycle since the requirement gathering is done very early in the process and once it has finished those are not re-checked. (Powell-Morse 2016.)

In modern industry time-to-market (TTM) is also a key factor for customers (Kwak & Stoddard 2004). That is, how much time it takes to start gaining value from the point the new idea was invented. The waterfall model is not optimal for this because the whole application needs to be finished before it can be deployed. This makes for long feedback times which raises possibilities for problems late in the development cycle (Powell-Morse 2016).

## 2.2 Agile

Agile methods were invented to overcome some of these inflexibility issues of the waterfall model. Ideas central to agile software development were introduced in the Agile Manifesto by Beck et al. (2001) which states: “We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.”

Faster releases and increased flexibility during the development process are often considered the strengths of the agile methods (Begel & Nagappan 2007). Agile methods often also improve communication both inside the project team and to the end customer (Begel & Nagappan 2007). These can be considered substantial competitive advantages on the fast-changing market.

There are various software development frameworks which adhere to the principles of the Agile manifesto. Out of those frameworks, Scrum is one of the most popular ones (Begel & Nagappan 2007). Scrum consists of short sprints during which small part of the software is developed according to priorities set by a customer. At the end of the sprint, software is supposed to be in a working condition. Quick reaction to problems is achieved with the daily scrum meetings which are short meetings where issues that have aroused are handled. (Schwaber & Sutherland 2018.)

### 2.3 Release management

While agile methods might reduce the time to market metric, there are other important factors related to software release costs. Significant portion of the total costs associated to a medium to large software project are typically related to the release process, project management and to the testing and quality assurance which are often part of release process (Saleh 2010; XebiaLabs 2018). In the market, there are multiple tools for helping with managing the release process such as XL Release from XebiaLabs and BuildMaster from Inedo (Inedo 2018a; XebiaLabs 2018).

Both BuildMaster and XL Release allow for example modeling the release process, inserting manual steps into the release process, inserting automatic steps into release process and setting approval gates between the steps. They also both visualize the state of the releases. Additionally, both tools integrate into many other services often used during the release such as issue trackers and continuous integration servers. (Inedo 2018a; XebiaLabs 2015; XebiaLabs 2018.) This makes it possible to have more control over the

release process providing more reliable and reproducible releases. Example view from BuildMaster is presented in Figure 2.

The screenshot displays the BuildMaster 6.0.3 release management tool interface. The top navigation bar includes links for Releases, Applications, Pipelines, and Plans. A dropdown menu shows 'Master's thesis' with a search icon. The main header area shows 'Master's thesis >' and 'Master's thesis First draft'. Below this is a sub-navigation bar with links for Overview, Releases, Packages, Plans, Pipelines, Issues, Assets, and Settings. The 'Release Overview' tab is selected. The main content area is divided into several sections: Details (Pipeline: Master's thesis, Created: 25.4.2018 0.07.11 by Admin), Release Variables (No variables to display), Notes (No notes have been added), Target Dates (Writing done: 1.4.2018 - 30.5.2018), Config File Versions (No configuration files for this application), Included Deployables (No deployables associated with this release), Issues (Task #1, Closed, Write conclusions), and SQL Scripts (No change scripts for this release). A 'Purge Release First draft' button is visible in the bottom right. The footer shows 'BuildMaster © 2018 Inedo, LLC' and 'Version 6.0.3 (Build 2) • Logged in as: Admin'.

**Figure 2.** Example view from BuildMaster 6.0.3 release management tool.

Thus, the usage of a release management tool could help improve the release process and reduce time wasted for example waiting for approval since the release management tool can notify required parties when the input from them is needed. However, both BuildMaster and XL Release are commercial programs (Inedo 2018b; XebiaLabs 2018). Therefore, the license costs should be considered when deciding their usage. As of April 2018, BuildMaster has also a community version which can be freely used, but it has the

limitation that each user is an admin on the service, so no proper access control is possible (Inedo 2018b). No information is available about the pricing of XL Release from the website of the product.



### 3 CONTINUOUS DELIVERY

#### 3.1 Background

Nowadays the agile methods are widely used in the software industry due to their ability to better respond to continuously changing customer needs. As mentioned before in the introduction, according to study by Rodríguez et al. (2012) 58 % of the studied Finnish software companies reported using some form of agile or lean development method.

Naturally, the widespread adaption of the agile methods has caused interest in software tools that can help in adopting the agile methods, and as a result, various tools and practices have emerged to support working according to those methodologies. As the software needs to be in a working condition at the end of each short sprint, it is vital to keep it in a functioning state continuously. This is needed to avoid excessive integration work at the end of each sprint. Doing the integration work late in the software life cycle is often costly and quickly leads to project delays (Duvall, Matyas & Glover 2007). Continuous integration (CI) and continuous delivery (CD) are practices often used for avoiding integration issues at the end of a software process (Fowler 2013).

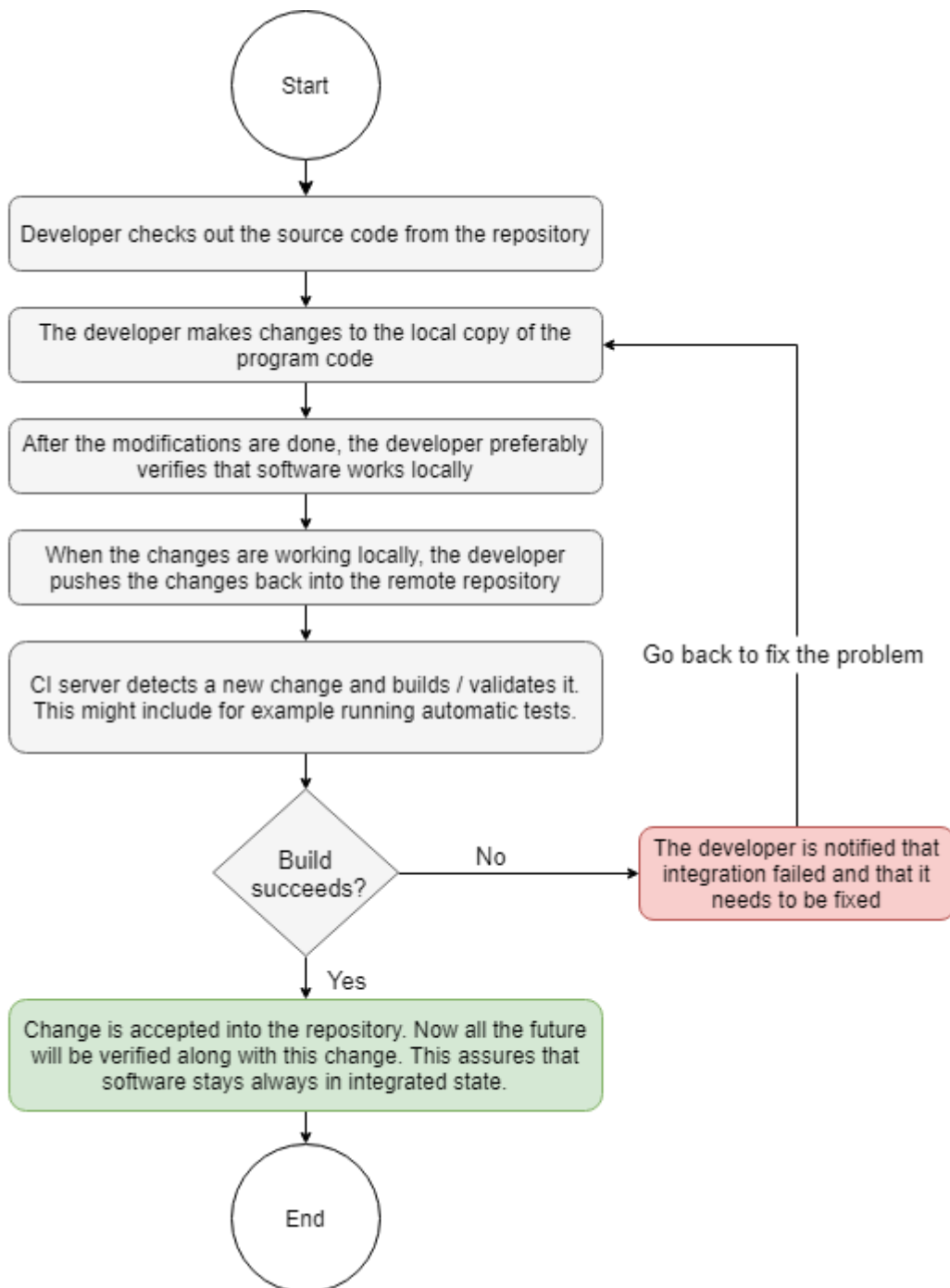
In the continuous integration developers integrate their work back into the main repository regularly. When the integration happens, a continuous integration server verifies that the integration is successful by validating the change against the rest of the repository. If integration fails, the developer is notified by the integration server. In the continuous delivery, this idea is taken further, and software is not only integrated, but also otherwise prepared for deployment such that the new version could be released each time the developer successfully integrates his work into the main repository. Integrating software regularly makes it possible to notice issues earlier and to release software more often. (Fowler 2013.) However, there might also be issues preventing the continuous integrations such as lack of the testing hardware (Lwakatare et al. 2016).

### 3.2 Continuous Integration

Continuous integration is one of the practices often used to help with implementing agile development methods. The term continuous integration is originating from one of the Extreme Programming's twelve practices. In the continuous integration team members integrate their work frequently into the main development branch. This integration usually happens at least once per day, and it can be automatic or manual. When the change is integrated, it is common to run some basic test set for it to catch possible integration issues early on. Doing the frequent integrations helps to keep the software in the releasable state during the whole development cycle. (Fowler 2006.)

The usual workflow for the developer in a continuous integration environment is described by Fowler (2006) followingly: as usual, the workflow process begins by upgrading one's local version of software sources from a remote version control repository. After that, the changes are made to the local version of the software, and it is verified that building the software still succeeds. Once verification is done, the developer can push changes back to the remote repository.

Then comes the actual CI part: the software is built for the second time on a separate CI machine (CI agent) which might execute various additional steps such as executing automatic tests during the integration pipeline. At this point, it is verified that developer's new modifications work well with everyone else's work. If something fails, the CI system sends an alert notification to the developer that there is something wrong with the updated version and that it failed to integrate cleanly with the main branch. This way, the issue is detected early in the process and can be fixed quickly. (Fowler 2006.) Figure 3 demonstrates the process.



**Figure 3.** The diagram describes the usual straightforward continuous integration process.

Of course, the described process is a straightforward one, and it could be easily taken further. For example, it might be a clever idea to produce an application installer as an output (artifact) of a successful build thus enabling the customers or other developers to

continuously test the latest version of the software (Fowler 2006). Another option is to go all the way to the continuous deployment: after the software is integrated successfully and tests pass it is possible to make yet another step that deploys the updated version of the software to the production server automatically (Fowler 2006).

Implementing the continuous integration provides a project with many benefits. One benefit is that CI system helps reducing assumptions by doing a rebuild of the software each time the change is made. CI can also be considered a vital part of project QA as it can be used for determining software health after each change. (Duvall et al. 2007: 24-25.) In the book by Duvall et al. (2007: 29) the high-level values of CI are described as: “

- Reduce risks
- Reduce repetitive manual processes
- Generate deployable software at any time and at any place
- Enable better project visibility
- Establish greater confidence in the software product from the development team“.

Additionally, CI system enables some other benefits. These include the ability to find and fix software bugs early, decrease the cost of new changes and ability to take software into use with smaller risk (Rasmusson 2010: 235).

However, adopting the continuous integration is not always a trivial task. Some problems found out were for example skepticism to benefits, the fear that implementing the CI will cost more than the benefit is, the poor maturity of tools required for supporting CI and the doubt of applicability of CI to all organizations and projects. In addition to these some more technical problems were found out such as too long feedback times for the CI system to be useful, too many manual tests to integrate frequently, the poor visualization of the build pipeline and the need for stricter software dependency management. (Debbiche, Dienér & Svensson 2014.)

Duvall et al. (2007) also point out some concerns commonly faced when thinking about taking a CI system into use. One problem mentioned is that people are worried that maintaining the CI system is too much extra work. Another problem that was mentioned is

that people might be worried that implementing the CI system in the middle of a project poses a too massive change causing a risk to the project. Other concern mentioned is that wrongly applied CI might cause some issues such as build instability which reduces the benefits of the system. Yet another concern comes from the needed software licenses and hardware costs. (Duvall et al. 2007: 32-33.)

In the market, there are various CI server tools available for different needs. In the embedded software project on-premises hosted solution is often necessary in order to get easy access to the tested hardware. Tools allowing this include TeamCity, Jenkins, Bamboo, CircleCI, and Travis CI Enterprise. Some of those are open source such as Jenkins and some are mainly cloud-based hosted solutions, even though they include commercial on-premises versions as well such as Travis CI. (Pecanac 2016.)

TeamCity is the CI solution that has been used in the case project for approximately two years now. TeamCity is a CI server by JetBrains which can be either self-hosted locally or hosted on one of the cloud providers. TeamCity provides a professional version free of charge. However, the professional version has some limitations which make it unsuitable for large software projects. Limitations are: only 3 build agents can be registered at the same time, and only 100 build configurations might be used. To overcome the limitations, JetBrains offers an enterprise edition of the TeamCity which has the same features, but the limitations can be scaled up by purchasing a suitable license. (JetBrains s.r.o. 2018a.)

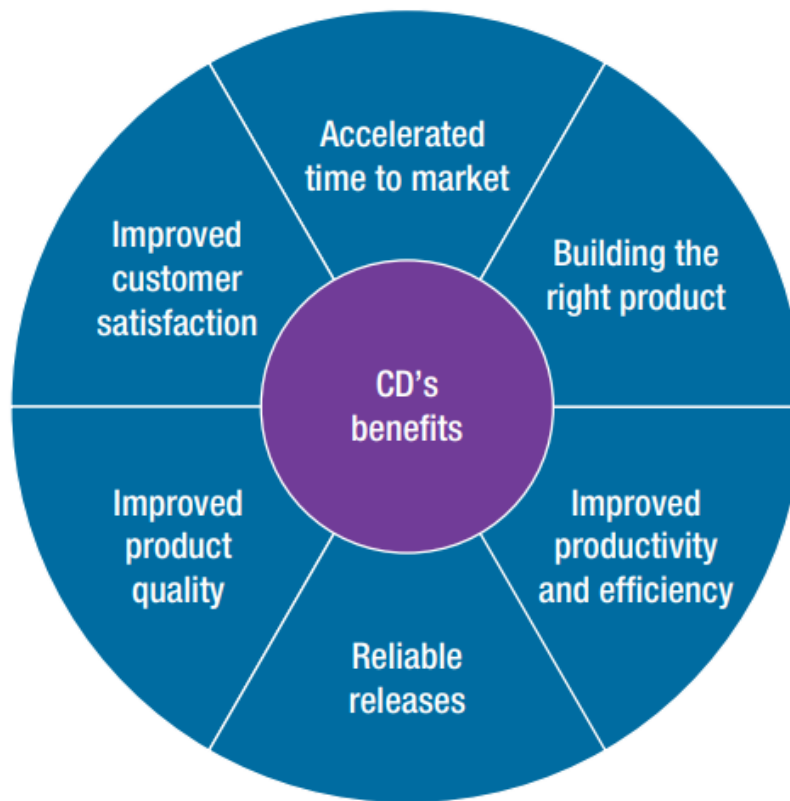
C and C++ environments used in the case software project are supported by TeamCity along with many different environments (JetBrains s.r.o. 2018c). In TeamCity 2017.2 official support for using Docker as part of the build pipeline was also included. This new feature allows, for example, running each software build inside a new Docker container. (JetBrains s.r.o. 2017.) The feature can be used to provide better isolation of the build environment and easier replication of the environment. This feature will be used in the case study and using Docker will be covered in more detail in Chapter 4.

### 3.3 Continuous Delivery

Continuous delivery builds on top of the continuous integration. While CI usually refers to the integrating, building and testing each change, this does not mean that everything needed for deployment is done in the CI pipeline. There might, for example, be additional work such as updating the environment, deploying the packages to the servers, updating the configuration files or other activities related to the release process which are not done as part of the continuous integration pipeline. Continuous delivery is filling the needed holes for the product to be ready for deployment at any point in the lifecycle. (Fowler 2013.)

Continuous delivery is an approach where software is kept in the releasable state during the whole life cycle so that it can be reliably released at any given time to the production. It is believed that there are numerous benefits from doing this such as the ability to bring new features and improvements rapidly and reliably to the market. This is often considered a substantial competitive advantage. Not using continuous delivery approach might lead to a situation where each release is developed for months and features completed early on the cycle unnecessarily wait a long time before they can be released to the customer. This might reduce or even completely remove the value that could have been acquired with the feature. (Chen 2015.)

Another problem with which the continuous delivery might help with is a disorganized release process. When the release is done only once in a few months and when the release process contains numerous manual steps the execution is often disorganized and error-prone. With the continuous delivery, the release process occurs more often which makes it easier to remember. Implementing CD frequently requires also stripping the unnecessary complications away from the process. Continuous delivery also often improves product quality and customer satisfaction because feedback for the changes is received more frequently. (Chen 2015.) Figure 4 lists benefits of applying continuous delivery to a software project.

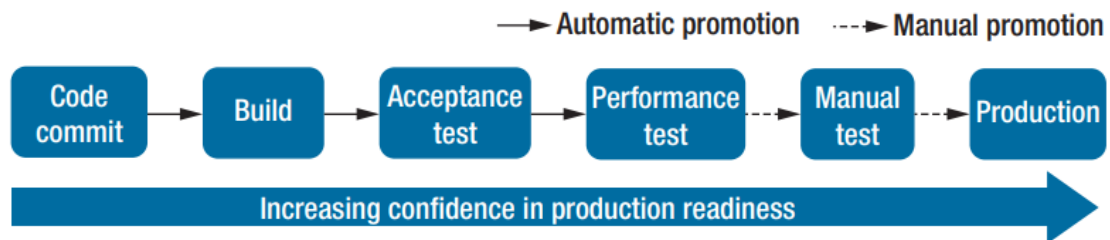


**Figure 4.** Typical benefits of practicing continuous delivery in a software project (Chen 2015).

Implementing the continuous delivery might sometimes be problematic. One problem mentioned by Chen (2015) is that release process usually involves many different teams that might have separate interests. For example, setting up the test environment might need support from operations team which might not be keen on giving too strong access rights to servers for another team as they might fear something will be broken. Another problem mentioned by Chen is that release processes often involve long bureaucratic steps which might take multiple days making delivery take too long time. Lastly, he mentions that there are currently no robust and comprehensive tools for supporting continuous delivery. Often necessary tools need to be developed by the developing organization themselves which takes lots of resources and might involve multiple tools for achieving all the requirements.

The continuous delivery pipeline can be automatic, semi-automatic or a manual. The pipeline often starts when the code is committed to the repository. After that, the CI server

usually builds the software and runs unit tests for it as was described in the previous chapter. However, in continuous delivery, there are usually more steps after it. For example, after the build has finished, there might be more extensive acceptance tests which are executed. There might also be manual tests and finally, the deploy step. The pipeline would advance to the next step only if the current step was executed successfully without problems. Promotion to the next step might be automatic such as when the next step begins if integration tests pass or manual such as when a release manager manually marks manual tests as executed leading to the product deploy phase to begin. Figure 5 represents an example CD pipeline.



**Figure 5.** Example pipeline with automatic and manual promotions between the steps (Chen 2015).

### 3.4 Continuous Delivery in embedded domain

Continuous delivery is most commonly used in the web domain as there are various tools for supporting it there. For example, virtualization and configuration management tools help setting up the test and production environments quickly and scaling up the processing power when needed. However, despite the benefits of continuous delivery, it has not been yet as commonly taken into use in the embedded system domain. This is not necessarily because the benefits of the continuous delivery would not be applicable to the embedded domain but rather because of the additional obstacles embedded systems development imposes. (Lwakatare et al. 2016.)



Lwakatare et al. (2016) conducted a multi-case study with an interpretive approach about the adoption of DevOps practices in the embedded systems domain. In the study, they collected data from four Finnish companies which develop embedded systems. They found out four key categories for issues of adopting the CD practices on the embedded domain.

The first problem found out was the usual organization structure. In the web companies, development is usually done by self-organizing feature teams with the required skills and tools to develop and test new features. On the embedded side, development is more often done in module teams which focus on some particular low-level part of a system. These teams tend to require specialization as they work closer to hardware. This kind of structure makes the importance of communication more crucial, since with the specialized teams it might easily happen that members of the team are not aware of what is happening outside the team. Moreover, having the hardware dependency often prolongs the development cycles and feature releases. (Lwakatare et al. 2016.)

The second frequent problem is the lack of proper test environments. Embedded software teams often do not have proper access to test environment which closely matches the one used by the customer. In the web domain creating a new test environment is easy but it is not the same in the embedded project where there are dependencies to the specific hardware used by the customer. (Lwakatare et al. 2016.)

The third problem found out by Lwakatare et al. is the lack of tools. While for the web domain there are various open source tools for automating the deployment process, very few tools exist for the embedded domain. They found out that there are no tools which would allow new software to be deployed reliably on a continuous basis to the target devices. This problem was even more severe in the critical embedded systems. (Lwakatare et al. 2016.)

The last found issue was about the lack of usage data. In the web domain companies often collect data about how their services are used and by whom. This data can be further processed to find out development targets for the continuous improvement. In the

embedded software domain, monitoring is often done only for the fault analysis and the feature usage information is not collected. The data is also often saved on the device or on the customer side leading to a situation where the developing company might not have easy access to it. This makes continuous improvement harder to do on the embedded domain. (Lwakatare et al. 2016.)

Despite the problems of adoption of continuous delivery for the embedded software projects, the benefits of practicing it would still be valid. For example, cutting the time-to-market time down by continuously building and testing software is not limited to the web application domain but instead would be beneficial to any project. Another reason why CD might be important in the future is cyber security of the embedded devices, which requires frequent updates.

In order to make continuous delivery more feasible in the embedded system domain, techniques have been studied for overcoming some of the limitations mentioned above. For example, one alternative solution to lack of test equipment is using simulation (Engblom 2015). In this solution real hardware is simulated using a virtual platform which runs on standard PCs and servers (Engblom 2015). This simulated platform can use code implemented for the embedded device and thus the testing becomes much more accessible (Engblom 2015). When the environment is running on a typical PC, for example, technologies used for web domain environment setup might be used.

## 4 BUILD ENVIRONMENT ISOLATION

### 4.1 Containers as build environment providers

Another part of a continuous delivery pipeline is setting up the build and test environment. Preferably the build environment should be reliable, isolated and easy to replicate. One option for build environment is to have a physical machine with the same operating system and environment as is used in the daily development. Another option is to replace the physical machine with a virtual machine. The machine can also be set up with a configuration management tool or in case of a virtual machine, from the snapshot image to ease the replication. However, another option is to use containerization technology such as Docker or LXD to provide an isolated environment for building and testing the software.

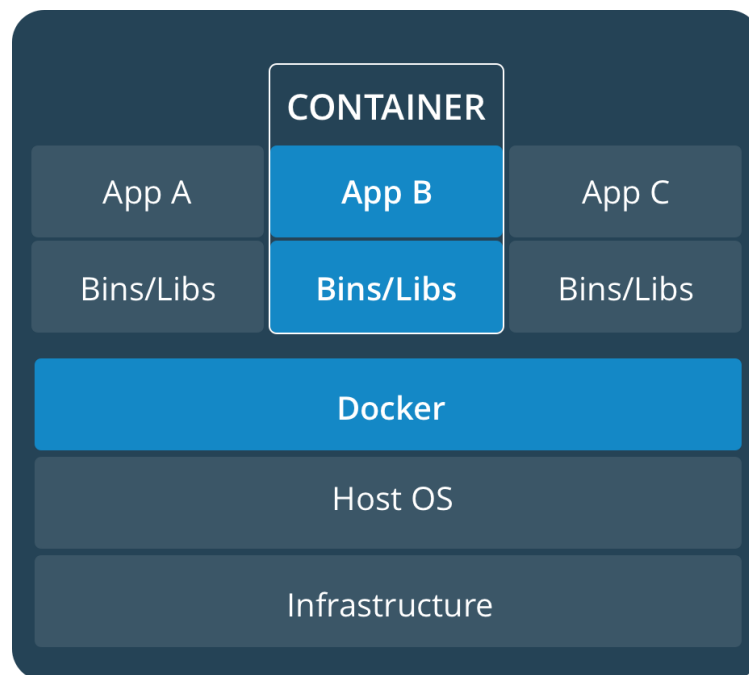
Docker is a software containerization platform. A Docker container is an environment created from a Docker image, which is a lightweight, stand-alone package, to provide the program and all the needed dependencies to run it. These containers will run similarly regardless of the platform on top of which Docker is running. Docker is available for Linux and Windows-based applications and it is based on open standards and is open source. (Docker Inc. 2018b; Docker Inc. 2018c.)

Containers also isolate the application from the surrounding environment avoiding the conflicts and improving the security (Docker Inc. 2018b; Docker Inc. 2018c.). The Docker container runs inside a separate namespace from which it cannot see the processes or filesystem outside the namespace. This isolation is provided on Linux using two pieces of the Linux kernel: namespaces and cgroups. (Anderson 2015.) On Windows the isolation is provided with Hyper-V or with process and namespace isolation technologies provided by the operating system (Brown et al. 2016).

On the other hand, there have also been studies if the Docker itself introduces security vulnerabilities. One example is that Docker daemon usually runs as a user who has full administrator rights on the host machine. As a result, if the access was gained from inside

the container to the host operating system, it could potentially compromise the entire system. This is something that should be considered when taking Docker into use in critical environments. (Merkel 2014.) Other points to the security issues are a possibility to turn off some of the Docker's security mechanisms and quite commonly used functionality to automatically update the environment from third-party registries. (Combe, Martin & Di Pietro 2016.)

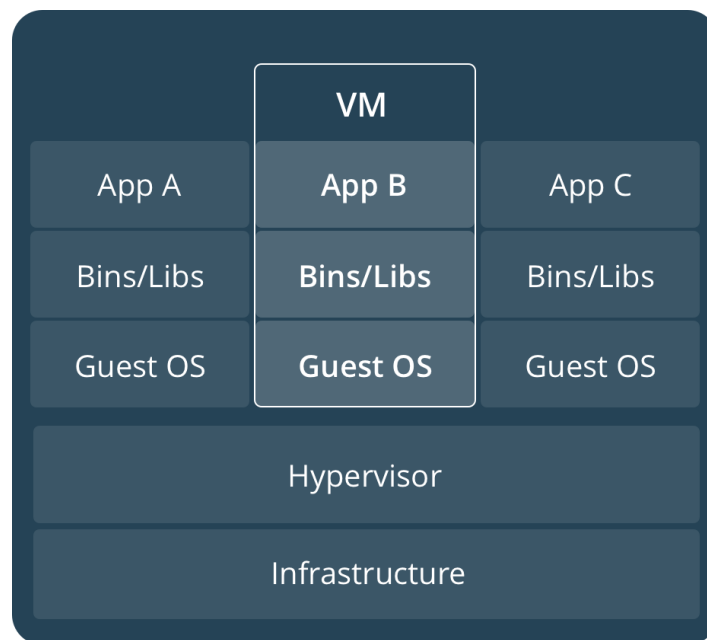
Docker containers typically use operating system kernel of the host machine. The filesystem of the container image is layered. This has the benefit that if changes are made in a single layer, only the layers above the modified layer need to be rebuilt and distributed saving both disk space and network bandwidth. Structure of Docker is shown in Figure 6. (Docker Inc. 2018b.)



**Figure 6.** Structure of the Docker. Docker is a layer on top of the application layer. The container includes an application with needed dependencies to run it. (Docker Inc. 2018b.)

Docker is sometimes used in place of a traditional virtual machine. However, it differs from the virtual machine in some noteworthy ways. Containers virtualize the operating

system instead of the hardware and thus the container does not need a hypervisor layer on top of the hardware. This should lead to less wasted computing resources which should lead to a better performance. This is backed up by the study from Felter et al. (2015) where Docker performance was found out to be the same or better than KVM-based virtualization although the difference was not huge. Containers are a layer on top of the application layer. Multiple containers can run on the same machine and they share the operating system kernel and resources. However, containers are isolated processes in the user space and do not have access to each other's internals. Due to these reasons, containers usually start almost instantly. (Docker Inc. 2018b.)



**Figure 7.** Structure of the virtual machine. Hypervisor works as an additional abstraction layer on top of the hardware. Each VM has its own operating system and applications. (Docker Inc. 2018b.)

Virtual machines, on the other hand, are at lower level. A virtual machine abstracts physical hardware into many machines. Each machine has its own copy of an operating system, including the kernel and applications. This means, for example, that starting up a virtual machine might take a long time as it needs to start up the whole operating system. Docker is often seen as a lightweight virtual machine because it does not need heavy hypervisor layer and full operating system installed on each image. However, Docker is

not technically a virtual machine and its architecture is rather different from usual virtualization. (Docker Inc. 2018b.) The difference in structure between Docker and the traditional virtual machine can be seen by comparing Figures 6 and 7.

Docker is typically used to deploy microservice-based applications to a cloud. This is useful because the container contains everything that is needed to run the software while it does not care about the underlying platform on which it is run. Another huge benefit with Docker is that it helps to manage the dependencies. Quite often applications have many components which all have their own set of dependencies. Sometimes these dependencies might even conflict with one another which leads to a situation known as “dependency hell”. With Docker, each component can be packaged along with its dependencies separately from the other components to avoid the issue. (Docker Inc. 2018c.; Merkel 2014.) Another area where this ability might be used is for packaging research environment along with the needed dependencies with the scientific work (Cito, Ferme & Gall 2016).

#### 4.2 Docker for Continuous Integration

Another use case for Docker is to use it for providing the build environments for the continuous integration server. In this situation, a separate Docker image is created specifically for building the application. This image contains all the tools necessary for building the application. At the beginning of the build, a new container is created from this special purpose image in a continuous integration build agent and the source code of the application is made available inside the created container. Then further steps of the build are executed inside this newly created container. After the build has finished, build artifacts can be fetched from inside the container and stored for use outside it. (Ledenev 2016.)

Using Docker like this for setting up the build environment has various benefits. One huge benefit is that it makes it is easy to switch between different environments. For example, if the application was previously built with the older toolchain, it is just a matter of building a new image with the new toolchain to test building with it. If the new

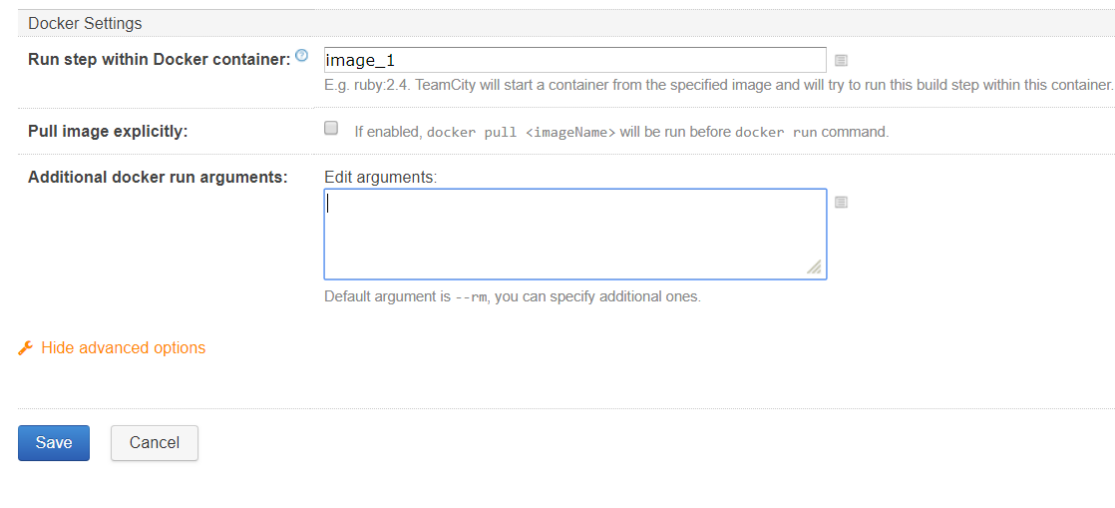
toolchain is not suitable, switching back is just a matter of falling back to the previous image. Another advantage is that build environment is easy to share with other developers since everything they need to build the software is bundled inside the image. This also supports build environment replicability. (Rancher Labs 2016.) Also, since one machine can host several Docker images at the same time as mentioned in the previous chapter, one physical machine can easily build many different programs without the risk of build environments conflicting with each other.

Since everything during the build is happening inside a Docker container, after the build has finished, it is easy to roll-back everything that was done (Rancher Labs 2016). This allows doing complicated environment setup during the build and cleaning the system once the build has finished which helps in build isolation and makes it possible to make heavier modifications to the build environment without possibly breaking other parts of the system.

Several CI server tools support using Docker for build environment setup. TeamCity has had this support since the end of 2017 (JetBrains s.r.o. 2017). TeamCity has extensive support for Docker. It allows creating Docker images in the build steps, uploading the created images to Docker registry and executing arbitrary build steps inside a Docker container created from the specified image, which can be fetched from the registry or stored locally on the machine. When build steps are executed inside the container, check-out directories and most of the environment variables are automatically passed inside it. As of Spring 2018 Docker support of the TeamCity has a limit that on Windows machines Docker works only in a “Windows container mode”. This means that Windows build machines cannot host Linux based build environments. (JetBrains s.r.o. 2018b.)

Build step execution inside a container works on per build step basis in TeamCity. A new container is created at the beginning of a build step, and it is automatically destroyed at the end of the build step. This means that the whole build does not share the same container unless the build has only a single build step. TeamCity also automatically makes sure that file permissions and ownerships are restored at the end of each build step to the state those were before the build step began. It is also possible to pass additional

parameters to Docker's run command that is executed by TeamCity to for example restrict resource usage or to mount additional locations inside the container. (JetBrains s.r.o. 2018b.) The configuration needed for using Docker is easy to configure in a build step configuration page provided by TeamCity as demonstrated in Figure 8.



The screenshot shows the 'Docker Settings' configuration page. It includes a text input for 'Run step within Docker container:' with the value 'image\_1' and a help text: 'E.g. ruby:2.4. TeamCity will start a container from the specified image and will try to run this build step within this container.' Below this is a checkbox for 'Pull image explicitly:' with the text 'If enabled, docker pull <imageName> will be run before docker run command.' The 'Additional docker run arguments:' section has a text area for 'Edit arguments:' and a note: 'Default argument is --rm, you can specify additional ones.' At the bottom, there is a 'Hide advanced options' link and 'Save' and 'Cancel' buttons.

**Figure 8.** Settings by TeamCity for running a build step inside a Docker container.

With the strong support for Docker in TeamCity introduced at the end of 2017, it is simple to use Docker for build environment provision and isolation as introduced earlier.



## 5 PLANNING

### 5.1 Current situation

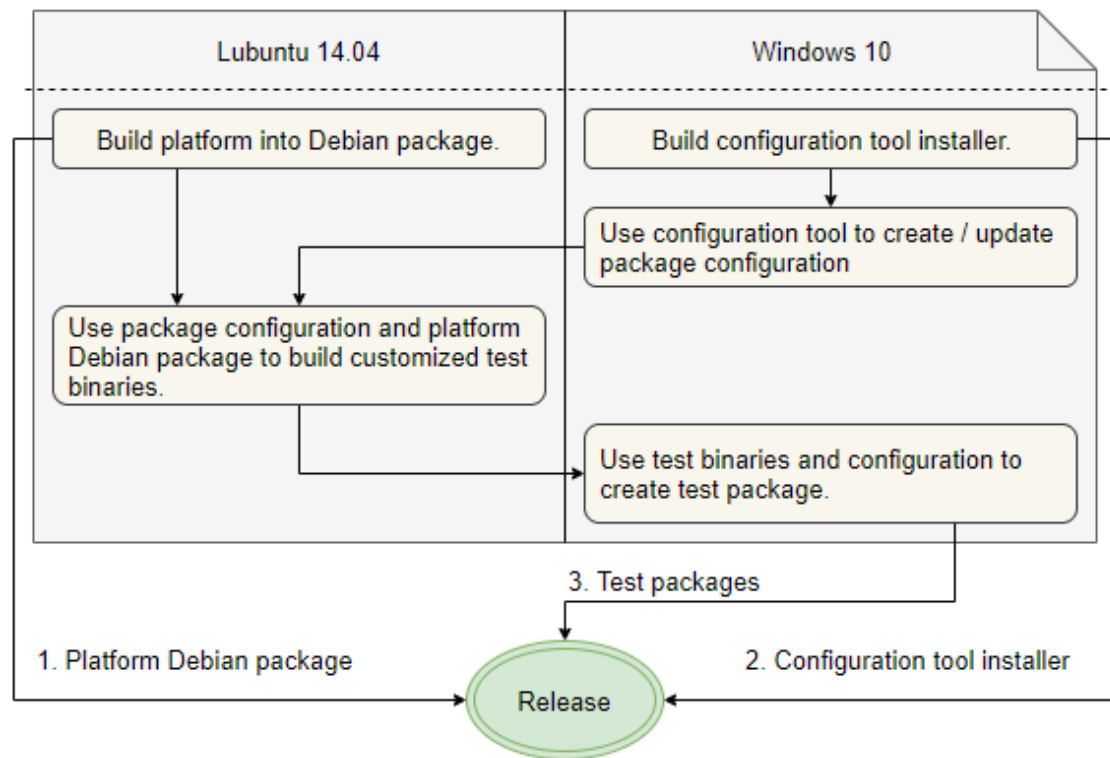
In the case project, major releases are currently done a few times per year due to the vast amount of manual work related to each release. The outputs of a release include three parts from the software perspective. On top of that, there are documents such as release notes and a test report.

First, the major release contains an embedded software platform which is packaged inside a Debian Linux package. This package can be considered the main release artifact from the embedded side since this is the package that the customer uses for developing their customized binaries. This package is basically a platform on top of which different applications for a specific purpose are built with. The platform provides the core functionality of a system such as communication methods between the embedded devices. Development and building of the platform occur mainly inside a specialized Linux environment which often runs inside a virtual machine. In the study, this part of the software will be referred as the “platform.”

Secondly, the release contains a desktop configuration and monitoring tool for the embedded devices. This tool currently works only on the Windows operating system. This tool can be used for example to configure the embedded system such as which devices are part of the system and how are the devices communicating with each other. The tool can also be used to define, for example, which version of the developed communication protocol is used by the system. Additionally, the tool allows monitoring and diagnosing the configured system. This tool is not given particular attention in this study since there is a parallel project for reducing the release time for it. When needed, this tool will be referred as the “configuration tool”.

Thirdly, the release contains four test system packages. These packages are pre-configured systems with different capabilities and devices enabled. For example, there is one

test system which is configured to include at least one of each device type. The test package contains a configuration created by the configuration tool and binaries for the embedded devices which can be downloaded to the devices using the configuration tool. The binaries are created inside the Linux development environment using the platform. Binaries are created based on the configuration created by the configuration tool since it decides which applications are enabled on which devices. The actual test package that is released is a specially structured zip archive created by the configuration tool. The package created by the configuration tool is later referred as a “test package” while the binaries created by the platform are referred as “test binaries”. Figure 9 represents the required release outputs with numbers 1 to 3.

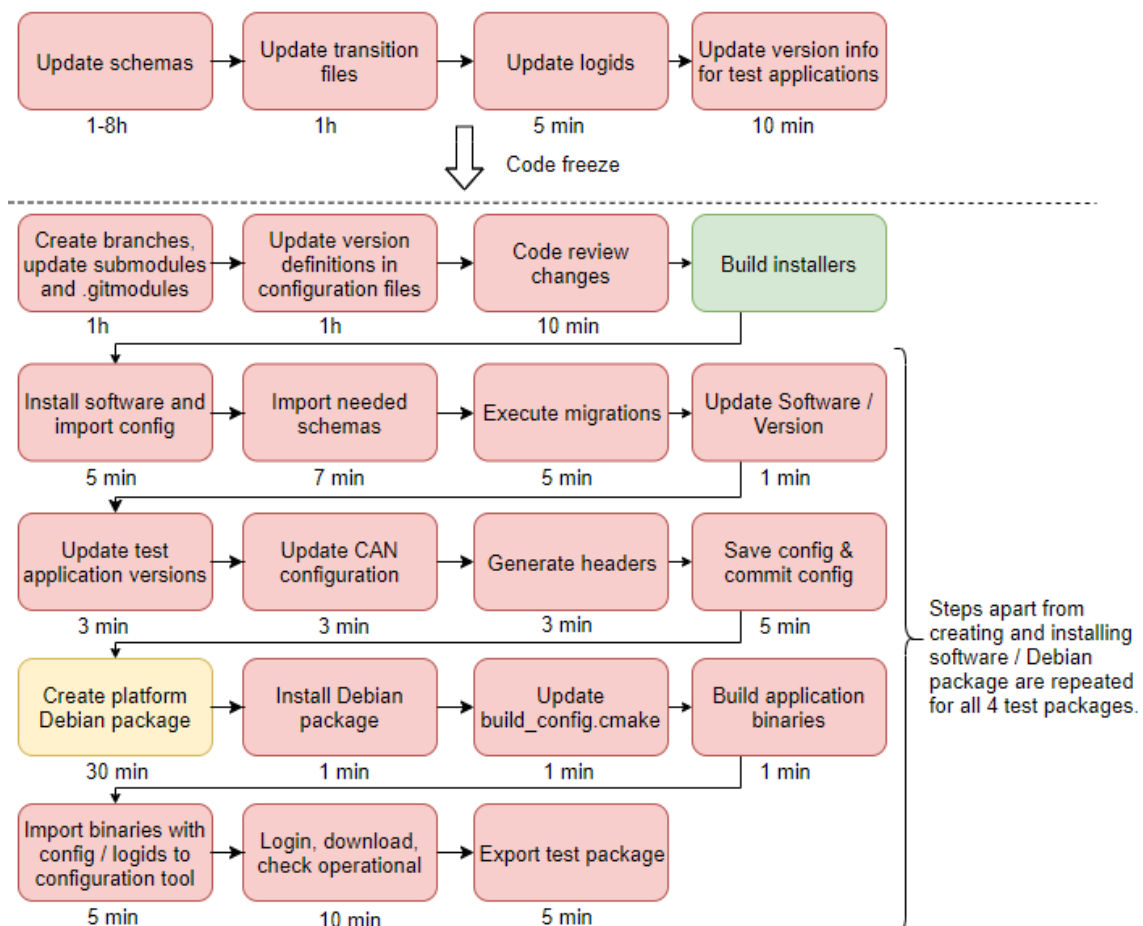


**Figure 9.** Figure representing different release outputs and situation where those are produced.

In case of an embedded software project, it is at times a bit hard to define what is the meaning of the continuous delivery. In this study, the definition of continuous delivery is that all the three release outputs mentioned above are provided and tested by the

continuous integration system. This is, once all the outputs from the Wapice are ready for delivery to the customer who can start using the provided files right away. This is needed, because it is not possible for this project to decide how and when the product is deployed to the end users.

The preparation for each release begins already before the code freeze, which is the point when all the changes for the release need to be committed to the master branch. Before the code freeze, however, release team is already doing some activities. The steps executed during the release are represented in Figure 10.



**Figure 10.** Overview of the steps executed during the release. Each step has a time estimate for executing it.

The steps with red color are currently done manually. The step with green color is done automatically by the CI system. The orange step would already be provided by CI system

but is still currently done manually for the release purposes. The process is represented in a diagram with the time estimates from the release team for executing each manual step. It is good to note these estimates are from an experienced release engineer and time estimates would probably be much higher for someone inexperienced with the steps.

**Update schemas** is the first step. In this step it is made sure that the schema files do not have unnecessary versions defined in them since those might have been added during the development for the testing purposes. If the version is no more in use and if it has not been released before, it should be removed from the file.

The second step is **update transition files**. These files define how the configuration is updated when updating some part of the system from one version to another. When new version of the schema is created, a corresponding transition file should also be created. Thus, when schema is removed, transition files should be updated accordingly.

The third step is **update logids**. There, an updated version of the log message IDs file is fetched from the external service and committed into the repository. A small utility is used for fetching the latest version of the file.

The last step before the code freeze is **update version info for test applications**. In this step, the versions of the test applications built on top of the platform are changed to the release version by modifying a configuration file.

Once the code freeze has started, the first step is to **create branches, update submodules and .gitmodules**. The step begins by creating release branches to the version control system (VCS). The VCS system in use is Git, and more specifically, Gerrit is used for managing it. Gerrit is hosting Git-repositories with extra functionalities such as with the support for peer-reviews. Once the release branches are created, “.gitmodules” file needs to be updated in each repository. This file handles which versions of the submodules are fetched. The project has 4 repositories. One for the platform, one for the configuration tool, one for automatic test scripts and one shared repository which is used to share common files between the three other repositories. In addition to this, the configuration tool

repository includes the platform repository as a submodule in its repository. Thus, .gitmodules file needs to be updated in three repositories and also the links to the correct revision needs to be updated in the same repositories.

Usually, at the same time as the submodules are updated, the step **update version definitions in configuration files** is also done. In this step, versions are updated to various configuration files such as to the documentation files. There are multiple files that need to be updated, and most of them have a different schema for the version string they expect. For the platform, there are currently three files which need to be updated, and they all have a different format for the version. After the changes are made, changes are pushed to Gerrit for a **code review**.

Once the changes are reviewed and merged, **build installers** step is executed. In this step an installer of the configuration tool is automatically built by the CI system and published as a build artifact.

Afterwards, a member of the release team **installs the configuration tool and imports a test configuration** into it. At the same time, one needs to fetch and **import needed schemas** from the platform's repository and point configuration tool to those.

After that, the user logs into the system with the configuration tool. At this point, the configuration tool might ask user to **execute migrations**. In that case, the release engineer has to select which migrations should be executed.

Next, the user should update versions in the system configuration. This includes **updating software / version** field with the release number and **updating test applications versions** to the latest ones available in the imported schemas.

Then one should use the configuration tool to trigger **update CAN-configuration** and to **generate header files** for the platform. These steps are simple to execute via the graphical user interface and are not explained in more detail. However, the generated header files should be moved to the platform repository.

Afterward, **configuration is saved and committed** to the VCS. This process from importing the configuration file to saving the updated version is repeated for all the 4 test system configurations. Along with the configuration, generated header files are also committed to the repository.

When the configurations are updated and committed to the repository for all the test systems, the build server **creates the platform Debian package**. This could then have been fetched by the release engineer and installed on one's local machine. However, it turned out that release engineer was building the Debian package on one's own local machine as well. This was mainly happening because one needed to build test applications locally anyway since those were not created by the CI system. One problem preventing the delivery of those by the CI system was that the created Debian package needed to be installed on the build machine before test binaries could be built against it. However, installing the development version of the Debian package poses a risk of breaking the build machine since the package might have post-install steps which might execute arbitrary commands. Thus, in a faulty situation, this could potentially break the whole test environment.

Typically, developers compile the test applications directly against the platform source code. However, during the release it is vital to verify that building the applications works also against the content of Debian package which is what is released to the customer. In order to do that, the **Debian package is installed** and file called **build\_config.cmake is updated** to point to the installed version of the platform before building the test binaries. Previously, build\_config.cmake file was updated four times, once for the test application binaries of each four systems. However, during the new process planning it turned out that this was unnecessary and updating the file once was enough.

Once build\_config.cmake is updated, **test application binaries are built**. Since the binaries are not created against the Debian package continuously during the development, this step often causes problems in a way that compilation fails.

Once the binaries are successfully created, they are moved back to Windows machine where they are **imported to the configuration tool with the configuration file and logids file** which were updated before.

Next, the release engineer logs into the system and **downloads the binaries and configuration to the actual embedded devices** using the configuration tool. Then it is verified using the configuration tool that the system is functioning correctly by **checking it goes to the operational mode**. Finally, the **system is exported as a test package** using the configuration tool. This is naturally also repeated for all the four test systems.

Now all the outputs needed for the release are ready. The Debian package has been built, and so has the installer of the configuration tool. Also, the test packages have been created using the Debian package as the source, and simple validation for it has been done. At this point, further manual validation is done, and if it passes, the created content will be published to the customer. If problems arise, then platform or configuration is changed, and the needed steps are repeated to provide new candidates for the release.

In order to better evaluate the results of the planned improvements, measures are in order about the current situation. Three measures were decided: total time of the process after the code freeze, the time between the beginning of the first manual step and the end of the last manual step after the code freeze and total active working time on the manual steps after the code freeze.

Table 1 contains the time estimates for different release steps. The estimates for the table are based on estimates from the members of the release team and verified by the author of the research by executing the same steps. Additionally, as mentioned before, due to application binaries not being build using the Debian package during the development, “build application binaries” step often takes much longer, up to at least 30 minutes extra time. This is included in the table with name “Fix problems in building binaries”. Based on the table, the whole process after the code freeze took about 7 hours and 40 minutes. This is also the time from the beginning of the first manual step to the end of the last manual step. Out of this time, active manual working was about 4 hours and 30 minutes.

**Table 1.** Table listing tasks related to release process when the practical part of the study began along with the total time and manual work time related to each step.

<b>Step description</b>	<b>Time</b>	<b>Manual work</b>
Create branches / update files	2 h	2 h
Code review changes	10 min	10 min
Build installers	50 min	0 min
Install software / import config	20 min	5 min
Import schemas	28 min	28 min
Execute migrations	20 min	1 min
Update software / version	4 min	4 min
Update sys. param and application versions	12 min	12 min
Update CAN configuration	12 min	5 min
Generate headers	12 min	5 min
Save config and commit it	20 min	10 min
Create platform Debian package	30 min	1 min
Install Debian package	1 min	1 min
Update build_config.cmake	4 min	4 min
Build application binaries	4 min	4 min
Fix problems in building binaries	30 min	30 min
Import system into the configuration tool	20 min	15 min
Login, download, check operational	40 min	10 min
Export test package	20 min	5 min
<b>TOTAL TIME</b>	<b>457 min</b>	<b>270 min</b>

The time before the code freeze is not taken into account for two reasons. Firstly, the long-lasting steps there require manual consideration with the other developers and are thus hard to automate. Secondly, and more importantly, the steps after the code freeze are



the steps which decide the actual deployment time. The other steps can be kept up-to-date during the project if needed, but the steps after the code freeze are significant in how long it takes to actually deliver the software once the decision for a release has arrived.

## 5.2 Reducing work

The goal of this study is to reduce the release cost and to decrease the time needed from code freeze to final release by removing the manual work required for each release. The process was modeled in Figure 10 with the time estimates from release engineers. In addition to asking time estimates, the release engineers were interviewed for recommendations about which steps they find the most troublesome to execute during the release. As a result, three focus areas were selected for this study.

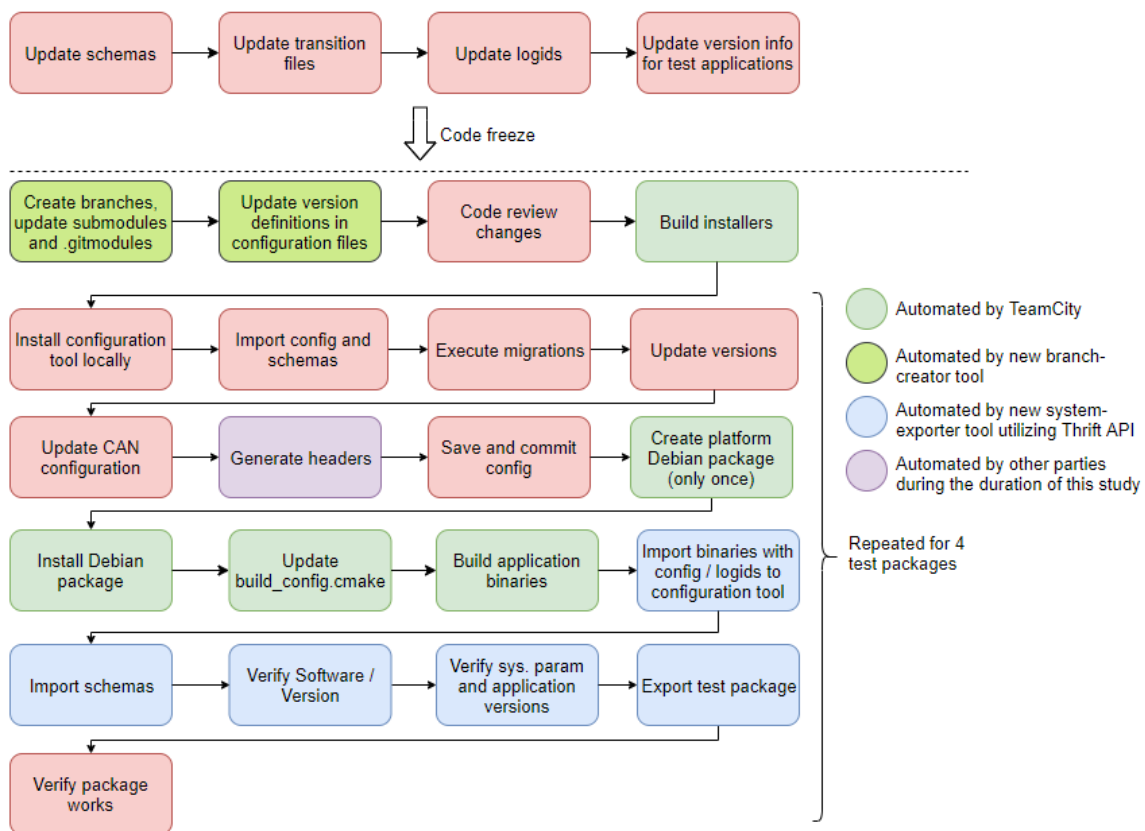
The first problem area is the branching and updating all the submodules and version files. This step requires working with four different Git-repositories and updating files on three of them. In addition, interaction with Gerrit web application is needed for creating the branches. Almost every file that needs to be updated also has a unique format for the version string. The step requires quite a bit of active manual working and is rather error-prone due to the need to jump around between different repositories and to update each file with the correct format.

The second problematic part is dealing with the test system configuration updates. This involves installing the configuration tool on the user's own machine, importing the configuration file and the needed schemas and doing the needed updates to the configuration. The process is tedious and error-prone especially since it needs to be repeated four times for different test packages.

The third focus area is building the test packages. This involves taking the upgraded configuration, moving it to the Linux-based development environment, installing the platform Debian package there, updating the build configuration file to point to the installed platform and to select which test binaries to build, building the binaries and moving them

back to Windows-environment, importing them along with the configuration file to the configuration tool and testing that everything works. After that the test package is exported using the user interface provided by the configuration tool. Additionally, in the old process, updating the build configuration file and building the binaries had to be done separately from each other for each test system.

Plans were made to improve the situation on all the three problem areas. The overview of the pipeline is represented in Figure 11 and presented in more detail in next paragraphs.



**Figure 11.** Overview of the planned new release process.

### 5.2.1 Automate branch creation and version file updates

At first, the focus is put on the branch creation and file updates. The proposed solution to this problem is creating a utility which handles the branch creation and file updates. The tool can interact with the Gerrit server using the REST-API that Gerrit provides (Gerrit 2018). The tool would need to have two main functionalities:

- Interact with the Gerrit server to manage the branches.
- Fetch the file contents from Gerrit, update the content and push it back to Gerrit for a review.

The release engineer could then merely execute this utility with the needed parameters such as the release name, Gerrit server address and project names and the utility would handle the branching and the file updates.

### 5.2.2 Updating configurations and building test packages

The second problem is updating the system configuration files. The first plan was to use the Apache Thrift API provided by the configuration management tool to automate the actions required. Apache Thrift is a software framework for scalable cross-language service development (Apache Software Foundation 2018a). Using the Thrift API, it is possible to use the most of the functionalities provided by the configuration tool programmatically. The bindings to the API are already generated for the Java programming language since those are also used by the automatic test scripts. The problem with this approach is that not everything in the process can be straightforwardly automatized. There is, for example, “execute migrations” step which would benefit from the user consideration about which of the migrations should be executed at which time. Thus, the fully automatic solution was ditched.

The solution idea for the third issue is first to isolate the build environment so that the Debian package can be installed inside it without risk. Then, updating `build_config.cmake` there will be automated, and test binaries will be compiled in the CI system against the Debian package. Then when this job finishes, it will trigger another job in CI system, which imports the created binaries, copies them to the correct places and logs into the system using the previously mentioned Thrift API.

After the system has been opened with the Thrift API, versions of the test applications and general software version defined in the configuration will be validated. In case some

version does not match the expected version, the build will report the situation as a TeamCity build problem. This will help the release engineer with the problem 2 for which completely automatic solution was abandoned. TeamCity can be configured so that build problem does not stop the build which shall be done so that package is exported even if some validation fails. The validation needs to get expected versions somewhere, and it can get them from the Debian package job. General software version will be parsed from the Debian package version, and the expected application versions will be formed using the schemas, which the Debian platform job will export as build artifacts.

Once the validation is done, this new software exports the actual test packages using the same Thrift API. This will result in one archive file per test system which is then published on the CI system. Implementing the tool this way will solve the problem 3, and problem 2 will be semi-automatized. The release engineer needs to update manually only the systems for which build problems are reported.

In order for the solution to the problems 2 and 3 be feasible, it must be possible to install the Debian package to the CI agent machine in order to use it for building the test binaries. For this TeamCity's Docker support will be utilized. The idea is to update the Docker image at the beginning of a build with the needed dependencies. Then Debian package will be created inside a clean container and installed there. After this, `build_config.cmake` will also be updated inside the container, and test binaries will be created there. Binaries will be published as build artifacts, and the created container will be removed from the system.

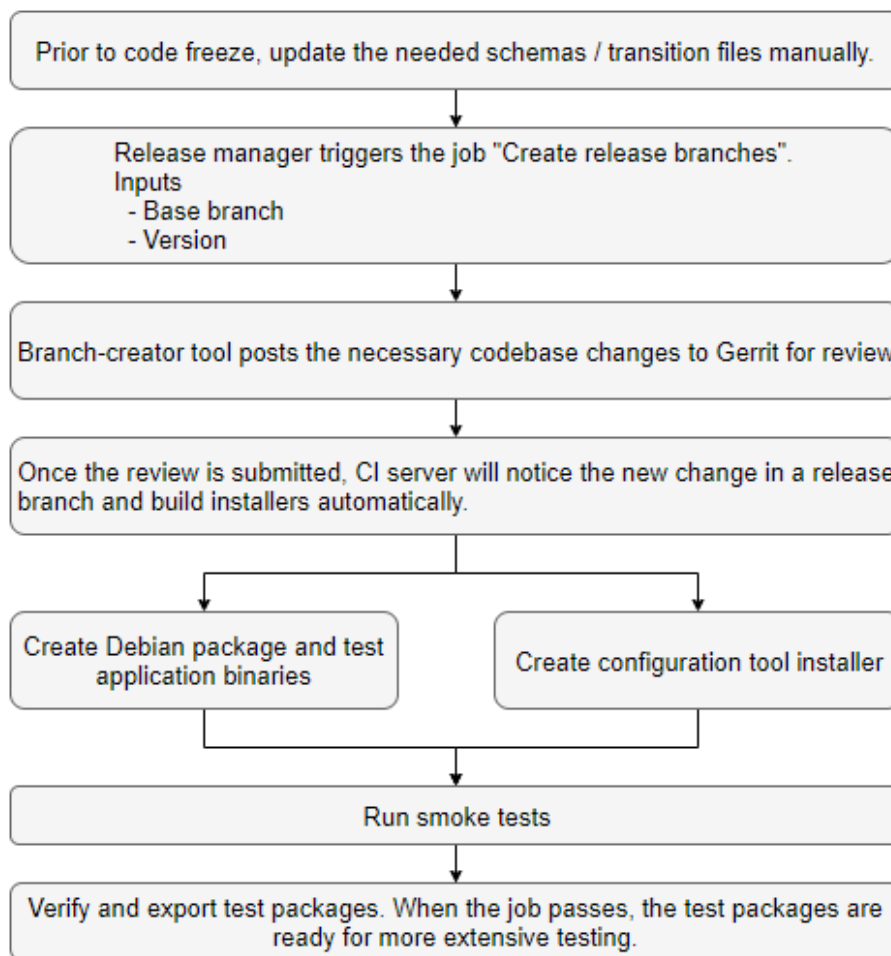
### 5.3 Controlling the pipeline

In order to control the complete build pipeline, multiple options were considered. First, release management tools like BuildMaster and XL Release were considered. However, the work required from setting up these was considered to be too massive at this point since the study already had multiple areas to implement such as Docker setup and the

schedule was already getting tight. XL Release is also rather expensive to use, which means it would need to have clear benefits to justify the cost.

The second option was to use Jenkins along with its pipeline plugin since it has support for asking the user input in the middle of the pipeline. Nevertheless, the idea was ditched when it became clear that the final pipeline would not benefit significantly from the support for manual steps which would have been the main selling point for Jenkins.

Instead, it was finally decided that the tools created as part of this study would at this point be controlled from the same TeamCity server as where the builds happen. The overview of planned TeamCity pipeline setup is described in Figure 12.



**Figure 12.** Overview of the planned TeamCity pipeline controlling the release process.

When the release begins, release manager triggers job to create release branches and update version in configuration files. Once those changes have been reviewed and merged to the repository, the rest of the pipeline is executed automatically. The needed user inputs for the “create release branches” will be asked before the build begins and those are stored as build parameters to control the build. The advantages of this solution are that it is both familiar to both teams involved in the process and TeamCity instance is already up and running so the cost of setting the system up will be minimal. This is beneficial also if the switch to a release management tool will happen at the later stage.

## 6 IMPLEMENTATION

### 6.1 Creating the branch creator for automatic branching and file updates

Improving the process is started by creating a tool for automating the branch creation and updating the necessary files. The process is started by collecting the exact requirements for the application. Identified main requirements are:

- Create release branches according to a user specified release name and base branches (base branch is the branch from which the new branch is created from).
- Base branch can be different for each project for which branches are created.
- It should support release candidate releases, final releases and maintenance releases.
- Update content of the files that require updating during the release. Currently, there are 5 distinct types of files that require updating in different projects. The file does not need to exist in all of the projects, but it might.

There are some additional features which could be considered such as also creating stable branches in addition to release branches. The stable branch is a branch which is created to continue development on a specific major release after the current release is done (usually bug fixes to the release). In order to fulfill the requirements, some hidden requirements also exist. These include a way to specify the correct Gerrit server and a possibility to interact with Gerrit's REST-API which requires user authentication. The created tool should also have an effortless way to include new file update abilities to the process since the files that need updating are changing from time to time. Also, the dependency to Gerrit should preferably be isolated from the rest of the system so that program can be modified for working with other tools apart from Gerrit if needed in the future.

The tool was decided to be implemented with Python programming language. There were multiple reasons for the choice. First, Python is familiar to every developer on the project's CI team. Secondly, its dynamic nature suits well the scripting use-case for the CI job. There also exists a third-party library `pygerrit2` for interacting with the Gerrit's REST-API and the library is MIT-licensed (Pursehouse 2018). Java was also considered since it would share most of the same advantages, but the need for compiling was considered troublesome for scripting inside the CI job.

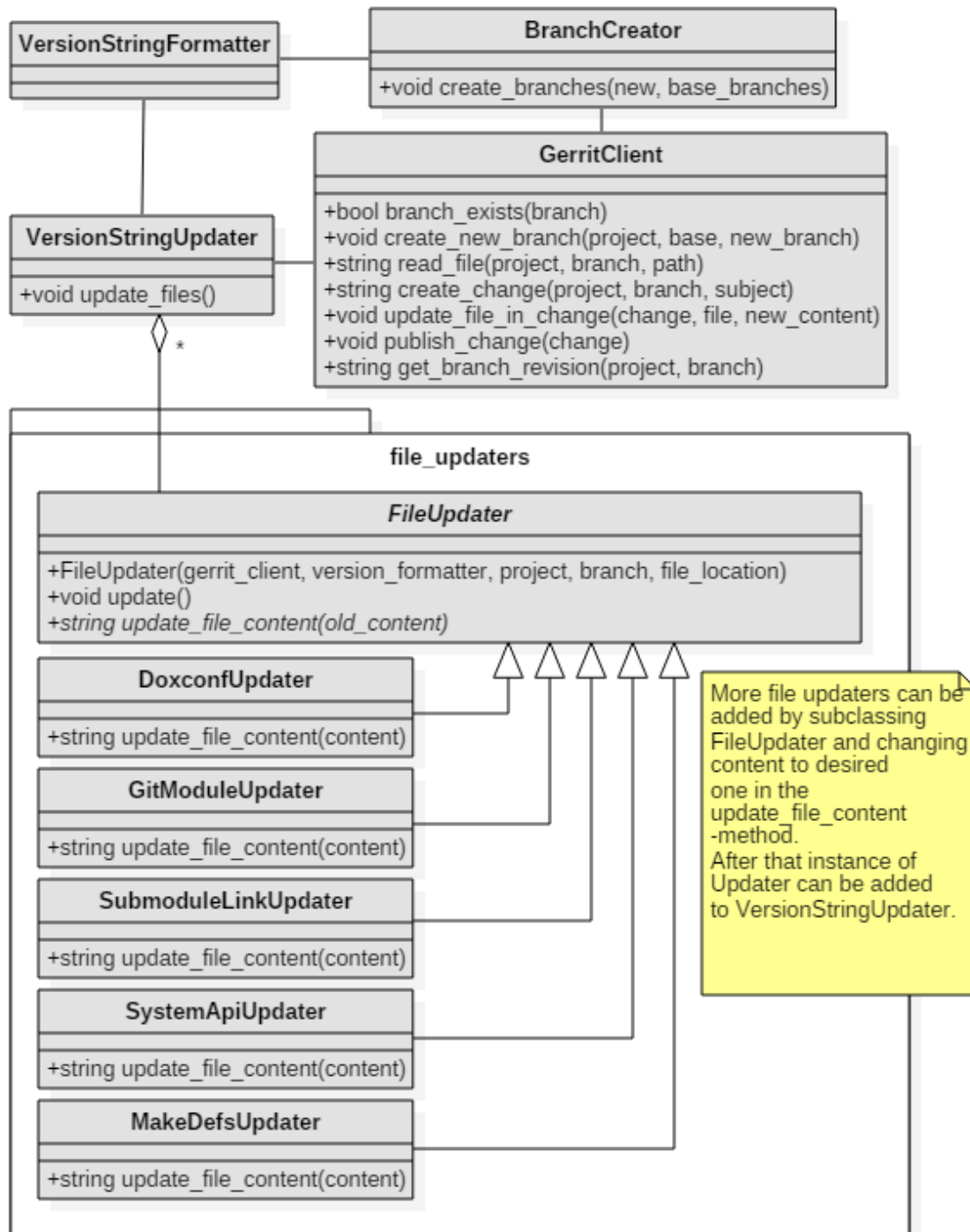
The development of the tool was started with planning the needed classes. Gerrit related functionality is isolated into the class `GerritClient` to fulfill the requirement that tool must be easy to modify to work with other VCS services apart from Gerrit. Also adding new file updaters is a matter of creating a new class and implementing the method called `update_file_content` there which gets the old file content as a string input and the method should return the updated file content. Then an instance of this new class should be added to the list of file updaters in `VersionStringUpdater` class along with the project and path where the file is updated. This is all that is needed for adding a new file to be updated to the tool.

Apart from the class diagram, the needed inputs for the program were designed. The program will take as inputs: address to Gerrit server, credentials to Gerrit (username and HTTP auth token), release name for which the branches are created in the form of MAJOR.MINOR[.MAINTENANCE] [RC X], e.g. 1.0.1 RC 1, the name of the default base branch along with the possible overwrites on per project basis and boolean toggle switches to decide if the files should be updated or if the stable branches should also be created. An example command invocation of the program could look like:

```
branchcreator --gerrit-url http://gerrit-address.com --
gerrit-username user --gerrit-password password --re-
lease 1.0.1 --base-branch master --override-projectA-
base-branch 1.0.0 --update-files --no-stable-branches
```

However, not all the parameters are required. The only necessary parameters are username, password, and release. Others have default values defined. The class diagram of the tool is represented in Figure 13.





**Figure 13.** Class diagram for the design of the branching utility.

After the design, implementation of the program was done. Details of this are not represented here since those do not add much to the study. After the implementation, the program was tested on test instance of Gerrit which is cloned from the production instance. Additionally, unit tests were written using ‘pytest’ testing framework. It is a framework that makes it easy to write small tests for the application using Python’s built-in assert statement (Krekel 2017). Unit tests included tests for all the version string formats provided from `VersionStringFormatter` class with different release types (release candidate, full, maintenance). Additionally, at least one test was made for each file updater in a way that valid file was given to it as an input, validator was executed, and the output content was validated to be what is expected. The application was also tested to be working with both Python 2 and Python 3 to make sure it works with both current and future machines.

Documentation for the application was also created at this stage. Deployment and dependency management of the application was done using program called ‘pipenv’, which is a tool to manage dependencies for the Python applications (Pipenv 2018). After the application worked, a new build configuration was created to TeamCity to allow smooth running of the tool. This is documented in greater detail in chapter 6.3.

## 6.2 Creating the system exporter for providing test packages from CI

The second task to do was to provide test packages from the continuous integration system. This task has basically two separate subtasks. The first task is to improve the build process of the platform such that the Debian package can be safely installed to the build agent. This is needed to closely resemble customer use case. The customer builds binaries against the Debian package and therefore the test binaries for release should also be build against it. This step requires constructing a way to isolate the build environment such that it is safe to install the created Debian package to the system without the risk of it breaking the whole environment. Basically, what is done during the build should be reversible after the build has finished.

The second part of this task is about taking the binaries created by the previous job, moving those to the correct locations on the file system so that the configuration tool can access them and then use the Thrift API of the configuration tool to import the needed schemas, validate the versions in the test system configuration and finally export the system as a test package archive. Then those archives are provided as build artifacts on the CI server for developers and testers to use. Additionally, the tool should report any issues it finds using the service messages of TeamCity, which allows reporting build problems. This basically requires just printing the error in special format. The exact format is specified in the documentation of TeamCity.

#### 6.2.1 Isolating the build environment

The first step is to handle the issue of isolating the build environment. During the literature review, it was found out that Docker can be used to fulfill this requirement. With Docker, it is possible to create a new container for each build step and clean the changes made at the end of the build step so that the next builds are not affected. However, a suitable Docker image is needed to utilize this option as well as a machine capable of running the Docker images.

A new virtual machine was created for hosting the Docker. The operating system used was CentOS 7 64-bit version, and it was installed to the machine already when the machine was handed out for the use in this project. As part of this study, Docker was installed and configured to the machine using the OverlayFS2 storage driver. Installation and configuration were done according to official documentation of Docker which is available in <https://docs.docker.com>.

The platform development is currently done inside a virtual machine which has 64-bit Ubuntu 14.04 installed in it. The environment fetches extra Debian packages such as development libraries from the custom Debian repository that resides inside the customer's network. Due to historical reasons the environment also uses Wine, which is a compatibility layer to run Windows applications for example in Linux (Wine 2018). This

is needed because some of the necessary tools required to run platform's unit tests are available only as Windows versions.

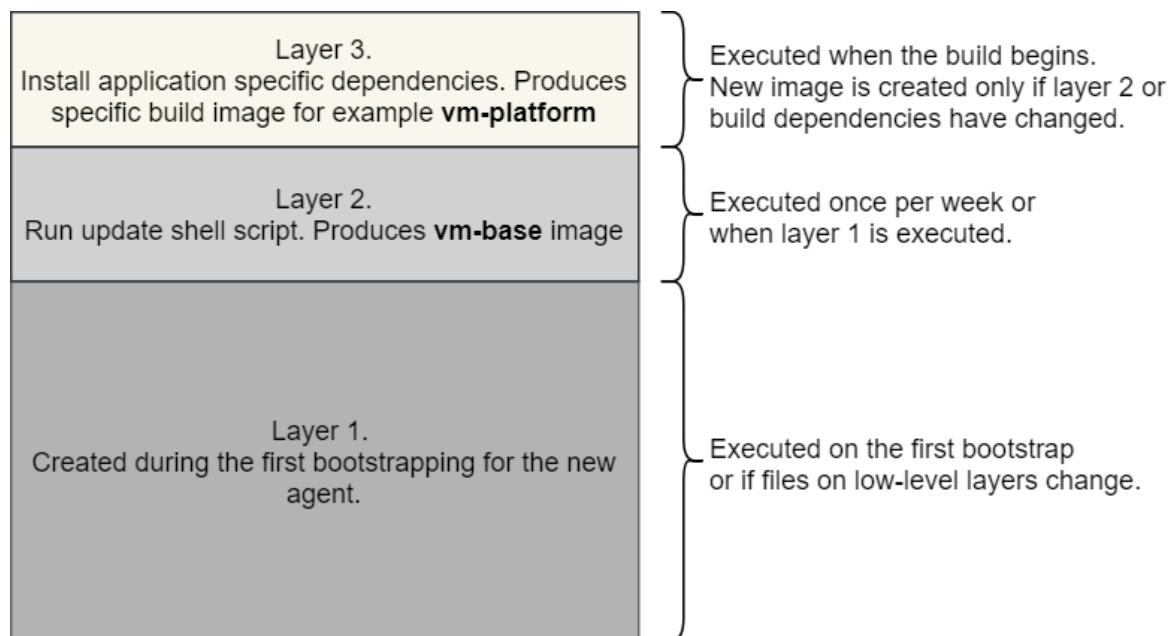
Other considerations when creating the image are that it should be performance optimized and keeping it up-to-date should be easy at this point, since it will be an additional way to build the platform and there will not be many resources dedicated to keeping it up-to-date at the beginning.

Based on these considerations, a base-image was built using the standard Docker procedure of using Dockerfile to build the image. The newly created image is based on 64-bit Ubuntu 14.04 image since this is the operating system used by the development virtual machines as well. The number of file-system layers was minimized in the Dockerfile by combining multiple commands into a single RUN-statement. This was done because optimizing the performance was one of the goals and because a vast number of layers might affect filesystem performance (Docker Inc. 2018a). To further support this, most of the operations for setting up the setup was split out to separate bootstrap shell-script which is invoked by a single RUN-statement. This has the advantage that layer count stays low. The disadvantage is that if there is even a minor change to the bootstrap-script, the whole big layer needs to be rebuilt. However, it is expected that there should not be too many changes into this layer.

To help to keep the image up-to-date, an additional mechanism was added to the Dockerfile. At the end of the Dockerfile, there is a step which is running the script that updates the environment. Before this step, an additional argument definition was added. With the help of this argument definition, CI system can pass a value for the argument which changes each time. When the value changes, the changed layer, and layers above it are invalidated and rebuilt. This way the script to keep the environment up-to-date can be forced to be executed each time. This update is triggered currently by a separate build job once a week.

The created image functions as a base image for more specific images. There are multiple applications which are built using this same Ubuntu 14.04 environment even though this

study focuses on only building the platform. The actual build job in the CI server is supposed to take this base image and add the needed per-application dependencies on top of it to form software specific final build environment. In case of the platform, this is done by using TeamCity's "Build Docker image" build runner to execute Dockerfile which takes the previously created base image as the base and runs a script which installs platform's dependencies on this environment, leading to a new image which is then used by the rest of the build steps. The overview of a Docker image creation process is represented in Figure 14.



**Figure 14.** The overview of the Docker image creation process. First, base-image is created, and later it is updated once a week. Then specific software build job takes the base image and builds the more specific image on top of the base image, which is then used in the rest of the build steps.

After the base image creation is complete, a new build job is created for the actual platform building. The build job needs to build software specific image, create a container from it, build the Debian package there, parse the version of the Debian package, install it, update the build\_config.cmake to point to the installed version and build the test binaries. Moreover, the build job should have the possibility to pass “--publishable” flag to the

Debian package creation process if the created Debian package is one that will be released to the customer repository.

In this build job, the first step creates the specific image using TeamCity's Build Docker image runner. It is configured to use base-image vm-base created in another job as a base for the new image. Name of the new specific image is set to be "vm-platform". Then on the rest of the steps "Run inside a Docker container" option is used and this "vm-platform" image is specified. On the second build step, the Debian package is created.

After it, version of the Debian package is parsed in step 3 using Linux utilities "sed" and "cut". The name of the Debian package is in form of "platform-a.b.c\_a.b.c\_amd64.deb" and the parsed value should be "a.b.c". The exact command used for parsing is:

```
version=$(echo platform-*_amd64.deb | sed 's/platform-  
//g' | cut -d'_' -f1)
```

Here, the echo is used to print the full name of the platform package. Then sed is used to remove the "platform-" part from the name. After this, cut is used to split the remaining part from "\_" character into half and take the first half leading to wanted "a.b.c". After this, the version is saved for the use by rest of the build steps by printing it using TeamCity service message format with the command:

```
echo    "##teamcity[setParameter      name='VERSION_STRING'  
value=' $version' ]"
```

Then, this Debian package is installed, and build\_config.cmake is updated to point to the newly installed Debian package using the previously parsed version. The file updating is also done using the sed-utility with the command:

```
sed -i 's/.*PLATFORM.*/set( PLATFORM "%VERSION_STRING%"  
) /g' build_config.cmake
```

Syntax "%VERSION\_STRING%" in the command is a way TeamCity provides for replacing parameter in command with the stored value. So, in this command TeamCity will

replace “%VERSION\_STRING%” with string “a.b.c”. After this, the test binaries are compiled against the installed platform which was installed.

The need to be able to pass `-publishable` flag to the build process is accomplished merely by using TeamCity’s checkbox type build parameter which is empty when not ticked and “`-publishable`” when ticked. This value is added to the end of build command in the same way as “%VERSION\_STRING%” was added to the sed-command in the previous paragraph.

Finally, the Debian package and test binaries are stored as build artifacts. This is done merely by using TeamCity’s functionality to select the files from the list of files which should be published. Additionally, schemas and test package configurations are also published as artifacts even though they are taken directly from the repository without any modifications during the build. This is done because those are later needed by the test package exporting tool which otherwise will not need access to platform repository. In the end, the created container is removed to roll-back the system to the point before the build with the exception that the specific image is now already updated for the next build.

### 6.2.2 Building test packages

At this point, these outputs are provided with the newly created job: Debian package of the platform, test binaries compiled using the platform from the Debian package, parsed version of the platform, schema files and test system configurations. The next step is to take these to the Windows machine, move them to the correct locations and validate and export the test packages using the Thrift API of the configuration tool.

Taking those to Windows machine and moving to correct locations is a simple part. TeamCity has built-in support for fetching files from other builds to the build machine. After this a trivial PowerShell-script was created which just moves the files to the correct location on the machine, removing the old ones before doing this and failing the build if one of the actions fail. This script in the simplest case only needs to use commands

“Remove-Item <path>”, “Copy-Item <source> -Destination <target>” and “mkdir <target>” and thus the script is not presented in greater detail.

Then, a new tool is created for validating and exporting the system. Additionally, this tool must handle importing the needed schema files. This is needed because not every test package should have every schema file in it and the information about the needed schema files are available only through the Thrift API. The tool is implemented using Kotlin-programming language. This language was chosen because of two reasons: firstly, the language that is chosen should be compatible with Java. Reason for this requirement is that files required for interacting with the Thrift API are already automatically generated for Java because those files are used by the automatic test scripts as well. Secondly, Kotlin was decided because our team wanted to have more experience with the Kotlin since it is gaining plenty of attraction. For example, TeamCity’s new domain-specific language for configuring the projects is based on Kotlin.

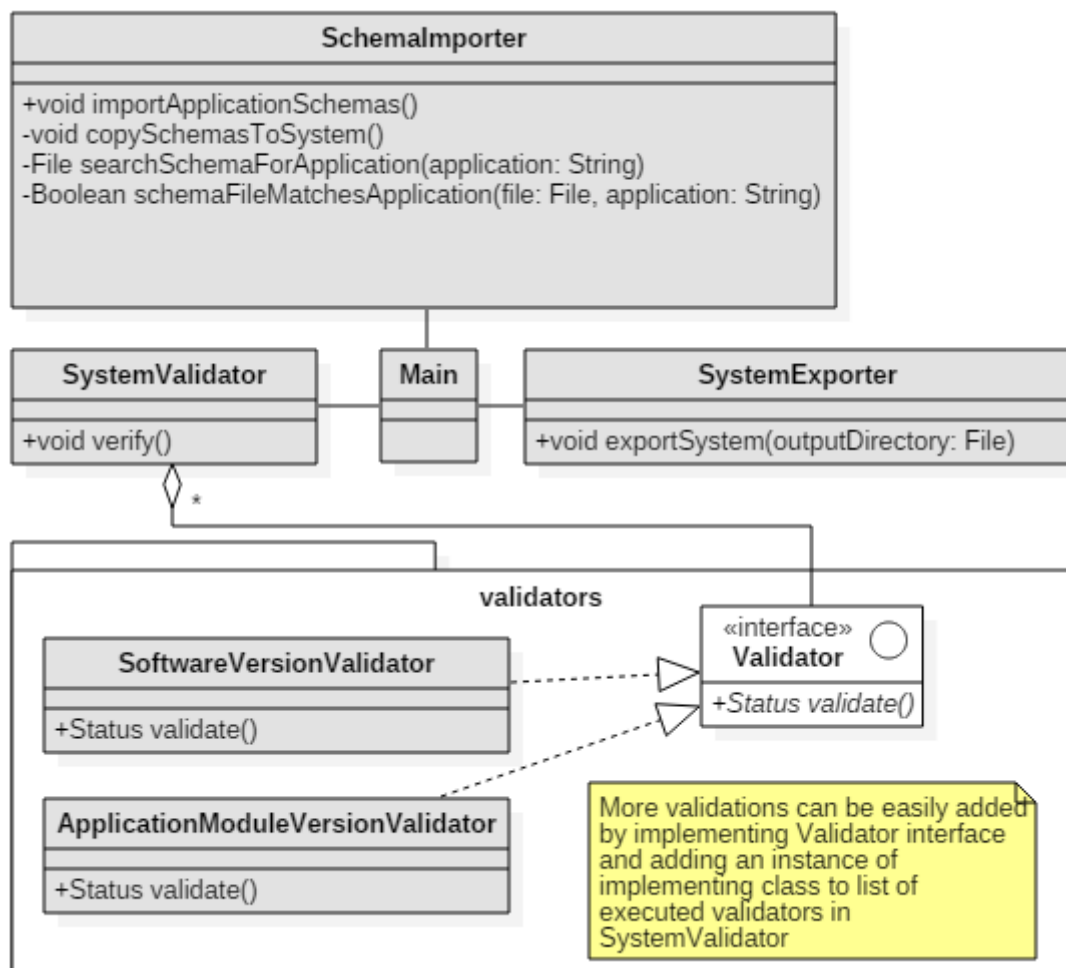
Similarly to branch creator utility, the design of this program started by collecting the requirements. The requirements found were:

- Use Thrift API of the configuration tool to export test packages
- Use Thrift API of the configuration tool to validate versions in the configuration of the test system.
- Use Thrift API to get information about the needed schema files and copy them from the user-defined path to the location where the Thrift API expects them to be when exporting the system.
- If an error happens, report it as a build problem to the TeamCity using the service message format. The details of the format are available at: <https://confluence.jetbrains.com/display/TCD10/Build+Script+Interaction+with+TeamCity>



- The program should take as an input a list of test systems to be validated / exported, the expected software version, and path to all schema files. It should output test packages and log files which tell what was done.
- Adding more validators to the software should be simple.

Based on these requirements, a class diagram was designed to fulfill the requirements. The diagram is visible in Figure 15.



**Figure 15.** The overview of the class structure of the system exporter tool.

The controlling of the application flow is done by the main function. It delegates the execution to the specific instances of the classes when it needs to, for example, to import schemas or to validate the system. Adding a new validator is a matter of creating a new class which implements interface `Validator` and adding an object of this new class to the list of executed validators in `SystemValidator` class.

The tool is developed as a Maven project. Maven is a tool for Java (and Kotlin) to handle for example the dependencies of the application (Apache Software Foundation 2018b). With Maven it is possible to depend on the generated Thrift API files for Java and re-use them for this project.

Similarly to branch creator project, the details of the implementation will not be covered as part of this paper. Testing the finished tool is done again in two steps. First unit tests were written for most of the methods. Secondly, the application was tested locally against multiple test system revisions, and the created test packages were compared to the expected output. After testing was finished, the tool was integrated into TeamCity as another build job as described in next chapter.

### 6.3 Managing the pipeline with TeamCity

Once both utilities were created and tested, it was time to integrate them into TeamCity. Running the branch creator can be considered mostly a separate step from the others. However, it is still the step that starts the whole pipeline. When the branch creator job is executed, the new changes are posted by the tool to the Gerrit for code review. Once these changes are approved and submitted, TeamCity will notice that new commits were pushed, and it will start builds for the desktop configuration tool and platform automatically. After this, the job to create test package is executed automatically twice per day or triggered manually by the release engineer.

In order to run branch creator tool, an interface was created using the tools provided by TeamCity. Basically, it was constructed by passing in the needed inputs to the release

creator as changeable configuration parameters, which are requested automatically when the job is triggered. The interface has fields for changing every significant parameter value. Some of the parameters were left to the default values such as Gerrit URL which is always the same for the project. The interface created for running branch creator is represented in Figure 16. In the figure, there are all the inputs that application needs. Release field has additionally also extra validation functionality, which prints out an error if the specified version string is not valid.

Run Custom Build [redacted] :: Release Automation :: Create release branches

General Changes Parameters \* Comment and Tags

The below parameters are marked as necessary for review

**Configuration parameters**

Base branch*	master	Common base branch, default one for all the projects unless explicitly overridden.	Reset
Also stable branch*	<input type="checkbox"/>	Create stable branch in addition to release one	Reset
Update files*	<input type="checkbox"/>	Also create changes in Gerrit which update needed files	Reset
Release name*		Release string. Format: <MAJOR>.<MINOR>[.<MAINTENANCE>][ RC <RC_NUMBER>]	Reset
Base branch ( )*		Specify if [redacted] base branch should differ from "Base branch"	Reset
Base branch (Platform)*		Specify if Platform base branch should differ from "Base branch"	Reset
Base branch			Reset

Run Build Cancel

**Figure 16.** The interface provided for the user to execute branch creator utility. Names of the projects are left out for confidentiality reasons since they are not significant for the study.

After the reviews have been submitted, another job starts which handles creation of the platform Debian package and test application binaries. This job usually starts automatically, but it has an extra parameter which can be defined by the user. It is the “`--publishable`” flag that was mentioned before for creating the Debian package without extra timestamp information in the package version. Usually, the Debian package has mangled name with a unique timestamp in order to make it easier to identify test versions from the official versions. With the `--publishable` flag this extra information is not added to the package name. Controlling this parameter happens similarly to controlling branch creator, using TeamCity’s build parameter which can be set before the build begins. Figure 17 shows this feature.

Run Custom Build :: [TESTING] Release step automation :: Create Debian package and Applica

General Changes Parameters Comment and Tags

Configuration parameter ▾

<Name> <Value> add

Configuration parameters

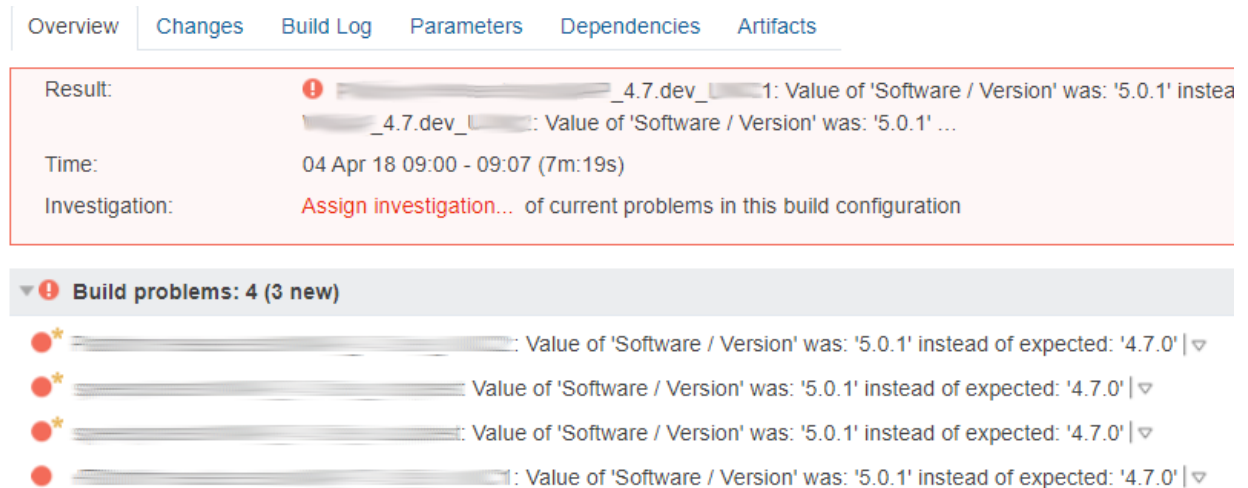
Build Platform in ☐ Publish mode Passes `--publish` flag to create-platform-pkg.py Reset

Run Build Cancel

**Figure 17.** Configuration options for the Debian package creation. The user can tick ‘Build Platform in Publish mode’ option to have Debian package without extra information included in the version.

Finally, once the Debian packaging job and desktop configuration tool installer jobs have finished, the final job will begin for exporting the test packages. This job does not require any user input. It will use outputs from both the mentioned jobs. The user can optionally select which build from each of the jobs is used for input files. After the build is executed, test packages will be available as build artifacts along with the log files. Additionally, if

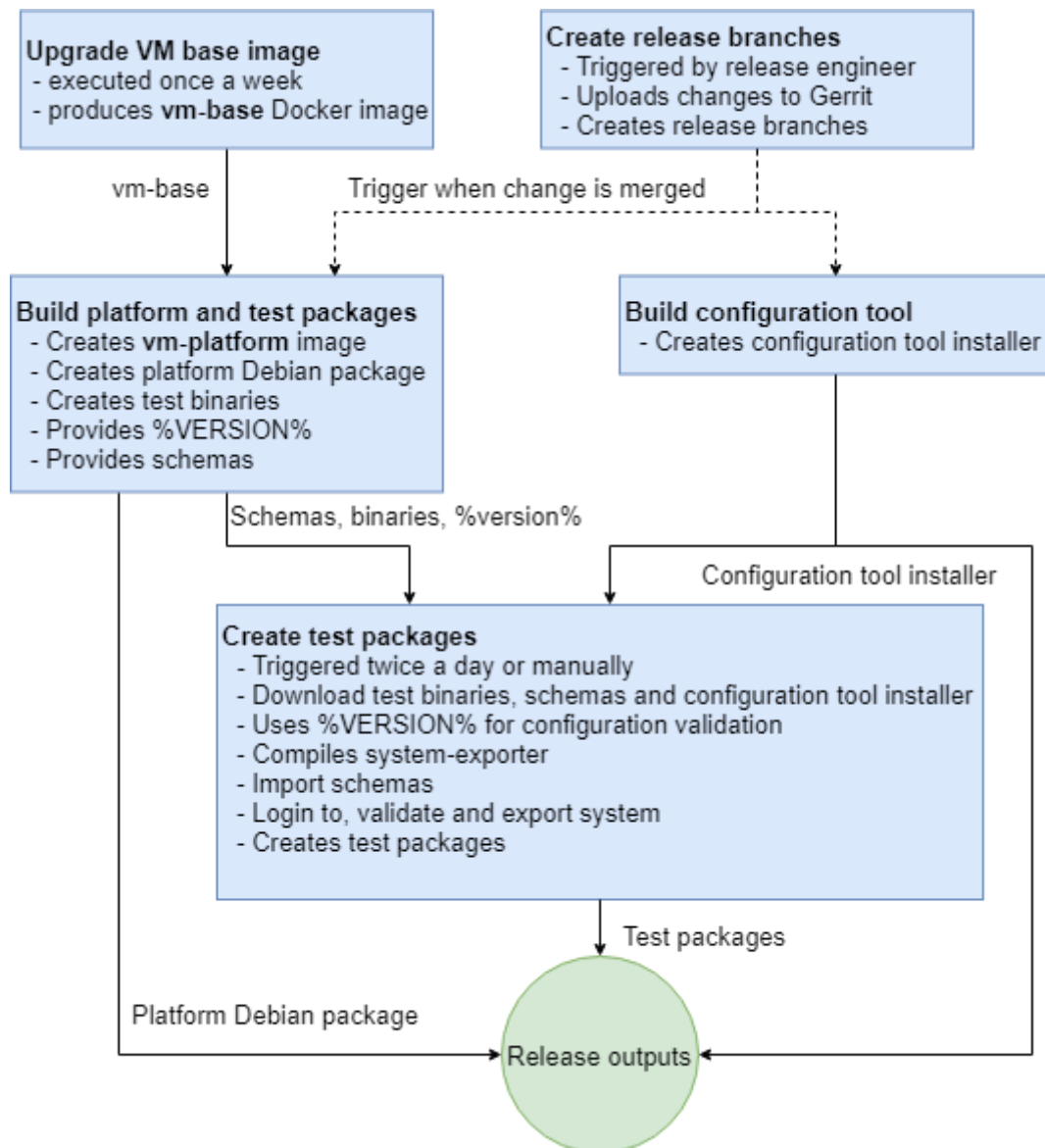
the validation step found any issues, those will be printed on the overview tab of the build as seen in Figure 18.



**Figure 18.** An example view from the final job. It lists problems found during the test package verifications. In the figure, there were problems found in 4 test packages. Names of the test packages are left out for confidentiality reasons since they are not significant for the study.

The problems reported on the overview page inform the release engineer in a simple way that there is still work that should be done before the packages can be delivered for further testing. If then further changes to the software or configuration need to be done, the developer makes the modifications, pushes them to the Gerrit, and the pipeline will be automatically executed again when TeamCity notices the new changes. Created test packages are available from the “Artifacts” tab which is visible in the Figure 18.

The overview of the complete pipeline is represented in Figure 19. It shows the process of getting all three different required release outputs and dependencies between different TeamCity build jobs. Each blue box represents a different TeamCity build job, while the green circle represents the release. Solid arrows between boxes represent artifacts that are moved between the jobs.



**Figure 19.** The overview of the complete pipeline. The blue boxes represent TeamCity build configurations. The green circle represents all the needed release outputs.

## 7 RESULTS

### 7.1 Results

After the changes were taken into use and tested, new estimates were done about the manual work required for a release after the code freeze with the release management team. New estimates are available in the Table 2 below. Prior to making the improvements, the whole process after the code freeze took about 7 hours and 40 minutes. This was also the time from the beginning of the first manual step to the end of the last manual step. Out of this time, active manual work was about 4 hours and 30 minutes.

After the improvements were made, the total time was cut down to about 4 hours. This is also the time between the start of the first manual step and the end of the last manual step. The most significant improvement happened in the amount of active manual work needed. Previously it was about four hours while after the improvements it was cut down at best to less than one hour.

Another consideration is that lots of recurring manual work with somewhat dull steps could be automatized. Overall this should lead to fewer mistakes and more reliable release process. It is also important to notice that steps executed are necessary first steps to achieving continuous delivery.

At the beginning of a study, it was considered that test system configuration updates would also be automatized. Some problems were faced while trying to do this such as manual intervention needed from the release engineer during the migration step. Because of this manual work could not be reduced more. Overall the results were good. The study achieved the goal of reducing release costs by removing the manual work required from the release team. At the same time, CI pipeline was further developed, and the risk for the mistakes during the release was lowered. With the improvements done it should also be more comfortable for new developers to learn steps necessary for a release.

**Table 2.** Table listing tasks related to release process along with the total time and manual work time related to each task before and after the improvements are done.

Step description	Total time		Manual work	
	Old	New	Old	New
Create branches / update files	2 h	<b>3 min</b>	2h	<b>1 min</b>
Code review changes	10 min	<b>10 min</b>	10 min	<b>10 min</b>
Build installers	50 min	<b>50 min</b>	0 min	<b>0 min</b>
Install software / import config	20 min	<b>20 min</b>	5 min	<b>5 min</b>
Import schemas (comes now from package)	28 min	<b>0 min</b>	28 min	<b>0 min</b>
Execute migrations	20 min	<b>20 min</b>	1 min	<b>1 min</b>
Update software / version	4 min	<b>4 min</b>	4 min	<b>4 min</b>
Update sys. param and application versions	12 min	<b>12 min</b>	12 min	<b>12 min</b>
Update CAN configuration (not needed)	12 min	<b>12 min</b>	5 min	<b>5 min</b>
Generate headers (automated outside the study)	12 min	<b>0 min</b>	5 min	<b>0 min</b>
Save config and commit it	20 min	<b>20 min</b>	10 min	<b>10 min</b>
Create platform Debian package	30 min	<b>30 min</b>	1 min	<b>0 min</b>
Install Debian package	1 min	<b>1 min</b>	1 min	<b>0 min</b>
Update build_config.cmake	4 min	<b>1 min</b>	4 min	<b>0 min</b>
Build application binaries	4 min	<b>4 min</b>	4 min	<b>0 min</b>
Fix problems in building binaries	30 min	<b>0 min</b>	30 min	<b>0 min</b>
Import system into the configuration tool	20 min	<b>5 min</b>	15 min	<b>0 min</b>
Login, download, check operational	40 min	<b>40 min</b>	10 min	<b>10 min</b>
Export test package	20 min	<b>10 min</b>	5 min	<b>0 min</b>
<b>TOTAL TIME</b>	457 min	<b>242 min</b>	270 min	<b>57 min</b>



## 7.2 Suggested next steps

There are multiple areas which could be developed next. The first one is automatizing testing of the created test packages. This should be reasonably easy to do with the used automatic test scripts. At the moment, test binaries are already used by the automatic test runs. Test system takes the binaries from the platform Debian package job and moves those to the correct locations manually. However, it would be good to change the system so that it would directly use the produced test packages which contains also the binaries. This would increase the trust in failure reproducibility since the same files would be used by automatic and manual testing. This would also reduce the time from the first manual step to the last manual step quite significantly since saving and committing config would be the last manual step after this.

The second focus area for the future is looking more thoroughly into automatizing test system configuration updates since this is the last time-consuming step after the code-freeze that exists and requires manual work. Another area where the study did not focus at all is reducing the time spent on manual testing the software. Currently, this is probably the area which causes significant amount of the costs associated with a release. Similarly, execution time of automatic tests is currently rather long and limited by the amount of available test hardware. This is also the same problem as found on the study by Lwakatare et al. (2006). One good focus area for the future would be reducing time required by the automatic tests for example by improving the utilization rate of the test hardware.

On the other side, using Docker for build environment setup could be investigated more. The system that was developed here could be easily developed further. The image created as part of this study was a recreation of the virtual machine inside a Docker image. In the future, it could be beneficial to make smaller specific Docker images for different purposes. If this was done, it would be possible to replace parts of the virtual machine-based development environment with small task-specific Docker images.

Release management tools could also be one suitable area for further research. There exist tools for helping the management of big software releases such as XL Release and

BuildMaster. The benefits and drawbacks of using one could be investigated in more detail.

## 8 CONCLUSIONS

The goal of this study was to find ways to decrease cost and time required to make a new software release for the embedded software project over at Wapice Ltd. The study produced a proposal for new improved release process with the focus on automation of the manual steps. By creating small utilities with Kotlin and Python programming languages many of the previously high effort manual steps could be simplified or completely automated.

The total deploy time after the code freeze could be cut to almost half from 7 hours and 40 minutes to a bit less than 4 hours. The active manual work was reduced by about 80% of what it was before from about 4,5 hours to a bit less than 1 hour. This means that both total time and active working time was saved with the actions made in this study, which would suggest that the study was at least somewhat successful since this was the main goal of it.

The benefit of cutting down the deployment time is not the only advantage of decreased time. Quite often other developers are waiting for the new test packages during the release and this waiting time is often not optimally used. Thus, the study should decrease the time wasted there. Another point to consider is that with the new process, the deployment time varies less. In the old process the time could vary a lot depending on the experience of the release engineer, because learning all the different steps would take some time. With the new process, the manual steps are simplified and are therefore much easier to master.

Moreover, with the help of Docker, it was possible to develop further the existing CI pipeline. Prior to the study, the installer for the configuration tool was only release output available from the build server. After the study, all the needed software outputs for the release are available from the build server. All the improvements are also already taken into use in the case project. These results strengthen the aspect that the study was useful.

However, the study could not completely achieve fully automatic continuous delivery pipeline in the same way as it is usually known over at the web development area. One

remaining issue to be solved is configuration file updates of the embedded systems which are needed during the release. The issue is that doing those currently require the judgment of a release engineer with the help from developers, and due to this no reliable and straightforward way was found for automating the task. This would be a good area for further research since this would remove the need for manual intervention in the middle of the pipeline. However, as mentioned in the literature, the continuous integration pipeline does not need to be fully automatic in order to be useful. The study also helped in identifying the next possible steps for achieving fully automatic deployment pipeline.

Another remaining issue was also mentioned in the literature as a problem for adopting continuous delivery in an embedded software project. This is, the total deploy time including the automatic tests is still higher than what is the length of the pipeline developed as part of this study because running the automatic tests with the created test packages takes a long time due to the limited amount of test hardware. Improving the testing capability would be a good candidate for further research.

On a higher level, this study supported the idea found from the literature that moving towards a continuous delivery can be beneficial for the software project and that CI pipeline can be utilized in embedded software projects as well. This can be seen from the fact that time needed for making a release could be significantly reduced by moving towards continuous delivery. However, in the literature, it was mentioned that embedded software projects need to develop custom solution often to achieve continuous delivery, which was also the case in this study.

In a situation where custom tools need to be developed, the process that was used in this study could be re-used. First begin by identifying the steps of the existing process for getting the desired outputs, find out the most troublesome parts of it by evaluating the amount of active manual work and total step length. Then start by automatizing those steps with the most significant effect on the total times. With this way, improvements to the CI pipeline can be gained even if the fully automatic continuous delivery is not immediately achieved as can be seen from this study.

## REFERENCES

- Anderson, C. (2015). Docker [Software Engineering]. *IEEE Software* 32: 3, 102-105 pp. ISSN: 0740-7459.
- Apache Software Foundation (2018a). Apache Thrift - Home [online]. [Referenced on: 21.5.2018]. Info about Apache Thrift. Available at: <https://thrift.apache.org>
- Apache Software Foundation (2018b). Maven / Introduction to the Dependency Mechanism [online]. [Referenced on: 22.5.2018]. Info about using Maven for Java project dependency management. Available at: <https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>
- Beck, K., M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt & R. Jeffries (2001). Manifesto for Agile Software Development [online]. [Referenced on: 23.4.2018]. Available at: <http://agilemanifesto.org>
- Begel, A. & N. Nagappan (2007). Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study. In: *Proceeding ESEM '07 Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, 255-264 pp. Ed. L. O'Conner. Washington: IEEE Computer Society. Madrid, Spain, September 20-21, 2007. ISBN: 0-7695-2886-4.
- Brown, T., R. Anderson, S. Cooley, R. Wike, M. Keating, D. Delimarsky & A. Childs (2016). About Windows Containers [online]. [Referenced on: 2.10.2018]. Available at: <https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/>.
- Chen, L. (2015). Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software* 32: 2, 50-54 pp. ISSN: 0740-7459.

- Cito, J., V. Ferme & H. C. Gall (2016). Using Docker Containers to Improve Reproducibility in Software and Web Engineering Research. In: *ICWE 2016: Web Engineering*, 609-612 pp. Eds. A. Bozzon, P. Cudre-Maroux & C. Pautasso. Cham, Switzerland: Springer International Publishing. Lugano, Switzerland, June 6-9, 2016. ISBN: 978-3-319-38791-8.
- Combe, T., A. Martin & R. Di Pietro (2016). To Docker Or Not to Docker: A Security Perspective. *IEEE Cloud Computing* 3: 5, 54-62 pp. ISSN: 2325-6095.
- Debbiche, A., M. Dienér & R. B. Svensson (2014). Challenges when Adopting Continuous Integration: A Case Study. In: *15th International Conference, PROFES 2014 Helsinki, Finland, December 10-12, 2014 Proceedings*, 17-32 pp. Eds. A. Jedlitschka et al. Cham, Switzerland: Springer. Helsinki, Finland, December 10-12, 2014. ISBN: 978-3-319-13835-0.
- Docker Inc. (2018a). Best Practices for Writing Dockerfiles [online]. [Referenced on: 22.5.2018]. Information on good practises when writing Dockerfiles. Available at: [https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices](https://docs.docker.com/develop/develop-images/dockerfile_best-practices)
- Docker Inc. (2018b). What is a Container [online]. [Referenced on: 15.5.2018]. Available at: <https://www.docker.com/what-container>
- Docker Inc. (2018c). What is Docker? [online]. [Referenced on: 15.5.2018]. Available at: <https://www.docker.com/what-docker>
- Duvall, P. M., S. Matyas & A. Glover (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. 1st ed. Boston: Pearson Education. 336 pp. ISBN: 978-0321336385.

- Engblom, J. (2015). Continuous Integration for Embedded Systems using Simulation [online]. *Embedded World 2015 Congress*. [Referenced on: 15.5.2018]. Available at: [https://www.researchgate.net/profile/Jakob\\_Engblom/publication/273119043\\_Continuous\\_Integration\\_for\\_Embedded\\_Systems\\_using\\_Simulation/links/54f70f7a0cf2ccffe9d99b8b.pdf](https://www.researchgate.net/profile/Jakob_Engblom/publication/273119043_Continuous_Integration_for_Embedded_Systems_using_Simulation/links/54f70f7a0cf2ccffe9d99b8b.pdf)
- Felter, W., A. Ferreira, R. Rajamony & J. Rubio (2015). An Updated Performance Comparison of Virtual Machines and Linux Containers. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 171-172 pp. Eds. IEEE. Philadelphia, USA, March 29-31, 2015. ISBN: 978-1-4799-1957-4.
- Fowler, M. (2006). Continuous Integration [online]. [Referenced on: 4.12.2017]. Available at: <https://martinfowler.com/articles/continuousIntegration.html>
- Fowler, M. (2013). ContinuousDelivery [online]. [Referenced on: 24.4.2018]. Available at: <https://martinfowler.com/bliki/ContinuousDelivery.html>
- Gerrit (2018). Gerrit Code Review - REST API [online]. [Referenced on: 21.5.2018]. Info about Gerrit REST-API. Available at: <https://gerrit-review.googlesource.com/Documentation/rest-api.html>
- Inedo (2018a). BuildMaster [online]. [Referenced on: 24.4.2018]. Page with general info about BuildMaster. Available at: <https://inedo.com/buildmaster/features>
- Inedo (2018b). BuildMaster [online]. [Referenced on: 24.4.2018]. Page describing BuildMaster's pricing model. Available at: <https://inedo.com/buildmaster/pricing>
- Isaias, P. & T. Issa (2015). *High Level Models and Methodologies for Information Systems*. 1st ed. New York, NY: Springer Verlag. ISBN: 978-1-4614-9254-2.
- JetBrains s.r.o. (2017). TeamCity 2017.2 is Released! | TeamCity Blog [online]. [Referenced on: 3.5.2018]. Info about new features of TeamCity 2017.2. Available at: <https://blog.jetbrains.com/teamcity/2017/11/teamcity-2017-2-released>

JetBrains s.r.o. (2018a). Buy TeamCity [online]. [Referenced on: 3.5.2018]. Info about TeamCity licensing. Available at: <https://www.jetbrains.com/teamcity/buy>

JetBrains s.r.o. (2018b). Integrating TeamCity with Docker [online]. [Referenced on: 20.5.2018]. Detailed information about TeamCity's Docker support. Available at: <https://confluence.jetbrains.com/display/TCD10/Integrating+TeamCity+with+Docker>

JetBrains s.r.o. (2018c). Integrations Support - Features | TeamCity [online]. [Referenced on: 3.5.2018]. Info about technologies supported by TeamCity. Available at: [https://www.jetbrains.com/teamcity/features/technology\\_awareness.html](https://www.jetbrains.com/teamcity/features/technology_awareness.html)

Krekel, H. (2017). Pytest: Helps You Write Better Programs — Pytest Documentation [online]. [Referenced on: 21.5.2018]. Info about pytest testing framework. Available at: <https://docs.pytest.org/en/latest>

Kwak, Y. H. & J. Stoddard (2004). Project Risk Management: Lessons Learned from Software Development Environment. *Technovation* 24: 11, 915-920 pp. ISSN: 0166-4972.

Ledenev, A. (2016). Docker Pattern: The Build Container [online]. [Referenced on: 20.5.2018]. Info about using Docker for Continuous Integration. Available at: <https://medium.com/@alexeiled/docker-pattern-the-build-container-b0d0e86ad601>

Lwakatare, L. E., T. Karvonen, T. Sauvola, P. Kuvaja, H. H. Olsson, J. Bosch & M. Oivo (2016). Towards DevOps in the Embedded Systems Domain: Why is it so Hard? In: *Proceedings of the 2016 49th Hawaii International Conference on System Sciences (HICSS)*, 5437-5446 pp. Eds. B. Tung & R. Sprague. Washington, DC, USA: IEEE. Kauai, Hawaii, January 5-8, 2016. ISBN: 978-0-7695-5670-3.

Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* 2014: 239. ISSN: 1075-3583.



- Pecanac, V. (2016). Top 8 Continuous Integration Tools [online]. [Referenced on: 3.5.2018]. Available at: <https://code-maze.com/top-8-continuous-integration-tools>
- Pfleeger, S. L. & J. M. Atlee (2010). *Software Engineering*. 4th ed. Boston; London: Pearson Education Inc. 782 pp. ISBN: 978-0-13-814181-3.
- Pipenv (2018). Pipenv: Python Dev Workflow for Humans [online]. [Referenced on: 21.5.2018]. Page describing the Pipenv tool. Available at: <https://docs.pipenv.org>
- Powell-Morse, A. (2016). Waterfall Model: What is it and when should You use it? [online]. [Referenced on: 22.4.2018]. Available at: <https://airbrake.io/blog/sdlc/waterfall-model>
- Pursehouse, D. (2018). Pygerrit2 - GitHub [online]. [Referenced on: 21.5.2018]. Pygerrit2's GitHub development page. Available at: <https://github.com/dpursehouse/pygerrit2>
- Rajlich, V. (2006). Changing the Paradigm of Software Engineering. *Commun ACM* 49: 8, 67-70 pp. ISSN: 0001-0782.
- Rancher Labs (2016). Docker-Based Build Pipelines (Part 1) - Continuous Integration and Testing [online]. [Referenced on: 20.5.2018]. Post about using Docker to provide build environment. Available at: <https://rancher.com/docker-based-build-pipelines-part-1-continuous-integration-and-testing>
- Rasmusson, J. (2010). *The Agile Samurai: How Agile Masters Deliver Great Software*. Raleigh (NC): Pragmatic Bookshelf. 262 pp. ISBN: 1-934356-58-1.
- Rodríguez, P., J. Markkula, M. Oivo & K. Turula (2012). Survey on Agile and Lean Usage in Finnish Software Industry. In: *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, 139-148 pp. New York: ACM. Lund, Sweden, September 19-20, 2012. ISBN: 978-1-4503-1056-7.

- Saleh, K. (2010). Effort and Cost Allocation in Medium to Large Software Development Projects. In: *Proceedings of the 10th WSEAS international conference on Applied computer science*, 33-37 pp. Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS). Iwate, Japan, October 4-6, 2010. ISBN: 9789-604742318.
- Schwaber, K. & J. Sutherland (2018). The Scrum Guide [online]. [Referenced on: 23.4.2018]. Available at: <https://www.scrumguides.org/scrum-guide.html>
- Wine (2018). WineHQ - Run Windows Applications on Linux, BSD, Solaris and macOS [online]. [Referenced on: 22.5.2018]. Page describing the Wine project. Available at: <https://www.winehq.org>
- XebiaLabs (2015). XL Release [online]. [Referenced on: 24.4.2018]. Page about available integrations to XL Release. Available at: <https://xebialabs.com/products/xl-release/plugins>
- XebiaLabs (2018). XL Release [online]. [Referenced on: 23.4.2018]. Page with general info about XL Release. Available at: <http://gallery.xebia.com/component/xlrelease>