Mikko Piuhola

Automation of container-based software build pipelines

Metropolia University of Applied Sciences

Bachelor of Engineering

Media Technology

Bachelor’s Thesis

25.3.2017

Tekijä Otsikko Sivumäärä Aika	Mikko Piuhola Konttipohjaisten sovellusjulkaisuketjujen automatisointi 71 sivua + 6 liitettä 25.3.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaajat	Group Manager Pia Kumpulainen Yliopettaja Kari Aaltonen

Insinöörityön tarkoituksena oli automatisoida sovelluskehitystiimin käytössä oleva jatkuvan integraation ja julkaisun järjestelmä ja kehittää sen pohjalta uudelleenkäytettävä malli yleiseen käyttöön. Jatkuvan integraation järjestelmä vastaa sovelluskehitystiimin tuottaman ohjelmakoodin ja järjestelmien testauksesta, julkaisuista ja raportoinnista. Automaation tarkoituksena oli lieventää vanhan järjestelmän ylläpito- ja käytettävyyshaasteita ja kehittää helposti käyttöönotettava sekä luotettava sovellusten testaus- ja julkaisujärjestelmä minkä tahansa sovelluskehitystiimin käyttöön.

Järjestelmä kehitettiin sovelluskonttiteknologioita ja jatkuvan integraation sekä julkaisun toimintatapoja hyödyntäen. Kehitetyn järjestelmän keskeisin osa oli jatkuvan integraation tuote, jota kehitystiimi oli aikaisemminkin käyttänyt. Sovelluskonttiteknologioiden käyttö mahdollistaa testausympäristöjen sovelluskohtaisen määrittelyn ja parantaa luodun järjestelmän toistettavuutta muissa ympäristöissä.

Järjestelmän konfigurointi automatisoitiin käyttämällä useita eri skriptausmenetelmiä. Sovellusten ja järjestelmien testaus- ja julkaisuputket kuvattiin versionhallintaan tallennettuina skripteinä. Tämä mahdollistaa testaus- ja julkaisuputkien kehittämisen sovelluskehittäjille tutuin menetelmin sekä järjestelmän laajamittaisen automaation. Skriptit ladataan versionhallinnasta automaattisesti ja ne sisältävät täydellisen kuvauksen sovellusten testaus- ja julkaisuputkista.

Työn tuloksena oli automatisoitu jatkuvan integraation ja julkaisun järjestelmä, joka voidaan pystyttää nopeasti ja helposti. Kehitystiimin mielestä luotu järjestelmä tarjoaa huomattavia parannuksia käytettävyydessä ja ylläpidettävyydessä verrattuna aiemmin käytössä olleeseen järjestelmään.

Avainsanat	Jatkuva integraatio, sovelluskehitys, automaatio, kontti

Author Title Number of Pages Date	Mikko Piuhola Automation of container-based software build pipelines 71 pages + 6 appendices 25 March 2017
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructors	Pia Kumpulainen, Group Manager Kari Aaltonen, Principal Lecturer

The purpose of this thesis was to automate existing software build pipelines for a development team at an IT company and to develop a re-usable model for widespread use. Build pipelines are used in continuous integration and deployment to form a set of tasks to test, publish and create reports of software. The automation intended to alleviate usability and maintainability issues that were common in the existing system. Another goal of this thesis was to create an easily replicable and usable continuous integration and deployment system that could be used by any software development team.

The build pipeline system was developed utilizing software container technologies, and continuous integration and deployment methods. The core of the system was a continuous integration application that the development team had previously used. Using software containers in the system allows developers to define their own build environments and simplifies the duplication of such systems elsewhere.

The system’s configuration was automated using different scripting methods. Build pipelines were implemented as version controlled scripts. The scripts will allow developers to define their own build pipelines easily with familiar coding techniques. Using scripts to define build pipelines also enabled the automation of build configurations in the system, not just the system itself.

The result was an automated continuous integration and deployment system that can be built from scratch quickly and easily. The development team agreed that the new system was a major improvement in usability and maintainability over the previous system.

Keywords	Continuous integration, software development, automation, container

Contents

Appendices

Abbreviations/Acronyms

CI — Continuous integration. The process of continuously executing tests against software source code and merging all work several times a day.

CD — Continuous delivery. The approach of producing software in short cycles so that it can be released at any time.

CLI — Command-line interface or command language interpreter. The means of interacting with a computer program where commands are issued using successive lines of text.

UI — User interface

RAM — Random-access memory. A form of computer data storage for frequently used program instructions.

API — Application programming interface. A set of functions to access the features and data of an application or other service.

XML — Extensible Markup Language. Text document format designed to store and transport data. Here used mostly for software configuration.

URL — Uniform Resource Locator. A reference to a web resource that specifies its location and a mechanism for retrieving it.

Glossary

Software development The act of producing applications and services, usually within a release cycle.

Software deployment The process of making an application or a service ready for use. Usually it involves some activities from the manufacturer or from the customer.

Build The process of constructing something that has an observable and tangible result. Within continuous integration the term often includes the steps for producing that result and testing it.

Build pipeline A group of parallel and linear stages of a build that form a cohesive whole that describes the flow of an application of service from build to testing to production.

Software environment A group of one or more computers, and possibly services, that form a single target for software deployment.

Production Refers to a location, usually a computer to which software deployments are made to make an application or service ready for use.

Open-source software Software which source code is available for the market and is often free to use.

Version control A method or a system where software source code is stored and can be tracked by its changes. Version control systems often include a concept of branching.

Branching The process of duplicating an object in software source code to allow making modifications to that source code in parallel to other branches.

Operating system The low-level software that enables the basic functions of a computer, such as scheduling tasks, and interacting with internal and external components.

Kernel The lowest or most core part of an operating system, often responsible for resource allocation, file management and security.

1 Introduction

Modern software development is built around the idea of fast feedback loops that allow software developers and their teams to quickly react to issues and create new features. Many technologies and methodologies are used to enable these fast feedback loops but one piece is crucial for minimizing the amount of time spent on unnecessary tasks, such as manually executing tests and software deployments: a functioning continuous integration system.

As the ultimate goal of software development is producing services and products for users, any time during development that is not used to design or produce that produce will hinder the progress of reaching that goal. Developers should instead be empowered to do what they do best: develop software.

Modern software development teams use a vast array of tools and techniques that promise to help the teams reach that goal without worrying about the extra stuff. One, and one of the most crucial categories of those tools are the continuous integration (CI) tools and applications. Continuous integration and continuous deployment (CD) tools are meant for automating the testing and deployment of software and systems. Different tools in this category take different approaches to testing and deploying software; some provide more granular inspection and reporting of test results, some expect external systems to take care of the details and focus more on visualizing the whole pipeline from source code to production deployments.

Though CI and CD tools promise to remove most manual steps from testing and deploying of software, they still require configuration of the tools themselves and, in most cases, of the software and systems being tested and deployed. The configuration is traditionally done using, often complex, graphical user interfaces and by copying and pasting non-human-readable configuration files.

Another facet of modern software development is the use of so-called containers, such as the Docker containers used in this thesis. Software container technologies provide means of describing and building applications and systems in a controlled and highly repeatable manner. These qualities make them an interesting pairing with CI and CD tools where repeatability and controllability provide a necessary basis for assuring the quality of applications and systems.

The goal of this thesis is to design and implement an automated continuous integration and deployment system that will alleviate usability and maintainability issues faced by a software development team at Digia Oy using their current continuous integration system. The existing system requires too much manual configuration and management, and provides poor usability for the developers, leading to slowdowns in development and unnecessary errors. This thesis also intends to study the use of software containers in build pipelines to allow developers to define their own build environments. Finally, the research and development done for this thesis project are used to create a re-usable template of the described build pipeline system for wider usage within and outside the company.

2 Project background

2.1 Project goals

The purpose of this thesis is to automate existing software build and deployment pipelines using continuous integration and deployment tools and processes. The pipelines use modern software container technologies for software deployments. Another target was to create a system that would be highly reusable and easily configurable for any software project within the company and outside of it. This thesis was done for Digia Oy (official logo show in Figure 1).

Figure 1 Official logo of Digia (1)

Digia is an information technology (IT) service company with software projects in many industries, such as banking, insurance, the public sector and telecommunications. Digia currently employs over 870 experts in Finland and Sweden and is expanding their international presence. Being an IT service company, continuous integration and build pipelines are a key part of the everyday business.

2.2 Project environment and requirement frame

The focus of this thesis is to automate an existing continuous integration and deployment system. This thesis uses that system to define its requirement frame. Automating an existing system also means that this thesis attempts to solve the specific problems and weaknesses in the current system, instead of developing a new system in a vacuum.

The current system utilizes Jenkins (2) as it continuous integration and delivery tool. The non-profit Software in the Public Interest, that holds the Jenkins trademark, describes Jenkins like this: “Jenkins is an open source automation server which enables developers around the world to reliably build, test, and deploy their software” (2).

One of the major pain points and a reason for implementing the system described in this thesis is the difficult configuration style of the current system. The current style can and has resulted in avoidable errors, as shown in chapter 2.4, that hinder the progress of the software development team.

The existing system is used by a team working on software high-security projects that handle classified information, due to which any usage of services located outside of Finland's borders should be limited to a minimum.

Due to the high security level requirements, any cloud-based continuous integration and deployment services were discarded as options, if they did not also provide a method of hosting the services on private servers. Though some of the most popular tools and services in this area are cloud-based, this requirement also provided the benefit of the template created in this thesis to be fully portable and usable in even more stricter environments.

The continuous integration tools also need to have at least some level of support for building so-called software containers, primarily Docker (3). Software container technologies are discussed further in this thesis.

The selected tools were also required to have some reporting capabilities, mainly for viewing software test results and test coverage reports and sending those reports forward. The tools need to support currently used formats, such as Cobertura (4, pp. 45-46) reports.

For source code version control, supporting Git (5) was the main requirement. GitHub Enterprise (6) is the source code management service provided by the project's customer, meaning that any automatic discovery of projects inside a version control system should support using the GitHub Enterprise platform. GitHub Enterprise is a privately hosted version of the popular GitHub version control service. Supporting other equivalent systems and services was seen as a positive but not an absolute requirement.

2.3 Research methods

The work in this thesis was divided into three main parts: baseline questionnaire, implementation and a review discussion. Due to time constraints set for the work, the implementation was not taken in production use but instead, a demonstration was given with some time for general discussion and review of the implementation and ideas.

The purpose of the baseline questionnaire was to gather information on the current state of continuous integration systems and understanding in the project environment, and to guide the implementation's focus. Though the idea for this thesis arose from the obvious need for automation in the project's continuous integration system, the questionnaire was necessary to allow for some level of qualitative analysis of the final implementation.

Next, the implementation was to be done as a template or demo version of such a system but with the requirement frame set by the current project environment. This is discussed further in chapter 2.4. The template can later be used to modify for actual usage in the target project, or any software project with a need for some level of continuous integration, which should include most modern software development projects.

After the implementation, a demonstration for the answerers of the baseline questionnaire was held. During the demonstration, any questions or concerns that arose were discussed and written down.

2.4 Baseline questionnaire

Overview

To get a good image for the state of the project's current continuous integration and deployment tools and practices, a baseline questionnaire was given to the project's team members and management in October 2016. The questionnaire also served as a guideline for the thesis work team to decide on what features to focus most heavily on, and which features might not be necessary at all.

The questionnaire was implemented as a Google Forms (7) questionnaire. Most questions required the answerers to choose one option multiple answers but some allowed multiple answers per question, and some had optional free text questions to get more detail out of some specific answers. The answerer background questions were mandatory to answer but others either had an “Other” option or were fully optional. Development and testing specific questions were also skipped for answerers who were told the questionnaire they were not currently in a development role.

The questionnaire had 13 participants in total, from varying job roles and backgrounds, as can be seen from the background question responses. Over half of the people the questionnaire was sent to, responded to the questionnaire. The questions and answers were originally in Finnish but have been translated for this thesis while attempting to convey original language and meaning. The full list of results is available in appendix 1.

Background information

A set of background information questions were set up to get some context for the given responses. The majority of answerers held at least a bachelor’s degree or equivalent and had at least 10 years of experience in the field (Figure 2).

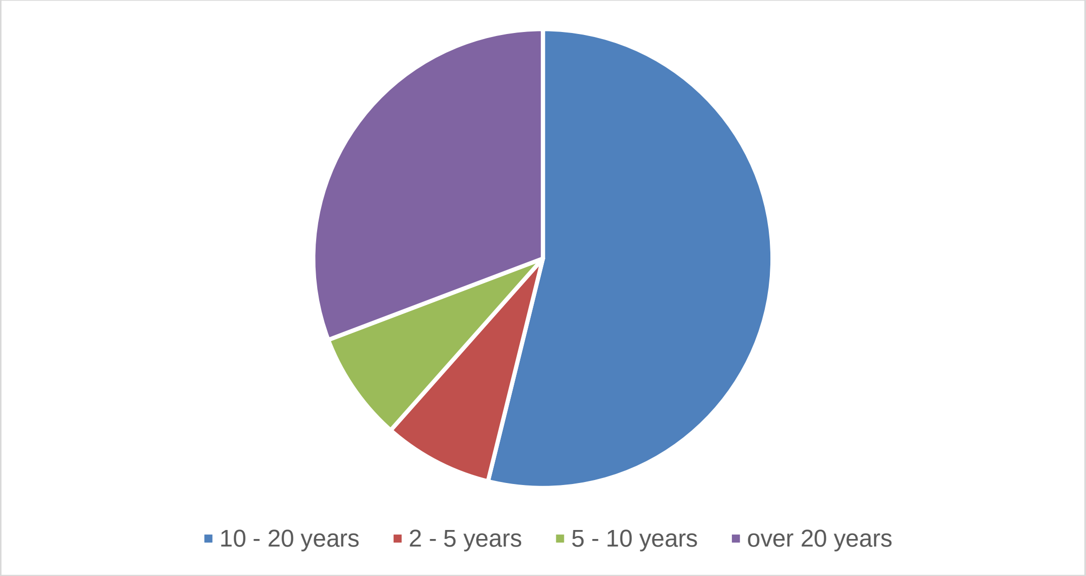


Figure 2 Work experience in years

When asked if the answerers do software deployments in their current role, roughly three quarters answered yes. The people who answered “No”, had either moved on from coding-type tasks or had not yet done any software deployments in their career. This also matches up with the responses for a question regarding their current role in the project, where roughly three quarters told they worked in a development role and the last third in project management. This is also in line with the group of people the questionnaire was originally sent to.

Previous experiences of continuous integration systems

The purpose of this section was to map the general level of familiarity of continuous integration and deployment systems amongst the answerers, and to find some common issues in the field.

Eleven out of thirteen answered yes, when asked whether they had used continuous integration systems before. Only just over half said they had previously configured CI systems, though this includes configuring jobs inside the systems not only the systems themselves.

The answerers were also asked to describe their answers in more detail, if possible. The levels of manual tasks had ranged from configured continuous integration jobs by hand to running scripts to add software to those systems. Automation had either been done based on some configuration files located in their respective version control systems or fully automatically based on version control branching.

Current continuous integration system

This section was designed to pinpoint the major issues in the current CI system. In the first few questions, the responses tell that most answerers were at least slightly familiar with continuous integration and deployment systems, and fewer with Jenkins CI specifically.

The responses were quite evenly distributed across the scale (Figure 3), when asked about the confidence level of adding their software into the current CI system. “Adding software”, here, means configuring the CI system to execute tests and possibly deployments against their software, as the questionnaire explained. The scale was from one to five, where one was described as meaning “Not at all” and five as “I’ve added multiple pieces of software”. Though many were relatively confident that they would be able to do it, roughly three quarters had not added any software to the current CI system.

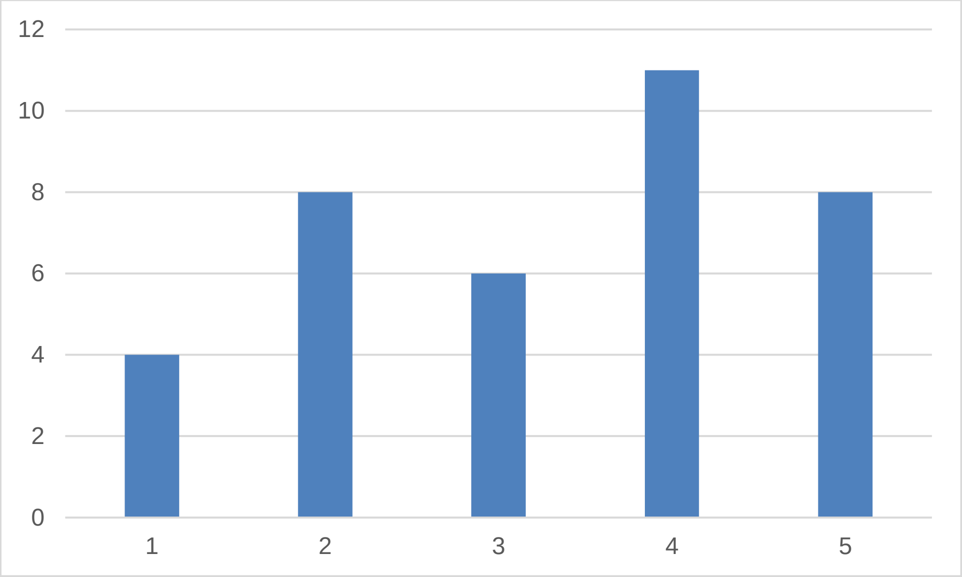


Figure 3 How confidently would you add a software project into the current CI system?

Furthermore, those that had done it, responded that they did not find it particularly easy; answers ranged from two to four, on a scale of one to five, where one was described as “Extremely difficult (would require guidance)” and five as “Extremely easy”. The responded even further towards the bottom of the scale when asked how well the answerers had understood what configurations and changes they had made.

The level of confidence and understanding was also reflected on the next question, where the responses told that errors in adding in software were not too rare. The most common reasons for those errors were said to either revolve around configuration difficulty or negligence. The thesis team understood the negligence answers as being more of a consequence of the difficulty of configuration rather than actual negligence.

All the answerers that had done at least some deployments on the current CI system said they would know how to do a deployment to the project’s test environment. In the current system, this usually involves either making source code changes to a specific version control branch or manually clicking a button in the CI system. Regardless, half of the answerers did not find the operation very easy but they were still confident in their ability to execute the deployment (Figure 4).

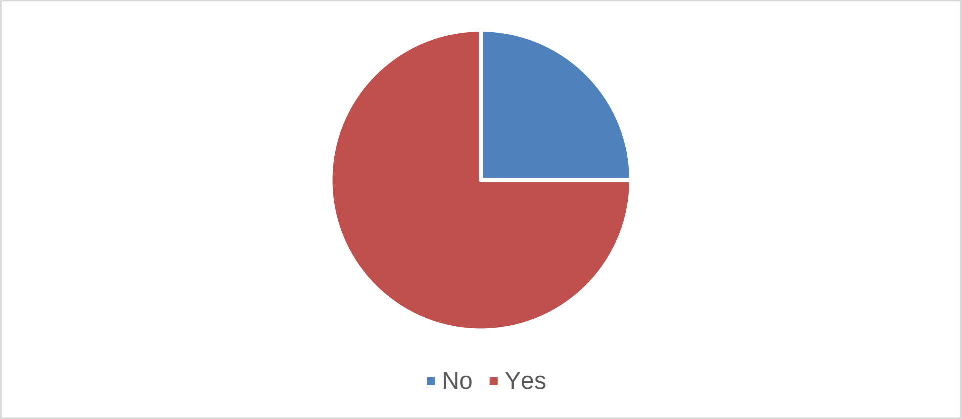


Figure 4 Would you be confident doing a software deployment to the test environment on the current CI system?

The people who answered “No” said they would rather first have someone who knows Jenkins CI to check the configurations. Most also said they sometimes do fully manual deployments outside the CI system. When asked for a reason for doing manual deployments, one person said it was too difficult to do with Jenkins CI and another wanted to test their changes in a real environment before committing their changes into version control.

Possibilities in a continuous integration system

The previous sections were meant more to map the current state of the CI systems but this section was designed to figure out the most important new features to implement in the demo system. First few questions of this section focused on the extensibility and level of manual control the answerers would desire: at least a portion of the answerers wanted the ability to do some manual deployments through the CI system – for example by defining what version control branch they wanted deployed (Figure 5).

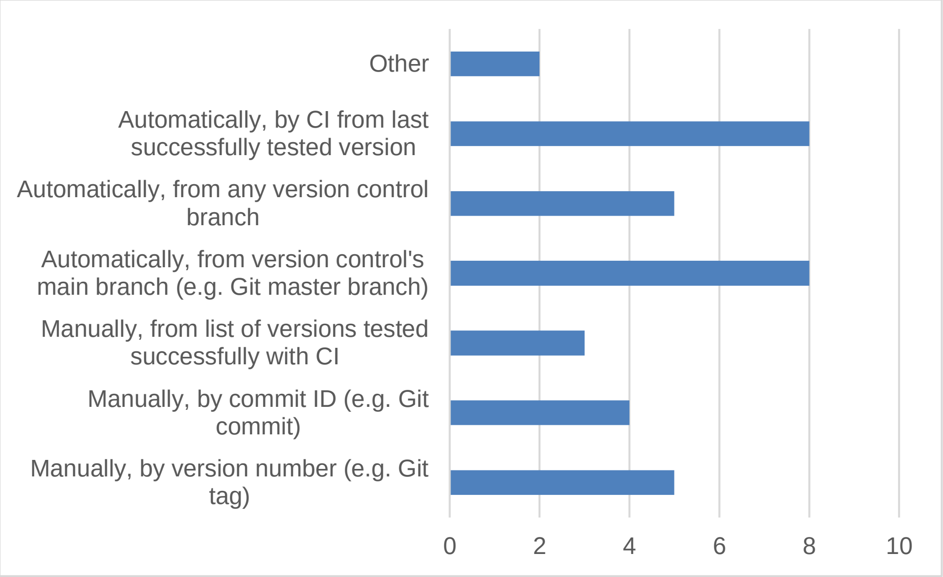


Figure 5 On what basis, should one be able to deploy software to the test environment?

The next two questions attempted to find out how common it is to need unique testing environments within a relatively large development team. As the demand was clear but not overwhelming, the thesis project team decided that creating such a feature would be desirable but not imperative. Nearly half of the answerers told they wanted CI system administrators to implement those unique environments in behalf of the developers. This is in line with the scale of answers to the previous question.

Altogether, the responses indicated that there is a clear demand for the capability of creating different testing environments for different occasions – between separate applications and even within the development cycle of a single application. One method of doing this was later implemented using software container technologies described in this thesis.

Developer requirements for a continuous integration system

For the thesis project team to be sure of which new features to develop and which to drop, a series of questions was set up for the questionnaire’s next section. These questions focused heavily on testing and test coverage since the team had requested it.

When asked whether the answerers needed test coverage data or reports for their software, the clear majority answered “Yes / for some of my software”. The thesis project team later pondered whether the people who had answered “No” were mostly working on tasks where testing should mostly be done to validate data not to cover, for example, business logic in an application. In those cases, test coverage might be seen by the answerers as unnecessary.

Another worrying set of responses, from the thesis team's perspective, was for the section on whether developers know when and why their tests fail on the current continuous integration system. First, over one fourth said they only knew their tests were failing when "someone else tells them about it". The current system does send constant feedback to a shared messaging platform but either the development did not follow those notifications or, for example, the volume was too large for the developers to pick out the important content from the noise. This was to be considered during the development of the new automated system but like the test coverage reporting, this was not to be the focus of this thesis.

Almost half of the answerers also said they only understood why their tests failed when they ran them locally (Figure 6), meaning that the output or visualizations in the current CI system was clearly inadequate. This situation could be improved with either a new CI tool that has improved usability in this regard or by using other methods of visualizing the test results in the current system, like the Jenkins Blue Ocean project (8).

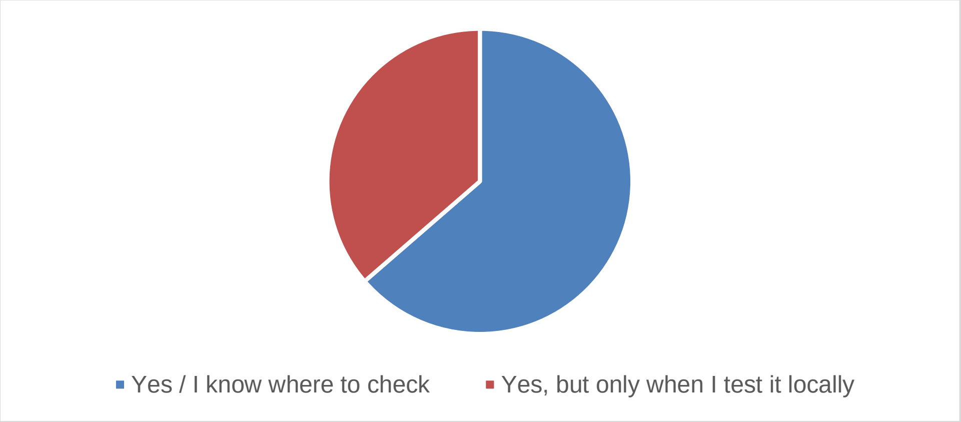


Figure 6 Do you know why your tests fail?

Almost half of the answerers said they didn't need artefacts built from their software. "Artefacts" were described as meaning a packaged state of an application, such as Docker images. This did not surprise the thesis project team as nearly half of the people who answered worked mainly on data integrations which often cannot be described or distributed as artefacts, per se, but instead as content.

General requirements for a continuous integration system

Next, the answerers were asked an open-ended question: “When the software is ready to deploy, who do you think should be able to approve the deployment to the test environment”? Most people wanted approval from at least two people, usually from someone from management. Some people wanted to deployment to be fully automatic after all tests have completed successfully against the source code and possible against the development environment. A few people also alternatively wanted approval from a dedicated testing person.

In the case of the team’s production environments, a clear majority wanted approval from a project manager, a customer or at least multiple people. The results for both questions were not surprising due to the current project environment the answerers are working with, where production deployments are done through a rigorous change management process.

Other questions and conclusions

The questionnaire finished with two open-ended questions: “Did this questionnaire bring up any other questions?” and “Do you have any needs from a CI/CD tool that were not included in this questionnaire?”. One answerer talked about the importance of competent and skilled tester: most testing automation systems, in the end, rely on a person to write the tests, meaning careless errors can always happen. Especially, developers who might not have that much testing experience cannot know every possible angle a piece of software should be tested.

Another answerer wondered how the CI/CD tools themselves can be operated in a way that assures they will always be up and running. This is an excellent point to consider when designing any kind of infrastructural systems, especially when its job is to assure the quality of other software.

Some people wanted the CI/CD tools to be configurable via files stored in version control, alongside the software itself. Using version controllable configuration files, forms the foundation of creating fully automated discovery of software in a version control system. It also helps with improving the transparency in CI/CD tool configuration, as the configurations are not done externally to the software being tested and deployed.

One answerer referred to the question about who should get test coverage reports and on what level; most reports found in CI/CD tools contain very low-level data, usually on the source code level, and in an often visually unappealing format. The reports for management-level personnel and customers should instead be on a higher level, and preferably stylized to match the project's visual look and feel.

In conclusion, the thesis project team mostly predicted the questionnaire's answers but the responses still helped with quantifying previous speculative thoughts on the current state of the project's CI/CD system. Some levels of distrust on the current system – such as the confidence to do deployments – were worrying but not entirely unexpected.

The thesis work team agreed to focus most of the work on creating an easily managed and reliable CI/CD tool installation with some version controlled way of generating build pipelines for new software automatically. The reporting features, though desirable, were seen as extra features considering the automation focus of this thesis, and would only be included if they could be implemented in the set timeframe. Software containers or equivalent solutions could be used to allow the desired customization of testing environments on a per project basis, or even more granularly if possible.

3 Continuous integration

3.1 Overview

Modern, agile software development teams often produce a lot of new or updated code. As software is built feature by feature, and often with multiple iterations, the need for making sure those new features are well integrated between themselves. Traditional methods require that development teams wait until all code-related tasks are completed before any changes can be combined, and eventually released. That kind of process, however, disagrees with the basic agile development ideas of fast feedback and iteration (visualized in Figure 7). This is essentially the problem the practice and philosophy of continuous integration attempts to solve. (9, pp. 94-95)

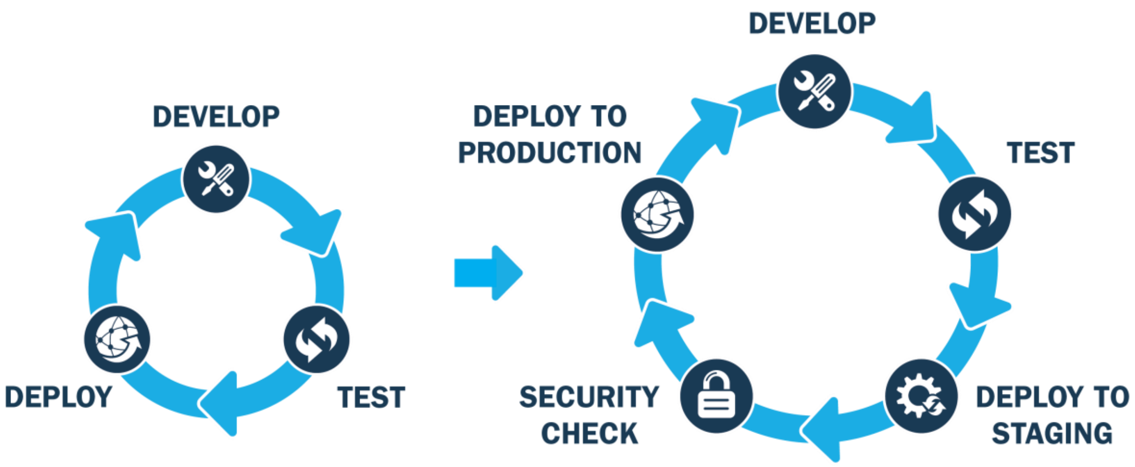


Figure 7 An example of the iterative continuous integration process (10)

As a technical solution, continuous integration involves an automated server or service that executes an integration process when changes are made to software source code. Continuous integration tools are used to build and test software at least with partial automation, preferably fully automatically. This process is not revolutionary, as most testing and integration tasks have always been done through some level of automation, for example by using simple scripts. Using a continuous integration tool simply attempts to remove those extra manual steps from the equation, like most automation tasks do. (9, pp. 95, 98)

The most obvious level of testing done in continuous integration systems is, of course, integration testing. As the name might suggest, integration tests are designed to verify the interoperation of all the components in a piece of software and even between different software (9, p. 79). Software is usually built out of multiple components, so making sure those components interact the way the developer expects, is key to good software development.

Some typical error cases for integration tests are: code method A calls an incorrect method B, method C calls method D with incorrect parameters and mistimed method calls or responses (9, p. 80). There are many strategies for implementing integration tests successfully but this thesis mainly focuses on providing an execution environment for those tests.

Another common level of testing is unit testing. As Linz puts it in he’s book “Testing in Scrum”: “unit test cases are designed to check whether individual software components

[…] work correctly” (9, p. 85). These kinds of tests are designed with some understanding of the internals of a specific component, and not just its interfaces. Unit tests are often the basis for calculating test coverage for source code. Test coverage simply describes which parts of source code have been covered in test cases. Those parts might be lines, code branches or class methods. (9, p. 85)

In advanced continuous integration systems, even operating system and hardware level changes can be tested to help with software’s quality assurance. This, however, is often more linked to continuous delivery, described in chapter 3.4. The two terms, continuous delivery and integration, are often linked together and the abbreviation “CI/CD” is commonly used when referring to the tools providing the features both processes entail.

For continuous integration systems to provide the fast feedback loop agile software development requires, they also need to have some methods of providing communications and notifications to the development teams and management. Having a notification system, such as email or an instant-messaging platform, allows developers to instantly know whether their tests failed and enable them to take immediate action.

Creating a stable and extensible continuous integration environment for a software development team is vital for it to function in the way it needs to. Making that environment automated, will remove unnecessary manual tasks from developers and free their time for actual design and development of software.

3.2 Importance of automation

The challenge in doing continuous integration without automation is that it can take up crucial time that could be used for more productive tasks when the systems need manual input or configuration. Manual configuration, especially, is a dangerous path to go down as amount of manual labor rises and error prone manual adjustments are made by hand. Furthermore, the processes involved in building and testing applications, generating test reports, and deploying software can be complex multistage ordeals, where involving a human-component can be detrimental to the processes’ completion. (11, pp. 23-25)

Automating a continuous integration system also alleviates the need for taking constant backups of the configuration files and contents as the configurations can always be reloaded from, for example, version control. Using version control to store and load CI/CD configurations also provides the benefit of all changes getting tracked and them being easily viewable by anyone with access to the version control system. It also makes a lot of sense to store the descriptions for testing an application right alongside the application's source code.

By making constant testing an automated process – instead of simply making it a requirement for some higher process – the team's job in assuring its software's quality moves away from being a separate step in the development process to being an everyday task that the team does not necessarily even need to care about. It also moves more of the responsibility of testing to the actual developers, instead of specialized testing teams. Of course, the benefit of having people especially trained in testing software cannot be understated. Automating the lower levels of testing, such as unit and integration testing, removes the same amount of unnecessary manual tasks from the testing personnel as from the development team.

The automation does not need to end in configuring individual build pipelines but, as this thesis intends to show, the generation of whole CI/CD systems can be automated to such a level that they can always be brought online at any given time, fully operational. This level of automation is readily available with the introduction of software container technologies described in chapter 4 and implemented as a part of this thesis.

3.3 Version control

Version control is a key part of software source code management. It is a system used to keep a record of changes made to one or more files. Version control comes in many forms: some systems simply keep every version of a file as a copy but systems designed for source code management, like Git, attempt to only keep a log of the changes made to files. Git is the version control system used in this thesis (official logo shown in Figure 8). (5)

Figure 8 Official logo of Git (12)

One fundamental feature of Git, and many other version control systems, is the ability of create so-called branches; a version control system’s branch allow developers to make parallel, and even conflicting changes to the same software. Git’s branching model is lightweight, making it ideal for short-lived development branches. All changes in Git are made in branches, even if no new branches are created, as the initial state is always a “mainline branch”, often named “master”. (13, p. 89)

Git also revolves around the concept of a shared central code repository (Figure 9). A repository is “a database containing all the information needed to retain and manage the revisions and history of a project” (13, p. 31). The changes to files in a repository are called commits and they contain some metadata: who made the changes, who added those changes to the repository, comments and the date when the changes were made. (13, pp. 31-32)

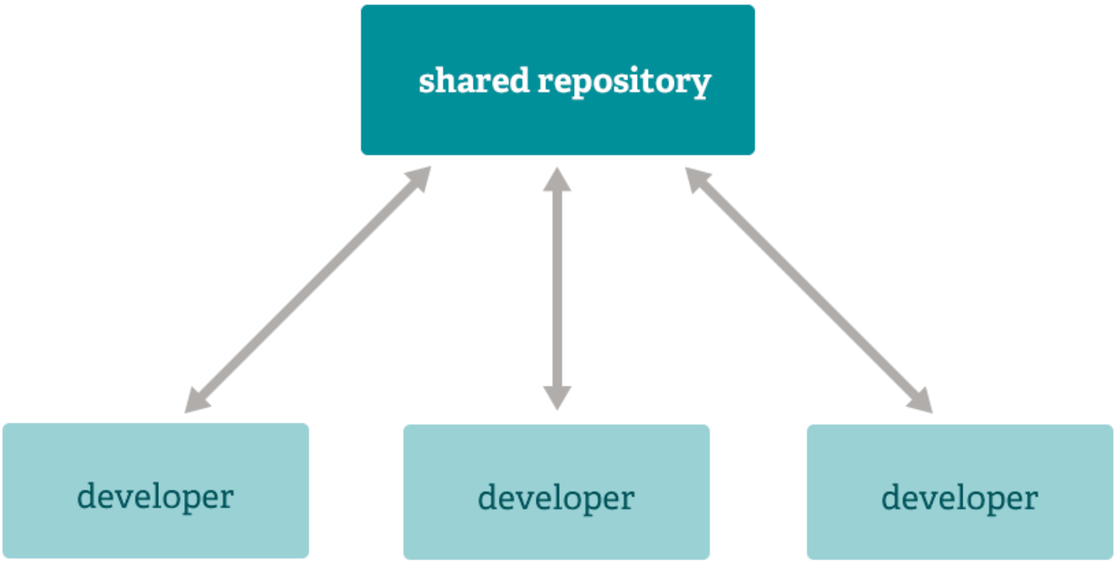


Figure 9 High-level view of a Git repository (5)

In continuous integration, branches can be used to clearly separate development into phases by deployment environment: for example, the current project team has a convention of using a branch named “master” to signify changes in the test environment and a branch named “dev” for the development environment (Figure 10). This practice, along with using separate, short-lived branches for feature development is a lightweight version of the Git branching model presented by Vincent Driessen in a blog post in 2010 (14). It is also not too dissimilar to the model presented by GitHub, called GitHub Flow (15). Using short-lived branches is key to enabling a good continuous integration process where all changes are merged often into the mainline branch. (13, pp. 89-90,9, pp. 94-95,14)

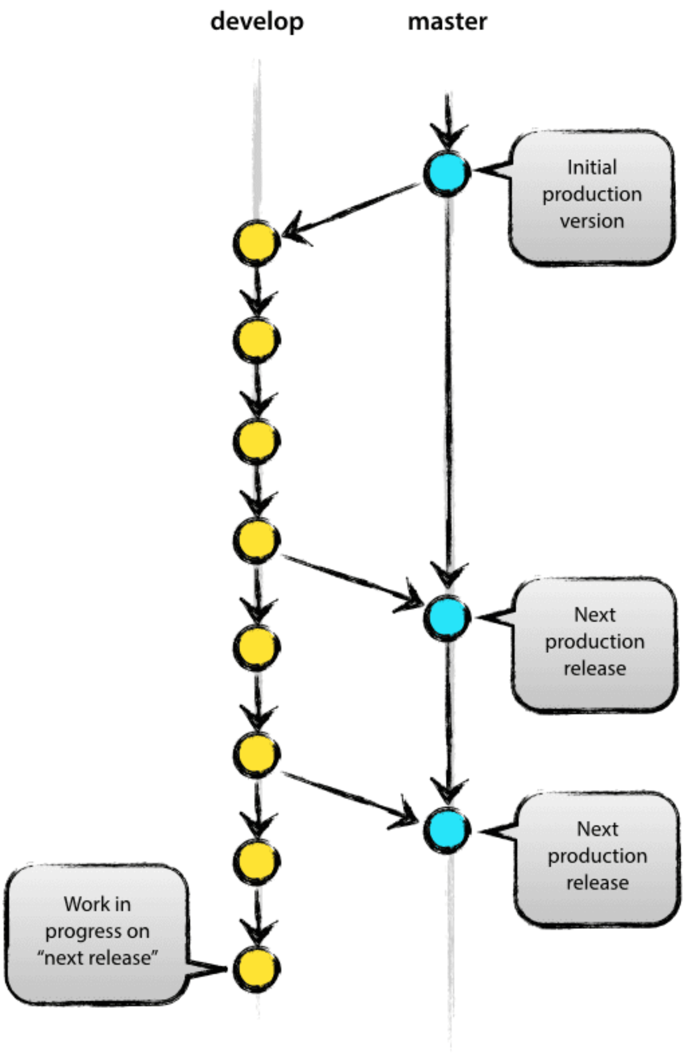

develop
master
Initial production version
Next production release
Next production release
Work in progress on "next release"

Figure 10 Main branches in the Git branching model presented by Vincent Driessen (14)

A continuous integration system can be used to generate separate workflows, or build pipelines for every version control branch. This allows developers to follow the test results of their own changes clearly. The CI process recommends developers to check in their changes to a shared central code repository at least daily, preferably after every change (9, p. 95). When changes are constantly and automatically checked against well-defined unit and integration tests, the process of assuring software quality is made rather simple as later merging those changes can be done with more confidence that they won't create conflicts or errors in the software.

3.4 Continuous deployment and build pipelines

While continuous integration is mostly about the testing and merging of software source code constantly to improve quality assurance, continuous deployment focuses on delivering that software to users and customers. After all, software has little value until it has users. The terms continuous integration and delivery – often written as CI/CD – are usually combined at least at some level. Continuous deployment relies on continuous integration in that software must be properly tested to allow reliable deployments of that software. Otherwise, continuous delivery would simply mean constant delivering “something” and not something of value.

Figure 11 shows an example of a CI/CD system visualizing the build pipeline from source control, all the way to preparing a production environment deployment. If the deployment to production will be made automatically, this would be called continuous deployment but if the build pipeline only prepares the deployment and waits for manual input, the appropriate term would be continuous delivery, as explained by Jez Humble (16).

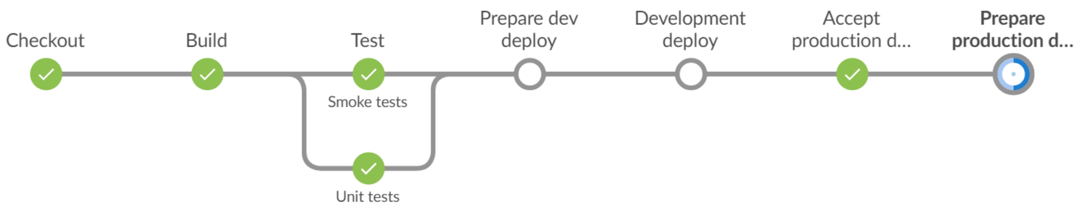


Figure 11 Build pipeline, as show in Jenkins Blue Ocean (8), in progress of preparing a production deployment

The difference between continuous delivery and deployment is also well visualized by Yassal Sundman in her blog post (17) as show in Figure 12. Continuous deployment implies continuous delivery, as the difference is mainly the final deployment step: in continuous deployment, all deployments are done fully automatically when all tests pass but in delivery some manual approval is required. This thesis’ implementation of continuous deployment is only partial; as defined in chapter 2.2, the project’s customer has a change management process for deploying changes to production, so that stage will always require some manual approval, but deployments to development and testing environments can be fully automated.

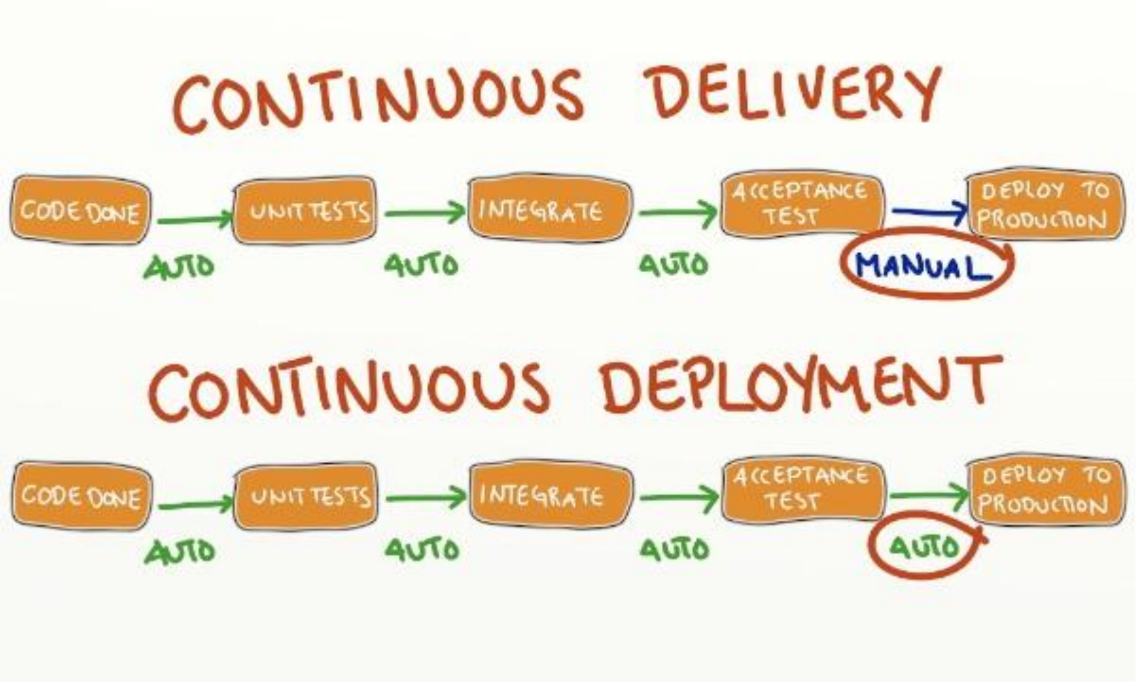
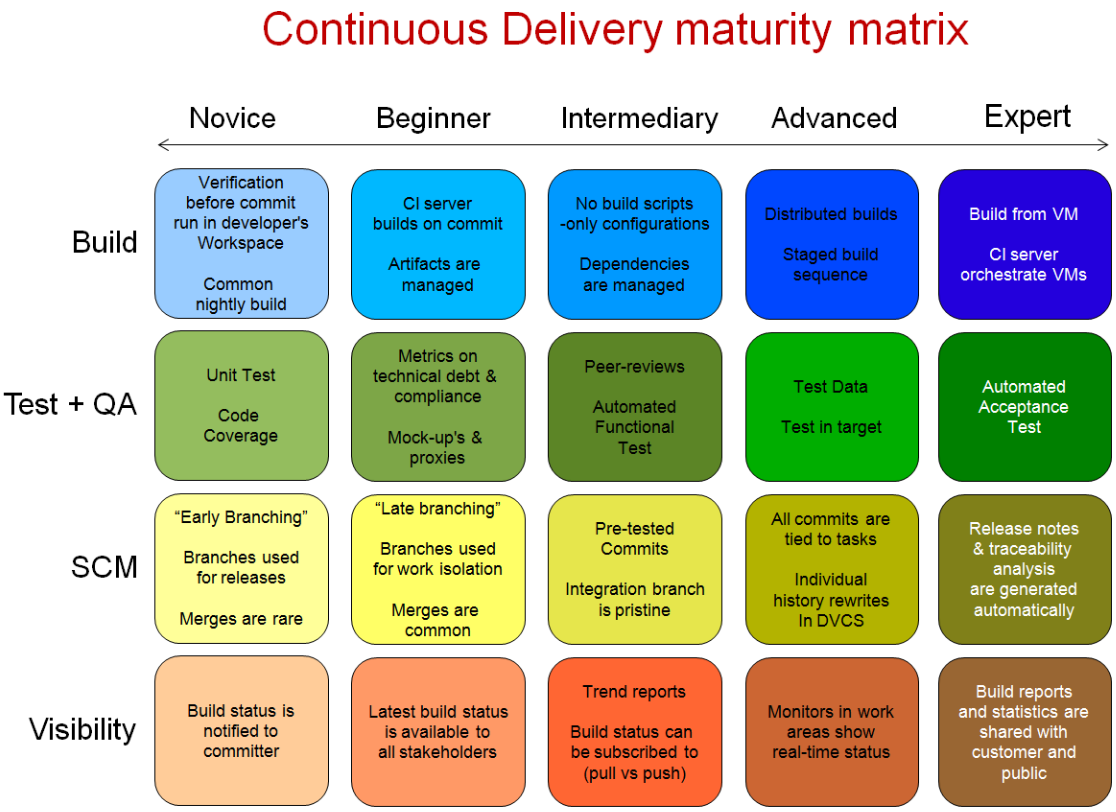


Figure 12 The difference between continuous delivery and deployment is the automation of the deployment step (17)

As Jez Humble explains: “if you can’t release every good build to users, what does it mean to be ‘done’ with a story?” (16). Without automatic deployments, traditional waterfall development starts taking ground back; features are only released when some larger whole is finished and not at the time the features themselves are done. Though, continuous delivery and deployment can be thought of as having different maturity levels, as visualized by Chris Shayan (18).



	Novice	Beginner	Intermediary	Advanced	Expert
Build	Verification before commit run in developer's Workspace Common nightly build	CI server builds on commit Artifacts are managed	No build scripts -only configurations Dependencies are managed	Distributed builds Staged build sequence	Build from VM CI server orchestrate VMs
Test + QA	Unit Test Code Coverage	Metrics on technical debt & compliance Mock-up's & proxies	Peer-reviews Automated Functional Test	Test Data Test in target	Automated Acceptance Test
SCM	“Early Branching” Branches used for releases Merges are rare	“Late branching” Branches used for work isolation Merges are common	Pre-tested Commits Integration branch is pristine	All commits are tied to tasks Individual history rewrites In DVCS	Release notes & traceability analysis are generated automatically
Visibility	Build status is notified to committer	Latest build status is available to all stakeholders	Trend reports Build status can be subscribed to (pull vs push)	Monitors in work areas show real-time status	Build reports and statistics are shared with customer and public

Figure 13 Continuous Delivery maturity matrix (18)

As the matrix in Figure 13 shows on its build row, continuous integration is an integral part of both continuous delivery and deployment. Without a solid base of automated builds and tests, proper continuous delivery will start to crumble and might even cause harm to the image of the development team as it cannot be seen to be deliver quality software when errors only show up after deployments.

Continuous deployment also relies heavily on the cultural aspect of trusting the CI/CD systems and developers to create and deploy software which quality can be assured. Without that trust, users will require long-spanning manual testing periods which slows down the iteration processes of development teams. Of course, simple unit and integration level tests should not be enough to fully automate deliveries all the way to production but instead some level of acceptance testing should be included in the build pipelines, as well. The acceptance tests can be codified and programmed, but it could be possible to integrate CI/CD systems to some other tools where manual testing is done but the acceptance is passed automatically to the CI/CD systems, skipping the need for extra communications.

Build pipelines (Figure 11) are a central part of continuous deployment. They are either conceptual of concrete groupings of stages software needs to go through to allow it to be deployed. Some tests might be run in parallel, such as long-running test cases, but mostly these pipelines are like any other production pipeline: if a previous stage finishes successfully, execute the next stage.

The importance of good visualizations of build pipelines are especially key for persons in the team that might not need the low-level information of specific unit tests but want to instead know whether a specific feature has shipped to users. This was one of the pains seen in the project team's current CI environment – as shown in chapter 2.4 – as separate stages did exist as individual tasks in the tool but they were separated from each other or were hard to combine and manage.

3.5 Continuous integration and deployment tools

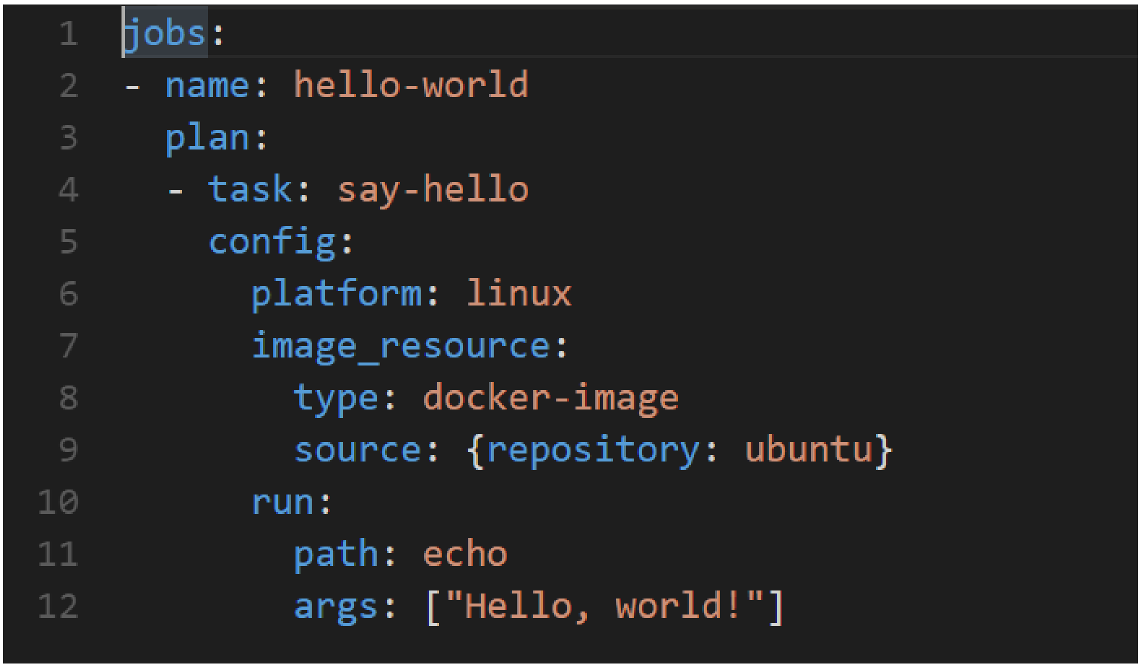
The premise of this thesis was initially to automate an existing continuous integration system but as the current tool, Jenkins CI, was seen as lackluster – especially in the user experience department – the thesis project deemed it necessary to evaluate other tools in the category, as well.

For a tool to be considered in this comparison, it had to have some concept of multi-stage builds, or build pipelines. Some paid tools were considered but mostly tools with at a free option were chosen as the results of this thesis are supposed to be used in demonstrations, for educational purposes and as a template for any kind of software project, regardless of financing.

The selected tools for the comparison were Concourse CI, TeamCity, Drone.io, GitLab CI, GoCD and the original Jenkins CI. The comparisons were mainly done at the feature level and by reading the respective systems' documentations. Some, more promising tools were tested within the given time frame. The results shown here should be considered from the point of view of this thesis' requirement frame and not as objective truths.

Concourse

The first tool in the comparison was a CI tool named Concourse. On its homepage, it immediately shows a clear visualization of a build pipeline – starting off with good promises. It also has a text document format (Figure 14) that is used to configure the build pipelines, ticking another requirement box. (19)

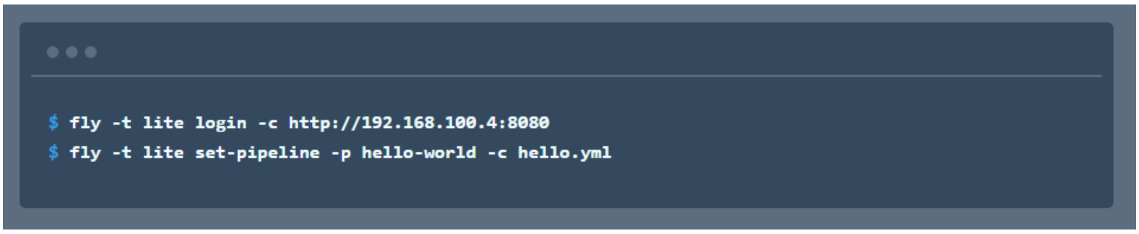


```yaml
jobs:
- name: hello-world
  plan:
  - task: say-hello
    config:
      platform: linux
      image_resource:
        type: docker-image
        source: {repository: ubuntu}
      run:
        path: echo
        args: ["Hello, world!"]
```

Figure 14 Concourse build pipeline configuration file (19)

Concourse's concepts for creating build pipelines also appeared clear: tasks as isolated execution environments for pipelines steps, with clear input and output interfaces, highly abstracted resources to start and end pipelines, like timers and version control systems, and jobs to tie those components together. (20)

But, where it fell short for the level of automation this thesis required was in the way those jobs were added to the CI tool: using the CI's own command-line interface (CLI), shown in the tool's homepage (Figure 15). Requiring the use of this tool conflicted with the idea that adding software to the CI/CD system should be as easy as clicking a few buttons or no manual interaction at all. The thesis project team also saw this as an unnecessarily complication of a relatively simple action – "add this configuration file to the system" – and though that it did not show Concourse in a good light, in this regard. This was the ultimate reason for not choosing Concourse. Though, as its pipelines concepts appeared excellent, at least in theory, the tool should be reconsidered later if they provide more automated means for adding software to the system.



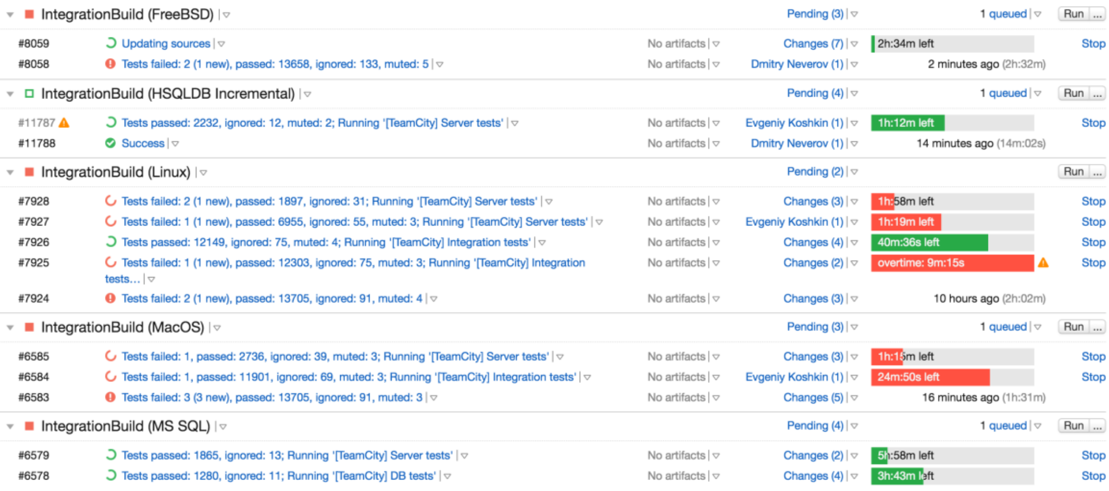
```
$ fly -t lite login -c http://192.168.100.4:8080
$ fly -t lite set-pipeline -p hello-world -c hello.yml
```

Figure 15 Usage example for Concourse's Fly command-line interface tool (19)

Another major feature that is missing from Concourse, is the ability to build version control branches separately from each other – a feature requested in the baseline questionnaire. This, too, might be fixed later but combined with the lack of any reporting capabilities, Concourse was discarded.

TeamCity

The next candidate, Team City – a CI tool from JetBrains – was quickly dismissed as too like the project's original Jenkins tool. Though it promises "powerful continuous integration" (21), it was obvious from the user interface screenshots (Figure 16) on the homepage that the user experience would not be a huge improvement over Jenkins.



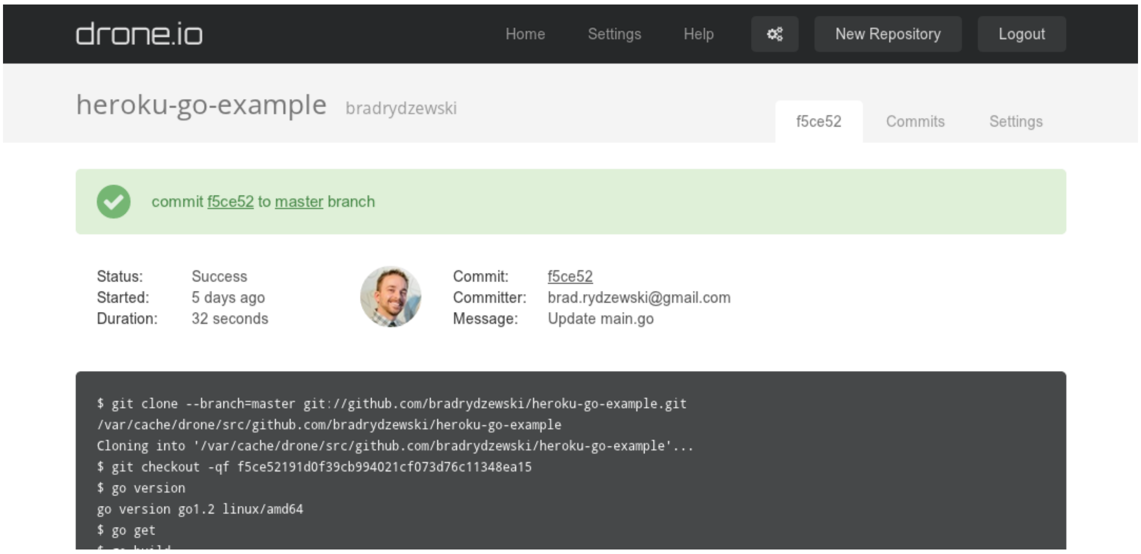
IntegrationBuild (FreeBSD)			Pending (3)	1 queued	Run ...
#8059	Updating sources	No artifacts	Changes (7)	2h:34m left	Stop
#8058	Tests failed: 2 (1 new), passed: 13658, ignored: 133, muted: 5	No artifacts	Dmitry Neverov (1)	2 minutes ago (2h:32m)	
IntegrationBuild (HSQLDB Incremental)			Pending (4)	1 queued	Run ...
#11787 ⚠	Tests passed: 2232, ignored: 12, muted: 2; Running '[TeamCity] Server tests'	No artifacts	Evgeniy Koshkin (1)	1h:12m left	Stop
#11788	✓ Success	No artifacts	Dmitry Neverov (1)	14 minutes ago (14m:02s)	
IntegrationBuild (Linux)			Pending (2)		Run ...
#7928	Tests failed: 2 (1 new), passed: 1897, ignored: 31; Running '[TeamCity] Server tests'	No artifacts	Changes (3)	1h:58m left	Stop
#7927	Tests failed: 1 (1 new), passed: 6955, ignored: 55, muted: 3; Running '[TeamCity] Server tests'	No artifacts	Evgeniy Koshkin (1)	1h:19m left	Stop
#7926	Tests passed: 12149, ignored: 75, muted: 4; Running '[TeamCity] Integration tests'	No artifacts	Changes (4)	40m:36s left	Stop
#7925	Tests failed: 1 (1 new), passed: 12303, ignored: 75, muted: 3; Running '[TeamCity] Integration tests...	No artifacts	Changes (2)	overtime: 9m:15s ⚠	Stop
#7924	Tests failed: 2 (1 new), passed: 13705, ignored: 91, muted: 4	No artifacts	Changes (3)	10 hours ago (2h:02m)	
IntegrationBuild (MacOS)			Pending (3)	1 queued	Run ...
#6585	Tests failed: 1, passed: 2736, ignored: 39, muted: 3; Running '[TeamCity] Server tests'	No artifacts	Changes (3)	1h:15m left	Stop
#6584	Tests failed: 1, passed: 11901, ignored: 69, muted: 3; Running '[TeamCity] Integration tests'	No artifacts	Evgeniy Koshkin (1)	24m:50s left	Stop
#6583	Tests failed: 3 (3 new), passed: 13705, ignored: 91, muted: 3	No artifacts	Changes (5)	16 minutes ago (1h:31m)	
IntegrationBuild (MS SQL)			Pending (4)	1 queued	Run ...
#6579	Tests passed: 1865, ignored: 13; Running '[TeamCity] Server tests'	No artifacts	Changes (2)	5h:58m left	Stop
#6578	Tests passed: 1280, ignored: 11; Running '[TeamCity] DB tests'	No artifacts	Changes (4)	3h:43m left	Stop

Figure 16 TeamCity user interface (21)

As TeamCity was also an equally old product as Jenkins – originally released in 2006 (22) – it carried the baggage of not being built from the ground up with support for containers and the concept of full-blown build pipelines. Another downside of TeamCity, especially considering the educational usage of the results of this thesis, is that it only provides very limited licenses free of charge (23).

Drone

The next candidate was Drone, the open source version of the paid service, Drone.io (24). Some previous team members had recommended it based on their own experiences with it. Drone is built around Docker (24), which means first-class support for container-based build pipelines. It also has a relatively modern looking user interface (Figure 17) but it does also lack reporting features. The lack of reporting features in all tools except Jenkins and TeamCity – the older tools – had become evident at this point.



commit f5ce52 to master branch

Status: Success
Started: 5 days ago
Duration: 32 seconds

Commit: f5ce52
Committer: brad.rydzewski@gmail.com
Message: Update main.go

```
$ git clone --branch=master git://github.com/bradrydzewski/heroku-go-example.git
/var/cache/drone/src/github.com/bradrydzewski/heroku-go-example
Cloning into '/var/cache/drone/src/github.com/bradrydzewski/heroku-go-example'...
$ git checkout -qf f5ce52191d0f39cb994021cf073d76c11348ea15
$ go version
go version go1.2 linux/amd64
$ go get
```

Figure 17 Drone user interface example (24)

One downside of Drone is that it is heavily focused on supporting GitHub and not any Git-based version control service or system. Using Drone might limit future options in regards to version control, which is not ideal. In regards to the missing reporting features, like with other tools, external tools could be used if Drone otherwise proved itself to be a good choice.

However, the modern user interface also lacked in the side of proper visualization of build pipelines, which was a major sticking point. Like Concourse, Drone might be better suited for smaller development teams with less need for advanced reporting features and management-level user interfaces but for the team this thesis intends to help, these features are crucial and Drone, too, was discarded.

GitLab CI

GitLab CI is a part of the Git repository manager, GitLab. GitLab is a similar service and application to GitHub, with optional self-hosting. Though GitLab CI is also available for use with GitLab itself – rendering it useless for this thesis – it was evaluated as a future option due to promising buzz heard within the company. (25)

As shown in Figure 18, it provides clear and simple visualizations for build pipelines, like Jenkins Blue Ocean (8). Like Drone, it is also built around Docker, meaning all stages in its build pipelines are run inside Docker containers, providing the ability to define the custom testing environments the team desired (chapter 2.4). (25)

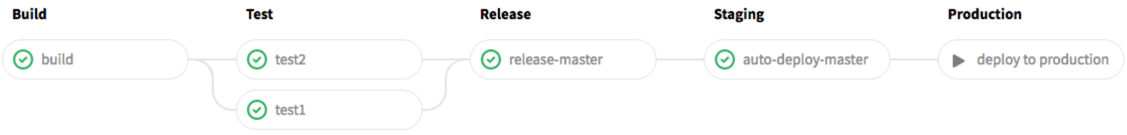


Figure 18 GitLab CI's visualization of a build pipeline (26)

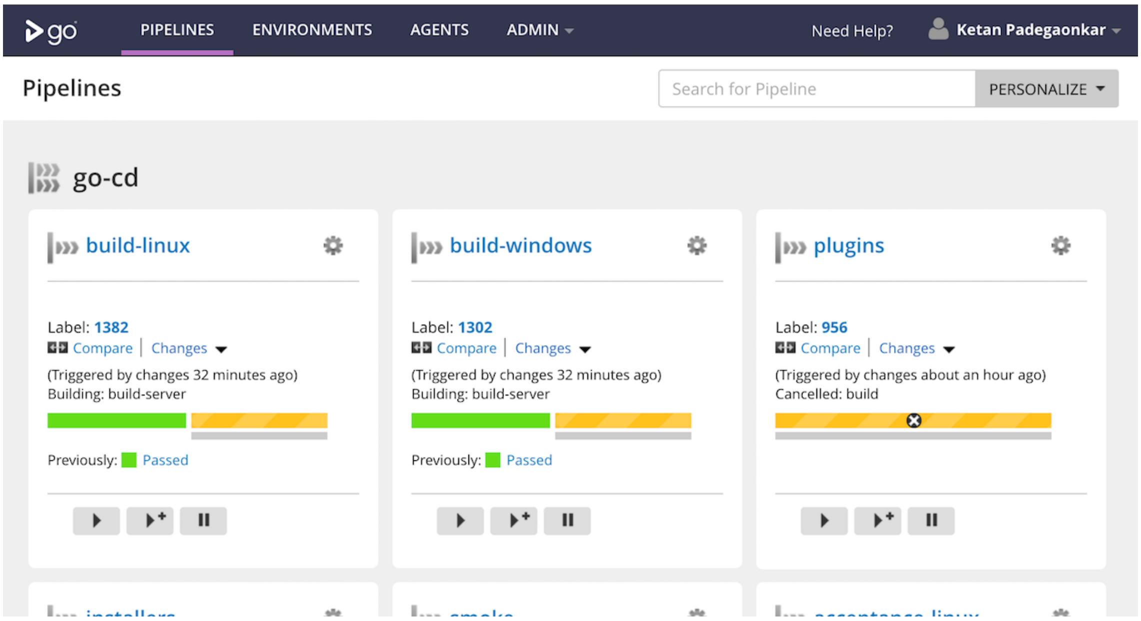
Though GitLab CI currently lack the capability of displaying test coverage and other reports in its own user interface, it provides similar means for publishing HTML-format reports like Jenkins (27). With the first-class support for pipelines, Docker and the ability to show test reports, GitLab CI would have been a perfect candidate. Unfortunately, due to it being limited to the GitLab service, it had to be discarded. For teams that have the ability to use GitLab and do not see themselves as switching away from it quickly, the thesis project team would not hesitate to recommend trying out GitLab CI based on this comparison. Actual performance was not evaluated, though, as a part of this thesis.

GoCD

The last tool to compare against Jenkins was GoCD by ThoughtWorks Inc is replacing the firm’s previous cloud-based continuous integration service, Snap CI (28). As the name suggests and tagline, “Simplify Continuous Delivery”, suggest, this tool’s focus is on the delivery part of CI/CD. (29)

GoCD’s plug-in landscape is similar to Jenkins’ in that it appears to rely heavily on open-source plug-ins, instead of providing the functionality out-of-the-box. One example of this is in the configuration files: GoCD only supports configuration with YAML files via an unofficial plug-in (30). Also, those plug-ins did not appear popular by looking at the number of watchers, stars and changes in one of the plug-in’s GitHub repository (30).

Its UI also appeared highly complex, or at least missing some higher-level visualizations for pipelines, from screenshots and from quick test runs (Figure 19). The version that was tested had, similarly to Jenkins’ classic user interface, deeply nested hierarchies of navigation and no proper dashboard view for quickly glancing the status of builds.



build-linux

Label: 1382

(Triggered by changes 32 minutes ago)
Building: build-server

Previously: 🟩 Passed

build-windows

Label: 1302

(Triggered by changes 32 minutes ago)
Building: build-server

Previously: 🟩 Passed

plugins

Label: 956

(Triggered by changes about an hour ago)
Cancelled: build

Figure 19 Example of GoCD's pipeline user interface (31)

GoCD also did not advertise any kind of support for Docker but some unofficial plug-ins were available for this too. The thesis project team saw the plug-in architecture as too fragile in all the same ways as Jenkins': it relies heavily on unofficial plug-ins and grants them a lot of control over the whole system, leading to a fragmented ecosystem. Overall, GoCD did not provide enough compelling reasons to warrant a change from a known system.

Jenkins

Jenkins (official logo show in Figure 20) is an automation server that the project is currently using as its continuous integration and delivery tool. As shown in chapter 2.4, though, Jenkins' classic user interface and configuration style has proven to be difficult and quite error prone for developers. Going into this comparison, the main benefit for Jenkins, still, was its popularity within the company and in the industry in general. A survey conducted by CloudBees shows that over 90 % of answerers considered Jenkins "to be mission-critical to the development and/or delivery process" (32, p. 10). It also holds the top position with 70 % among other tools on the market in 2014 (33).

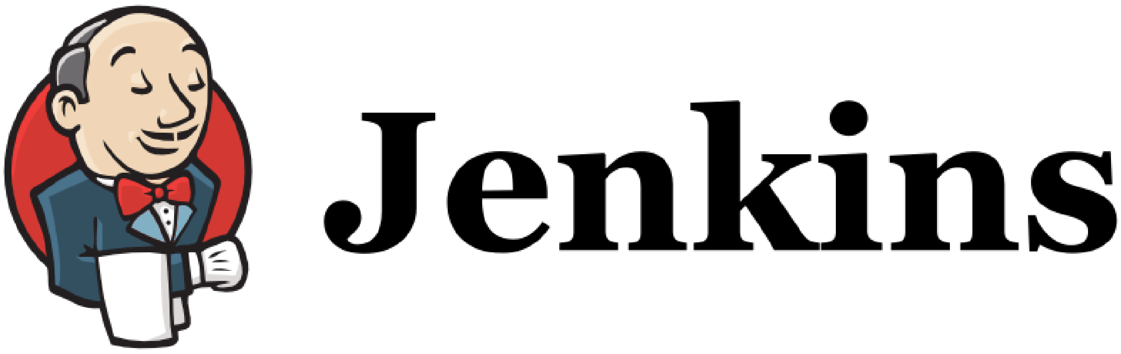


Figure 20 Official logo of Jenkins (34)

Also of interest for this thesis was that the aforementioned CloudBees survey showed that of the people who said the practiced continuous delivery, over 50 % defined their delivery pipelines as code. The thesis project team later selected this method to use in the implementation of build pipelines. It is known as Jenkins Pipeline and is further discussed in chapter 5.1.

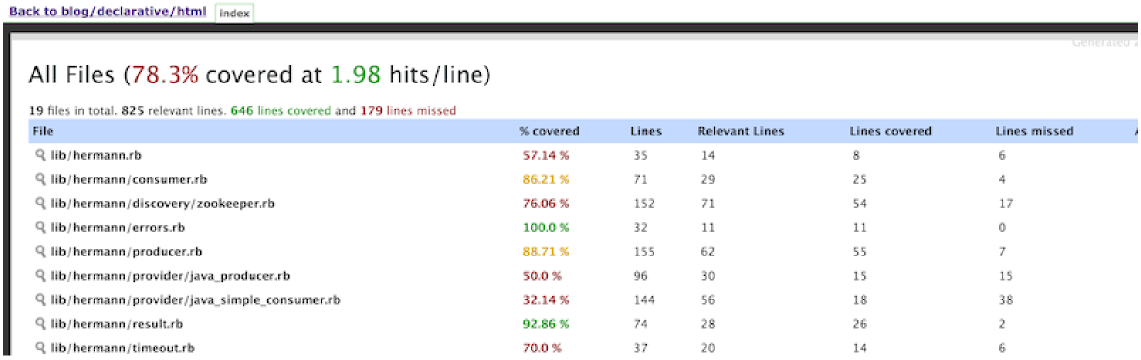
As for the actual results of the comparison, Jenkins did not fair greatly: its level of documentation is heavily lacking in content and information; for some parts of the configuration of pipelines, no documentation is available and for some its only source code level documentation, meaning it is mainly helpful for developers already familiar with the source code.

Jenkins also did not start out with a concept of build pipelines, so some parts still feel tacked on: one of the most common things to see in a continuous delivery system is a visualization of the whole pipeline, and this is hidden behind multiple clicks in the user interface. The usability is slightly remedied, though, by the new Blue Ocean Project which promises to “rethink the user experience of Jenkins” and is “designed from the ground up for Jenkins Pipeline” (8).

The project team also felt that the plug-in architecture of Jenkins hindered its usability; by relying on plug-ins for core functionality and allowing them to create their own user interfaces, Jenkins does not seem like a cohesive system, at least for those not familiar with the system as shown by the baseline questionnaire in chapter 2.4.

Docker support is still new in Jenkins: for example, the plugin “Docker Pipeline” used in this thesis was first released in 2015 (35). It was also initially limited to the paid version of Jenkins, called CloudBees Jenkins (36). In comparison, the original release year of Hudson, the project behind Jenkins, was 2005 (37). The state of plugins and Docker is further discussed in chapter 6.

One major feature Jenkins has going for it, at least compared to the other tools in this comparison, is its ability to generate, display and send out all kinds of reports (38). For example, builds can generate HTML formatted reports of test coverage, as shown in Figure 21, which can then be linked to those builds in Jenkins. As shown in the baseline questionnaire results in chapter 2.4, viewing test coverage reports is a heavily requested feature for the current project team. The benefit of having this reporting capability in the same tool that manages the build pipelines is that it reduces the number of new tools to learn. Jenkins can also use external tools, such as SonarQube (39). This, though, cannot be counted against other tools, as most external tools use simple network interfaces to communicate with the continuous integration tools.



All Files (78.3% covered at 1.98 hits/line)

19 files in total. 825 relevant lines. 646 lines covered and 179 lines missed

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed
lib/hermann.rb	57.14 %	35	14	8	6
lib/hermann/consumer.rb	86.21 %	71	29	25	4
lib/hermann/discovery/zookeeper.rb	76.06 %	152	71	54	17
lib/hermann/errors.rb	100.0 %	32	11	11	0
lib/hermann/producer.rb	88.71 %	155	62	55	7
lib/hermann/provider/java_producer.rb	50.0 %	96	30	15	15
lib/hermann/provider/java_simple_consumer.rb	32.14 %	144	56	18	38
lib/hermann/result.rb	92.86 %	74	28	26	2
lib/hermann/timeout.rb	70.0 %	37	20	14	6

Figure 1. Figure 21 Test coverage report published as a HTML report from Jenkins (38)

While Jenkins' lack of history with build pipelines is damning, its ability to execute nearly any kind of automated tasks, not just build-related tasks, was seen by the project team as hugely positive. The development team is currently using Jenkins to monitor the status of some infrastructure services in their development and production environments. Though these kinds of tasks could be constructed as a build pipelines in other applications, they are conceptually very different and Jenkins already has built-in capability and plug-ins for visualizing them separately from build pipelines.

Although the results of this comparison do not show Jenkins in great light, it was, however, seen by the thesis project team as the most viable option amongst the compared tools. The main factor in this decision was the level of familiarity within the company but also the reporting capabilities and the support for non-build related actions. Other tools could be considered for smaller scale operations where, for example, the reporting functionality is not as necessary, as some of them appeared to do their specific focuses very well.

Overall, for the use case of this thesis, Jenkins was still the best choice in the opinion of the thesis project team but as the time for this comparison was limited, exploring external tools for reporting and pipeline visualization was not included. Later, it might be useful to revisit those product categories and either compliment Jenkins with them or switch CI tools but for now, no tool gave compelling enough reasons to warrant a large-scale switch.

4 Container technologies

4.1 Overview

Software containers are essentially a level of virtualization that operate on the operating system level, whereas more traditional virtualization technologies operate on the physical hardware. Their purpose is to be able to execute code in a relatively isolated environment on a single or multiple host computers. Software container are not in themselves a completely new technology (40, p. 7) but it has recently picked up more speed, especially with the introduction of Docker (3,41). One indicator of Docker's popularity are the Google search trends for Docker compared to other, older container technologies, like chroot and LXC, as seen in Figure 22. (40, pp. 7-8,42)

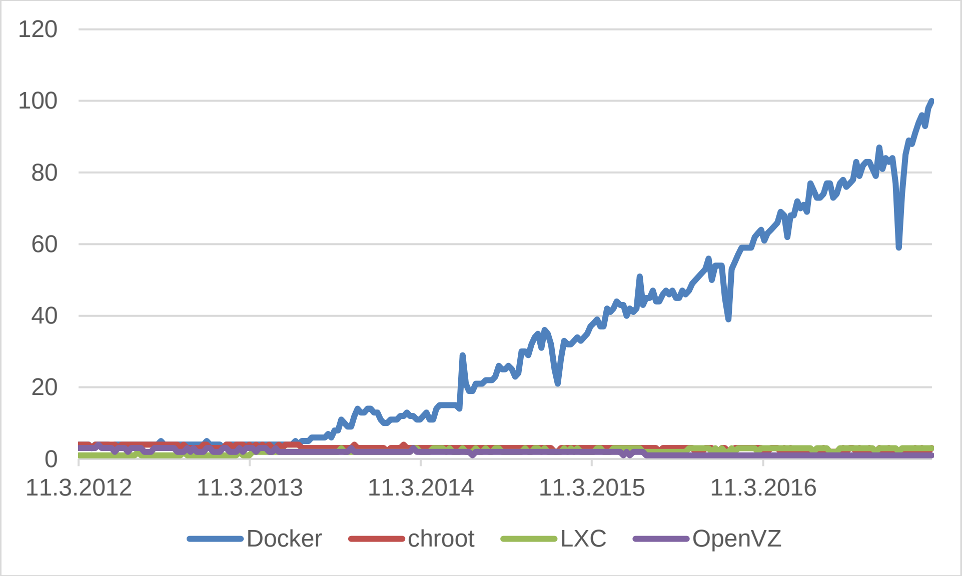


Figure 22 Worldwide Google Trends by search topic from 2012 to 2017 for different container technologies (42)

As container technologies rely on an operating system's kernel – the core of the operating system, supporting for example resource allocation and security functions – they can require a smaller set of resources compared to a full-blown operating system (Figure 23). Though, relying on the operating system's kernel also limits them to virtualizing other software that is built to work on that operating system's kernel; only operating systems and software built for the Linux kernel can operate in a container on a machine using that Linux kernel and vice versa for the Microsoft Windows kernel. Containers also do not provide the same level of isolation as so-called hypervisor virtualization technologies that operate directly on the physical hardware, but container technologies usually provide their own methods of security and isolation. (40, pp. 7-8)

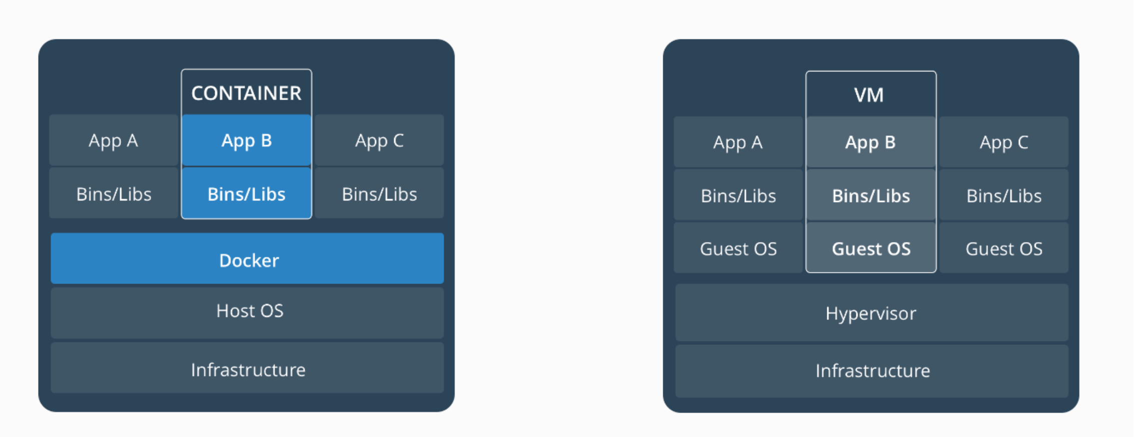


Figure 23 Architectural view of Docker containers vs virtual machines (43)

Probably two of the most attractive sides of container virtualization are its leanness when it comes to resources and its ability to be described in source code or a text document, such as Dockerfiles (44) for Docker containers. They also provide an improved separation of concerns where developers can focus on building their software on top of containers and operations personnel is mainly responsible for providing a stable environment for running those containers (40, p. 10). As containers often contain their own operating systems working on top of the host systems kernel, the developers can configure their systems in relative isolation in relation to other containers running on the host system. This enables the use of wildly conflicting configurations on a single host machine.

Although most of the functions described above and generally possible with containers are not unattainable of even difficult with more traditional methods of virtualization, containers provide a clear benefit to the fast development and testing loop, that modern software development is built on, with their lightweight nature and easy reproducibility due to their inherent configurability. That lightweight nature enables them to be used even in continuous integration and delivery environments (40, pp. 16, 185) where fast build times and quick teardowns are key for providing a fast feedback loop for the developers and software development teams.

The existing system this thesis was built on was already using Docker and it was seen by the project team as the currently most popular container technology, so Docker was chosen as the container technology for this thesis’ continuous integration needs. Most continuous integration and delivery tools also focused their container support on Docker (45) or had additional software to support it (36), so it was an obvious choice for this thesis project.

4.2 Docker container software

Docker (official logo shown in Figure 24) can be considered as a platform but for the purposes of this thesis, the focus will be on its container technologies.

Figure 24 Official Docker logo (46)

Docker container software has four major components: the Docker client and server, Docker images, Docker image registries and containers (40, pp. 11-12). Docker server handles the containers and the client communicates with that server – either locally on the same host machine or remotely via network communications. Images are the basis for Docker containers, and as James Turnbull puts it: “You can consider images to be the ‘source code’ for your containers” (40, p. 13). Containers are the actual instances of those images running through a Docker server. (40, pp. 11-14)

Before late 2016, Docker could only be run on x64 hosts with a modern Linux kernel (40, p. 18), but recently Docker for Windows Server was announced (47) enabling developers to create Docker containers for the Windows kernel. Currently, the project this thesis work was done for, does not have a need for that but it speaks for the versatility of Docker technology moving forward.

4.3 Docker images and containers

Docker images

Docker images are like virtual machine images in that they contain all the data files and configurations required for running the systems they represent. They are stored as binary images that can be easily redistributed via Docker image registries. They are made up of filesystems layered on top of each other (Figure 25) not totally unlike some version control systems work. Layers form the base for a container's root filesystem (48). Where Docker images differ from virtual machine images is that they rely on the host machines' kernels, meaning that they cannot be run without that kernel. (49, pp. 3-4,40, pp. 77-80)

The first layer in a Docker image is called a boot filesystem or "`bootfs`" (40, p. 78). Most development tasks with containers will never interact with this low-level layer but are instead built on top of existing operating system Docker images that do the heavy-lifting. By sharing the kernel with the host machine, these images can shed a bulk of the low-level portions of operating systems such as resource allocation operations and task scheduling. (40, p. 78,49, p. 4)

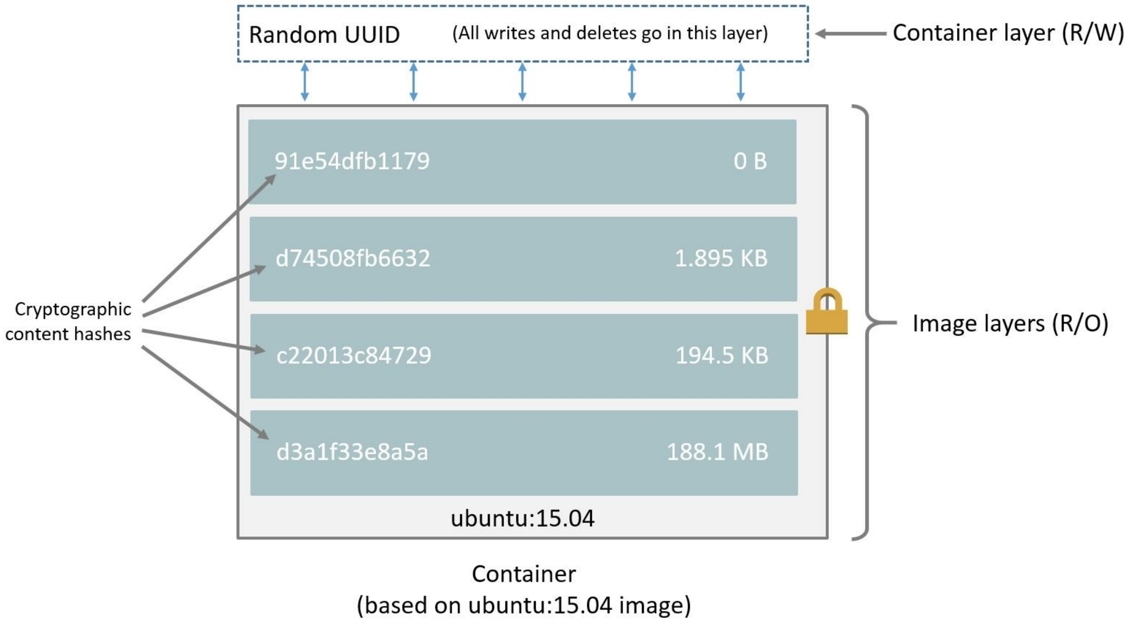


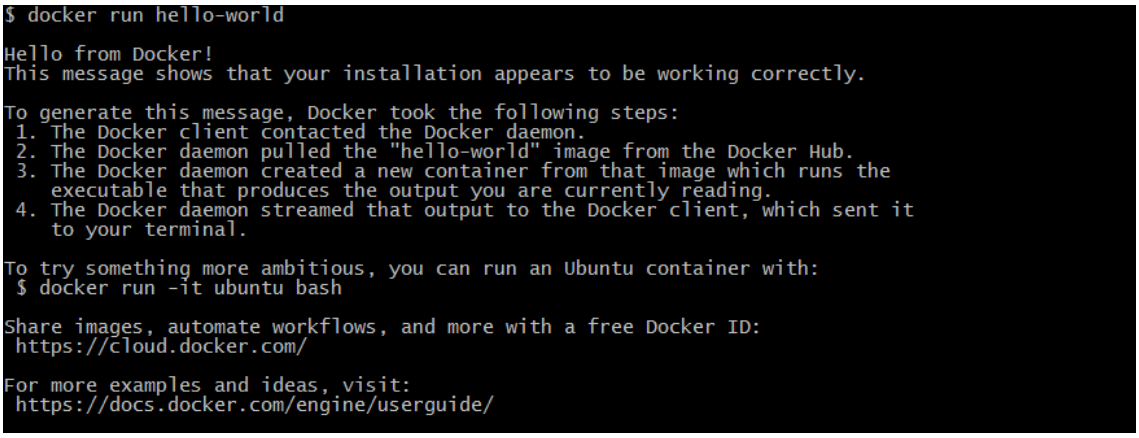
Figure 25 Diagram of a Docker container based on the Ubuntu 15.04 image (48)

Docker images are also packaged in a way that they should run identically in any environment with the same kernel version. Of course, any dependencies towards the host system from the image – such as requiring specific files to exist on the host – can make images' behavior differ. (49, p. 6)

Docker containers

Docker containers are instances of Docker images running on a server. Whereas images are part of the building and packaging phase of software, containers are the result of deploying that software into an environment. (40, p. 14) Images are essentially blueprints for containers; an image can be used infinitely to create new containers.

Containers are launched from Docker images (Figure 26) and have one or more processes running in them. Containers can also be in a stopped state where no processes are running but any filesystem changes made within them are persisted until they are destroyed. The operations inside a container are described both in the image and in the file contents inserted into the container. (40, pp. 14, 110-111)



```
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

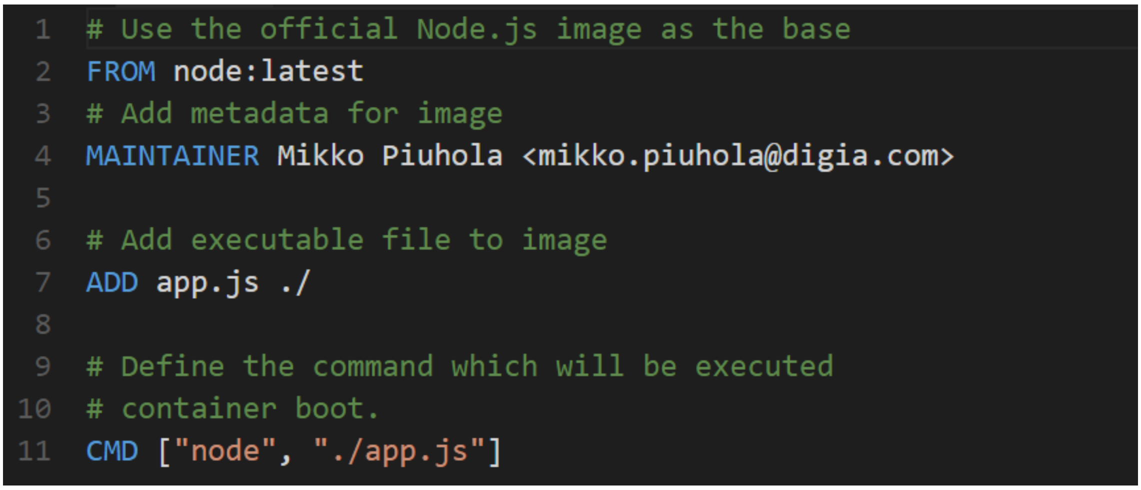
Figure 26 Creating and executing a Docker container from the image named “hello-world”

As such, nothing prevents running Docker containers inside Docker containers, as any kernel-level operations would simply be passed through the first layer. In reality, though, security features of Docker, namely SELinux and AppArmor, can get misconfigured. Some of the filesystems Docker uses are also not designed to be run this way and can result in write and read errors. (50)

4.4 Dockerfile

Dockerfile (44) is a text document format that is used to create and build Docker images. It uses a domain-specific language (51) which contains a limited set of possible operations construct images (44). Docker themselves describe Dockerfile as being a “recipe which describes the files, environment and commands that make up and image” (52).

Figure 27 shows a simplistic Dockerfile for an application built on top an official Docker image for the Node.js JavaScript environment (53). The command “FROM” is used to define the image this Dockerfile and se subsequent image will be based on – the first layer of this image. Next, the “MAINTAINER” command is used to define a common metadata field: who is the maintainer of this Dockerfile. Dockerfile also has the command “LABEL” for setting more freeform metadata. (44)



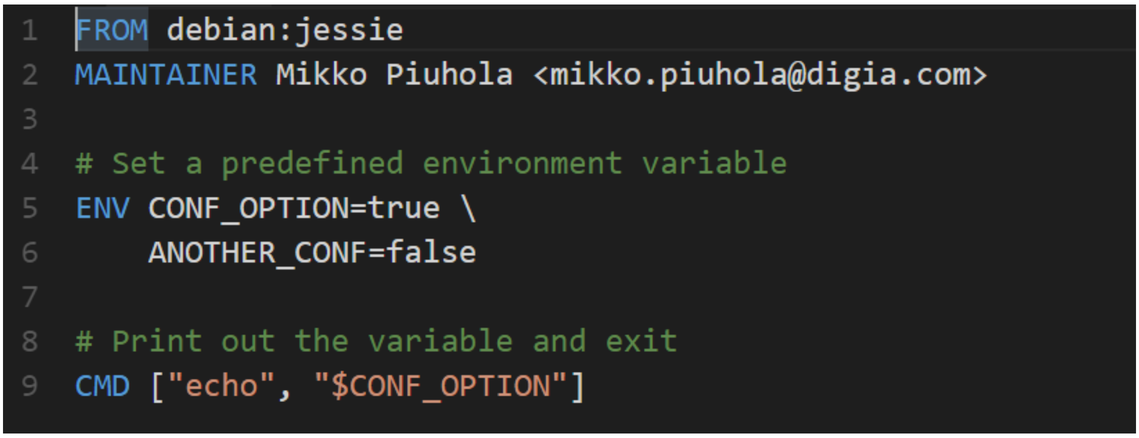
```
# Use the official Node.js image as the base
FROM node:latest
# Add metadata for image
MAINTAINER Mikko Piuhola <mikko.piuhola@digia.com>

# Add executable file to image
ADD app.js ./

# Define the command which will be executed
# container boot.
CMD ["node", "./app.js"]
```

Figure 27 Dockerfile example for running a simple Node.js application

The command “ADD” is used to, as the name implies, add files into the image. This creates a new filesystem layer in the image. Most commands create new filesystem layers on the generated images. Reducing the number of layers in a Docker image is a common optimization tactic (54). Creating multiple layers can result in inefficiency for the image’s filesystem. (44)

Finally, “CMD”, or sometimes “ENTRYPOINT”, is used to define the command to be executed when a container is started from this image. In the example shown in Figure 27 the command will start a Node.js process with the added “app.js” file. (44)

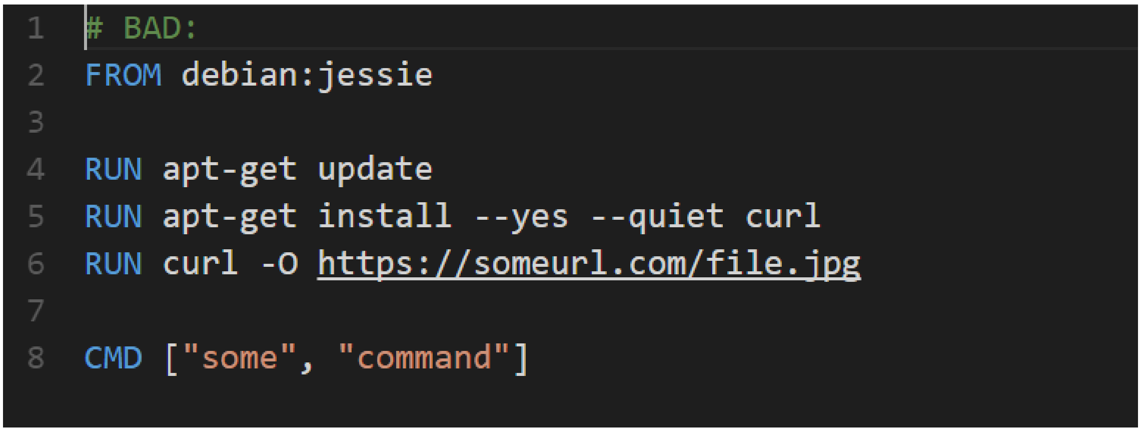
Other common commands are: “EXPOSE” and “ENV”. EXPOSE is a metadata instruction to inform Docker that the generated container will listen on the given network port or ports. ENV, as the name suggests, defines an environment variable (Figure 28). Environment variables are effectively public key-value-pairs. Docker images often use them as their configuration method: for example, the official MySQL Docker image has an environment variable “MYSQL_ROOT_PASSWORD” to set a password for the database’s root user (55).



```
1  FROM debian:jessie
2  MAINTAINER Mikko Piuhola <mikko.piuhola@digia.com>
3
4  # Set a predefined environment variable
5  ENV CONF_OPTION=true \
6      ANOTHER_CONF=false
7
8  # Print out the variable and exit
9  CMD ["echo", "$CONF_OPTION"]
```

Figure 28 Dockerfile containing the ENV command and printing its value out

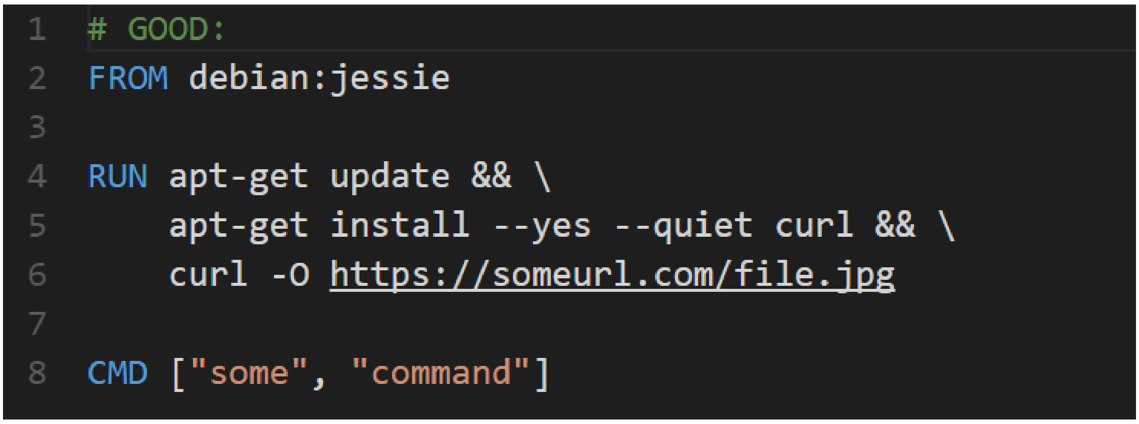
As mentioned earlier, reducing the number of layers created by a Dockerfile is a common optimization. For the ENV command, this can be done as shown on Figure 28’s rows five and six, where an equal sign is used to allow multiple environment variables to be defined in a single ENV command (44). The backwards slash on row five is an escape character that allows writing the command on multiple rows. This best practice is often ignored with the “RUN” command, which executes the command or set of commands it is given. The harmful way of writing RUN commands is shown on Figure 29, where rows four through six would all create their own filesystem layers for unnecessary cruft in the resulting image.



```
# BAD:
FROM debian:jessie

RUN apt-get update
RUN apt-get install --yes --quiet curl
RUN curl -O https://someurl.com/file.jpg

CMD ["some", "command"]
```

Figure 29 Dockerfile example that will create extra layers due to missing layer optimizations

Instead of separating each command to its own RUN block, this same Dockerfile should be written as shown in Figure 30. There the RUN blocks' contents are merged using the common “&&” symbol, meaning that the next command will be executed if the previous was executed successfully.



```
# GOOD:
FROM debian:jessie

RUN apt-get update && \
    apt-get install --yes --quiet curl && \
    curl -O https://someurl.com/file.jpg

CMD ["some", "command"]
```

Figure 30 Dockerfile example following best practices for layer optimization

Although the result of merging multiple RUN commands can sometimes get crowded and hard to read, it has clear benefits for the optimization of storage usage. In some cases, though, it is wise to not merge some commands: if it is known that the resulting layers from a set of commands will never change after they've been created and that those layers might be reused in some other image, it can be useful to keep them as a separate unit from the next layers which are more likely to change. This is often quite futile, though, as most layers are not highly reusable between different images, unless they are used to form an actual image. (44)

5 Software build pipelines

Build pipelines formalize the concept of developing and deploying software in stages. They describe groups of stages that need to be completed for a version of software to be considered built. Usually, they are executed in order and later stages are only executed when previous stages finished successfully. A single stage can contain multiple smaller tasks, and those can consist of multiple phases. Great build pipeline tools also provide high-level visualizations of the processes that can be understood at a glance (Figure 31).

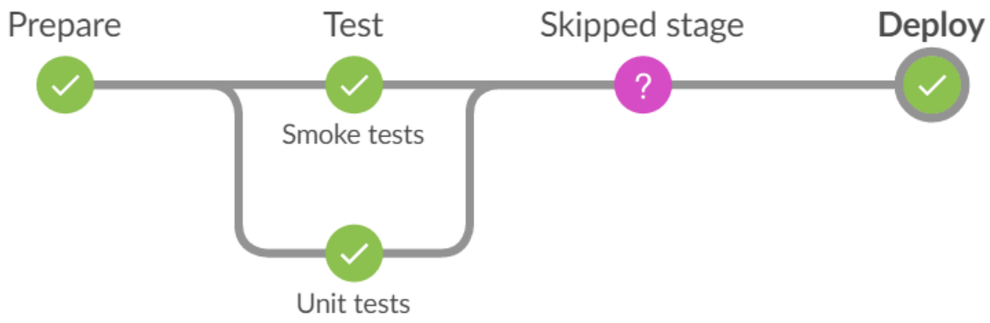


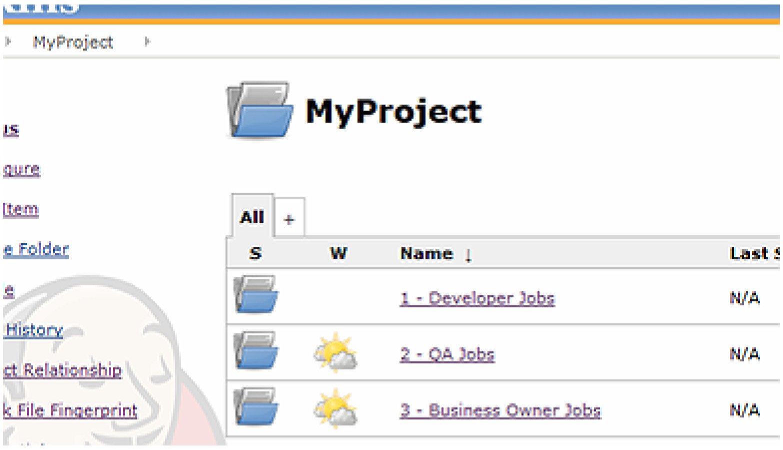
Figure 31 Build pipeline visualization for one of the example build pipelines

Like manufacturing pipelines, the most common stages in a software build pipeline are: building the software, testing it and deploying it to a particular environment. In some cases, testing can contain different kinds of testing separated into their own stages, and deployments can be done to multiple environments with even more testing in between.

The selected continuous integration tool, Jenkins CI, provides a few previously unavailable methods for creating and managing software build pipelines. Some, like Jenkins Pipeline, relate more to the actual configuration of pipelines and some, like Blue Ocean, focus on the management and user experience side of the equation.

5.1 Describing continuous integration tasks queues with Jenkins Pipeline

Pipeline as Code, or Jenkins Pipeline as it is commonly called by plug-ins and tutorials, is a method of describing and running build pipelines in Jenkins (56). By using a configuration file called Jenkinsfile, it allows users to store versioned descriptions of their build pipelines in the version control system of choice. Previously, creating pipelines in Jenkins was done more conceptually than concretely; jobs – roughly equal to stages in a build pipeline – would be configured separately from each other and later linked together by various methods, including a simple folder structure provided by the CloudBees Folder Plugin (57,58).



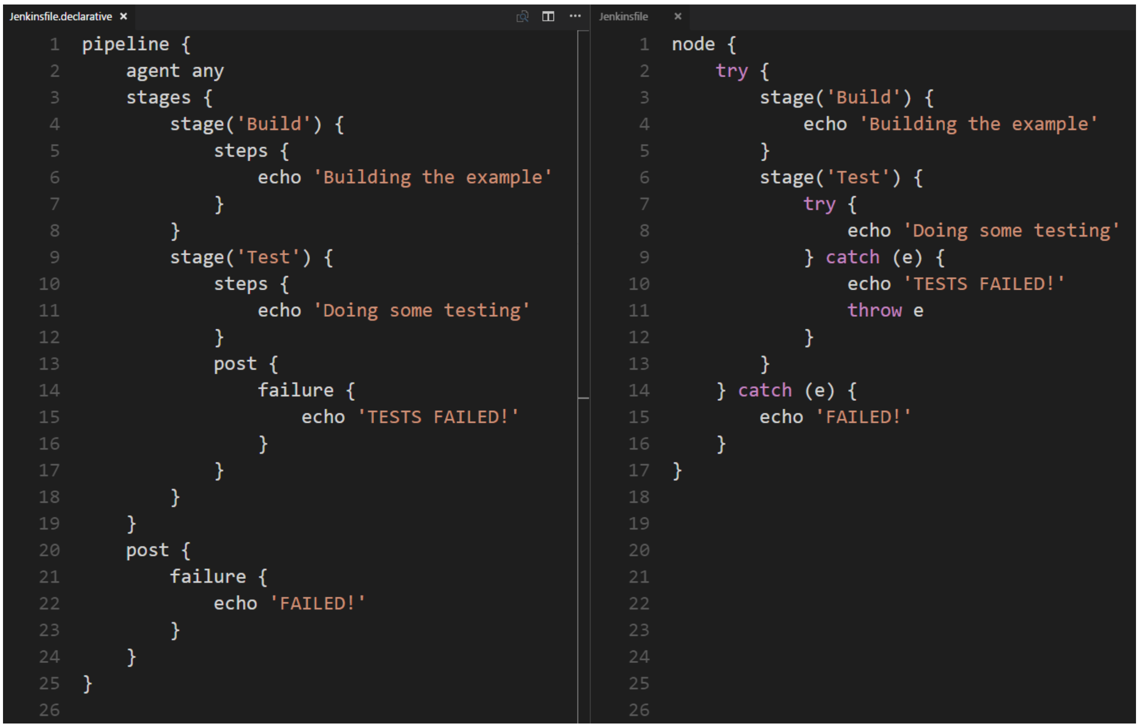
S	W	Name ↓	Last S
		1 - Developer Jobs	N/A
		2 - QA Jobs	N/A
		3 - Business Owner Jobs	N/A

Figure 32 The CloudBees Folders Plugin's view for a pipeline (58)

Jenkins Pipelines require a Jenkinsfile to be present in the source code repository as that is what is used to configure the build pipelines. The inclusion of proper version control to build pipelines is a huge boon for development teams in general for the transparency it provides and especially for larger teams that have review processes in place for applications, who can now review the concrete build pipelines too. Beyond the benefits of code review, version control also simplifies rolling back changes in the build configurations in case errors are made – which are inevitable as humans are still required to write at least some portions of those configurations.

Jenkinsfiles can be written in two formats: in an older scripted style written in the Groovy programming language (56) or in a new declarative style, in more configuration-like syntax. The declarative syntax was officially released as a beta version during the work phase of this thesis, in December 2016 (59). Pipeline as code should also help with the difficulty of configuring Jenkins jobs – shown to be an issue in the baseline questionnaire – as code is more likely to have a consistent user interface than the fully custom graphical user interfaces created by plug-in creators for the Jenkins web interface, which were a core part of the confusion around build configuration.

As anyone familiar with comparing scripted and declarative style configurations can guess, the declarative syntax is much simpler but also less expressive due to its inherent nature of being an abstraction over the scripted style. Figure 33 shows a comparison of a simplified build pipeline with two stages – build and test –, an error handler for the test stage and a separate error handler for the whole pipeline. In this example, the scripted style is more succinct but the difference to the declarative style on the left is quite clear: catching errors needs to be done with the Groovy try-catch block, instead of simply stating that something should be done if the current stage fails. This is also more powerful, though, as the decision is left to the user to decide whether that failure should bubble up and fail the whole build – requiring the rethrow of the exception as shown on row 11 – or if it some known error case that should be logged and suppressed.



```
 1  pipeline {
 2      agent any
 3      stages {
 4          stage('Build') {
 5              steps {
 6                  echo 'Building the example'
 7              }
 8          }
 9          stage('Test') {
10              steps {
11                  echo 'Doing some testing'
12              }
13              post {
14                  failure {
15                      echo 'TESTS FAILED!'
16                  }
17              }
18          }
19      }
20      post {
21          failure {
22              echo 'FAILED!'
23          }
24      }
25  }
26
```

```
 1  node {
 2      try {
 3          stage('Build') {
 4              echo 'Building the example'
 5          }
 6          stage('Test') {
 7              try {
 8                  echo 'Doing some testing'
 9              } catch (e) {
10                  echo 'TESTS FAILED!'
11                  throw e
12              }
13          }
14      } catch (e) {
15          echo 'FAILED!'
16      }
17  }
18
19
20
21
22
23
24
25
26
```

Figure 33 Same Jenkins Pipeline configuration written in the declarative style (left) and the scripted style (right)

What the declarative syntax does provide, though, is a syntax that most users can understand at least roughly without knowing any Groovy or code structures. It also allows the use of so-called script-blocks, where most of the actions available in the scripted syntax can be used. It should be noted, however, that the declarative pipeline syntax had its first non-preview release in February 2017 (60) so it is still quite a new syntax and plug-in. For example, the plug-in did not initially provide all the necessary options for creating fully custom build environments using Docker and its usage for scrapped for this thesis after some experimentation. After the work for this thesis was completed, though, it already had made some of those features available and the syntax should be revisited later on.

Jenkins pipelines can also use a feature called shared libraries (61) which enable developers to share common parts of their build pipelines. Developers can even create standardized builds that only require a few parameters like shown later in chapter 6, effectively defining their own versions of the declarative pipeline syntax – for a better or worse.

The ability to define build pipelines as code is a huge improvement, in the opinion of the thesis project team, as previously recreating build configurations in Jenkins relied on having backups of XML files that are not easily understood at a glance and especially not meant to be hand-configured. When pipelines are defined in Jenkinsfiles residing in the same source code repositories as the applications in development, recreating them from scratch is simply a process of importing the Jenkinsfile in Jenkins or pointing the automation system created in this thesis at the Git repository. This means that even in the event of a catastrophic loss, the CI system should be easy to bring back up online. To alleviate the pain of reconfiguring Jenkins using the web interface in this case, not just the configuration of pipelines but Jenkins itself should be done automatically with scripts.

5.2 Script-based configuration of Jenkins

Using Jenkinsfiles to automatically configure build pipelines for Jenkins is a great step towards automating the CI system but it still has one flaw: someone will need to manually configure Jenkins and document that configuration somewhere, or at the very least setup a backup system for the configuration files. Fortunately, though, Jenkins provides a method of configuring itself using scripts written in the Groovy programming language

(62). These scripts are called hook scripts and will be automatically executed by Jenkins upon startup when placed in specific locations. A common example is to configure the URL where Jenkins should be available (Code example 1).

```
// Configure Jenkins’ external URL
if (jlc.getUrl() != env['CONF_JENKINS_URL']) {
  println "--> setting Jenkins url to ${env['CONF_JENKINS_URL']}"
  jlc.setUrl(env['CONF_JENKINS_URL'])
  jlc.save()
}
```

Code example 1. Groovy script excerpt to configure Jenkins’ external URL

Using these hook scripts is the recommended method for preconfiguring Jenkins. Unfortunately, as with most plug-ins and code-based adjustments to Jenkins, their documentation is lacking in quantity and quality; most APIs are only documented at the code method level – without any descriptions on when and why to use them – and some are not documented at all, and users need to go sift through plug-ins’ source code. Most configurations done at the Jenkins instance level are, fortunately, through official APIs which are more likely to be better documented than non-official plug-in APIs.

The next obvious step from configuring general Jenkins settings is to automate the generation of Jenkins jobs. Even though Jenkinsfiles are used to configure what exactly happens in the build pipelines, jobs need to be configured in Jenkins to fetch those repositories containing Jenkinsfiles. The Job DSL Plugin (63) provides a scripting language to configure any type of Jenkins job (Figure 34).

```
def project = 'quidryan/aws-sdk-test'
def branchApi = new URL("https://api.github.com/repos/${project}/branches")
def branches = new groovy.json.JsonSlurper().parse(branchApi.newReader())
branches.each {
    def branchName = it.name
    def jobName = "${project}-${branchName}".replaceAll('/','-')
    job(jobName) {
        scm {
            git("git://github.com/${project}.git", branchName)
        }
        steps {
            maven("test -Dproject.name=${project}/${branchName}")
        }
    }
}
```

Figure 34 Simple Groovy script to configure a Jenkins job with a single step (63)

The plug-in can then be used through the hook scripts to automatically generate a new job immediately when the CI system is first started up. This removes yet another manual step from the equation. Beyond creating the initial scripts to configure Jenkins, the jobs to fetch Git repositories and the Jenkinsfiles, a system using these features should be mostly automated: Jenkins will configure itself using the hook scripts and create jobs to fetch and update Git repositories with Jenkinsfiles that automatically configure their own build pipelines.

6 Implementation of the automated build pipeline

After the initial questionnaire (chapter 2.4) was completed in October 2016, a high-level plan for the implementation was formed. The plan was to create an example environment with the automated CI system and one or more example applications with build their build pipelines. The CI system would connect to a public GitHub organization – GitHub's method of grouping Git repositories together logically – as no actual content should be used for the thesis due the project's high-security classification. The CI should automatically scan all desired repositories in the GitHub organization and have some method of separating build pipelines by software and by their version control branches.

The thesis team also created a public GitHub organization for the example applications and libraries (64). This organization acts as a mockup of the existing GitHub Enterprise organizations the project uses. The example application has some unit tests that produce

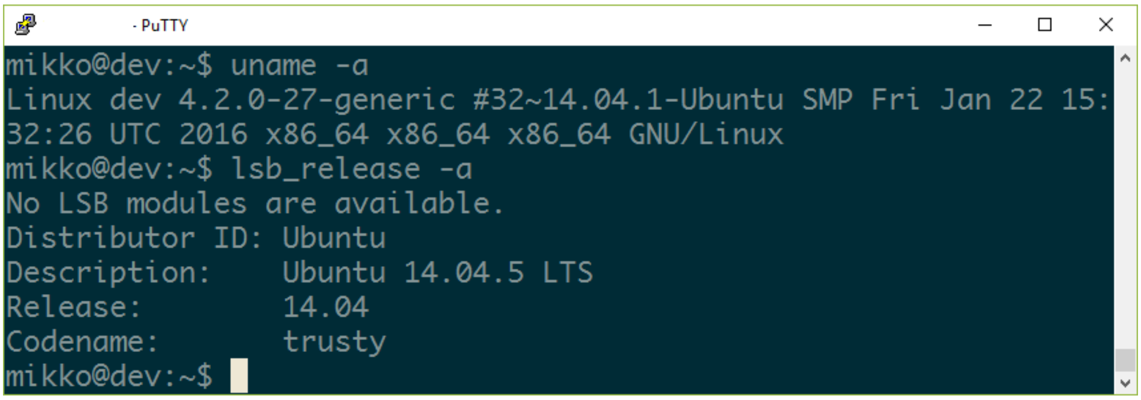
JUnit test success reports, and HTML-format and Cobertura-format test coverage reports. Its structure was based on existing builds for the project's applications with publicly available tools using the Node.js platform. For this thesis' purposes, the contents of the applications and how they are built is irrelevant as long as they produce the supported formats of test reports and have some steps that use Docker, to test its viability for creating custom build environments.

All source code required for creating the automated CI system and the example build pipelines shown here are available in the appendices and public GitHub repositories (64,65).

Environment setup

The first step was to create an easily replicable environment for the CI system. The thesis project team decided that the environment would consist of two virtual servers: one for the CI system and for deploying to "development", and another to host a simple Docker registry and to act as the "production" environment. These virtual servers were provided by Digia to the thesis project team. Both servers would only have Docker and Docker containers running in them as the CI system was to be built as a Docker container and all deployments were to be done using Docker.

To get a fast start to the implementation, the thesis project team chose Ubuntu 14.04 LTS (Figure 35) as the operating system as it was known to have decent default settings for Docker – at least compared to operating systems like CentOS and RedHat Enterprise Linux which prefer using a more issue-ridden storage driver for Docker, called devicemapper (66). Storage driver is what handles Docker's read and write operations and there are multiple to choose from based on kernel support (67). The devicemapper storage driver had been proven itself to have some stability issues in the project's existing environments. Both servers were ordered with identical configurations on the operating system level.

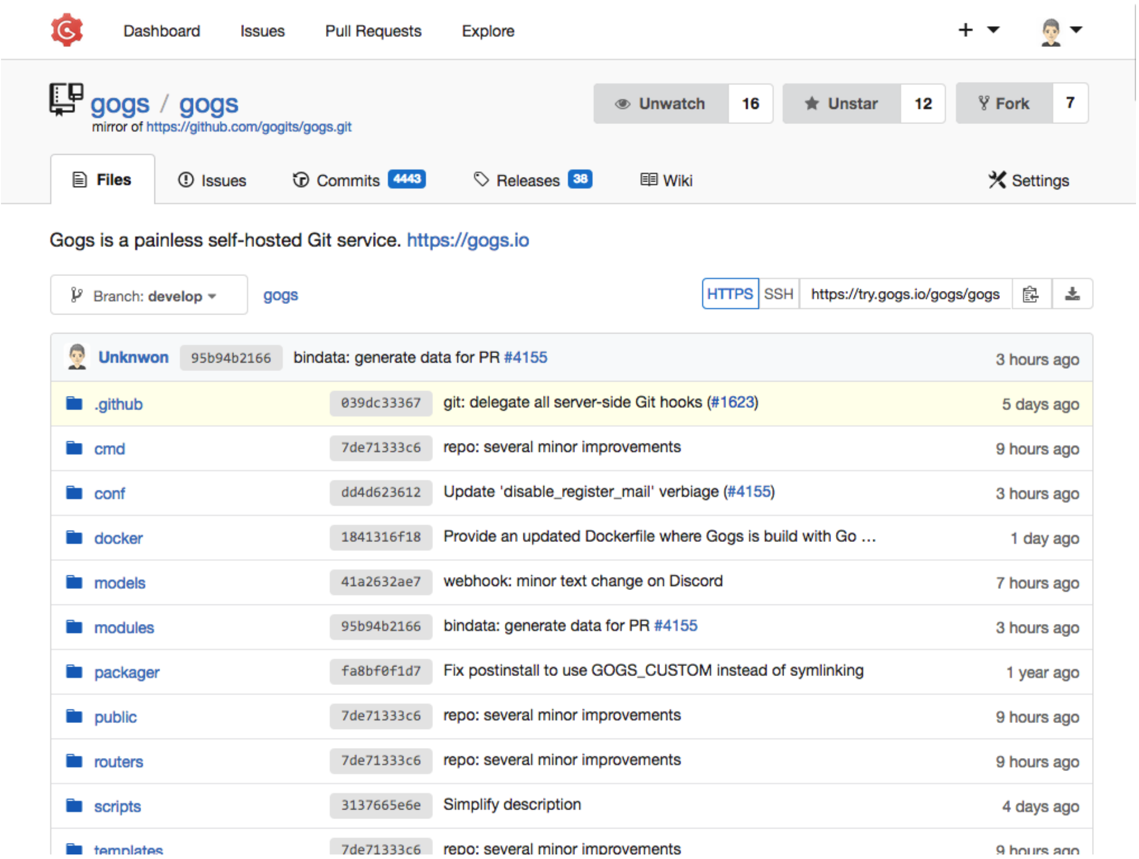


```
mikko@dev:~$ uname -a
Linux dev 4.2.0-27-generic #32~14.04.1-Ubuntu SMP Fri Jan 22 15:
32:26 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
mikko@dev:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 14.04.5 LTS
Release:        14.04
Codename:       trusty
mikko@dev:~$
```

Figure 35 Operating system and kernel version of one of the example servers

As Jenkins CI requires a fair amount of memory (68), the CI server was configured with 8 GB of RAM and with 5.8 GB of usable hard drive space. Hard drive space would not be an issue as the CI server would host just a few build pipelines and get recreated from scratch constantly during the iterative development. The other server would host a simple Docker registry and a small web server, both which do not require much memory or storage space, so it was configured with an identical hard drive and 1 GB of RAM. Both servers had identical two core processors as the workloads would not be very high on either of the servers.

Initially, the thesis project thought about using a self-hosted service called Gogs (69), which attempts to be an API compatible clone of GitHub. It is also visually quite similar to GitHub (Figure 36). The idea was entertained as it would allow simulating the real project’s real situation where the GitHub Enteprise service is not accessible from the internet. This was quickly abandoned though, as Jenkins only had a single plug-in that would support interacting with Gogs’ equivalent to GitHub’s organizations and it is still, as of the time of writing, only had an early preview version released (70). The equivalent Jenkins plug-ins for GitHub and GitHub Enterprise were seen by the thesis project team as more stable and ready for production use.



Gogs is a painless self-hosted Git service. https://gogs.io

Unknwon	95b94b2166	bindata: generate data for PR #4155	3 hours ago
.github	039dc33367	git: delegate all server-side Git hooks (#1623)	5 days ago
cmd	7de71333c6	repo: several minor improvements	9 hours ago
conf	dd4d623612	Update 'disable_register_mail' verbiage (#4155)	3 hours ago
docker	1841316f18	Provide an updated Dockerfile where Gogs is build with Go ...	1 day ago
models	41a2632ae7	webhook: minor text change on Discord	7 hours ago
modules	95b94b2166	bindata: generate data for PR #4155	3 hours ago
packager	fa8bf0f1d7	Fix postinstall to use GOGS_CUSTOM instead of symlinking	1 year ago
public	7de71333c6	repo: several minor improvements	9 hours ago
routers	7de71333c6	repo: several minor improvements	9 hours ago
scripts	3137665e6e	Simplify description	4 days ago
templates	7de71333c6	repo: several minor improvements	9 hours ago

Figure 36 Gogs' user interface for a Git repository (69)

After a round of basic security update to the servers, the servers were ready to have Docker installed on them. At the time, the latest stable version of Docker was 1.12.3 so it was selected as the version to install. Though the existing CI system currently uses Docker 1.8, a much older release, the thesis team decided it would be better to build the new system against a newer version of Docker as the existing system would have its Docker version upgraded anyway. Docker was installed using a simple script based on official installation guides (Code example 2).

```
# Install recommended extra packages
# to allow the use of AUFS storage driver.
apt-get install --yes \
    linux-image-extra-$(uname -r) \
    linux-image-extra-virtual

# Setup the Docker repository
apt-get install --yes \
    apt-transport-https \
        ca-certificates \
        curl \
        software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
apt-key fingerprint 0EBFCD88
add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"

# Update package manager cache
apt-get update

# Install Docker Engine 1.12.3
apt-get install --yes docker-engine=1.12.3-0~ubuntu-trusty
```

Code example 2. Docker 1.12.3 installation script for the Ubuntu operating system

The installation was then evaluated against the CIS Docker 1.11.0 Benchmark by Center for Internet Security (71), which is a revered set of guidelines for creating a secure basis for a Docker installation. The evaluation was done using version 1.1.0 of the official Docker Bench for Security (72). Some parts, such as logging and separate hard drives for Docker and the operating system, were skipped as irrelevant for this thesis' purposes. As per the benchmark, access to so-called legacy registries was disabled. This disables the use of older, more insecure protocols for Docker registries. Other meaningful options were set to secure values by default in the engine installation. The final report made using the Docker Bench for Security is available in appendix 2. The servers were now ready to host the thesis' Docker containers.

Creating the Jenkins Docker image

Next, Digia created a GitHub repository for the thesis' Jenkins Docker image where the image could be inspected and distributed after the thesis was done (65). The repository hosts the Dockerfile, and all required scripts and source code for creating the Docker image for this thesis' automated CI system. It also contains short guides for the usage and further configuration of the system.

To start off, the official Jenkins Docker image was chosen as the image's basis (73) due to its already excellent quality and level of documentation. Some environment variables were added to allow later configuration of Docker and the CI system. The Dockerfile was also adjusted slightly to merge unnecessary extra layers to optimize the size of the image slightly as described in chapter 4.4. Code example 3 shows a snippet from the final Dockerfile, where the ENV commands were merged to optimize the number of layers in the image and four new environment variables were added: DOCKER_VERSION, DOCKER_SHA256, SEED_JOBS_DIR and CONF_GITHUB_TOKEN. The meaning of these variables will be explained later in this chapter. The full Dockerfile is available in the public GitHub repository and in appendix 3.

```
ENV JENKINS_HOME=/var/jenkins_home
ENV JENKINS_SLAVE_AGENT_PORT=50000 \
    JENKINS_UC=https://updates.jenkins.io \
    COPY_REFERENCE_FILE_LOG=${JENKINS_HOME}/copy_refer-
ence_file.log \
    TINI_VERSION=0.9.0 \
    TINI_SHA=fa23d1e20732501c3bb8eeeca423c89ac80ed452 \
    DOCKER_VERSION=1.12.5 \

DOCKER_SHA256=0058867ac46a1eba283e2441b1bb5455df846144f9d9b
a079e97655399d4a2c6 \
    JENKINS_OPTS=-Djenkins.install.runSetupWizard=false \
    SEED_JOBS_DIR=/usr/share/jenkins/ref/seed-jobs \
    CONF_GITHUB_TOKEN_ID=github-org-token
```

Code example 3. Environment variables and an example of Docker image layer optimization from the final Dockerfile

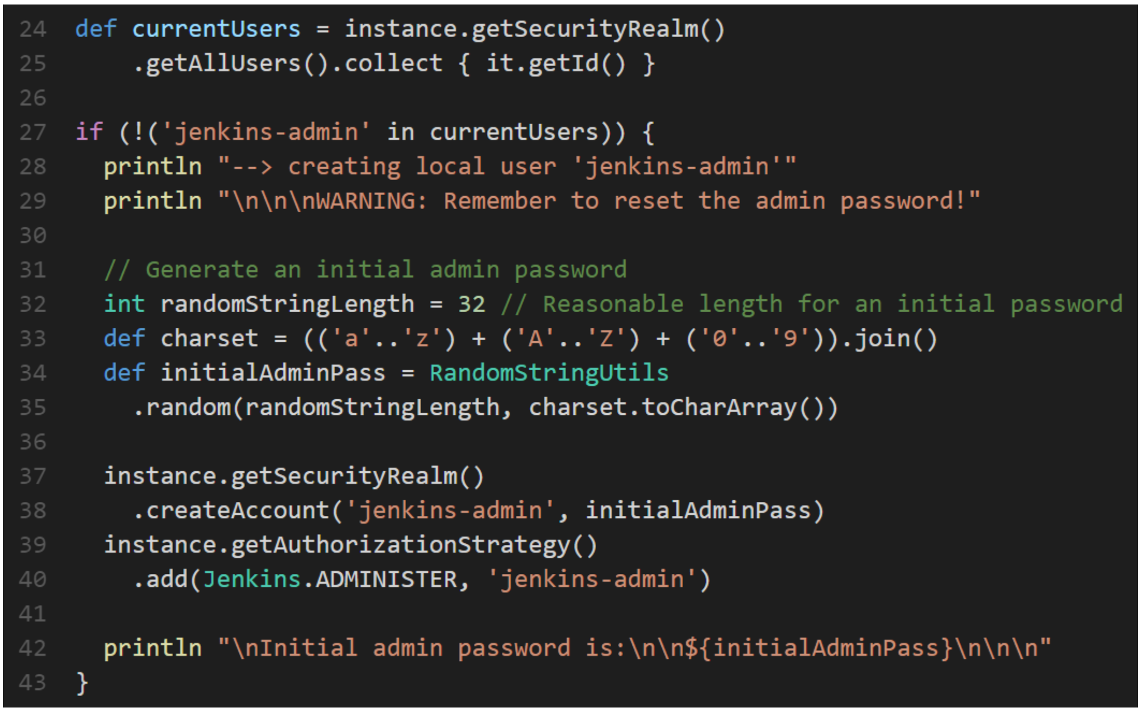
For Jenkins to be able to use Docker to build the example applications' images and possibly to create custom build environments, as desired in the baseline questionnaire, a Docker client needed to be installed in the Docker image. As can be seen from appendix 3's lines 59 through 63 this is quite straightforward: download the executable files from a secure Docker server and allow their execution in the image's environment. But, as the downloaded files are simply the Docker client, they need to connect to the host server's Docker server. This is done by mounting the docker.socket file from the host server – essentially allowing read-access to a file on the host – during runtime. This way, all Docker commands executed inside the Jenkins image, will control the host's Docker server. The previously mentioned environment variables, DOCKER_VERSION and DOCKER_SHA256, are used here to allow the selection of a Docker version.

It should be noted, however, that allowing Jenkins to access the host's Docker server is highly insecure as it can allow full root-level access to the host server (74). This was done, however, with the understanding that all builds would be done by trusted personnel and only source code from that personnel would ever make it to the CI system. Unfortunately, though, this means that the use of Docker in Jenkins would not be viable in every situation. The situation might improve later down the line but during this thesis, there was no known, reasonably unconvoluted solution and the risk was taken. For open-source projects accepting contributions from unknown entities, it might be reasonable to vet the changes even before any automated tests are run, therefore preventing access to the CI system but that will increase the workload for the project.

In addition to installing the Docker client and optimizing layer usage in the image, a plug-in installation script – provided in the official Jenkins Docker image repository – was added, along with a text document containing a machine-readable list of plug-ins to install automatically when the Docker image is built. The configuration file could then be used as a kind of dependency list for the Jenkins image, and whenever a plug-in should be updated the file would be updated and a new version of the Docker image be built – enabling even higher levels of version control for this CI system.

Script-based automation of Jenkins configuration

After the research and evaluation of the script-based configuration of Jenkins, as described in chapter 5.2, was done, creating the Groovy hook scripts for relatively straightforward. Few key features needed to be configured: an initial admin user to skip Jenkins' installation wizard (Figure 37), login tokens for accessing the target GitHub organization, instant messaging integration for build notifications, shared global libraries for build pipelines and the automatic seeding of a job that would scan the desired GitHub organization for repositories. Note, however, that all configuration shown in this thesis pertains only to the example applications and actual configuration values should be decided based on user needs. For example, the authorization strategies will be the most likely to vary as only a single master admin user was created for this example system.

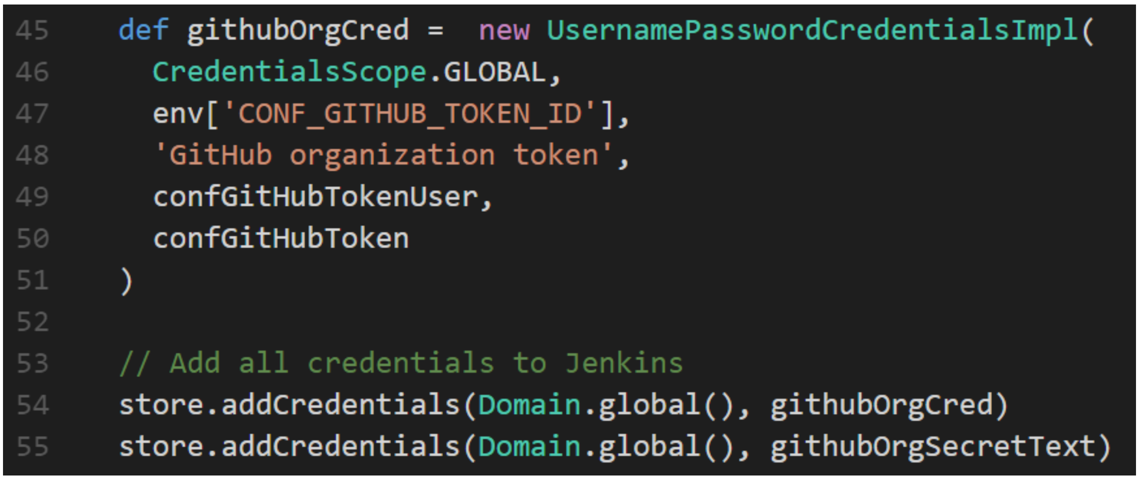


```
def currentUsers = instance.getSecurityRealm()
    .getAllUsers().collect { it.getId() }

if (!('jenkins-admin' in currentUsers)) {
  println "--> creating local user 'jenkins-admin'"
  println "\n\n\nWARNING: Remember to reset the admin password!"

  // Generate an initial admin password
  int randomStringLength = 32 // Reasonable length for an initial password
  def charset = (('a'..'z') + ('A'..'Z') + ('0'..'9')).join()
  def initialAdminPass = RandomStringUtils
    .random(randomStringLength, charset.toCharArray())

  instance.getSecurityRealm()
    .createAccount('jenkins-admin', initialAdminPass)
  instance.getAuthorizationStrategy()
    .add(Jenkins.ADMINISTER, 'jenkins-admin')

  println "\nInitial admin password is:\n\n${initialAdminPass}\n\n\n"
}
```

Figure 37 Excerpt from hook script to set an initial admin password for Jenkins in case the admin user does not already exist.

As show in the excerpt in Figure 37, some considerations needed to be made to the fact that these hook scripts will execute on every boot of the Jenkins Docker container; the admin user’s password shouldn’t be reset if the user already exists as that would confuse users that have changed the initial password to something else.

Inversely, the GitHub login tokens should be set every time the container is started as the tokens are defined by the users as environment variables for the Docker container, meaning they can change during the lifetime of the container’s data. User’s shouldn’t need to start from scratch when they simply want to update a set of login tokens – perhaps after a security breach. As the login tokens allow access to secure data, they should not be stored in plain text on Jenkins, but instead using Jenkins’ built-in data store for credentials (Figure 38). This also allows simpler access to them by using a unique identification field, configured using the environment variable “CONF_GITHUB_TOKEN_ID” as show in Figure 38.

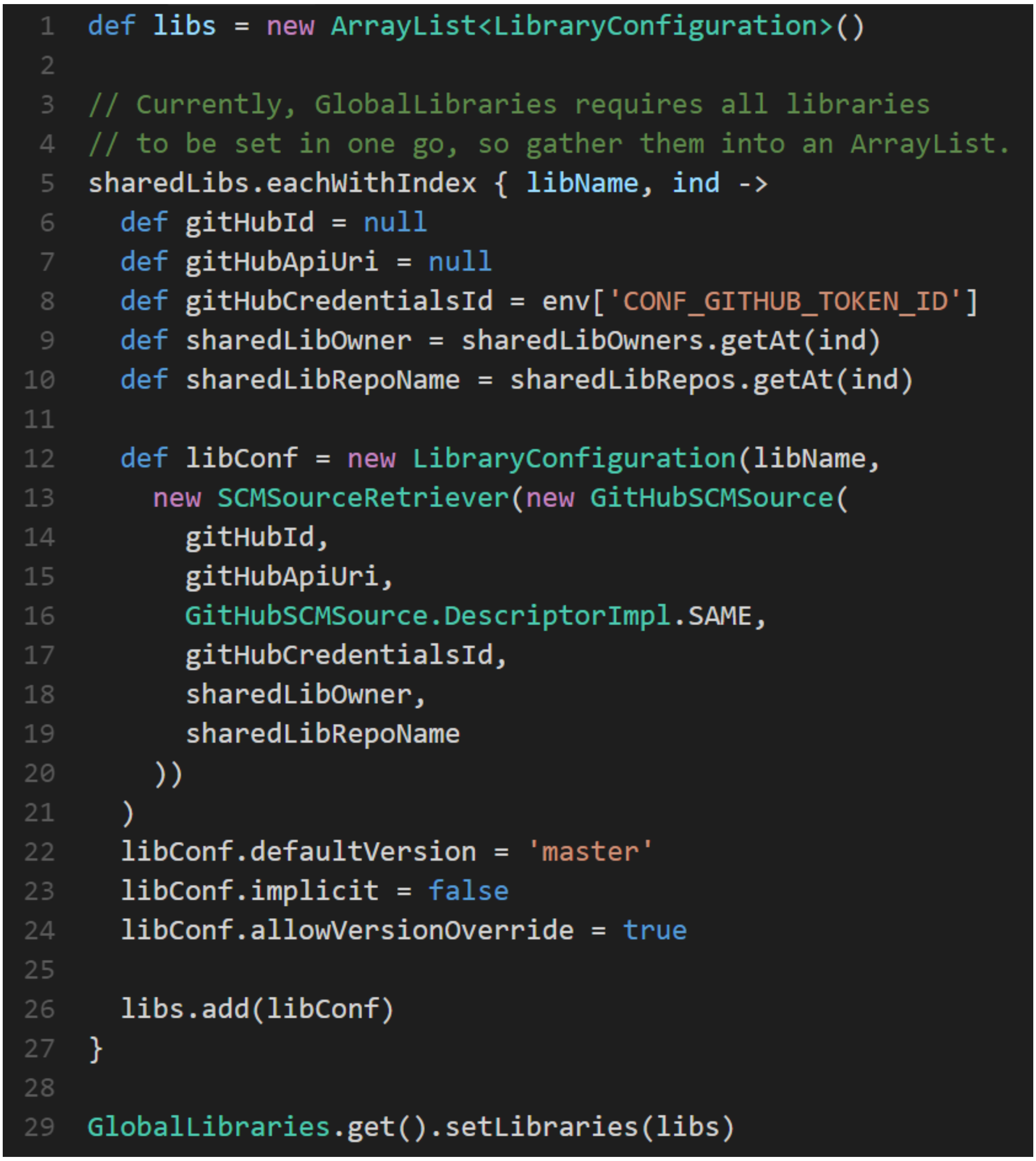


```
def githubOrgCred =  new UsernamePasswordCredentialsImpl(
  CredentialsScope.GLOBAL,
  env['CONF_GITHUB_TOKEN_ID'],
  'GitHub organization token',
  confGitHubTokenUser,
  confGitHubToken
)

// Add all credentials to Jenkins
store.addCredentials(Domain.global(), githubOrgCred)
store.addCredentials(Domain.global(), githubOrgSecretText)
```

Figure 38 Excerpt from script to setup GitHub login tokens for Jenkins

For the build notifications, Slack was chosen due to its mature Jenkins plug-in and clear API. Similarly to the GitHub tokens, Slack tokens are input using the environment variable “CONF_SLACK_TOKEN” and stored in Jenkins’ credential store. By configuring Slack settings globally, it allowed removing private data like the Slack integration’s name from application specific build configurations and instead allowed them to use the global default values wherever possible.

The shared library feature in Jenkins Pipelines allows the use of external Groovy code that can be used to share common functionality. As it is very likely that the number of these libraries will often be larger than one, the thesis project team decided to allow the configuration on multiple libraries at the same time. Like with other configurations, this was done using environment variables which, this time, included a separator character to make it possible to insert multiple library configurations simultaneously. The configurations were then converted into global libraries in Jenkins as shown in Figure 39.



```groovy
def libs = new ArrayList<LibraryConfiguration>()

// Currently, GlobalLibraries requires all libraries
// to be set in one go, so gather them into an ArrayList.
sharedLibs.eachWithIndex { libName, ind ->
  def gitHubId = null
  def gitHubApiUri = null
  def gitHubCredentialsId = env['CONF_GITHUB_TOKEN_ID']
  def sharedLibOwner = sharedLibOwners.getAt(ind)
  def sharedLibRepoName = sharedLibRepos.getAt(ind)

  def libConf = new LibraryConfiguration(libName,
    new SCMSourceRetriever(new GitHubSCMSource(
      gitHubId,
      gitHubApiUri,
      GitHubSCMSource.DescriptorImpl.SAME,
      gitHubCredentialsId,
      sharedLibOwner,
      sharedLibRepoName
    ))
  )
  libConf.defaultVersion = 'master'
  libConf.implicit = false
  libConf.allowVersionOverride = true

  libs.add(libConf)
}

GlobalLibraries.get().setLibraries(libs)
```

Figure 39 Excerpt from hook script to configure shared Jenkins Pipeline libraries

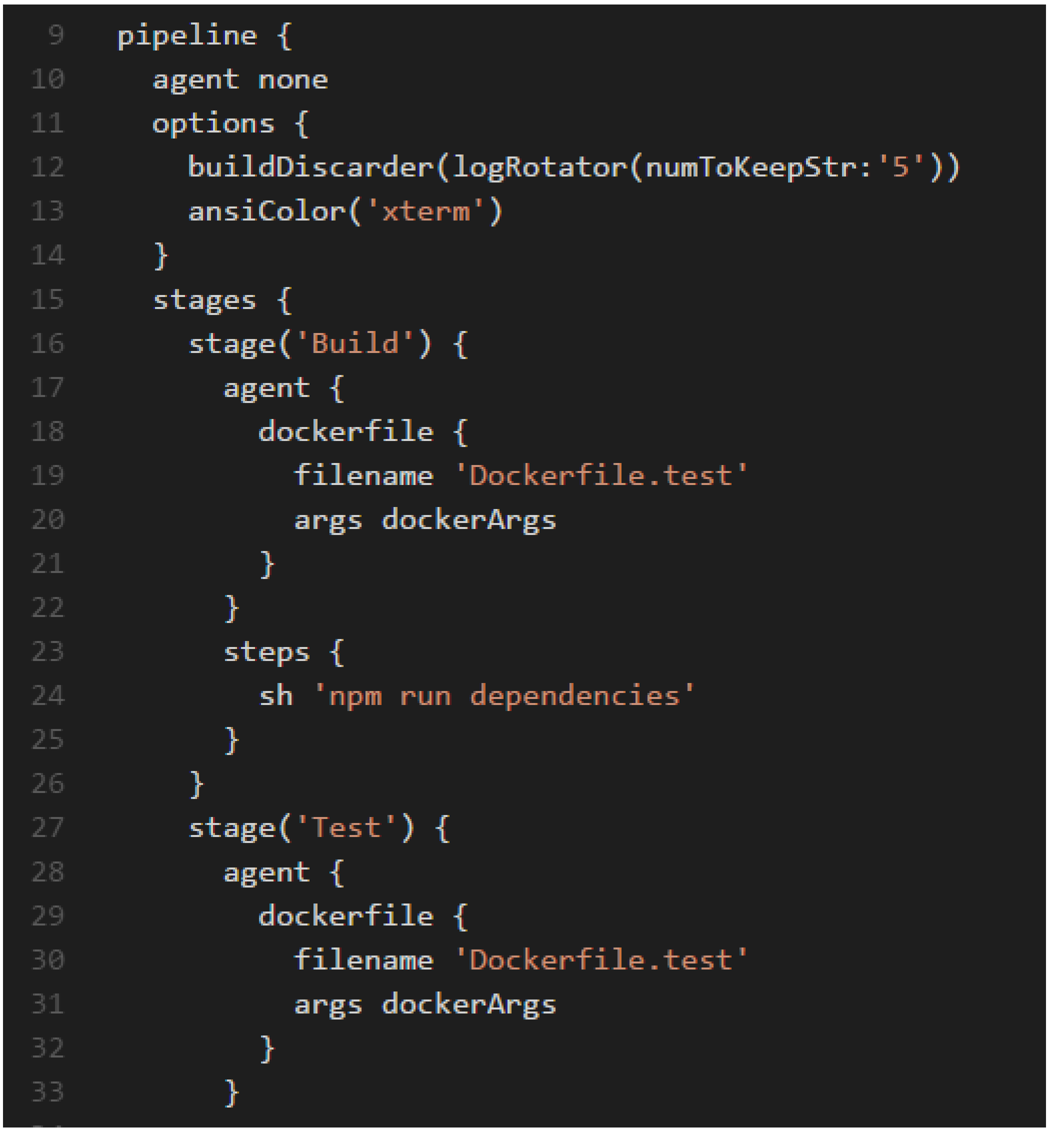
To automatically generate jobs using the Job DSL Plugin (63), a simple script was made that reads all files that have filenames ending in “.groovy” in a designated directory. The directory will get populated during the Docker image build with Job DSL scripts for the desired jobs. Here, the only necessary job was a so-called GitHub Organization Folder provided by the GitHub Branch Source Plugin (75). The folder will automatically scan a given GitHub organization – a group of Git repositories – and, for the ones that contain a Jenkinsfile, create build pipelines. Those build pipelines will also be separated by version control branch and immediately created, updated and removed based on events coming from GitHub. This feature was initially provided by a plugin of the same name, GitHub Organization Folder Plugin, but that plug-in was deprecated during the development of this thesis and all functionality was migrated to the new plugin as described in an official Jenkins blog post (76). The full script for setting up the automatic scanning job is available in appendix 4.

Build pipeline for an example application

The main example application has the following features: it is built using common Node.js build tools, its final form is a Docker image containing a web server hosting static files, it outputs JUnit test reports and HTML-formatted test coverage reports and it has two deployment scripts: one for development and another for production. The deployment scripts are meant to simply demonstrate deployments and are not designed to be fully functional.

One of the specifications defined after the baseline questionnaire, was to create a build pipeline that takes a Dockerfile specification and uses the resulting Docker container for the pipeline’s build environment. As mentioned in chapter 5.1, this was not initially possible and using Docker required using publicly available Docker images instead of ones in the example applications’ Git repositories. The pipeline also had to have some method of requiring manual user input before production deployments.

After some testing with the newly released declarative style for Jenkinsfile – shown in Figure 40 – it was rejected for the final product as too incomplete, especially for the manual input before production deployments. Though it gained the ability to use fully custom Dockerfiles during the development of this thesis, it lacked the ability to share the created Docker containers between stages without using the same container throughout the pipeline. The issue with this is that Jenkins has a concept of executors which essentially defines the number of concurrent builds that can be run on a single Jenkins instance and the as the pipelines were expected to wait even for weeks in the stage before production deployment, it would have been unacceptable to have them consume these executors. The manual input step can be run outside of the main executors without consuming much resources but this requires that the build is not currently using a Docker container, in this instance. Without the ability to share Docker containers or execute stages outside the test environment container, it would have required creating new containers for every single stage except the manual input stage (Figure 40) which would have consumed too much resources in the long run.

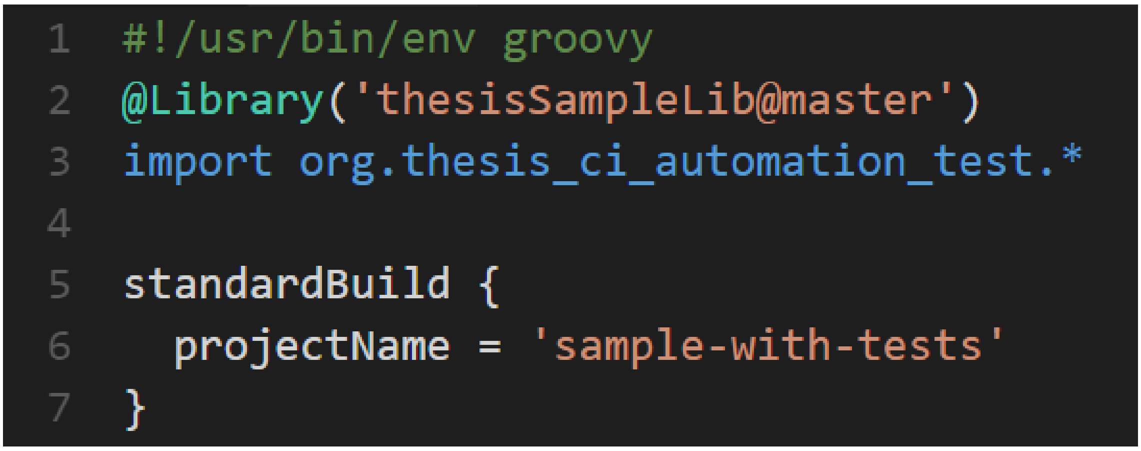


```
pipeline {
  agent none
  options {
    buildDiscarder(logRotator(numToKeepStr:'5'))
    ansiColor('xterm')
  }
  stages {
    stage('Build') {
      agent {
        dockerfile {
          filename 'Dockerfile.test'
          args dockerArgs
        }
      }
      steps {
        sh 'npm run dependencies'
      }
    }
    stage('Test') {
      agent {
        dockerfile {
          filename 'Dockerfile.test'
          args dockerArgs
        }
      }
```

Figure 40 Using a separate Docker containers for each pipeline stage by defining a new “agent” block every time

The source code for the declarative style pipeline is available in appendix 5 for further comparisons but it should be noted that it is missing some features compared to the final scripted pipeline, like test coverage reporting. The pipeline refers to an older version of the shared pipeline libraries used in the final pipeline but other portions are fully functional and are presented merely to show the difference between the scripted and declarative approaches.

The final build pipeline was implemented using a shared library created for this thesis. The example application's Jenkinsfile is very simple as a direct result of this decision as can be seen in Figure 41. The shared library – included here using the "@Library" annotation and the "import" command – exposes a method called standardBuild that takes a single parameter: the example application's name. This approach might be the best for the team this thesis was done for as their builds are mostly identical from the build configuration's perspective and would remove most of the boilerplate code moved around in the Git repositories. However, for more varied scenarios, the full scripted build is available in the shared library's standardBuild method.



```
#!/usr/bin/env groovy
@Library('thesisSampleLib@master')
import org.thesis_ci_automation_test.*

standardBuild {
  projectName = 'sample-with-tests'
}
```

Figure 41 The final Jenkinsfile for the example application's build pipeline

The shared library feature of Jenkins Pipelines allowed for two levels of common modules: utility modules for things like Slack integration and high-level abstractions like the "standardBuild" used in Figure 41. Though the level of abstraction is extremely high for the final example, users can create their own abstractions suitable for their specific needs. One idea the thesis team had was that a method could be exposed that contained some standardized set of stages which would then take their actual contents from the applications' Jenkinsfiles. This could unify all build pipelines in a project and allow for a more consistent experience compared to every developer creating their own definitions of build pipelines.

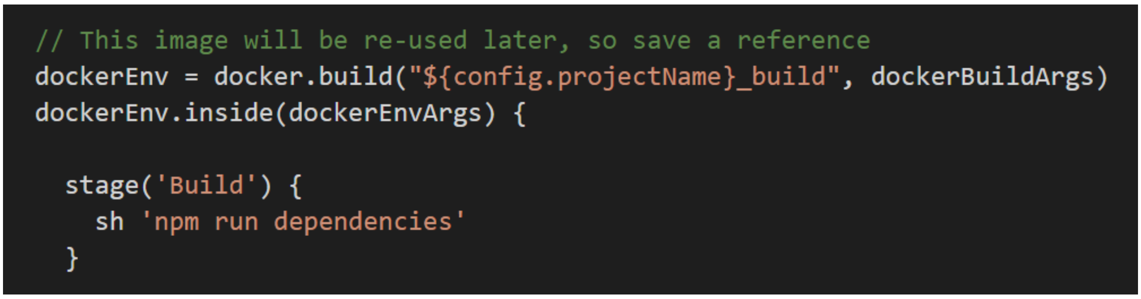
In the final pipeline shown in appendix 6, custom test environments are created by placing a file named "Dockerfile.test" in an application's Git repository which is then used to build an image and started automatically as a container in which all the build stages are run – except for the manual input stage, which is executed outside any containers. For example, the main example application needs some common Node.js build tools to create its distribution files and run its tests. The application's Git repository can then have the Dockerfile.test file that will build an image which already has those tools preinstalled (Code example 4).

```
FROM node:6

# Install required Node.js tools
RUN npm install -g bower grunt-cli && \
    echo '{ "allow_root": true }' > /root/.bowerrc

# Define working directory.
WORKDIR /data

# Define default command.
CMD ["bash"]
```

Code example 4. Dockerfile.test example for a pipeline using common Node.js build tools

Building the same application in the existing system required that those same tools were installed on the Jenkins instance itself. This then meant that all builds had to support the same versions of those tools, leading to some painful upgrade processes when multiple applications needed to be upgraded at the same time to use some newer version of, for example, Grunt. There are ways to circle that issue in the existing system but all of them require that those environments are defined in Jenkins itself and not in every application. That would then slow down the process of upgrading dependencies and create pain points for the developers, as shown in the baseline questionnaire.

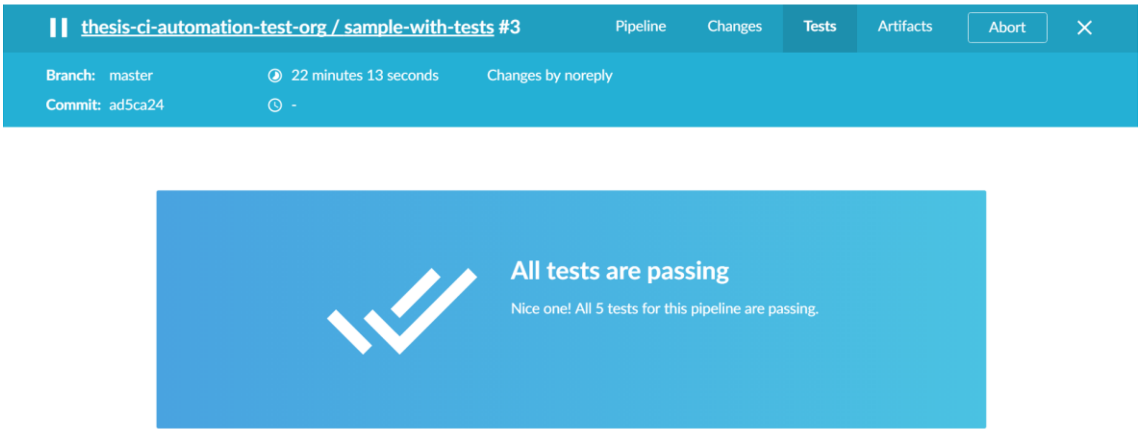
The shared library can instead use the provided Dockerfile.test definition to build an image and run the pipeline stages inside a container using that image as shown in the excerpt in Figure 42. There is at least one issue with this approach, though: as the example application wants to build its own Docker image from the results of its build, it requires allowing access to the host machine's Docker server, like in the Jenkins container. As with Jenkins itself, this can allow dangerous access to the host server, so considerations need to be made when selecting what Docker images to use for pipelines' custom environments. Using Docker also means that environment variables available in the standard build environment need to explicitly passed through to the custom environment as Docker will not automatically set any environment variable values.



```
// This image will be re-used later, so save a reference
dockerEnv = docker.build("${config.projectName}_build", dockerBuildArgs)
dockerEnv.inside(dockerEnvArgs) {

  stage('Build') {
    sh 'npm run dependencies'
  }
```

Figure 42 Running pipeline stages inside a Docker container

The test coverage reporting is done using the HTML-formatted reports provided by the example application as the Cobertura Plugin is not currently compatible with Jenkins Pipeline (77). Although the Blue Ocean UI can show the test results (Figure 43) it currently lacks the ability to show test coverage reports completely. This was one issue that could not be resolved during this thesis and the classic Jenkins UI must be used to link to the generated coverage reports.



thesis-ci-automation-test-org / sample-with-tests #3

Branch: master

Commit: ad5ca24

22 minutes 13 seconds

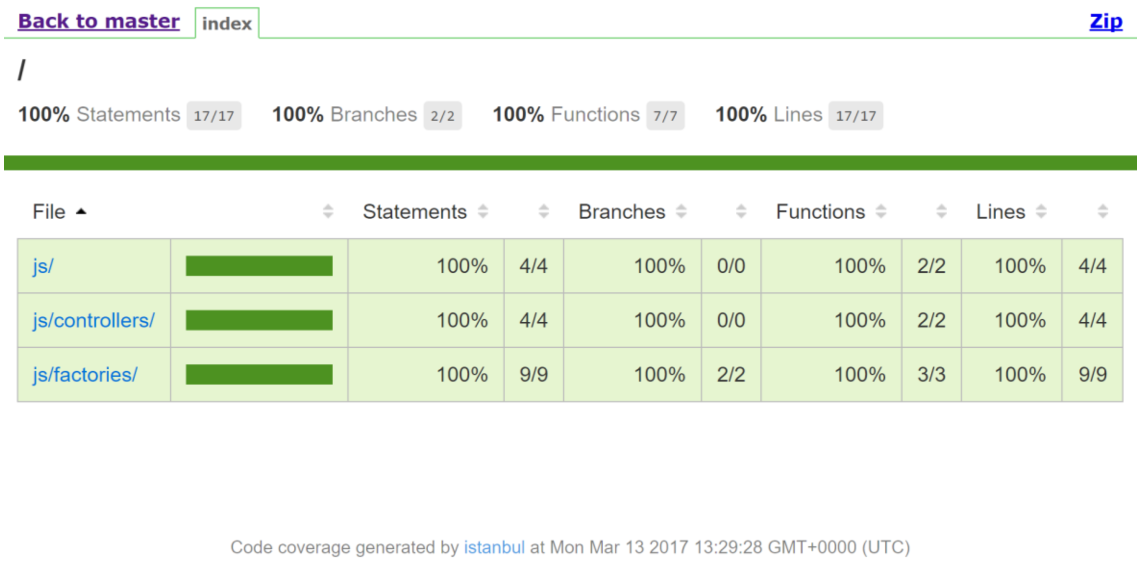
Changes by noreply

All tests are passing

Nice one! All 5 tests for this pipeline are passing.

Figure 43 Jenkins shows that all tests are passing for a build pipeline

The test coverage system used in the example application and by the project team does provide a visually pleasing and clear HTML report for test coverage (Figure 44), meaning that this feature was not completely lost. This report can be linked in the classic Jenkins UI and the thesis project team believes that a similar feature or full support for proper Cobertura coverage reporting will at some point be available in the Blue Ocean UI as well.



/

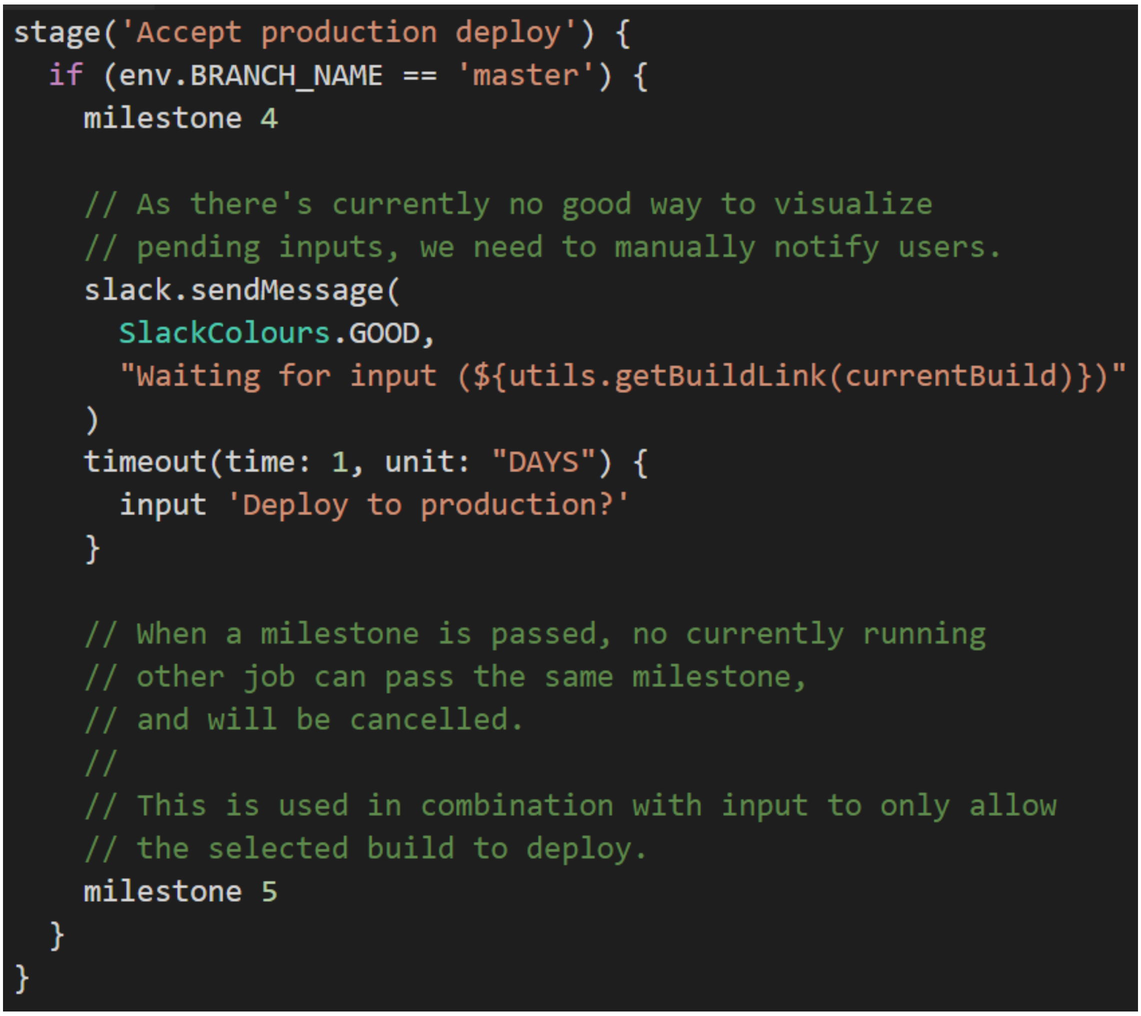
100% Statements 17/17 100% Branches 2/2 100% Functions 7/7 100% Lines 17/17

File ▲		Statements		Branches		Functions		Lines	
js/	🟩	100%	4/4	100%	0/0	100%	2/2	100%	4/4
js/controllers/	🟩	100%	4/4	100%	0/0	100%	2/2	100%	4/4
js/factories/	🟩	100%	9/9	100%	2/2	100%	3/3	100%	9/9

Code coverage generated by istanbul at Mon Mar 13 2017 13:29:28 GMT+0000 (UTC)

Figure 44 Test coverage report from the example application

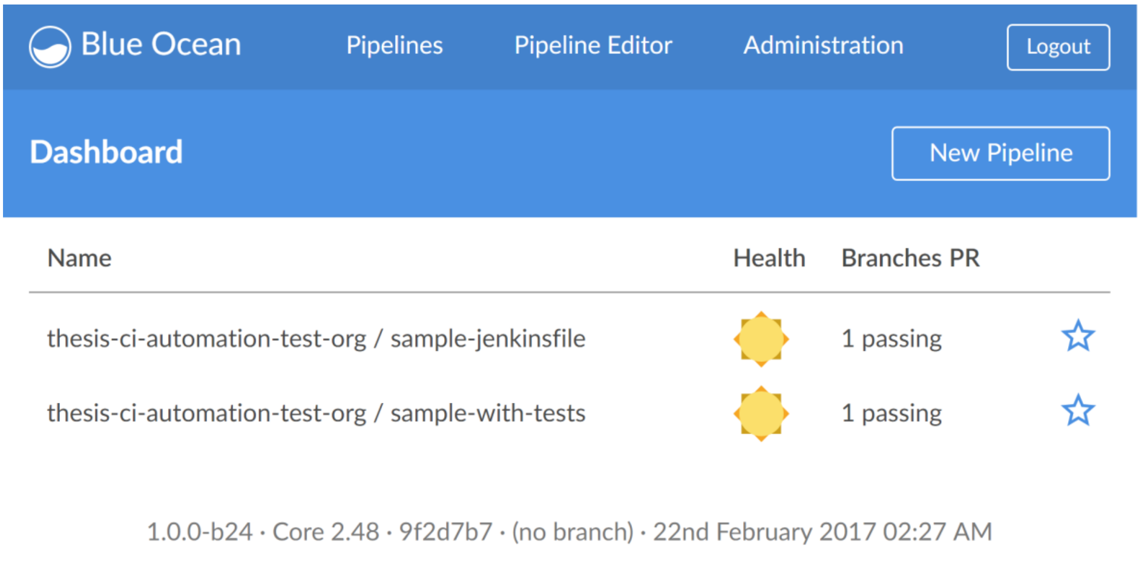
Finally, to prevent fully automatic deployments to production environments, a step called input was introduced; the input step pauses the build pipeline and waits for user input indefinitely, unless an explicit timeout is specified. The final pipeline has this step inside a stage called “Accept production deploy” (Figure 45). This stage is skipped unless the version control branch being built has the name “master”, meaning production deployments are only done from the master branch. It also uses steps called milestone that stop all other builds when some other build passes a milestone step they haven’t passed.



```
stage('Accept production deploy') {
  if (env.BRANCH_NAME == 'master') {
    milestone 4

    // As there's currently no good way to visualize
    // pending inputs, we need to manually notify users.
    slack.sendMessage(
      SlackColours.GOOD,
      "Waiting for input (${utils.getBuildLink(currentBuild)})"
    )
    timeout(time: 1, unit: "DAYS") {
      input 'Deploy to production?'
    }

    // When a milestone is passed, no currently running
    // other job can pass the same milestone,
    // and will be cancelled.
    //
    // This is used in combination with input to only allow
    // the selected build to deploy.
    milestone 5
  }
}
```

Figure 45 Final pipeline stage to wait for user input before a production deployment

Unfortunately, the state that a pipeline is waiting for input is not visible in the classic Jenkins UI or in the new Blue Ocean UI (Figure 46). Without any other feedback, this would mean that users would have to go through all pipelines to check whether one of them was waiting for input. That is clearly unacceptable from a usability perspective. Fortunately, though, as Slack integration had already been introduced to the build pipelines for instant notifications on build failures, the notification system could be used for this purpose, too: when a pipeline reaches the stage where manual input is required, a notification is sent to notify everyone on the team that an application is ready for production deployment (Figure 45) and the team can then take action if required.

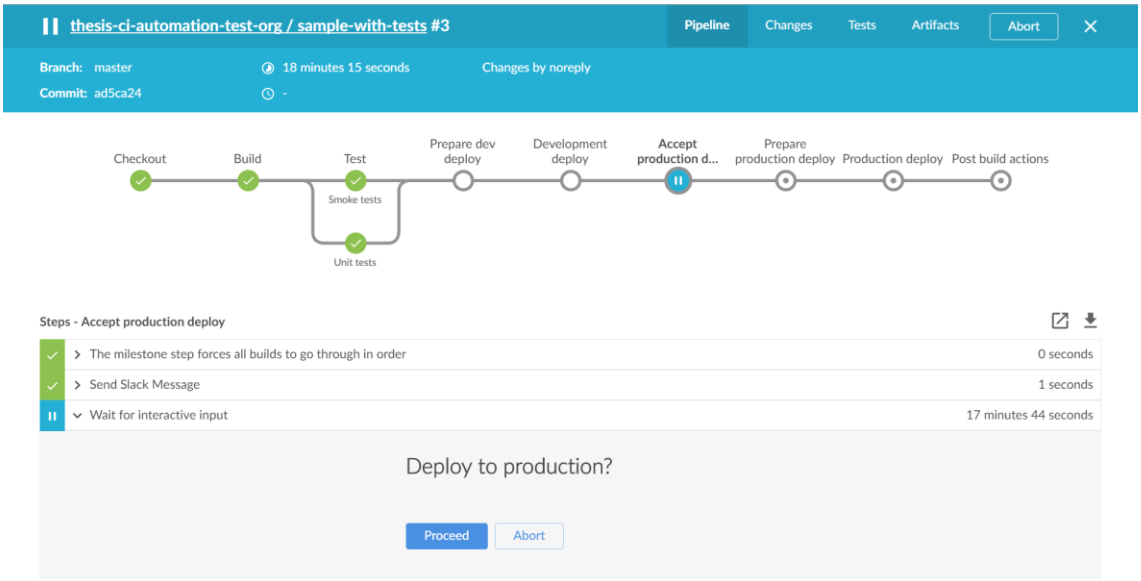


Name	Health	Branches	PR
thesis-ci-automation-test-org / sample-jenkinsfile		1 passing	☆
thesis-ci-automation-test-org / sample-with-tests		1 passing	☆

1.0.0-b24 · Core 2.48 · 9f2d7b7 · (no branch) · 22nd February 2017 02:27 AM

Figure 46 Blue Ocean UI not showing that a pipeline is waiting for user input

From the Slack notification, users are directed to the build pipeline waiting for input. This view then shows the buttons to either allow the production deployment or to abort the build. Jenkins also visualizes this status by marking the stage waiting for input with a pause symbol and by changing the overall color scheme of the page (Figure 47).



thesis-ci-automation-test-org / sample-with-tests #3

Branch: master
Commit: ad5ca24
18 minutes 15 seconds
-
Changes by noreply

Steps - Accept production deploy

✓	The milestone step forces all builds to go through in order	0 seconds
✓	Send Slack Message	1 seconds
II	Wait for interactive input	17 minutes 44 seconds

Deploy to production?

Figure 47 Build pipeline waiting for input before production deployment

In conclusion, combining Docker for the custom environments with the script-based configuration of Jenkinsfiles and the new Blue Ocean UI with improved usability for users and managers, resulted in a nearly fully automated CI/CD system that the thesis team thinks can help alleviate many of the issues faced by the project team on the existing system. The new system could be configured fully automatically from Jenkins to the build pipelines, and destroyed and brought back up quickly on a whim.

7 Results and conclusion

The final version of the developed continuous integration and deployment system was presented to the software development team at Digia Oy whose existing system this thesis attempted to improve and automate. After a quick demonstration and a high-level introduction to the design goals of the build pipeline system, the group discussed the end product and how it reflects on the goals, and compares to the original system it intended to replace. As the thesis team had expected, the overall reception was positive and the development team appeared eager to take the system into use.

The demonstration included the CI/CD system's and builds' configuration using different scripting methods described in this thesis and using Docker in build pipelines. Especially the script-based automation of system configuration and pipelines got a great reception, and the team saw these features essential for improving the maintainability and extensibility of the whole CI/CD system. One of the original questionnaire answerers had already taken some cues from the work done in this thesis to another team and had seen positive results, solving the same type of problems this thesis intended to solve.

Blue Ocean was also on display as it was essential to the usability improvements made in this thesis. The development team agreed – at least on the basis of the quick demonstration – that the new Blue Ocean user interface brings forth the most crucial information, such as what builds are running, and can help users find the information they need more quickly than with the previous system.

The final system is not without its flaws, though: reporting – which was seen as lacking in the baseline questionnaire – was not improved much compared to the existing system, some features related to software deployment specific to the development team had not been fully tested and the manual acceptance step of the example pipeline did not include any method of authorizing the approval. Although most of these issues are solvable, it was unfortunate that they could be resolved during the work done for this thesis.

In conclusion, this thesis reached its main goals: usability and maintainability improvements using automation techniques, and improvements in the workflow of software developments teams via clearly defined, and extensible build pipelines. The system created in this thesis will be taken into use by the original development team and has already formed the basis for a company-wide template for similar systems.

References

1 Kuvia [Online]. Digia Oyj. <http://digia.com/yritys/kuvia/>. Accessed 9.3.2017.

2 Jenkins Press Information [Online]. Software in the Public Interest. <https://jenkins.io/press/>. Accessed 9.3.2017.

3 Docker [Online]. Docker Inc. <https://www.docker.com/>. Accessed 9.3.2017.

4 Kolehmainen, Antti. OHJELMISTOTESTAUS: Opas ohjelmistokehittäjille. Opinnäytetyö. Jyväskylän ammattikorkeakoulu; 2009.

5 Git - About Version Control [Online]. Git. <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>. Accessed 9.3.2017.

6 GitHub Enterprise - The best way to build and ship software [Online]. GitHub, Inc. <https://enterprise.github.com/home>. Accessed 9.3.2017.

7 Google Forms [Online]. Google. <https://docs.google.com/forms/>. Accessed 9.3.2017.

8 Blue Ocean [Online]. Software in the Public Interest. <https://jenkins.io/projects/blueocean/>. Accessed 9.3.2017.

9 Linz T. 2014. Testing in Scrum Sebastopol: O'Reilly Media.

10 Taschner, Chris. 2014. Security in Continuous Integration [Online]. <https://insights.sei.cmu.edu/sei_blog/2014/12/security-in-continuous-integration.html>. Accessed 10.3.2017.

11 Dustin E, Garrett T, Gauf B. 2009. Implementing Automated Software Testing Stoughton, Massachussets: Pearson Education, Inc.

12 Git - Logo Downloads [Online]. Git. <https://git-scm.com/downloads/logos>. Accessed 10.3.2017.

13 Loeliger J, McCullough M. 2012. Version Control with Git Sebastopol: O'Reilly Media Inc.

14 Driessen, Vincent. 2010. A successful Git branching model [Online]. <http://nvie.com/posts/a-successful-git-branching-model/>. Accessed 10.3.2017.

15 Understanding the GitHub Flow - GitHub Guides [Online]. GitHub, Inc. <https://guides.github.com/introduction/flow/>. Accessed 10.3.2017.

16 Humble, Jez. 2010. Continuous Delivery vs Continuous Deployment - Continuous Delivery [Online]. <https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/>. Accessed 10.3.2017.

17 Sundman, Yassal. 2013. Continuous Delivery vs Continuous Deployment [Online]. <http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment>. Accessed 10.3.2017.

18 Shayan, Chris. 2013. Continuous Delivery Matrix - Chris Shayan - Confluence [Online]. <https://chrisshayan.atlassian.net/wiki/display/my/2013/07/23/Continuous+Delivery+Matrix?utm_source=Codeship&utm_medium=CI-Guide>. Accessed 10.3.2017.

19 Concourse: CI that scales with your project [Online]. Pivotal Software, Inc. <https://concourse.ci/index.html>. Accessed 10.3.2017.

20 Concepts [Online]. Pivotal Software, Inc. <https://concourse.ci/concepts.html>. Accessed 10.3.2017.

21 Team City -- Your 24/7 Build Engineer [Online]. JetBrains. <https://www.jetbrains.com/teamcity/>. Accessed 10.3.2017.

22 Technology-leading software development firm specializing in the creation of intelligent development tools [Online]. JetBrains. <https://www.jetbrains.com/company/>. Accessed 10.3.2017.

23 Buy TeamCity [Online]. JetBrains. <https://www.jetbrains.com/teamcity/buy/#license-type=new-license>. Accessed 10.3.2017.

24 Drone and Docker, Open Source CI - Drone [Online]. Drone.IO Inc. <http://blog.drone.io/post/docker-drone-open-source/>. Accessed 10.3.2017.

25 GitLab Continuous Integration | GitLab [Online]. GitLab Inc. <https://about.gitlab.com/gitlab-ci/>. Accessed 10.3.2017.

26 Introduction to pipelines and jobs - GitLab Documentation [Online]. GitLab Inc. <https://docs.gitlab.com/ee/ci/pipelines.html>. Accessed 10.3.2017.

27 Publish Code Coverage Report with GiLab Pages | GitLab [Online]. GitLab Inc. <https://about.gitlab.com/2016/11/03/publish-code-coverage-report-with-gitlab-pages/>. Accessed 10.3.2017.

28 Snap is Going Away | Snap CI's The Pipeline Blog [Online]. ThoughtWorks, Inc. <https://blog.snap-ci.com/blog/2017/02/06/2017-02-06-snap-announcement/>. Accessed 10.3.2017.

29 GoCD - Open Source Continuous Delivery Server | GoCD [Online]. ThoughtWorks, Inc. <https://www.gocd.io/>. Accessed 10.3.2017.

30 tomzo/gocd-yaml-config-plugin: Plugin to declare Go pipelines and environments configuration in YAML [Online]. tomzo. <https://github.com/tomzo/gocd-yaml-config-plugin>. Accessed 10.3.2017.

31 Releases - Version notes - 16.12.0 | GoCD [Online]. ThoughtWorks, Inc. <https://www.gocd.io/releases/>. Accessed 9.3.2017.

32 CloudBees, Inc. 2016. The State of Jenkins - 2016 Community Survey Results [Online]. <https://www.cloudbees.com/sites/default/files/2016-jenkins-community-survey-responses.pdf>. Accessed 9.3.2017.

33 White, Oliver. 2014. Java Tools and Technologies Landscape for 2014 [Online]. <https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/12/>. Accessed 9.3.2017.

34 Logo - Jenkins - Jenkins Wiki [Online]. Software in the Public Interest. <https://wiki.jenkins-ci.org/display/JENKINS/Logo>. Accessed 9.3.2017.

35 Docker Pipeline - Jenkins Plugins [Online]. Software for the Public Interest. <https://plugins.jenkins.io/docker-workflow>. Accessed 9.3.2017.

36 CloudBees Jenkins Enterprise

37 Whitehead, Nicholas. 2008. Continuous integration with Hudson | JavaWorld [Online]. <http://www.javaworld.com/article/2077956/open-source-tools/continuous-integration-with-hudson.html>. Accessed 9.3.2017.

38 Declarative Pipeline: Publishing HTML Reports [Online]. Software in the Public Interest. <https://jenkins.io/blog/2017/02/10/declarative-html-publisher/>. Accessed 9.3.2017.

39 Continuous Code Quality | SonarQube [Online]. SonarSource. <https://www.sonarqube.org/>. Accessed 9.3.2017.

40 Turnbull J. 2014. The Docker Book [Unknown]: James Turnbull.

41 ClusterHQ. 2016. Container Market Adoption Survey 2016 [Online]. <https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf>. Accessed 9.3.2017.

42 Docker, chroot, LXC, OpenVZ - Explore - Google Trends [Online]. Google. <https://trends.google.com/trends/explore?date=2012-03-11%202017-03-09&q=%2Fm%2F0wkcjgj,%2Fm%2F02sm3c,%2Fm%2F0crds9p,OpenVZ>. Accessed 9.3.2017.

43 What is a Container | Docker [Online]. Docker, Inc. <https://www.docker.com/what-container>. Accessed 9.3.2017.

44 Dockerfile reference [Online]. Docker Inc. <https://docs.docker.com/engine/reference/builder/>. Accessed 9.3.2017.

45 Using Docker Images - GitLab Documentation [Online]. GitLab Inc. <https://docs.gitlab.com/ce/ci/docker/using_docker_images.html>. Accessed 9.3.2017.

46 Docker Legal Terms - Marks and Logos [Online]. Docker, Inc. <https://www.docker.com/brand-guidelines>. Accessed 9.3.2017.

47 Introducing Docker for Windows Server 2016 [Online]. Docker, Inc. <https://blog.docker.com/2016/09/dockerforws2016/>. Accessed 9.3.2017.

48 Understanding images, containers and storage drivers [Online]. Docker, Inc. <https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/#content-addressable-storage>. Accessed 9.3.2017.

49 Boettiger, Carl. 2014. An introduction to Docker for reproducible research, with examples from the R environment [Online]. <https://arxiv.org/pdf/1410.0846.pdf>. Accessed 9.3.2017.

50 Petazzoni, Jérôme. 2017. Using Docker-in-Docker for your CI or testing environment? Think twice [Online]. <https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/>. Accessed 9.3.2017.

51 About Domain-Specific Languages [Online]. Microsoft. <https://msdn.microsoft.com/en-us/library/bb126278.aspx>. Accessed 9.3.2017.

52 Build your own image - Docker Documentation [Online]. Docker, Inc. <https://docs.docker.com/engine/getstarted/step_four/>. Accessed 9.3.2017.

53 Node.js [Online]. Node.js Foundation. <https://nodejs.org/en/>. Accessed 9.3.2017.

54 Best practices for writing Dockerfiles - Docker Documentation [Online]. Docker, Inc. <https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/>. Accessed 9.3.2017.

55 library/mysql [Online]. Docker, Inc. <https://hub.docker.com/_/mysql/>. Accessed 9.3.2017.

56 Pipeline as Code [Online]. Software in the Public Intereste. <https://jenkins.io/doc/book/pipeline-as-code/>. Accessed 12.3.2017.

57 Glick, Jesse. 2017. CloudBees Folders Plugin - Jenkins - Jenkins Wiki [Online]. <https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Folders+Plugin>. Accessed 12.3.2017.

58 Kawaguchi, Kohsuke, Andrew Phillips. 2014. Orchestrating Your Delivery Pipelines with Jenkins [Online]. <https://www.infoq.com/articles/orch-pipelines-jenkins>. Accessed 12.3.2017.

59 Wolf, Patrick. 2016. Announcing the beta of Declarative Pipeline Syntax [Online]. <https://jenkins.io/blog/2016/12/19/declarative-pipeline-beta/>. Accessed 12.3.2017.

60 Bayer, Andrew. 2017. Pipeline Model Definition Plugin [Online]. <https://wiki.jenkins-ci.org/display/JENKINS/Pipeline+Model+Definition+Plugin>. Accessed 12.3.2017.

61 Extending with Shared Libraries [Online]. Software in the Public Interest. <https://jenkins.io/doc/book/pipeline/shared-libraries/>. Accessed 12.3.2017.

62 Kawaguchi, Kohsuke. 2013. Groovy Hook Script - Jenkins - Jenkins Wiki [Online]. <https://wiki.jenkins-ci.org/display/JENKINS/Groovy+Hook+Script>. Accessed 12.3.2017.

63 Ryan, Justin. 2016. Job DSL Plugin - Jenkins - Jenkins Wiki [Online]. <https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin>. Accessed 12.3.2017.

64 thesis-ci-automation-test-org [Online]. Digia Oy. <https://github.com/thesis-ci-automation-test-org>. Accessed 11.3.2017.

65 DigiaFactory/automated-jenkins-pipeline [Online]. Digia Oy. <https://github.com/DigiaFactory/automated-jenkins-pipeline>. Accessed 11.3.2017.

66 Docker and the Device Mapper storage driver - Docker Documentation [Online]. Docker, Inc. <https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/>. Accessed 11.3.2017.

67 Select a storage driver - Docker Documentation [Online]. Docker, Inc. <https://docs.docker.com/engine/userguide/storagedriver/selectadriver/>. Accessed 11.3.2017.

68 Kawaguchi, Kohsuke. 2016. Installing Jenkins - Jenkins - Jenkins Wiki [Online]. <https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins>. Accessed 11.3.2017.

69 Gogs [Online]. Gogs. <https://gogs.io/>. Accessed 11.3.2017.

70 Releases · kmadel/gogs-branch-source-plugin [Online]. kmadel. <https://github.com/kmadel/gogs-branch-source-plugin/releases>. Accessed 11.3.2017.

71 CIS Docker 1.11.0 Benchmark [Online]. Center for Internet Security. <https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.11.0_Benchmark_v1.0.0.pdf>. Accessed 11.3.2017.

72 docker/docker-bench-security [Online]. Docker, Inc. <https://github.com/docker/docker-bench-security/tree/v1.1.0/distros>. Accessed 11.3.2017.

73 docker/Dockerfile at 79e871d7ea9085cabb287dd53705b4432e36cf6f · jenkinsci/docker [Online]. jenkinsci. <https://github.com/jenkinsci/docker/blob/79e871d7ea9085cabb287dd53705b4432e36cf6f/Dockerfile>. Accessed 11.3.2017.

74 Don't expose the Docker socket (not even to a container) [Online]. lvh. <https://www.lvh.io/posts/dont-expose-the-docker-socket-not-even-to-a-container.html>. Accessed 11.3.2017.

75 Connolly, Stephen. 2017. GitHub Branch Source Plugin [Online]. <https://wiki.jenkins-ci.org/display/JENKINS/GitHub+Branch+Source+Plugin>. Accessed 12.3.2017.

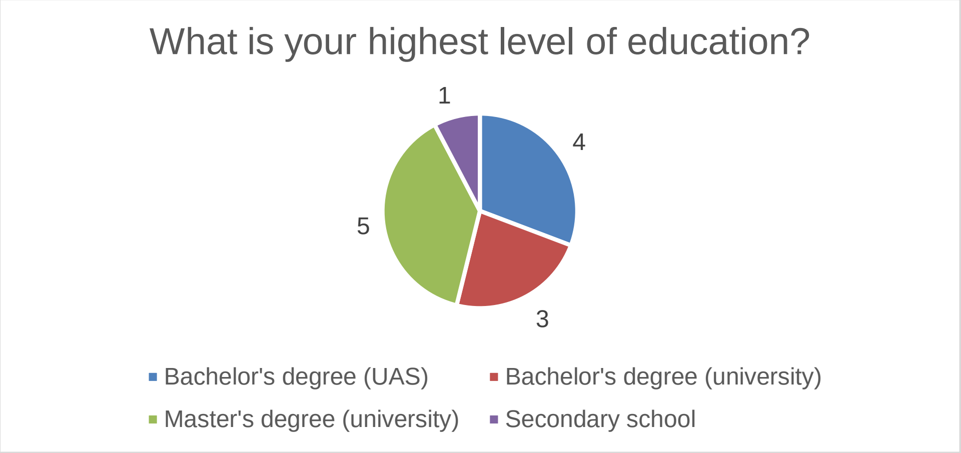
76 Connolly, Stephen. 2017. SCM API turns 2.0 and what that means for you [Online]. <https://jenkins.io/blog/2017/01/17/scm-api-2/>. Accessed 12.3.2017.

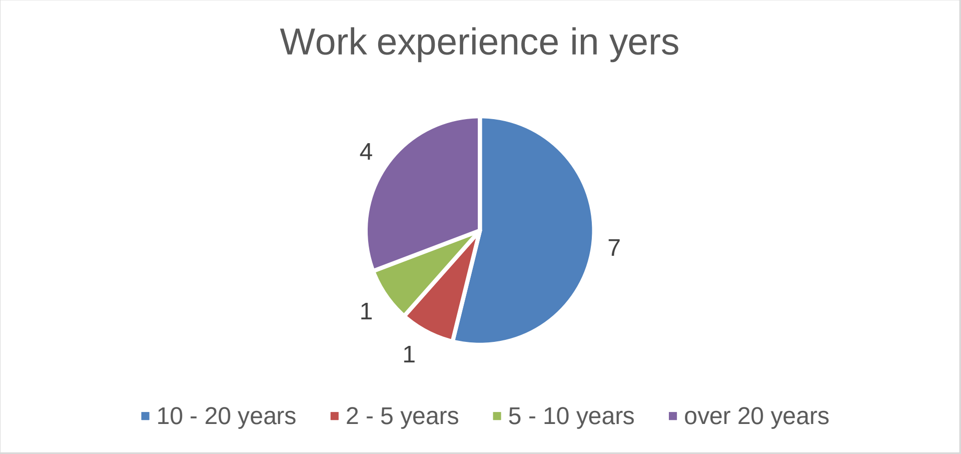
77 Make compatible with Jenkins Workflow/Pipeline plugin for report publishing [Online]. 60secs. <https://github.com/jenkinsci/cobertura-plugin/issues/50>. Accessed 13.3.2017.

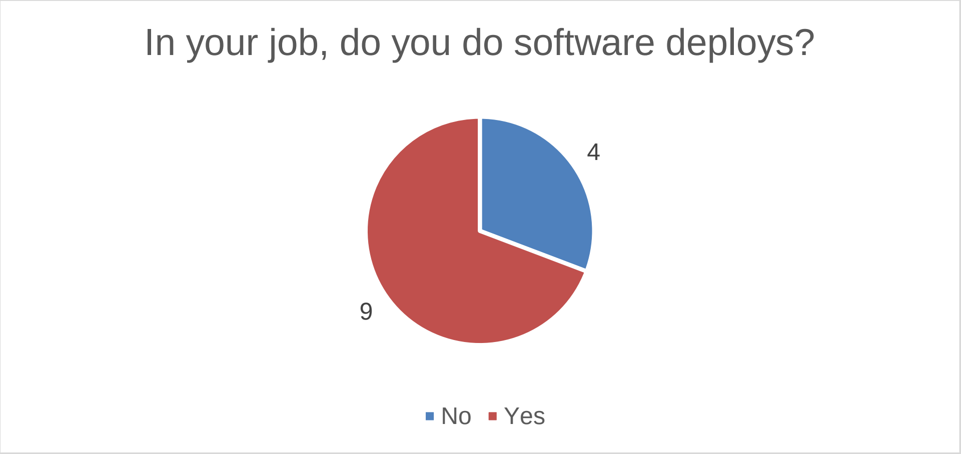
78 Richardson J, Gwaltney WJ. 2007. Ship It! USA: Pragmatic Bookshelf.

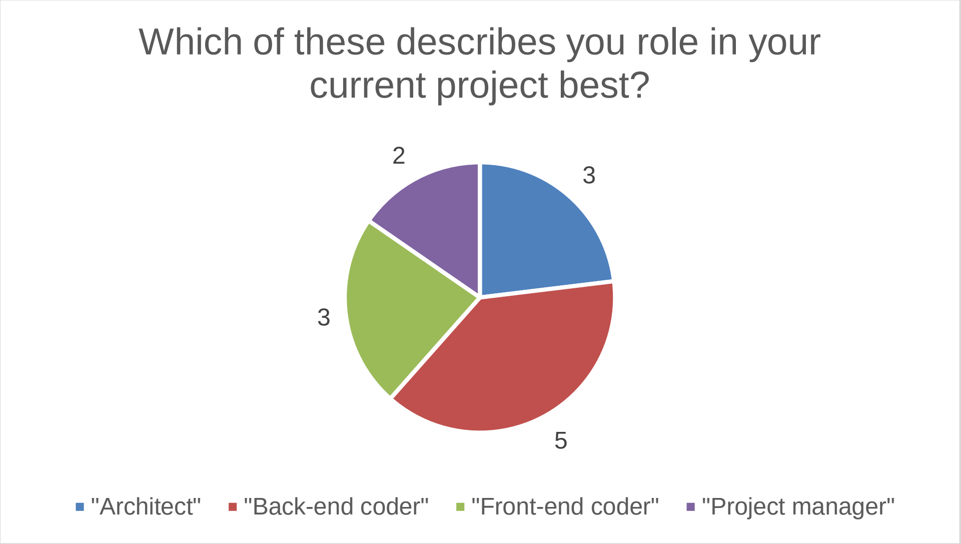
Baseline questionnaire results

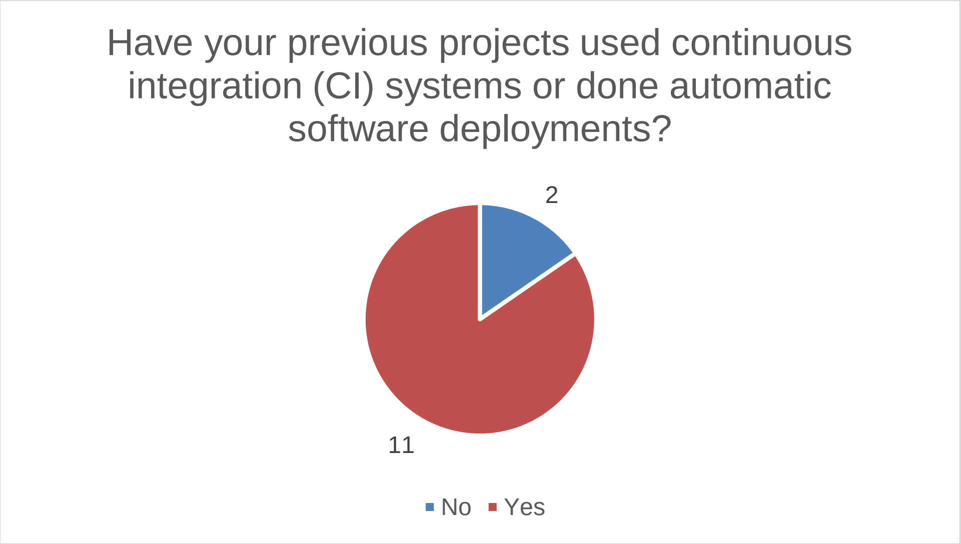
Original questions and answers were in Finnish, but have been translated to English with an attempt to maintain original language and meaning.

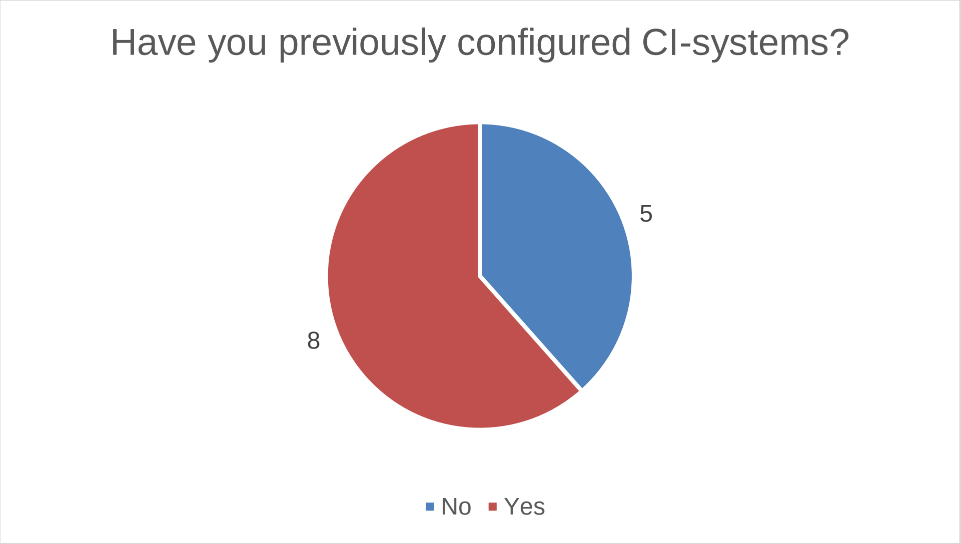


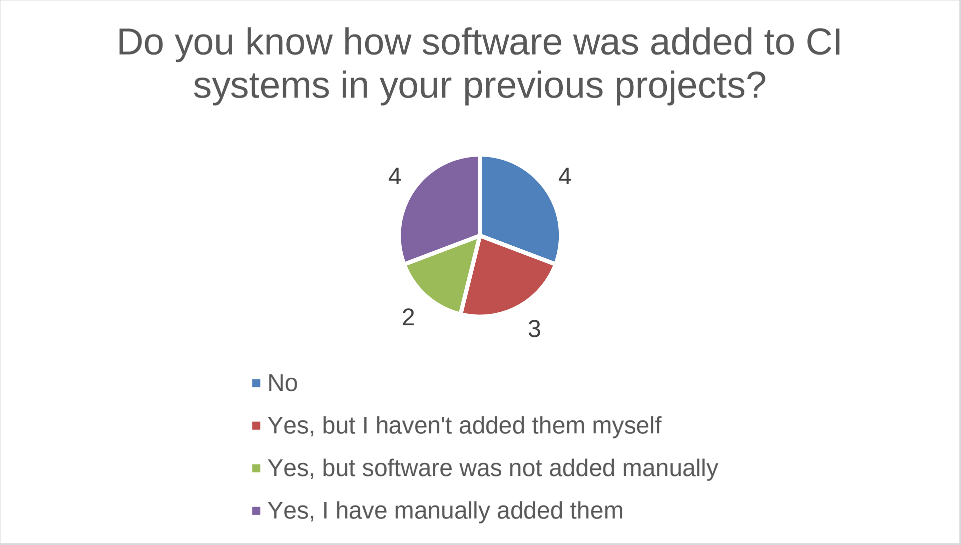


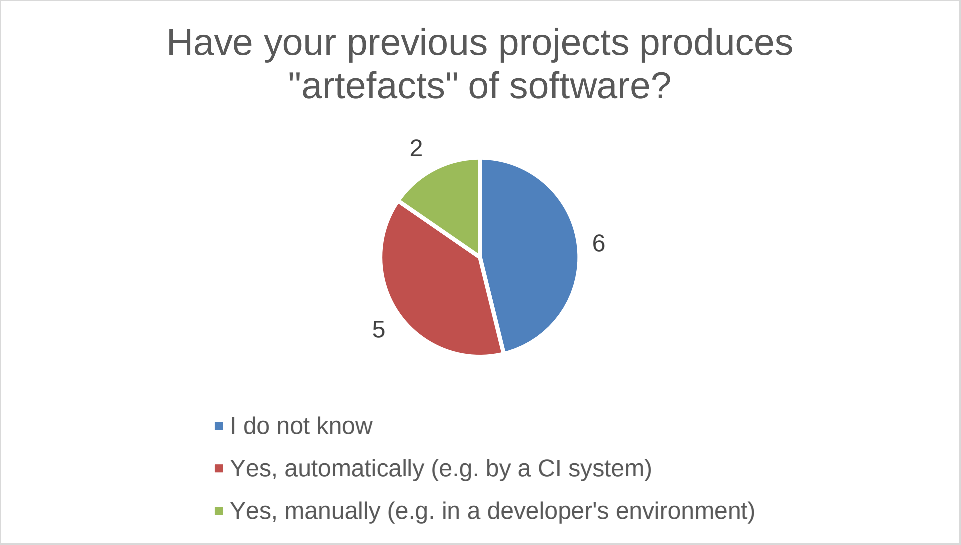


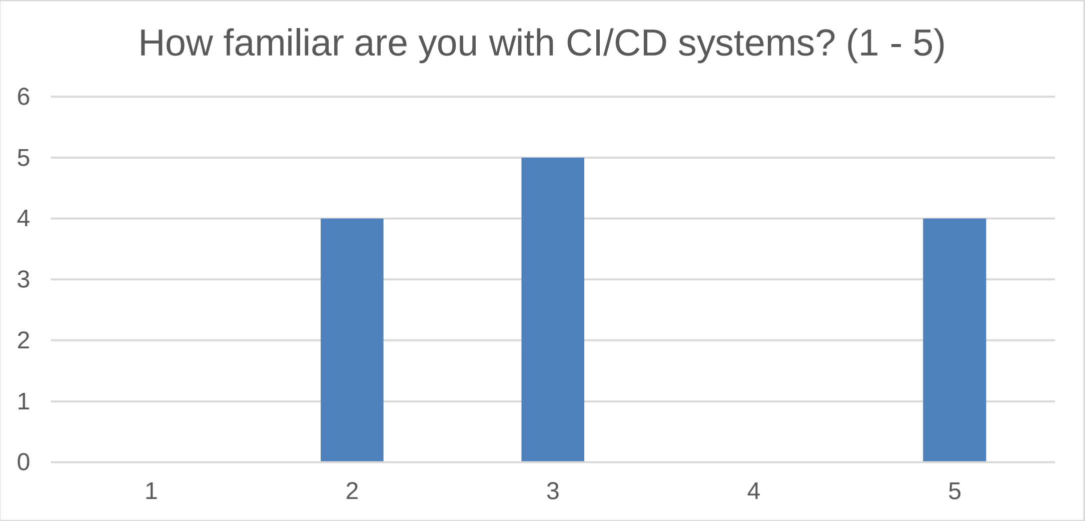

Which of these describes you role in your current project best?
2
3
3
5
■ "Architect"
■ "Back-end coder"
■ "Front-end coder"
■ "Project manager"

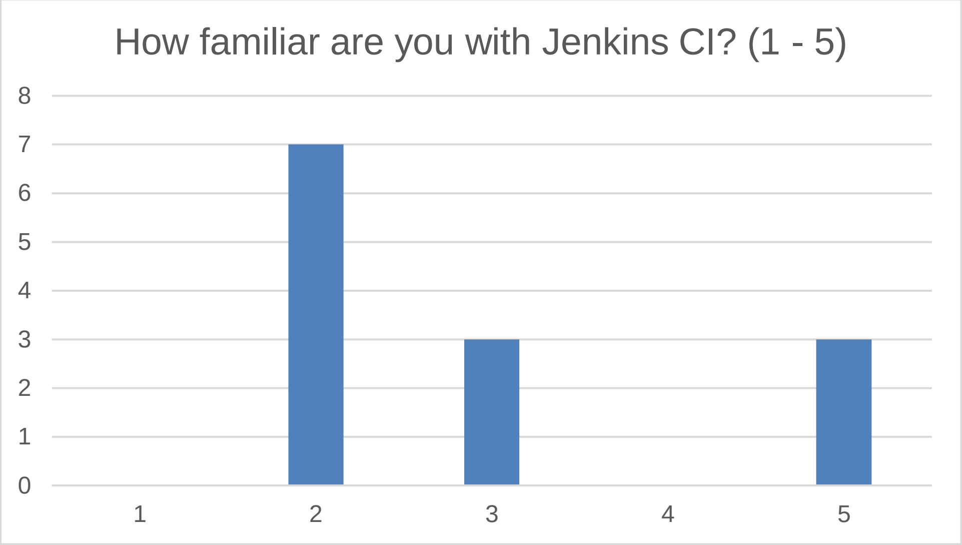

Have your previous projects used continuous integration (CI) systems or done automatic software deployments?
2
11
■ No
■ Yes

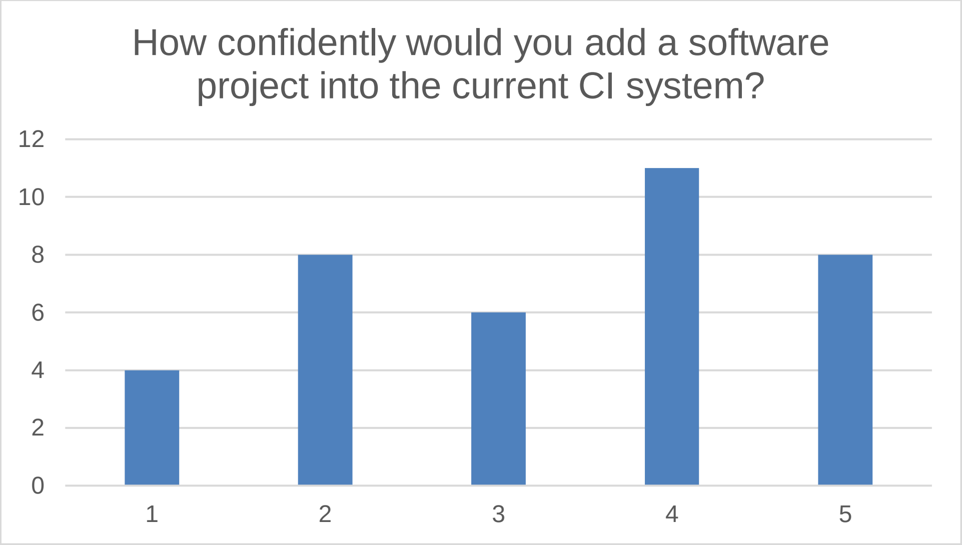

Have you previously configured CI-systems?
5
8
■ No
■ Yes

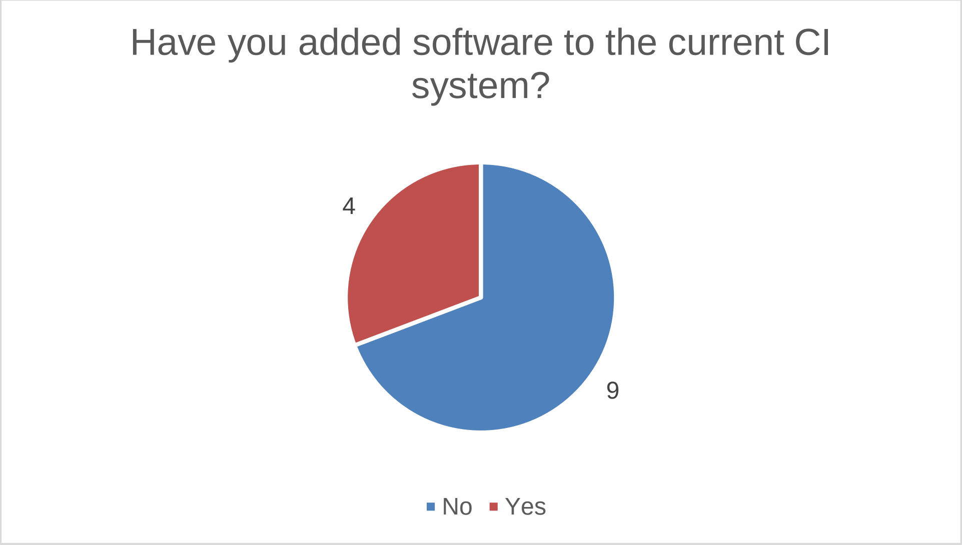

Do you know how software was added to CI systems in your previous projects?

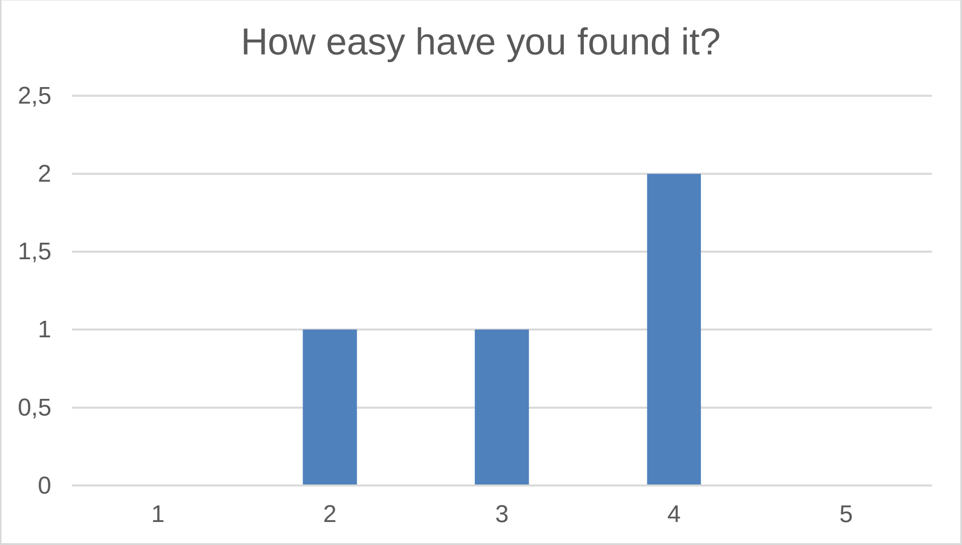

Have your previous projects produces "artefacts" of software?

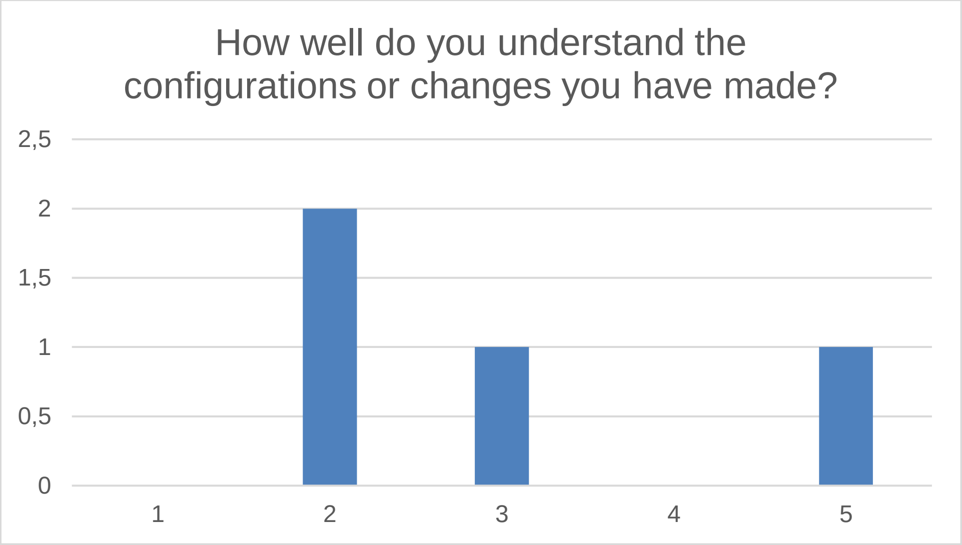

How familiar are you with CI/CD systems? (1 - 5)

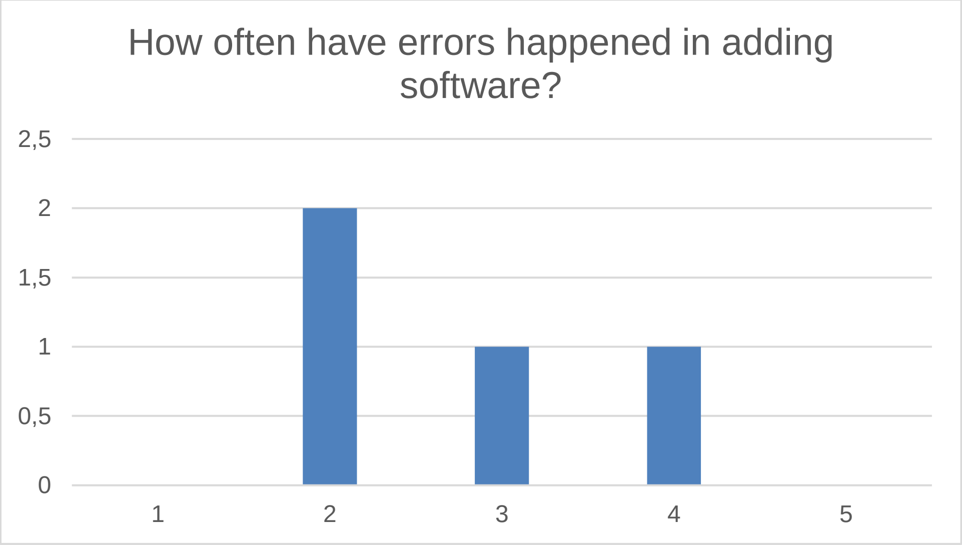

How familiar are you with Jenkins CI? (1 - 5)
8
7
6
5
4
3
2
1
0
1
2
3
4
5

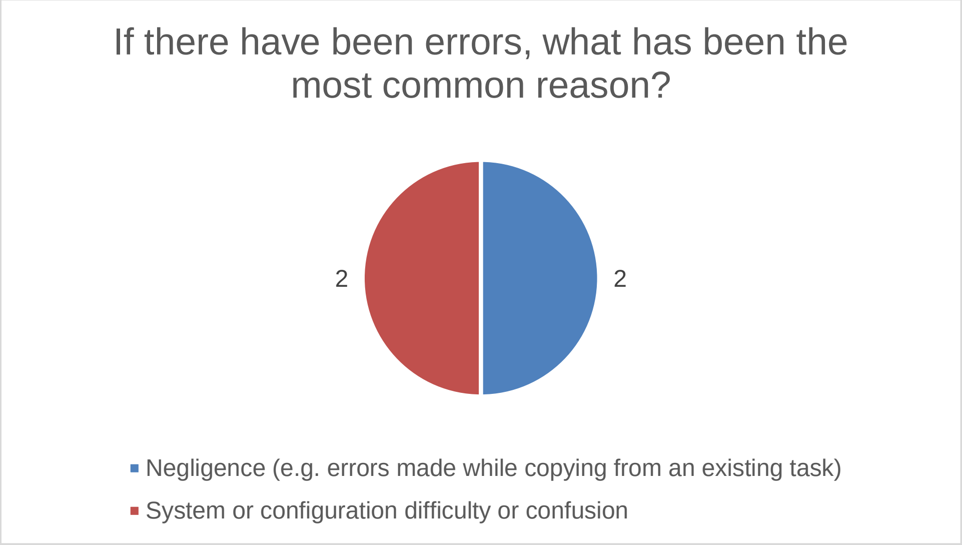

How confidently would you add a software project into the current CI system?
12
10
8
6
4
2
0
1
2
3
4
5


Have you added software to the current CI system?
4
9
■ No ■ Yes

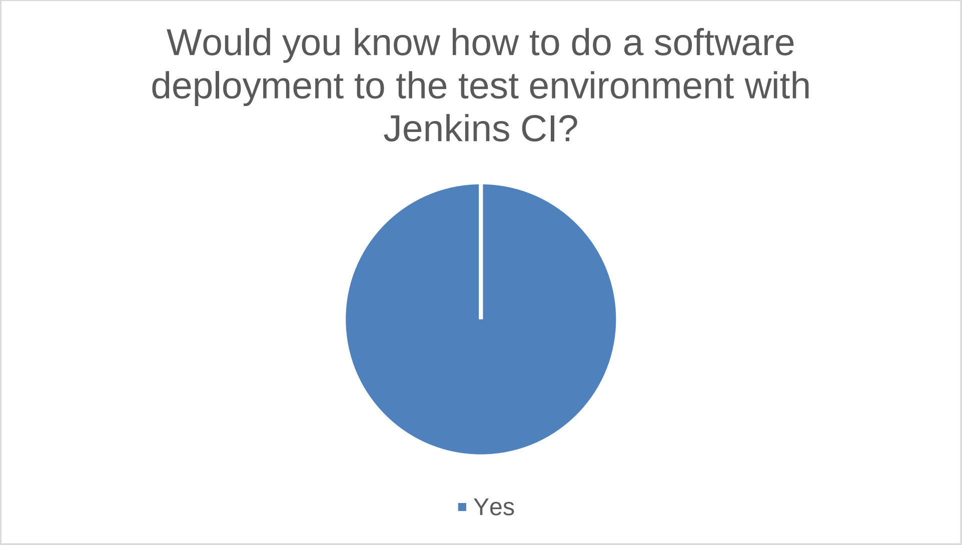

How easy have you found it?
2,5
2
1,5
1
0,5
0
1
2
3
4
5


How well do you understand the configurations or changes you have made?
2,5
2
1,5
1
0,5
0
1
2
3
4
5

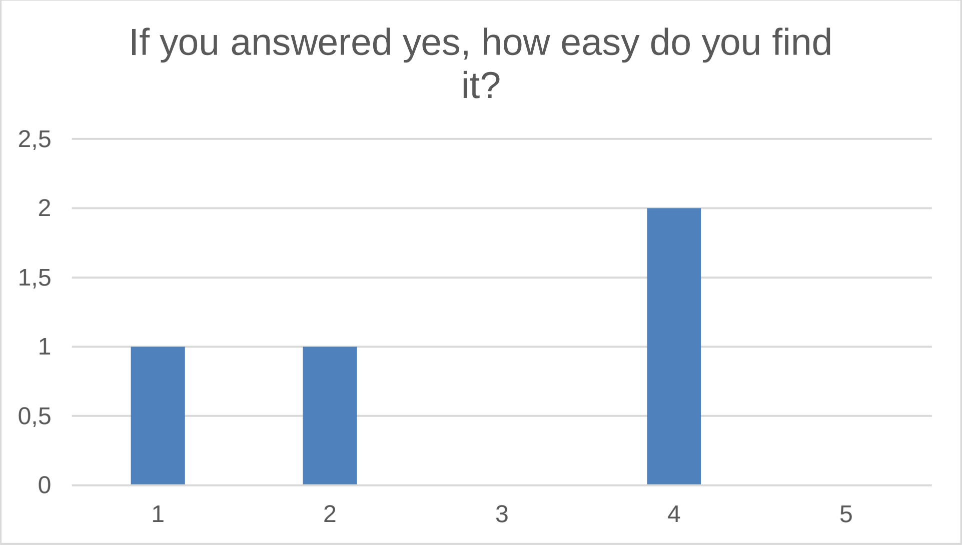

How often have errors happened in adding software?
2,5
2
1,5
1
0,5
0
1
2
3
4
5



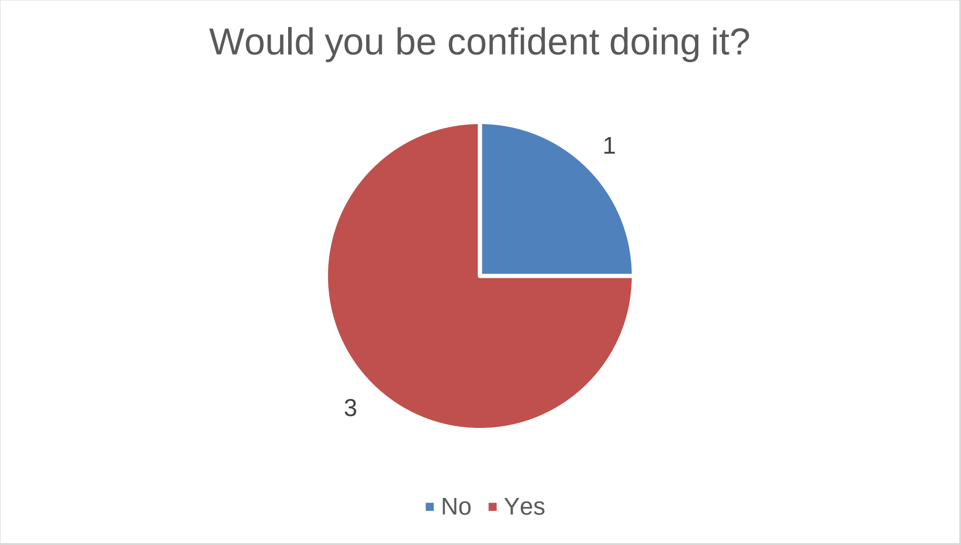
If there have been errors, what has been the most common reason?

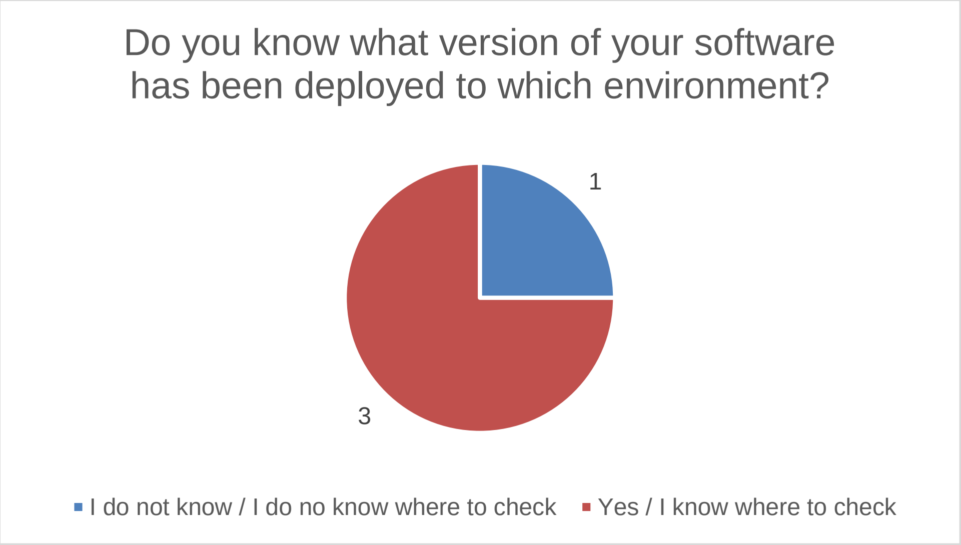


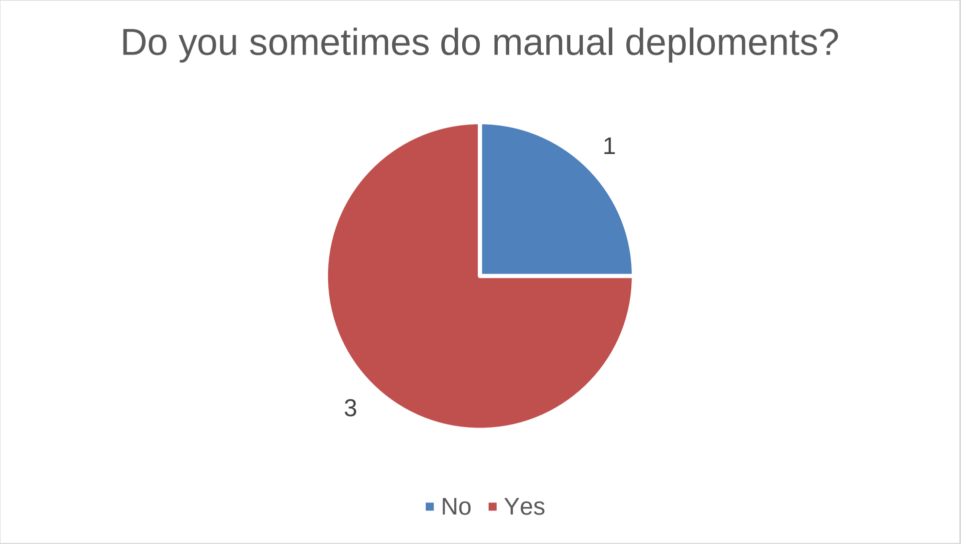
Would you know how to do a software deployment to the test environment with Jenkins CI?

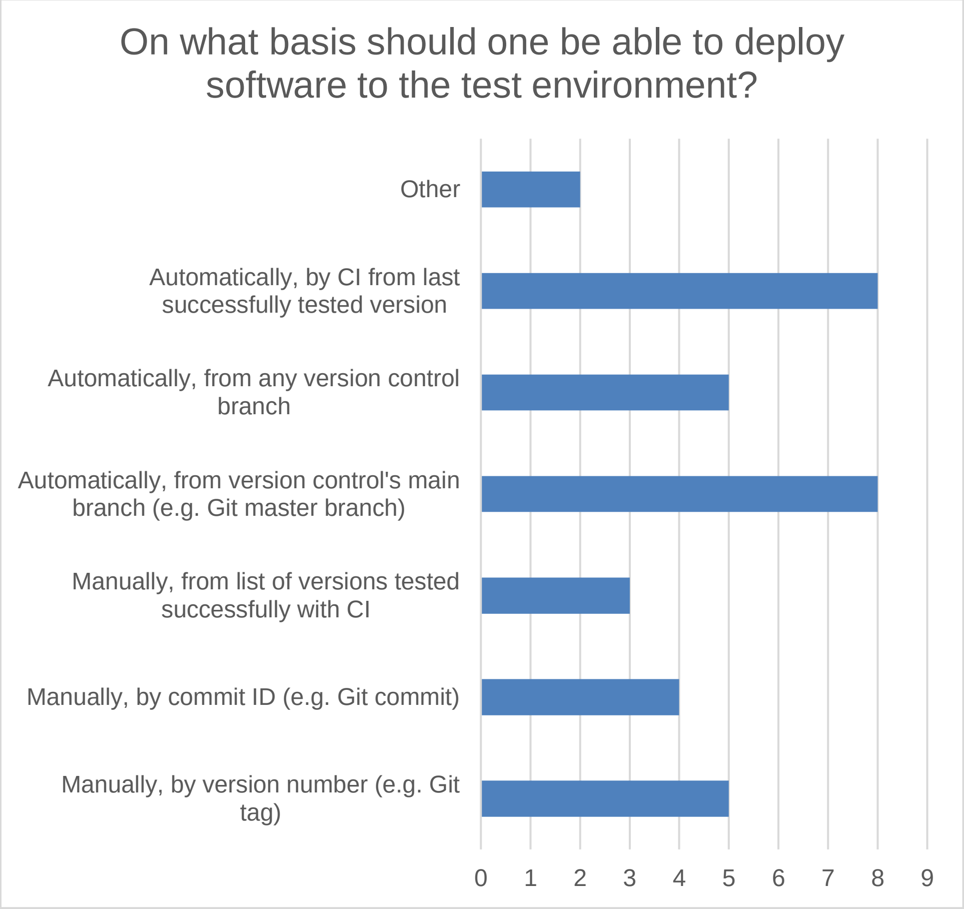


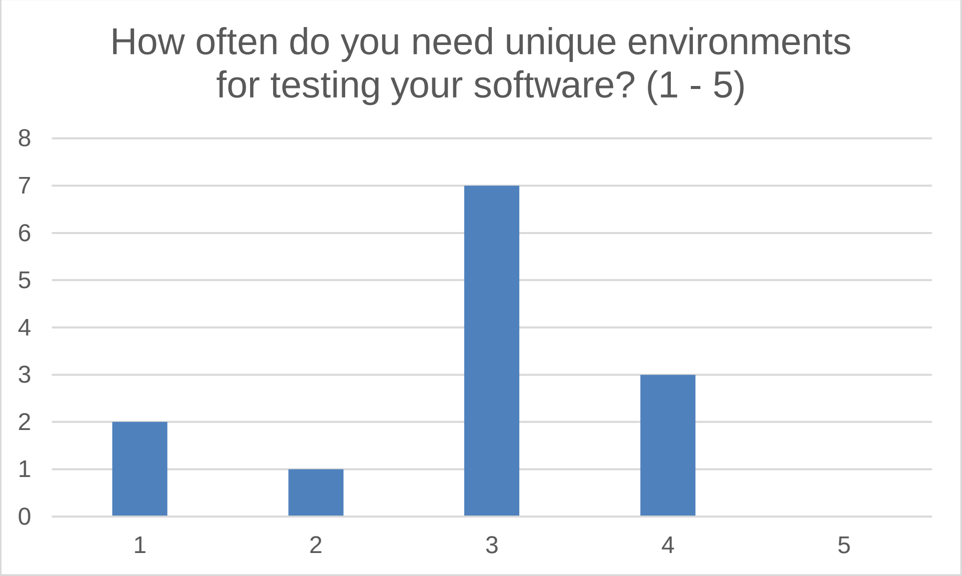
If you answered yes, how easy do you find it?

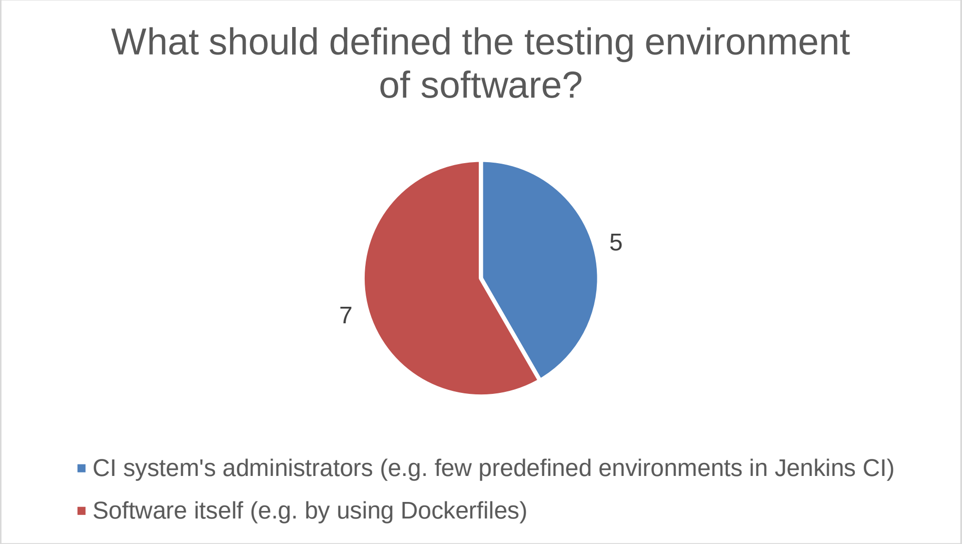

Would you be confident doing it?
1
3
No
Yes

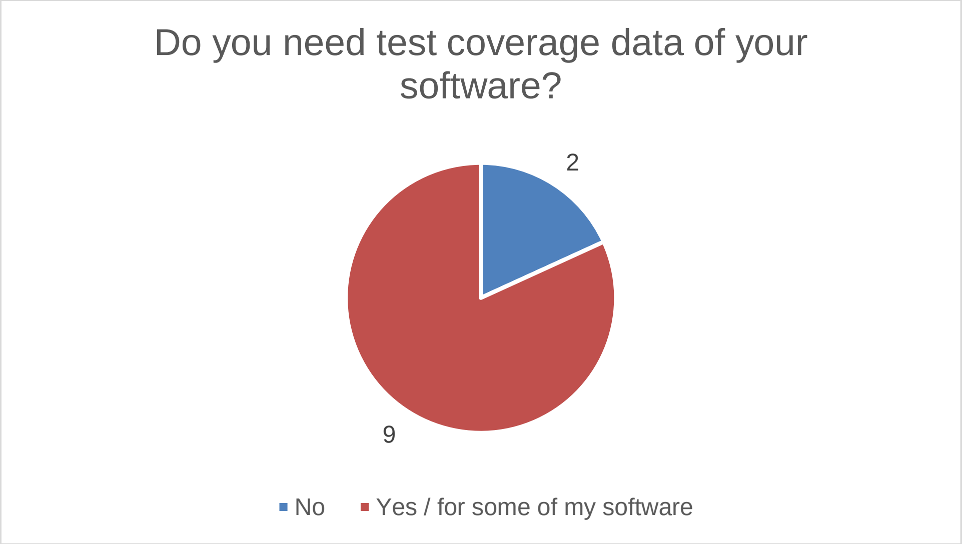

Do you know what version of your software has been deployed to which environment?
1
3
I do not know / I do no know where to check
Yes / I know where to check

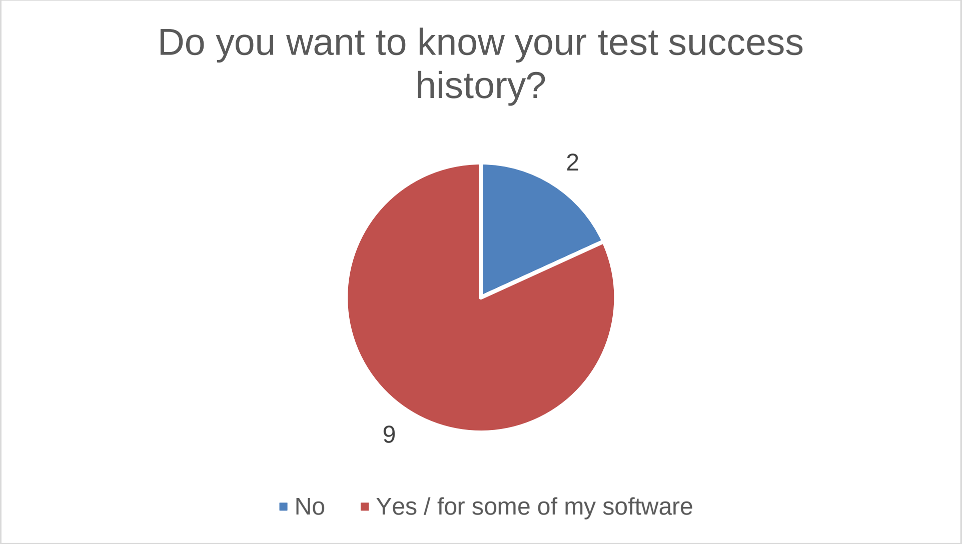

Do you sometimes do manual deploments?
1
3
No
Yes

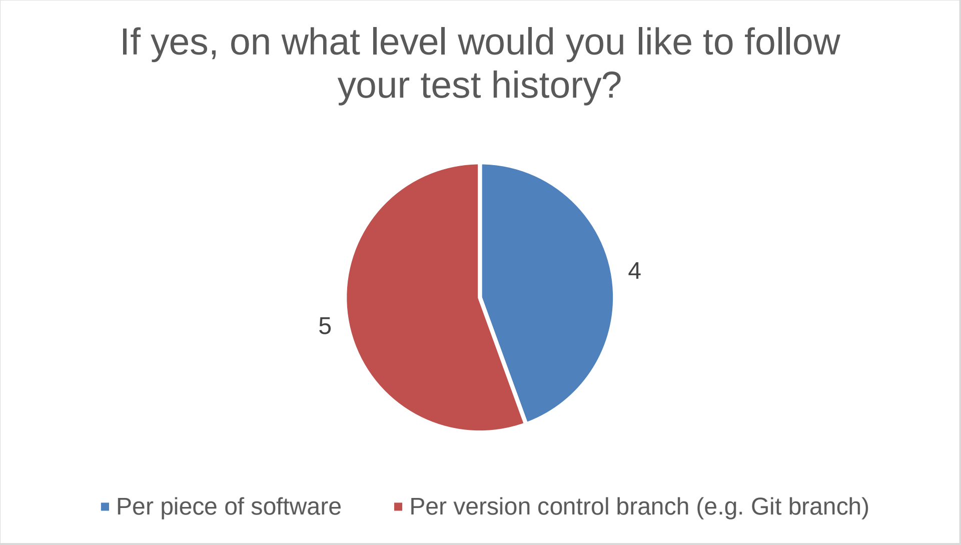

On what basis should one be able to deploy software to the test environment?
Other
Automatically, by CI from last successfully tested version
Automatically, from any version control branch
Automatically, from version control's main branch (e.g. Git master branch)
Manually, from list of versions tested successfully with CI
Manually, by commit ID (e.g. Git commit)
Manually, by version number (e.g. Git tag)
0
1
2
3
4
5
6
7
8
9

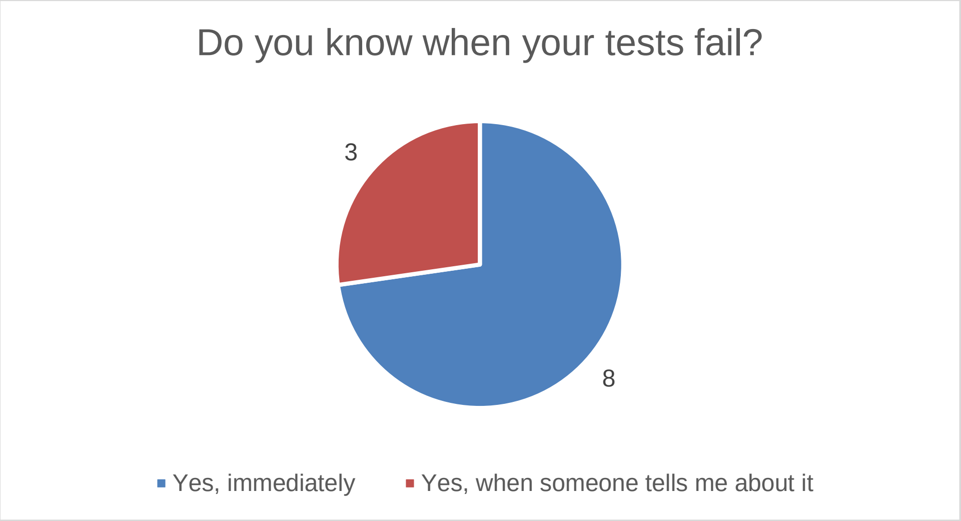

How often do you need unique environments for testing your software? (1 - 5)
8
7
6
5
4
3
2
1
0
1
2
3
4
5

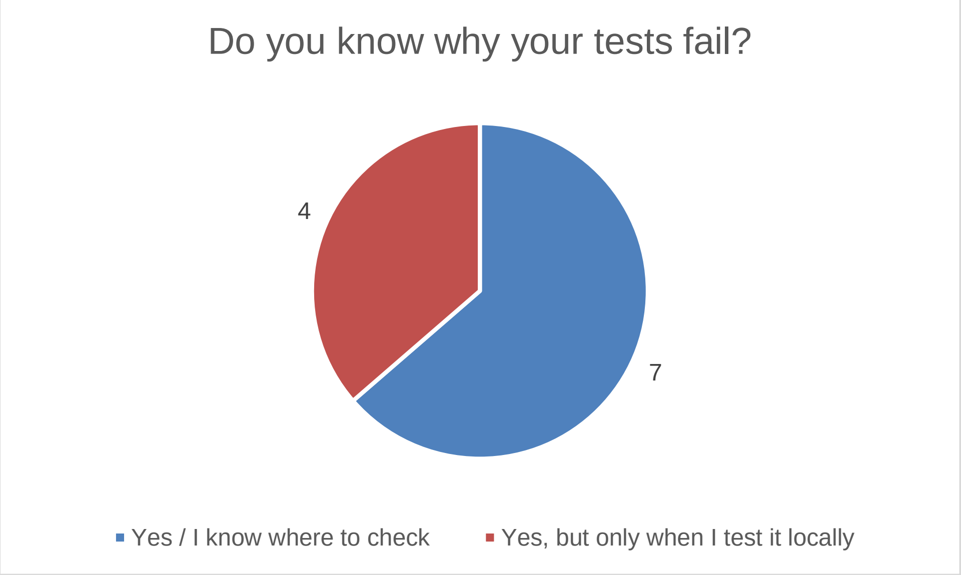

What should defined the testing environment of software?
5
7
■ CI system's administrators (e.g. few predefined environments in Jenkins CI)
■ Software itself (e.g. by using Dockerfiles)

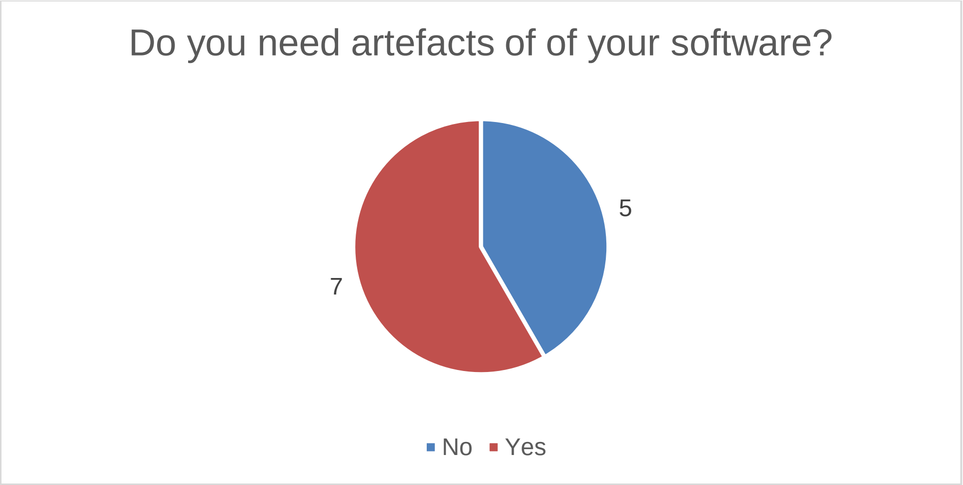

Do you need test coverage data of your software?
2
9
■ No
■ Yes / for some of my software

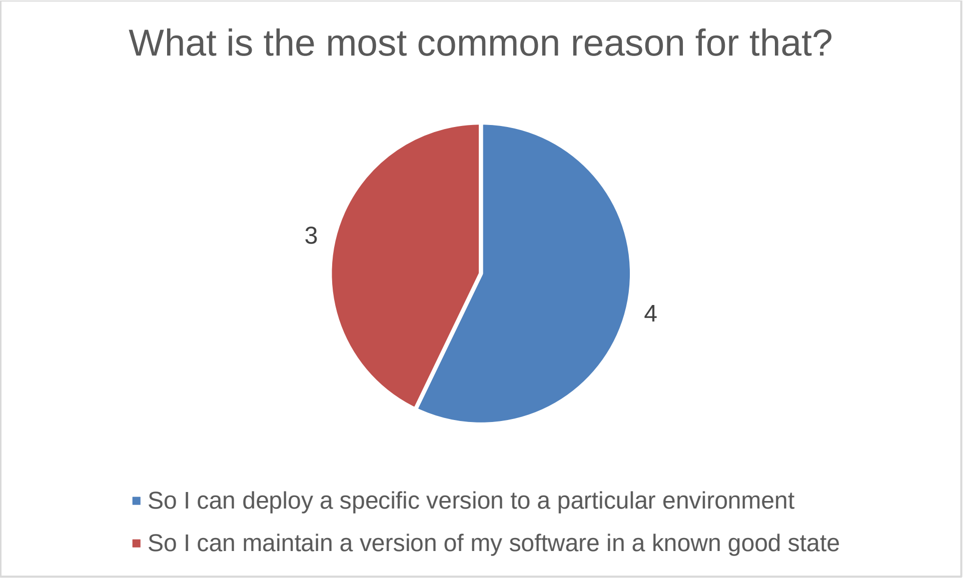

Do you want to know your test success history?
2
9
■ No
■ Yes / for some of my software

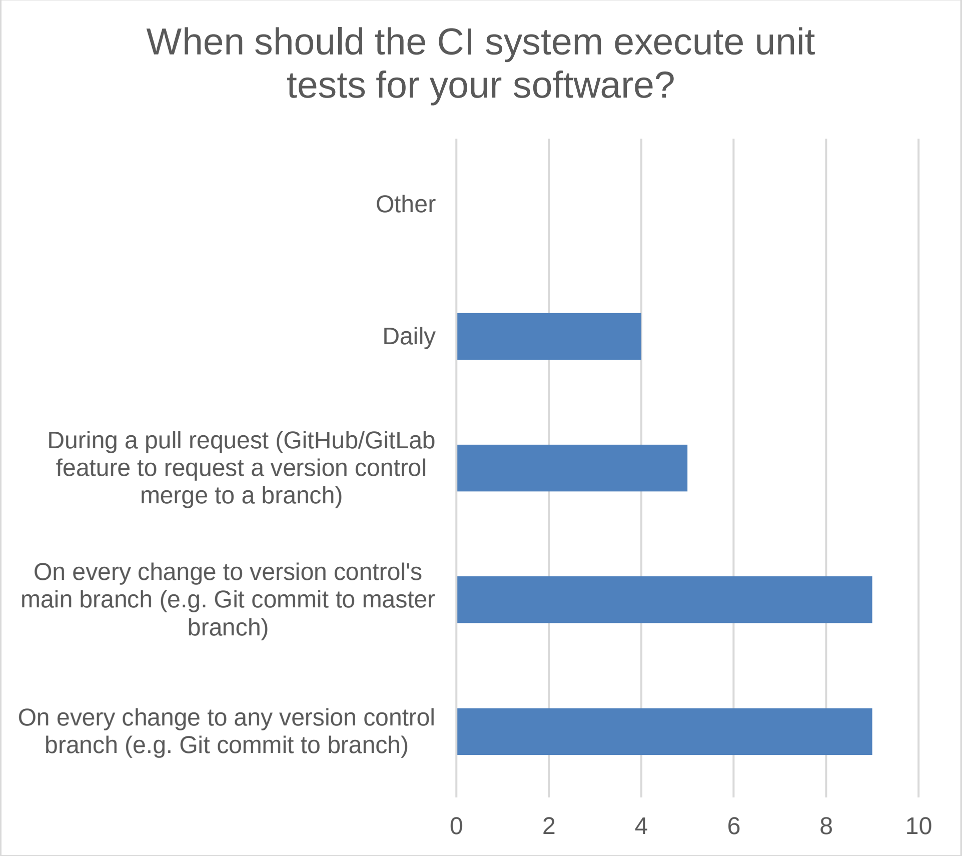

If yes, on what level would you like to follow your test history?
4
5
■ Per piece of software
■ Per version control branch (e.g. Git branch)

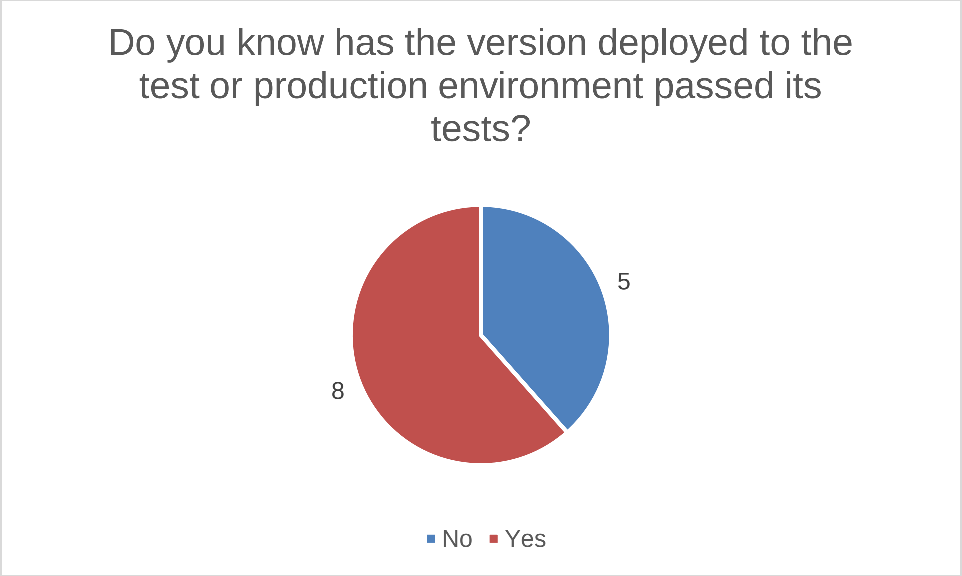

Do you know when your tests fail?
3
8
■ Yes, immediately
■ Yes, when someone tells me about it

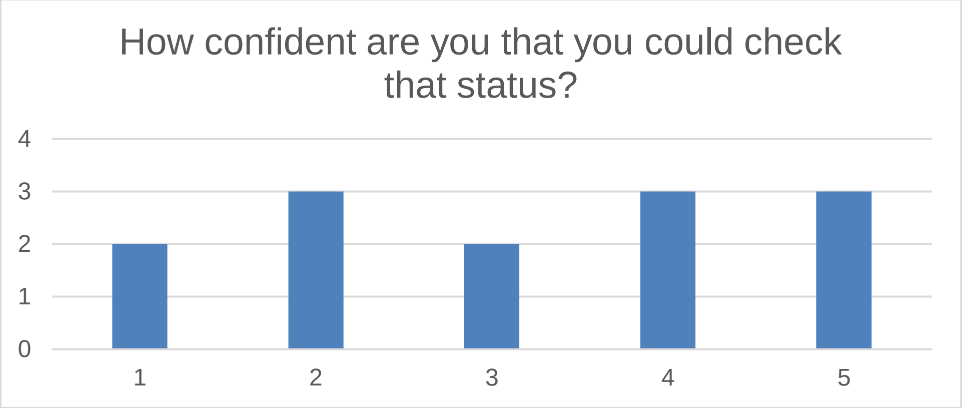

Do you know why your tests fail?
4
7
■ Yes / I know where to check
■ Yes, but only when I test it locally

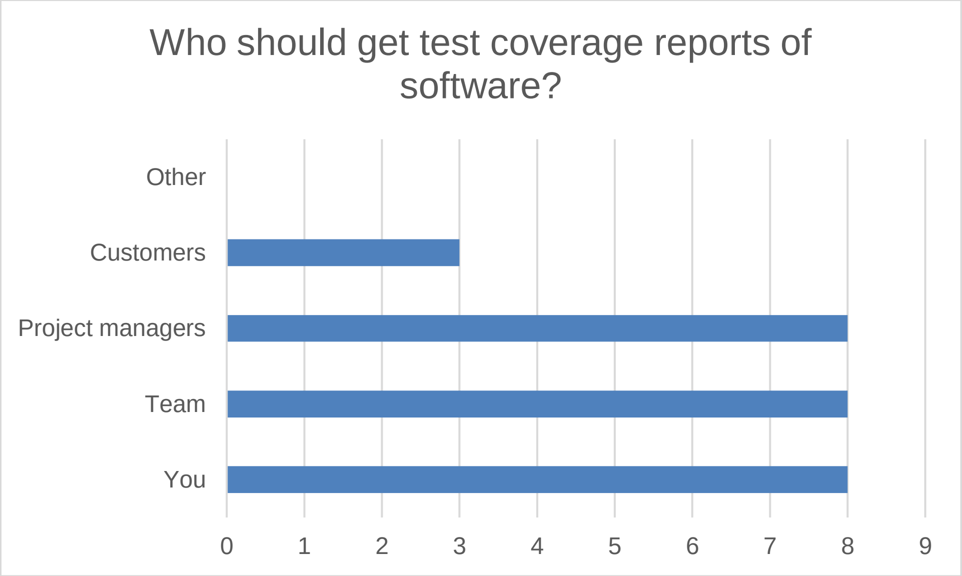

Do you need artefacts of of your software?
5
7
■ No
■ Yes

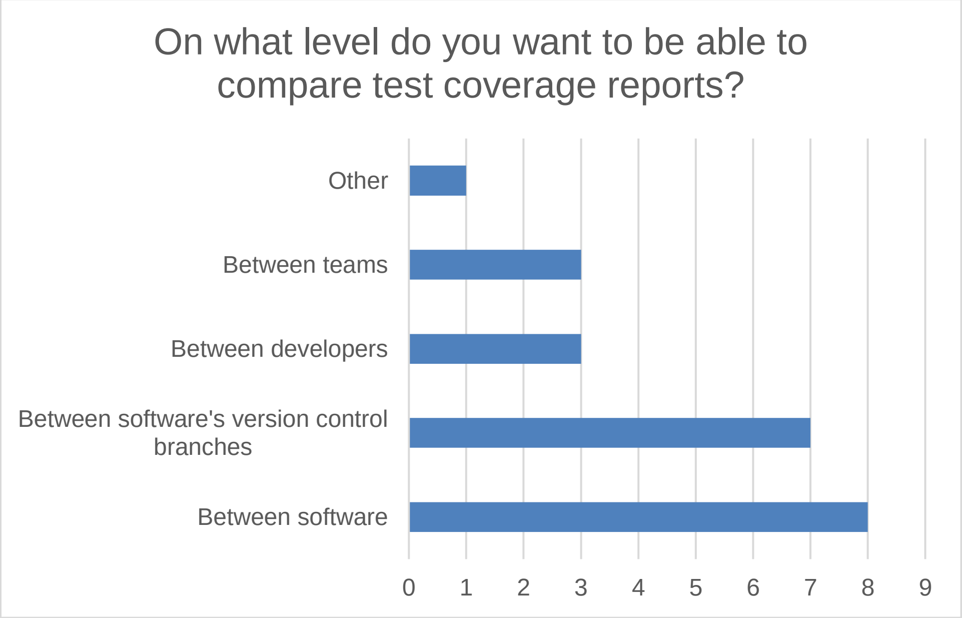

What is the most common reason for that?
3
4
■ So I can deploy a specific version to a particular environment
■ So I can maintain a version of my software in a known good state


When should the CI system execute unit tests for your software?
Other
Daily
During a pull request (GitHub/GitLab feature to request a version control merge to a branch)
On every change to version control's main branch (e.g. Git commit to master branch)
On every change to any version control branch (e.g. Git commit to branch)
0
2
4
6
8
10


Do you know has the version deployed to the test or production environment passed its tests?
5
8
No
Yes


How confident are you that you could check that status?
4
3
2
1
0
1
2
3
4
5


Who should get test coverage reports of software?
Other
Customers
Project managers
Team
You
0
1
2
3
4
5
6
7
8
9


On what level do you want to be able to compare test coverage reports?
Other
Between teams
Between developers
Between software's version control branches
Between software
0
1
2
3
4
5
6
7
8
9

Docker Bench for Security v1.1.0 report

```
# ------------------------------------------------------------------------------
# Docker Bench for Security v1.1.0
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker 1.11 Benchmark:
# https://benchmarks.cisecurity.org/downloads/show-single/index.cfm?file=docker16.110
# ------------------------------------------------------------------------------

Initializing Wed Nov 23 12:38:58 EET 2016


[INFO] 1 - Host Configuration
[WARN] 1.1  - Create a separate partition for containers
[PASS] 1.2  - Use an updated Linux Kernel
[PASS] 1.4  - Remove all non-essential services from the host - Network
[PASS] 1.5  - Keep Docker up to date
[INFO]       * Using 1.12.3 which is current as of 2016-10-26
[INFO]       * Check with your operating system vendor for support and security maintenance for docker
[INFO] 1.6  - Only allow trusted users to control Docker daemon
[INFO]      * docker:x:999:mikko
[PASS] 1.7  - Audit docker daemon - /usr/bin/docker
[PASS] 1.8  - Audit Docker files and directories - /var/lib/docker
[PASS] 1.9  - Audit Docker files and directories - /etc/docker
[INFO# ------------------------------------------------------------------------------
# Docker Bench for Security v1.1.0
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker 1.11 Benchmark:
# https://benchmarks.cisecurity.org/downloads/show-single/index.cfm?file=docker16.110
# ------------------------------------------------------------------------------

Initializing Wed Nov 23 12:38:58 EET 2016


[INFO] 1 - Host Configuration
```

```
[WARN] 1.1  - Create a separate partition for containers
[PASS] 1.2  - Use an updated Linux Kernel
[PASS] 1.4  - Remove all non-essential services from the host - Network
[PASS] 1.5  - Keep Docker up to date
[INFO]       * Using 1.12.3 which is current as of 2016-10-26
[INFO]       * Check with your operating system vendor for support and security maintenance for docker
[INFO] 1.6  - Only allow trusted users to control Docker daemon
[INFO]      * docker:x:999:mikko
[PASS] 1.7  - Audit docker daemon - /usr/bin/docker
[PASS] 1.8  - Audit Docker files and directories - /var/lib/docker
[PASS] 1.9  - Audit Docker files and directories - /etc/docker
[INFO] 1.10 - Audit Docker files and directories - docker.service
[INFO]      * File not found
[INFO] 1.11 - Audit Docker files and directories - docker.socket
[INFO]      * File not found
[PASS] 1.12 - Audit Docker files and directories - /etc/default/docker
[INFO] 1.13 - Audit Docker files and directories - /etc/docker/daemon.json
[INFO]      * File not found
[PASS] 1.14 - Audit Docker files and directories - /usr/bin/docker-containerd
[PASS] 1.15 - Audit Docker files and directories - /usr/bin/docker-runc


[INFO] 2 - Docker Daemon Configuration
[WARN] 2.1  - Restrict network traffic between containers
[PASS] 2.2  - Set the logging level
[PASS] 2.3  - Allow Docker to make changes to iptables
[PASS] 2.4  - Do not use insecure registries
[WARN] 2.5  - Do not use the aufs storage driver
[INFO] 2.6  - Configure TLS authentication for Docker daemon
[INFO]      * Docker daemon not listening on TCP
[INFO] 2.7 - Set default ulimit as appropriate
[INFO]      * Default ulimit doesn't appear to be set
[WARN] 2.8  - Enable user namespace support
[PASS] 2.9  - Confirm default cgroup usage
[PASS] 2.10 - Do not change base device size until needed
[WARN] 2.11 - Use authorization plugin
[WARN] 2.12 - Configure centralized and remote logging
[PASS] 2.13 - Disable operations on legacy registry (v1)


[INFO] 3 - Docker Daemon Configuration Files
[INFO] 3.1  - Verify that docker.service file ownership is set to root:root
[INFO]      * File not found
```

```
[INFO] 3.2  - Verify that docker.service file permissions are
set to 644
[INFO]      * File not found
[INFO] 3.3  - Verify that docker.socket file ownership is set to
root:root
[INFO]      * File not found
[INFO] 3.4  - Verify that docker.socket file permissions are set
to 644
[INFO]      * File not found
[PASS] 3.5  - Verify that /etc/docker directory ownership is set
to root:root
[PASS] 3.6  - Verify that /etc/docker directory permissions are
set to 755
[INFO] 3.7  - Verify that registry certificate file ownership is
set to root:root
[INFO]      * Directory not found
[INFO] 3.8  - Verify that registry certificate file permissions
are set to 444
[INFO]      * Directory not found
[INFO] 3.9  - Verify that TLS CA certificate file ownership is
set to root:root
[INFO]      * No TLS CA certificate found
[INFO] 3.10 - Verify that TLS CA certificate file permissions
are set to 444
[INFO]      * No TLS CA certificate found
[INFO] 3.11 - Verify that Docker server certificate file owner-
ship is set to root:root
[INFO]      * No TLS Server certificate found
[INFO] 3.12 - Verify that Docker server certificate file permis-
sions are set to 444
[INFO]      * No TLS Server certificate found
[INFO] 3.13 - Verify that Docker server key file ownership is
set to root:root
[INFO]      * No TLS Key found
[INFO] 3.14 - Verify that Docker server key file permissions are
set to 400
[INFO]      * No TLS Key found
[PASS] 3.15 - Verify that Docker socket file ownership is set to
root:docker
[PASS] 3.16 - Verify that Docker socket file permissions are set
to 660
[INFO] 3.17 - Verify that daemon.json file ownership is set to
root:root
[INFO]      * File not found
[INFO] 3.18 - Verify that daemon.json file permissions are set
to 644
[INFO]      * File not found
[PASS] 3.19 - Verify that /etc/default/docker file ownership is
set to root:root
[PASS] 3.20 - Verify that /etc/default/docker file permissions
are set to 644
```

```
[INFO] 4 - Container Images and Build Files
[INFO] 4.1  - Create a user for the container
[INFO]      * No containers running
[WARN] 4.5  - Enable Content trust for Docker


[INFO] 5  - Container Runtime
[INFO]      * No containers running, skipping Section 5


[INFO] 6  - Docker Security Operations
docker: "inspect" requires a minimum of 1 argument.
See 'docker inspect --help'.

Usage:  docker inspect [OPTIONS] CONTAINER|IMAGE|TASK [CON-
TAINER|IMAGE|TASK...]

Return low-level information on a container, image or task
[INFO] 6.4 - Avoid image sprawl
[INFO]      * There are currently: 1 images
[INFO] 6.5 - Avoid container sprawl
[INFO]      * There are currently a total of 0 containers, with
0 of them currently running

] 1.10 - Audit Docker files and directories - docker.service
[INFO]      * File not found
[INFO] 1.11 - Audit Docker files and directories - docker.socket
[INFO]      * File not found
[PASS] 1.12 - Audit Docker files and directories - /etc/de-
fault/docker
[INFO] 1.13 - Audit Docker files and directories -
/etc/docker/daemon.json
[INFO]      * File not found
[PASS] 1.14 - Audit Docker files and directories -
/usr/bin/docker-containerd
[PASS] 1.15 - Audit Docker files and directories -
/usr/bin/docker-runc


[INFO] 2 - Docker Daemon Configuration
[WARN] 2.1  - Restrict network traffic between containers
[PASS] 2.2  - Set the logging level
[PASS] 2.3  - Allow Docker to make changes to iptables
[PASS] 2.4  - Do not use insecure registries
[WARN] 2.5  - Do not use the aufs storage driver
[INFO] 2.6  - Configure TLS authentication for Docker daemon
[INFO]      * Docker daemon not listening on TCP
[INFO] 2.7 - Set default ulimit as appropriate
[INFO]      * Default ulimit doesn't appear to be set
[WARN] 2.8  - Enable user namespace support
[PASS] 2.9  - Confirm default cgroup usage
[PASS] 2.10 - Do not change base device size until needed
[WARN] 2.11 - Use authorization plugin
```

```
[WARN] 2.12 - Configure centralized and remote logging
[PASS] 2.13 - Disable operations on legacy registry (v1)


[INFO] 3 - Docker Daemon Configuration Files
[INFO] 3.1  - Verify that docker.service file ownership is set
to root:root
[INFO]      * File not found
[INFO] 3.2  - Verify that docker.service file permissions are
set to 644
[INFO]      * File not found
[INFO] 3.3  - Verify that docker.socket file ownership is set to
root:root
[INFO]      * File not found
[INFO] 3.4  - Verify that docker.socket file permissions are set
to 644
[INFO]      * File not found
[PASS] 3.5  - Verify that /etc/docker directory ownership is set
to root:root
[PASS] 3.6  - Verify that /etc/docker directory permissions are
set to 755
[INFO] 3.7  - Verify that registry certificate file ownership is
set to root:root
[INFO]      * Directory not found
[INFO] 3.8  - Verify that registry certificate file permissions
are set to 444
[INFO]      * Directory not found
[INFO] 3.9  - Verify that TLS CA certificate file ownership is
set to root:root
[INFO]      * No TLS CA certificate found
[INFO] 3.10 - Verify that TLS CA certificate file permissions
are set to 444
[INFO]      * No TLS CA certificate found
[INFO] 3.11 - Verify that Docker server certificate file owner-
ship is set to root:root
[INFO]      * No TLS Server certificate found
[INFO] 3.12 - Verify that Docker server certificate file permis-
sions are set to 444
[INFO]      * No TLS Server certificate found
[INFO] 3.13 - Verify that Docker server key file ownership is
set to root:root
[INFO]      * No TLS Key found
[INFO] 3.14 - Verify that Docker server key file permissions are
set to 400
[INFO]      * No TLS Key found
[PASS] 3.15 - Verify that Docker socket file ownership is set to
root:docker
[PASS] 3.16 - Verify that Docker socket file permissions are set
to 660
[INFO] 3.17 - Verify that daemon.json file ownership is set to
root:root
[INFO]      * File not found
```

```
[INFO] 3.18 - Verify that daemon.json file permissions are set
to 644
[INFO]      * File not found
[PASS] 3.19 - Verify that /etc/default/docker file ownership is
set to root:root
[PASS] 3.20 - Verify that /etc/default/docker file permissions
are set to 644


[INFO] 4 - Container Images and Build Files
[INFO] 4.1  - Create a user for the container
[INFO]      * No containers running
[WARN] 4.5  - Enable Content trust for Docker


[INFO] 5  - Container Runtime
[INFO]      * No containers running, skipping Section 5


[INFO] 6  - Docker Security Operations
docker: "inspect" requires a minimum of 1 argument.
See 'docker inspect --help'.

Usage:  docker inspect [OPTIONS] CONTAINER|IMAGE|TASK [CON-
TAINER|IMAGE|TASK...]

Return low-level information on a container, image or task
[INFO] 6.4 - Avoid image sprawl
[INFO]      * There are currently: 1 images
[INFO] 6.5 - Avoid container sprawl
[INFO]      * There are currently a total of 0 containers, with
0 of them currently running
```

Final Dockerfile for Jenkins

```
# Heavily based on the official Jenkins image:
# https://github.com/jen-
kinsci/docker/blob/79e871d7ea9085cabb287dd53705b4432e36cf6f/Dockerfile

FROM openjdk:8-jdk
LABEL maintainer="Mikko Piuhola <mikko.piuhola@digia.com>"

# Jenkins version being bundled in this docker image
ARG JENKINS_VERSION=2.47
# jenkins.war checksum, download will be validated using it
ARG JENKINS_SHA=16d7e0762964bd5fbc43a7ad5121cccf88fb4816

# Jenkins settings
ENV JENKINS_HOME=/var/jenkins_home
ENV JENKINS_SLAVE_AGENT_PORT=50000 \
    JENKINS_UC=https://updates.jenkins.io \
    COPY_REFERENCE_FILE_LOG=${JENKINS_HOME}/copy_reference_file.log \
    TINI_VERSION=0.9.0 \
    TINI_SHA=fa23d1e20732501c3bb8eeeca423c89ac80ed452 \
    DOCKER_VERSION=1.12.5 \

DOCKER_SHA256=0058867ac46a1eba283e2441b1bb5455df846144f9d9ba079e976553
99d4a2c6 \
    JENKINS_OPTS=-Djenkins.install.runSetupWizard=false \
    SEED_JOBS_DIR=/usr/share/jenkins/ref/seed-jobs \
    CONF_GITHUB_TOKEN_ID=github-org-token

# Can be used to customize where jenkins.war get downloaded from
ARG JENKINS_URL=https://repo.jenkins-ci.org/public/org/jenkins-
ci/main/jenkins-war/${JENKINS_VERSION}/jenkins-war-${JENKINS_VER-
SION}.war

# Names & IDs for Jenkins user
ARG user=jenkins
ARG group=jenkins
ARG uid=1000
ARG gid=1000

# Install general and build-time dependencies,
# clear apt cache (no need to leave in image, wasting space).
# Jenkins is run with user `jenkins`, uid = 1000
# If you bind mount a volume from the host or a data container,
# ensure you use the same uid.
# `/usr/share/jenkins/ref/` contains all reference configuration we
want
# to set on a fresh new installation. Use it to bundle additional
plugins
# or config file with your custom jenkins Docker image.
RUN apt-get update && apt-get install -y \
    git \
    curl \
    && rm -rf /var/lib/apt/lists/* \
    && groupadd -g ${gid} ${group} \
    && useradd -d "$JENKINS_HOME" -u ${uid} -g ${gid} -m -s /bin/bash
${user} \
    && mkdir -p /usr/share/jenkins/ref/init.groovy.d \
    && mkdir -p /usr/share/jenkins/ref/seed-jobs

# Use tini as subreaper in Docker container to adopt zombie processes.
# Next, install Jenkins itself.
# Install Docker CLI to enable usage of Docker inside Jenkins jobs.
RUN curl -fsSL https://github.com/krallin/tini/releases/download/v${TINI_VERSION}/tini-static -o /bin/tini && chmod +x /bin/tini \
    && echo "$TINI_SHA  /bin/tini" | sha1sum -c - \
    && curl -fsSL ${JENKINS_URL} -o /usr/share/jenkins/jenkins.war \
    && echo "${JENKINS_SHA}  /usr/share/jenkins/jenkins.war" | sha1sum
-c - \
    && chown -R ${user} "$JENKINS_HOME" /usr/share/jenkins/ref \
    && curl -fsSLO https://get.docker.com/builds/Linux/x86_64/docker-
${DOCKER_VERSION}.tgz \
    && echo "${DOCKER_SHA256}  docker-${DOCKER_VERSION}.tgz" |
sha256sum -c - \
    && tar --strip-components=1 -xvzf docker-${DOCKER_VERSION}.tgz -C
/usr/local/bin \
    && chmod +rx /usr/local/bin/docker \
    && chmod +s /usr/local/bin/docker

# Jenkins home directory is a volume, so configuration and build history
# can be persisted and survive image upgrades
VOLUME /var/jenkins_home

# Web UI & slave machines:
EXPOSE 8080 50000

# No need to give Jenkins root privileges,
# run as custom user.
USER ${user}

# From a derived Dockerfile one can use `RUN plugins.sh active.txt`
# to setup /usr/share/jenkins/ref/plugins from a support bundle.
COPY scripts/ /usr/local/bin/

# Install the initial plugins
COPY plugins.txt /usr/share/jenkins/plugins.txt
RUN /usr/local/bin/install-plugins.sh $(cat /usr/share/jenkins/plugins.txt | tr '\n' ' ')

# Copy after installing to prevent unnecessary
# extra layers from plugin installs.
COPY init.groovy.d/ /usr/share/jenkins/ref/init.groovy.d/
COPY seed-jobs/ /usr/share/jenkins/ref/seed-jobs/

ENTRYPOINT ["/bin/tini", "--", "/usr/local/bin/jenkins.sh"]
```

Groovy script to set up a GitHub Organization Folder

```
#!/usr/bin/env groovy

// Seed a single GitHub organization multibranch folder
// that auto-scans all repositories and their branches.

import javaposse.jobdsl.dsl.*

// Include access to Jenkins.instance for inspecting e.g `Jenkins.in-
stance.allItems.each{p-> println "job:" +p.name}` or `Jenkins.in-
stance.getItemByFullName('somejobname')`
import jenkins.model.*
import hudson.model.*

def env = System.getenv()
def jobName = env['CONF_GITHUB_ORG'] // Use organization's name as the
project name

// Need to use Jenkins Credentials global secret username-password for
Github username and Github Personal Access Token as follows:
// https://github.com/settings/tokens/new?scopes=repo,public_repo,ad-
min:repo_hook,admin:org_hook&description=Jenkins+Access
def jenkinsGithubuserAccesstokenId = 'github-org-token'

def seedJob = job(jobName) {
    displayName jobName
    configure { project ->
        // This job config.xml is for a Jenkins 2.0 type Github Organ-
ization Folder
        project.name = 'jenkins.branch.OrganizationFolder'
        // Jenkins will automatically pin plugin version attributes
for the config.xml, but can also set attributes for top node like
this: project.attributes()['attrib'] = 'something'
        project / 'projectFactories' / 'org.jenkinsci.plugins.work-
flow.multibranch.WorkflowMultiBranchProjectFactory' {}
        project / 'icon' (class: 'jenkins.branch.MetadataActionFold-
erIcon') / 'owner' (class: 'jenkins.branch.OrganizationFolder', refer-
ence: '../..') // sets class and reference attributes
        project / 'properties' / 'jenkins.branch.NoTriggerOrganiza-
tionFolderProperty' {
            branches '.*'
        }
        project / 'folderViews' (class: 'jenkins.branch.Organization-
FolderViewHolder') / 'owner' (reference: '../..')
        project / 'healthMetrics' / 'com.cloudbees.hud-
son.plugins.folder.health.WorstChildHealthMetric'
        project / 'orphanedItemStrategy' (class: 'com.cloudbees.hud-
son.plugins.folder.computed.DefaultOrphanedItemStrategy') {
            pruneDeadBranches true
            daysToKeep 0
            numToKeep 3
        }
        // Trigger repository scans every 1 hour
        project / 'triggers' / 'com.cloudbees.hud-
son.plugins.folder.computed.PeriodicFolderTrigger' {
            spec 'H * * * *'
            interval 3600000
        }
        project / 'navigators' / 'org.jen-
kinsci.plugins.github__branch__source.GitHubSCMNavigator' {
            repoOwner "$jobName"
            scanCredentialsId "$jenkinsGithubuserAccesstokenId"
            checkoutCredentialsId 'SAME'
            pattern '.*'
            buildOriginBranch true
            buildOriginBranchWithPR true
            buildOriginPRMerge false
            buildOriginPRHead false
            buildForkPRMerge false
            buildForkPRHead false
        }

        // For Jenkins 2.0 Organization Folder job - some of the Free-
styleJob generated config.xml is not needed.
        project.remove(project / scm)
        project.remove(project / publishers)
        project.remove(project / builders)
        project.remove(project / buildWrappers)
        project.remove(project / concurrentBuild)
        project.remove(project / blockBuildWhenDownstreamBuilding)
        project.remove(project / blockBuildWhenUpstreamBuilding)
        project.remove(project / keepDependencies)
        project.remove(project / canRoam)
        project.remove(project / disabled)
    }
}

// Want to immediately schedule a build of the new job for computing
sub-folders
// so that branches and pull-requests of any origin get included.
queue(seedJob) // https://github.com/jenkinsci/job-dsl-
plugin/wiki/Job-DSL-Commands
```

Declarative pipeline example

```
#!/usr/bin/env groovy
@Library('thesisSampleLib')
import org.thesis_ci_automation_test.*

def slack = new SlackNotifier()
def utils = new Utils()
def dockerArgs = '-v /var/run/docker.sock:/var/run/docker.sock'

pipeline {
  agent none

  options {
    buildDiscarder(logRotator(numToKeepStr:'5'))
    ansiColor('xterm')
  }

  stages {
    stage('Build') {
      agent {
        dockerfile {
          filename 'Dockerfile.test'
          args dockerArgs
        }
      }

      steps {
        sh 'npm run dependencies'
      }
    }

    stage('Test') {
      agent {
        dockerfile {
          filename 'Dockerfile.test'
          args dockerArgs
        }
      }

      steps {
        sh 'grunt unit'
      }

      post {
        always {
          junit 'test-results/**/unit-test-results.xml'
        }
      }
    }

    stage('Prepare dev deploy') {
      agent {
        dockerfile {
          filename 'Dockerfile.test'
          args dockerArgs
        }
      }

      when {
        branch 'dev'
      }

      steps {
        sh 'npm run build:dev'
      }
    }

    stage('Development deploy') {
      agent {
        dockerfile {
          filename 'Dockerfile.test'
          args dockerArgs
        }
      }

      when {
        branch 'dev'
      }

      steps {
        milestone 1
        lock(resource: 'dev-server', inversePrecedence: true) {
          milestone 2
          retry(count: 3) {
            sh './deploy.dev.sh'
          }
        }
      }
    }

    stage('Accept production deploy') {
      when {
        branch 'master'
      }

      steps {
        milestone 3
        script {
          slack.sendMessage(
            SlackColours.GOOD.colour,
            "Waiting for input (${utils.getBuildLink(env)})"
          )
        }
        input 'Deploy to production?'
        milestone 4
      }
    }

    stage('Prepare production deploy') {
      agent {
        dockerfile {
          filename 'Dockerfile.test'
          args dockerArgs
        }
      }
```

```
      when {
        branch 'master'
      }

      steps {
        sh 'npm run build:prod'
      }
    }

    stage('Production deploy') {
      agent {
        dockerfile {
          filename 'Dockerfile.test'
          args dockerArgs
        }
      }

      when {
        branch 'master'
      }

      steps {
        milestone 5
        lock(resource: 'prod-server', inversePrecedence: true) {
          milestone 6
          retry(count: 3) {
            sh './deploy.prod.sh'
          }
        }
      }
    }
  }

  post {
    always {
      script {
        slack.notify(currentBuild, currentBuild.getResult(), env)
      }
    }
  }
}
```

Final pipeline definition of a standardized build

```
#!/usr/bin/env groovy
import hudson.AbortException
import org.jenkinsci.plugins.workflow.steps.FlowInterruptedException
import org.thesis_ci_automation_test.*

// This is an example of a highly standardized Pipeline
// See https://github.com/jenkinsci/workflow-cps-global-lib-plugin
//
// In a project's Jenkinsfile, this should be used like:
//
// @Library('this-library') _
// standardBuild {
//   projectName = 'my-project'
// }

// Using Pipeline libraries, all libraries' vars/*.groovy
// scripts with call-methods are exposed as [fileName] functions.
def call(body) {
  // Read the body-closure's content
  // into config and evaluate it (body()).
  def config = [:]
  body.resolveStrategy = Closure.DELEGATE_FIRST
  body.delegate = config
  body()

  // ACTUAL BUILD BEGINS

  def slack = new SlackNotifier()
  def utils = new Utils()

  def dockerEnv = null
  def dockerBuildArgs = '-f Dockerfile.test .'
  def dockerEnvArgs = '-v /var/run/docker.sock:/var/run/docker.sock'
  def DOCKER_REGISTRY_NAME = 'my-registry:8082'
  def DOCKER_REGISTRY_URI = "http://${DOCKER_REGISTRY_NAME}"

  // Keep only last 5 builds
  properties([
    buildDiscarder(logRotator(numToKeepStr: '5'))
  ])

  // In regular Jenkinsfile (not declarative), we need to
  // manually manage errors and post-actions,
  // so wrap everything in try-catch-finally.
  try {

    ansiColor('xterm') {
      node {
        stage('Checkout') {
          //deleteDir() // TODO: Uncomment when done demoing
          checkout scm
        }
        withCredentials([[$class: 'UsernamePasswordMultiBinding', credentialsId: 'docker-login', usernameVariable: 'USERNAME', passwordVariable: 'PASSWORD']]) {
          // This image will be re-used later, so save a reference
          dockerEnv = docker.build("${config.projectName}_build", dockerBuildArgs)
          dockerEnv.inside(dockerEnvArgs) {

            stage('Build') {
              sh 'npm run dependencies'
            }

            stage('Test') {
              try {
                parallel 'Unit tests': {
                  sh 'grunt unit'
                }, 'Smoke tests': {
                  sleep 10
                  echo 'Do some rudimentary smoke tests here'
                }
              } finally {
                // Test results should always be saved (or attempted)
                junit 'test-results/**/unit-test-results.xml'
                publishHTML(target: [
                  reportName: 'Coverage report',
                  reportDir: 'test-results/html',
                  reportFiles: 'index.html',
                  keepAll: true,
                  alwaysLinkToLastBuild: true,
                  allowMissing: false
                ])
              }
            }

            stage('Prepare dev deploy') {
              // Deployment's should only be made from the dev branch.
              // Blue Ocean will also mark this stage "Skipped",
              // as there are no steps executed in else-case.
              //
              // This is clearer than skipping whole stages,
              // as then they would not be rendered at all.
              if (env.BRANCH_NAME == 'dev') {
                milestone 1
                sh "docker login --username=${USERNAME} --password=${PASSWORD} ${DOCKER_REGISTRY_URI}"
                sh 'npm run build:dev'
                sh 'npm run publish:dev'
              }
            }

            stage('Development deploy') {
              if (env.BRANCH_NAME == 'dev') {
                milestone 2

                // We should only allow a single deploy at a time
                lock(resource: 'dev-server', inversePrecedence: true) {
                  milestone 3
                  retry(3) {
                    sh './deploy.dev.sh'
                  }
                }
              }
            }

          }
        }
      }

      stage('Accept production deploy') {
        if (env.BRANCH_NAME == 'master') {
          milestone 4

          // As there's currently no good way to visualize
          // pending inputs, we need to manually notify users.
          slack.sendMessage(
            SlackColours.GOOD,
            "${currentBuild.getFullDisplayName()} - Waiting for input
(${utils.getBuildLink(currentBuild)})"
          )
          timeout(time: 1, unit: "DAYS") {
            input 'Deploy to production?'
          }

          // When a milestone is passed, no currently running
          // other job can pass the same milestone,
          // and will be cancelled.
          //
          // This is used in combination with input to only allow
          // the selected build to deploy.
          milestone 5
        }
      }

      node {
        withCredentials([[$class: 'UsernamePasswordMultiBinding', cre-
dentialsId: 'docker-login', usernameVariable: 'USERNAME', passwordVar-
iable: 'PASSWORD']]) {

          // Re-use the previously created Docker image
          dockerEnv.inside(dockerEnvArgs) {

            stage('Prepare production deploy') {
              // Production deploys should only be made from master
              if (env.BRANCH_NAME == 'master') {
                sh "docker login --username=${USERNAME} --pass-
word=${PASSWORD} ${DOCKER_REGISTRY_URI}"
                sh 'npm run build:prod'
                sh 'npm run publish:prod'
              }
            }

            stage('Production deploy') {
              if (env.BRANCH_NAME == 'master') {
                milestone 6
                lock(resource: 'prod-server', inversePrecedence: true)
{
                  milestone 7
                  retry(3) {
                    sh './deploy.prod.sh'
                  }
                }
              }
            }

          }
        }
      }
    }

  } catch (FlowInterruptedException|AbortException err) {
    currentBuild.result = 'ABORTED'
    throw err
  } catch (err) {
    echo "${err}"
    currentBuild.result = 'FAILURE'
    throw err
  } finally {
    // Use a separate stage for visualization purposes,
    // otherwise these steps won't be shown in Blue Ocean (at least
currently).
    stage('Post build actions') {
      slack.notify(currentBuild, currentBuild.result, env)
    }
  }
}
```