Development of Continuous Delivery at Mentech Innovation

Bachelor's Thesis

Electrical and Automation Engineering

Valkeakoski

February 2019

Polina Rymshina

Electrical and Automation Engineering
Valkeakoski

Author	Polina Rymshina	**Year**	2019
Subject	Development of Continuous Delivery at Mentech Innovation		
Supervisor	Mika Oinonen		

ABSTRACT

The software development standards of the modern world require the companies to respond quickly to new opportunities and build on top of feedback from customers. To help the developer teams to adapt to the need of fast software production new methods have been created. One of these methods is Continuous Delivery - essentially an automated process of building, testing and releasing software. The aim of this thesis project was to examine whether the concept of Continuous Delivery would improve the process of software production at the case company - Mentech Innovation, - and if so, how?

Mentech Innovation is a health care technology startup aiming to improve the quality of life of people with mental disabilities. The developers at Mentech work on an emotion sensing and regulation platform HUME, which in the end of 2019 will be turned into production grade software. For that, automated quality assurance and reliable and repeatable releases needed to be set up.

During the thesis project a literature research on the concept of Continuous Delivery, its benefits, disadvantages and risks was made. Additionally, the Continuous Delivery software deployment pipeline for the HUME website of Mentech Innovation was created to assess the feasibility of Continuous Delivery usage for all software at Mentech Innovation.

As a result of the work, the created deployment pipeline of the HUME website has proved to work correctly. The benefits of the usage of the Continuous Delivery concept have been validated, as the concept improved the process of software testing and release by making it fully automated, controlled and therefore reliable. Additionally, it saved time for the developers in that instead of manual testing and releasing time can now be spent on developing new features.

In conclusion, by examining literature sources and implementing Continuous Delivery for the HUME website of Mentech Innovation, it was proved that the concept of Continuous Delivery can benefit the overall process of software production at Mentech Innovation by making the process fully automated, fast, repeatable and reliable.

Keywords Continuous Delivery, pipeline, deployment, testing
Pages 49 pages including appendices 2 pages

Contents

List of Tables

List of Figures

Glossary

artifact By-product produced during the development of software (e.g. project source code, dependencies, binaries or resources). 2, 14, 15, 27, 30, 35, 39

container Unit of software that contains an environment. i, 32, 35

Docker Container technology, allowing the developer to run the application inside a certain environment. 32, 33, 35, 36, 38, 39

environment Libraries, tools, and other files necessary to be able to run the application. i, 3–5, 14, 15, 19–22, 27, 29, 32–40, 42

Git Version control system to track changes in the software code. 4, 10, 22, 23, 29, 30

MoSCoW method A way of determining the importance of requirements for a project. MoSCoW stands for Must haves, Should haves, Could haves, Won't haves. 8

Nginx Web server to serve static assets (for example send files to clients). 36

production Environment where the software is available to the customers. i, 2–4, 10, 11, 15, 19, 20, 27, 29, 34–37, 39

staging Environment made for software testing. Nearly exactly resembles a production environment. 2–5, 15, 27, 29, 34, 35, 37, 39, 42

web service HTTP server application. Used for managing the data from clients and receiving analytic results from the sensors. 29, 36, 38, 39

web socket Communications protocol enabling a two-way communication session between the user's browser and a server. 29

Webpack Webpack bundles web assets (such as images, CSS and JavaScript files) for use in a browser. 33, 35, 37, 39

In the following text the above mentioned terms can be found, and in an electronic version of the thesis on click the reader will be redirected to the glossary page for a term definition.

1 INTRODUCTION

In the modern software development world the quality of the software product of a company is as important as the company's ability to respond quickly to new opportunities and issues. For all the software development companies, big and small, especially for startups, it is essential to move fast, building on top of feedback from customers. After all, software only brings value when it is in the hands of the user (Humble & Farley, 2011, 14).

In the past couple of decades the complexity of written software as well as used tools has increased. New methods were needed to help the developer teams to adapt to the need of fast writing, testing and releasing of software. One of these methods is the concept of Continuous Delivery - an important and trending idea in software production.

Continuous Delivery, essentially, is an automated process of building, testing and releasing of software. The teams practising Continuous Delivery produce the software in short cycles and ensure that the software can be released on demand of management or a customer at any time (Chen, 2015, 50), whereas the conventional methods of testing, packaging and releasing of software can take from weeks to months to get the software in the hands of users, and the release process is not repeatable or even reliable (Humble & Farley, 2011, 14).

The case company for this thesis is Mentech Innovation - a health care technology startup based in Eindhoven, The Netherlands. Mentech Innovation aims to impact the quality of life and happiness of people with mental disabilities. The developers at Mentech work on an emotion sensing and regulation platform HUME, which reads body parameters and outputs them in a form of a graph, simultaneously evaluating the mood of a patient.

The aim of this thesis project was to examine the concept of Continuous Delivery by performing a research on its benefits and drawbacks, workflows, ways for risk management and strategies of deployment. Additionally, an implementation of Continuous Delivery for the HUME website was needed to assess the feasibility of Continuous Delivery usage in the company. In the end, a conclusion whether the concept of Continuous Delivery is beneficial for Mentech Innovation was made.

2 ASSIGNMENT

The assignment that was given to the thesis author is presented in this chapter. First, the background of the assignment is described - the case company (commissioner) and its products. Secondly, the thesis assignment description is given, with the main requirements described shortly. Then, a problem definition is derived from the assignment description. Finally, the hypothesis and the research and design questions are defined.

2.1 Background

Mentech Innovation is a health care technology company creating emotion sensing products and services for a better mental health care. With its products, Mentech Innovation wants to impact the quality of life and happiness of people with a mental disability or dementia. Mentech is a young company, a startup, currently in the state of feasibility and market validation.

The company develops an emotion sensing and regulating platform called HUME. The technology is based on the sensing of body parameters with wearables (EmoKit), the detection of emotional arousal via smart models and deep learning algorithms (EmoRadar), as well as the methodologies for emotion regulation.

In the third and fourth quarters of 2018 Mentech Innovation used the first version of HUME to test the feasibility of general arousal sensing in carehouses. In the second quarter of 2019 it is planned to present the second version of the product identifying positive and negative arousal.

Mentech Innovation consists of three departments: hardware, software and data science teams. For this thesis assignment the student joined the software team to work on the automation of the delivery process of the software created in the department.

2.2 Description of assignment

As mentioned before, Mentech Innovation is developing a technology for emotion sensing and recognition in mentally disabled people and people with dementia. This technology includes a wearable with sensors, a data streaming platform, a website, a web application and a database. In addition to this, Mentech develops software for the company partner, En-Gager, including a mobile and a web application.

During the second half of 2018, Mentech Innovation has developed a prototype of its product. In the second half of 2019 they want to turn it into a production grade software. For that they needed quality assurance and reliable and repeatable releases. At the start of this thesis project the software at Mentech Innovation was delivered ad-hoc, without standardisation, automated testing or security. As the testing and delivery stages of software releasing were not automated, the software and the deployments of it were prone to human errors.

A possible solution to the problem of faulty software delivery would be the concept of Continuous Delivery, which could improve the process of testing and deploying the software through automating it.

The conventional way to release software is to have a set date release. With this model there is a concept of cycle time which is the time measured from deciding to make a change in the software till having it in production. In many teams the one cycle can last for weeks or even months, which can have an impact on the user satisfaction with the product and cost the company money. Additionally, having a set date for a release means that the days before the release are stressful, as the developers try to fix the possible bugs in the last minute (Humble & Farley, 2011, xxiii). The concept of Continuous Delivery allows the company to avoid these problems, as the developers just release the feature when they are done working on it and it gets automatically tested. Further in the document the other benefits of Continuous Delivery are described and the decision is made whether the transfer from the conventional set date release model to Continuous Delivery model is beneficial.

The pattern that is central to Continuous Delivery is the deployment pipeline. It is, in essence, an automated implementation of the build, deploy, test and release process (Humble & Molesky, 2011, 7). An example of such a pipeline can be seen in Figure 1.

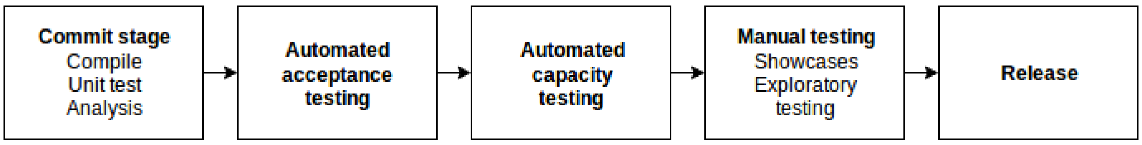


Figure 1: Example of a deployment pipeline (Humble & Farley, 2011, 4)

The deployment pipeline that initially was used at Mentech Innovation can be seen in Figure 2. Compared to the Figure 1 the initial Mentech pipeline lacks all the testing stages, except for unit testing. Additionally, the releasing of software using this pipeline was not automated.

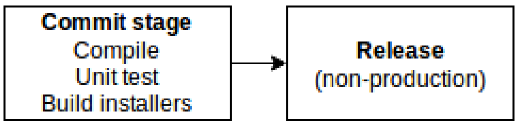


Figure 2: Initial deployment pipeline at Mentech Innovation

The aim of this assignment was to examine the concept of Continuous Delivery, evaluating the benefits and risks of it for Mentech Innovation, as well as looking into the ways of implementing Continuous Delivery for the Mentech Innovation software. Then the Continuous Delivery tools and the infrastructure of Continuous Delivery should have been configured for the Mentech Innovation HUME website to assess the feasibility of the Continuous Delivery usage for all software at Mentech Innovation. The exact process of work, including the work phases and steps, is described in chapter 3.

Additionally, together with the Mentech Innovation management and the lead engineer, the thesis author had set up a list of requirements for the software and the tool choices. These requirements were used when working on Continuous Delivery and choosing tools and methods of implementation for it.

The list of the created requirements can be seen below:

- Repeatability of the deployment process
- Cost of tools - preferably free, otherwise about 100 eur/month
- Preferably open source tools
- Manual code review should be possible
- Rollbacks should be possible
- Deployment should be done to staging environment (continuously)
- Zero downtime for applications
- Static analysis for code formatting, common bugs and duplicate code should be set up
- Testing should be possible to do before merging
- The tools and working methods should be compatible with Git

2.2.1 Scope

By definition, Continuous Delivery is a practice to automatically build, test code changes and release the software product to a production-like environment (Halonen, 2017, 6). Therefore this thesis work does not include the other steps of product releasing, such as management-related decisions (such as opportunity assessments, meetings, etc.), deploying to production, and post-deployment software monitoring. This project focuses solely on the research and implementation of Continuous Delivery: the build, test and release (to non-production) stages, configurations and tools for them.

2.3 Problem definition

As mentioned before, the software at Mentech Innovation was initially delivered without automated testing, standardisation or security. The process of deployment was not repeatable or reliable, which lead to human errors and time spent on fixing them. Additionally, the deployments were not documented so the information about the errors was not saved. As the deployment process was not automated, every deployment turned into a time-consuming process, which was repetitive and hard to test.

The lack of automated build-test-release process is a problem for several additional reasons. Teams that do not practice Continuous Delivery have to either spend time on manually testing the software they write or hire extra people to perform that job, which wastes time and money of the company. Manual testing increases the time until the release of the end product which means the clients do not get new features or bug fixes of the application for a long time. The process of releasing of these new updates is manual as well, therefore every time it is done, it might be different. The configuration of the system, the release process, even the software update itself - it all might cause errors during the release process, or, even worse, bugs released to the end users. The process of releasing is not monitored, so the developers team will not be informed immediately when something goes wrong with the software, and even then, the root cause of the problem will be hard to track down. In addition, without monitoring it is not possible to see how the system behaves when the certain changes to the software are introduced, or, for example, when the amount of users increases or new servers are added. Without that, the planning of the future improvements of the system to meet the demands of business and customers is very hard. (Humble & Farley, 2011, 4-10.)

The system that would improve the situation with the software releases at Mentech Innovation should have solved the aforementioned problems. The build-test-release process should have been automated. So, by one push of a button, the developers from Mentech would be able to commit their changes, have them compiled, unit tested, statically analysed for common bugs and duplicate code, then tested for compatibility with the other parts of the system and capacity tested and after that released to a staging environment. Only then the software could be released to the users on demand of the company management.

2.4 Research questions and hypothesis

From the assignment described in chapter 2.2 several questions were formulated with the main one being:

> Would the concept of Continuous Delivery improve the process of writing, testing and releasing software at Mentech Innovation and if so, how?

From this main question several subquestions and a hypothesis were derived. The subquestions helped to answer the main question. They are listed in chapters 2.4.1 and 2.4.2. The hypothesis is presented in chapter 2.4.3.

2.4.1 Literature research questions

The questions that had to be answered during the literature research phase of the thesis project were:

- What should the general structure of the deployment pipeline for the Mentech Innovation software be?
- What are the methods of risk management for the pipeline?
- What are the strategies of deployment and which one suits Mentech Innovation best?
- Which development workflows exist nowadays and are compatible with Continuous Delivery?

2.4.2 Design questions

The questions that needed to be answered during the design and implementation phase of the thesis project were:

- What is the structure of the deployment pipeline for the HUME website?
- What are the best tools for the implementation of the deployment pipeline for the HUME website?

2.4.3 Hypothesis

The hypothesis that was to be proved or disproved during this research:

> The concept of Continuous Delivery will improve the process of software releasing at Mentech Innovation by making it fast, frequent, repeatable and reliable.

3 PROJECT APPROACH

In this chapter the methods chosen for the research project are presented including the time limits set for them. The project was divided into two phases. First, a literature research was conducted to explore the concept of Continuous Delivery, compare the development workflows, deployment strategies etc. The literature research phase was followed by a design and implementation phase. During this phase the Continuous Delivery strategy was applied to the HUME website, which required a choice and configuration of all the necessary tools.

3.1 Research methods

The literature research phase lasted ten weeks. In this phase the problem and the basic requirements for the Continuous Delivery pipeline were defined. Based on this information a literature base was created. Using the found literature the benefits and drawbacks of Continuous Delivery were evaluated. Then a general structure of a deployment pipeline was drawn. A research was conducted on the risk management methods for the pipeline and the development workflows. The deployment strategies were described and had their advantages and disadvantages evaluated. The research phase concluded with several decisions:

- A decision was made whether the concept of Continuous Delivery is beneficial for Mentech Innovation
- A development workflow suitable for the Mentech Innovation software developers was chosen
- A deployment strategy for software applications was chosen

3.2 Design and implementation methods

The implementation and design phase lasted nine weeks. For the work method Scrumban was chosen. Scrumban is an agile project management methodology. It is based on the features of Scrum and Kanban - two other frameworks for managing knowledge work (Nikitina, Kajko-Mattsson, & Stråle, 2012).

In Scrum, the work time is split in short fixed-length cycles called sprints. The deliverables for each sprint are selected beforehand. The work is sorted by priority and relative effort. Then the sprint is ”locked” - new deliverables cannot be added during the sprint. After a couple of weeks - a usual duration of a sprint - all the work of a sprint should be done.

In Kanban a visualisation tool (e.g. a whiteboard) is used to illustrate the work phases, for example ”To Do”, ”Ongoing” and ”Done” as columns. Then all the deliverables of a project are placed into these columns and are implemented according to their position. For instance, an item can be in the ”To Do” column first when no work has been performed on it. When the developer decides to implement it the item goes to the ”Ongoing” column. When it is implemented the item is put to the ”Done” column. Items can be added to the board if new ideas appear or issues arise. In Kanban the number of items in each column can be restricted.

Scrumban has the features of both Scrum and Kanban. Applied to this project it meant that a list of requirements and deliverables was made. These requirements and deliverables were prioritised using the MoSCoW method. A board was used to visualise the work phases - ”Open”, ”To Do”, ”Doing”, ”Testing” and ”Closed”. ”Open” column featured all the deliverables and features for the project. ”To Do” included the deliverables that were worked on in the nearest time. The items that were being worked on were in the ”Doing” column. The items that needed to be tested were in the ”Testing” column. The deliverables that have been implemented were placed in the ”Closed” column. The list of deliverables and features could be edited (added to) during the project. The thesis author was working according to the requirements implementing them one by one.

With Scrumban the flow of work is continuous. There are no sprints, no time limits. The deliverables are just implemented one by one and based on the feedback the new features or issues can be addressed. Therefore, Scrumban is a good match with the idea of Continuous Delivery.

4 LITERATURE RESEARCH

In this chapter an analysis of literature information is given to answer the research questions of chapter 2.4.1. For that the concept of Continuous Delivery was examined, including the deployment pipeline structures, ways for risk management, development workflows and the strategies of deployment.

Getting software released to users is often a painful, risky, and time-consuming process (Humble & Farley, 2011). A proposed solution for easing the process is Continuous Delivery. The concept of Continuous Delivery emerged in 2010 when Jez Humble and David Farley released a book called Continuous Delivery (Sharma, 2018). It was proposed to be an extension to an already existing Continuous Integration development practice. The difference between the two can be seen in Figure 3. Continuous Integration takes the software from the phase where it is being written to the step where the software is being tested. Continuous Delivery goes even further and allows the developers to automatically release the new software updates.

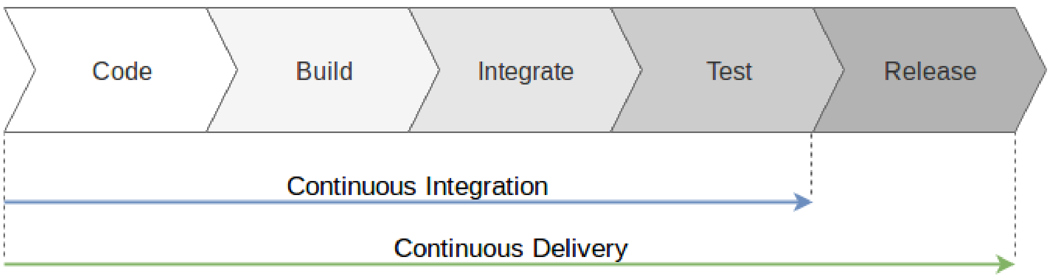


Figure 3: Comparison of Continuous Integration and Continuous Delivery

Continuous Delivery is similar to a traditional production line: just like products, software needs to be assembled, tested, verified and packaged, and delivered to the users. This should be automated. For example, if the general tests for software have been written beforehand, they can be applied automatically to every release of software to ensure it works. Then the testing of software will turn into just one push of a button.

4.1 Continuous Delivery rules

There are no strictly defined rules to follow when implementing Continuous Delivery. However, there are good practices and recommendations to follow to essentially achieve the benefits of Continuous Delivery within the company (Humble & Farley, 2011, 24-29; see also Farcic, 2017). These principles are listed and shortly described below.

Automate the deployment pipeline All the steps of the deployment pipeline should be automated. The only exception from this rule is the possible manual testing stage (showcases or exploratory testing). All the other testing stages as well as the commit and release stages should be performed automatically.

Integrate frequently With every implemented feature, integration should be performed with the rest of the project. This way the delivery of the new features can be continuous.

Be Agile Agile teams deliver work in small increments, which results in faster value delivery. The requirements and feedback are evaluated continuously to be able to quickly respond to change.

Keep everything in version control All the code, test scripts, configurations and documentation for a project should be kept in version control. This is done to be able to start up the project on any machine on demand and to be able to fallback to the previous version if needed.

Fix bugs as soon as they appear The problem will be found and fixed faster while the code is still fresh in the developer's mind. The developer is not supposed to work on anything else until the pipeline is finished.

Practice test-driven development Tests need to be present before the new code is committed, otherwise the buggy code will go to production. Tests can be written before the changes to the code are made and they should be based on the user requirements. It is also possible to write tests right after the new code is written.

Have a fast deployment pipeline According to Farcic (2017), the average time for the pipeline to complete should be 15 minutes. Following this recommendation will help the developers to stay in focus. They are not supposed to work on the new features until the run of the pipeline is finished successfully. Otherwise it will not be possible to integrate the new features frequently.

Commit only to master branch or short-lived branches When working with Git, if branching is abused and merging the feature branches with the master branch happens rarely (less than once a day), then the integration is postponed, and the company does not practice Continuous Delivery. Ideally, if the deployment pipeline is trustworthy, and the developers run the local version of the pipeline before committing, it would be much better and faster to commit directly to master. However, it might be challenging as it requires high discipline of the developers. It is easier to have short-lived branches and ensure they are merged with the master as soon as work on them is done.

Run commit tests before merging new commits This can be done by either the developer manually or the Continuous Integration server automatically (as mentioned before, manual local testing requires high discipline therefore it is challenging, so automatic run is preferred). First, the developer should update their copy of a project by pulling from the version control system. Then, a local build should be initiated and the tests should be run. This is done to ensure that the developer has the latest version of the project when he/she commits, so that the build will not run into merge issues when the developer pushes the new updates to the version control system. Additionally, it reduces the chance that the developer introduces bugs in the central repository of version control.

4.2 Advantages and disadvantages

Continuous Delivery advocates claim that, if implemented correctly, the concept can make the software releases a repeatable, reliable and predictable process (Chen, 2015, 50). However, implementing and actually following the rules of the Continuous Delivery approach can be too challenging. To make a decision whether Continuous Delivery is beneficial for Mentech Innovation both the advantages and disadvantages of the concept had to be listed and evaluated.

The advantages of Continuous Delivery according to Chen (2015, 52) are:

+ **Saving time and money** If the testing and releasing is being performed manually the company has to hire a testing (Quality Assurance) and deployment team or the software developers of the company have to spend their time testing and releasing. Automating of the test-release process will help the company save money on the new employees and the software developers can spend their time implementing new features and client requests instead.

+ **Improved product quality** With Continuous Delivery implemented, after the developer commits changes to the code the whole code base undergoes a series of tests. These tests include checking whether all the functions of an application work as intended, as well as checking if the app is still working well together with the database or other applications, meets the requirements, etc. These tests help to reduce the risk of bugs appearing because of human errors and manual configurations. Additionally, one of the rules of Continuous Delivery is that if a test has failed, the developer has to fix it immediately and not leave it for later. Because of that, the amount of new features reaching production will increase - they will not anymore be put on a long waiting list for the items that need to be fixed.

+ **Standardisation** The Continuous Delivery pipeline standardises the procedures of deploying the software. Manual deployment processes are hardly the same between different updates, because it is very common that the deployment steps are not well documented or memorised. Using an automatic deployment process the tests, commands, tools used will be the same for each update to the software. Additionally, each deployment process is automatically documented with a deployment script.

+ **No application downtimes** With a manual deployment process it is easy to accidentally push a bug to production. In the worst case scenario this will break the whole application and will result in a disappointment and loss of clients. With automatic testing enabled, an accidental push to production is very unlikely to happen, as tests will fail if there are bugs present in the code.

+ **Stronger relationship with the customers** With the Continuous Delivery rule of committing changes whenever a new requirement has been implemented the application will be updated often. The customer requests for new features can be taken in account as soon as they arrive. Therefore the application can follow the user requirements as closely as possible. Additionally the application is always in a running state. As customers can always see the new ideas and requests turn into working features the relationship between the company and the clients improves.

+ **Lowering stress level** Manual releases into production are big events. They are usually surrounded with a lot of stress, because of the bugs that might occur, human errors that went unnoticed and configuration and compatibility problems. Manual releases require a lot of work of the testing and deployment teams. If the release can be performed automatically by just one push of a button and each release is backed up by version control, then the stress level associated with releasing a product reduces significantly. (Humble & Farley, 2011, 17-22.)

The disadvantages of Continuous Delivery according to Chen (2015, 53) are:

- **Challenging rules** The rules of Continuous Delivery (listed in chapter 4.1) can be challenging to adopt. They require a lot of initial work, team collaboration, discipline and time. However these rules need to be followed as only then can a team deliver value to the clients continuously.
- **Very good team collaboration needed** A very good collaboration and coordination is needed in the team to successfully implement continuous practices (Shahin, Babar, & Zhu, 2017, 3925). The team members involved in writing software have to understand the concept and follow the rules, for instance invest time in writing tests, integrating with every new requirement implemented and fixing the code immediately after any bugs appeared. Additionally the team members need to know what features have been implemented and what the status of the project is at all times. This should be visualised.
- **Little research on problem solving** Very little research has been done on how to introduce Continuous Delivery in a team (Chen, 2015, 53). This means that even though there is a lot of advice on the internet on how to adopt the concept more smoothly, there is no common strategy to ensure the acceptance and collaboration on Continuous Delivery. If some complications arise there are no common practices to tackle them efficiently.
- **Complicated implementation** The practices associated with Continuous Delivery as well as the configuration and usage of the Continuous Delivery tools require a set of soft and hard skills which are usually not taught in a university. Therefore the learning curve of implementing the continuous practices can be a bottleneck. The implementation of Continuous Delivery takes a lot of time and money, because hiring a person skilled enough is expensive.

To compare the advantages and the disadvantages of the Continuous Delivery approach a comparison table was made (Table 1). The table features the benefits of Continuous Delivery on the left and the drawbacks of it on the right. Each item in the table has a weight to it. The weights are given based on the preference of the thesis author and they have been approved by the lead engineer of Mentech Innovation. Benefits have the weights ranging from 1 to 5, drawbacks have the negative weights from -1 to -5, where 5 is highly favourable and -5 is highly unfavourable. Under each column the end score for the column is counted. After that, the total score is counted by subtracting the negative value from the positive value. If the resulting total score is positive, then the advantages outweigh the disadvantages and the evaluated concept is generally profitable. Otherwise, the concept is not recommended for use.

As we can see from Table 1 the total score is positive, which means that the Continuous Delivery approach could theoretically be beneficial to be used at Mentech Innovation.

Table 1: Evaluation of advantages and disadvantages

Pros		Cons	
Saving time and money	+3	Challenging rules	-4
Improved product quality	+4	Very good team collaboration needed	-4
Standardisation	+3	Little research on problem solving	-2
No application downtimes	+2	Complicated implementation	-3
Stronger relationship with customers	+3		
Lowering stress level	+2		
End score:	17	End score:	-13
Total score: 4			

4.3 Value stream map

Value stream map is a visualisation of the software delivery process including the stages the software goes through and the time spent on these stages. Creating a value stream map is a low-tech process. It aims to depict the software delivery process from a business point of view starting with the concept stage and ending with the client stage (Humble & Farley, 2011, 107-108).

In the case of Mentech Innovation, creating a value stream map was useful to visualise the problem and in the end of this thesis work to compare the new value stream map with the one created before the project. This helped to prove the hypothesis that the concept of Continuous Delivery is beneficial for Mentech Innovation and improves the process of software releasing in the company.

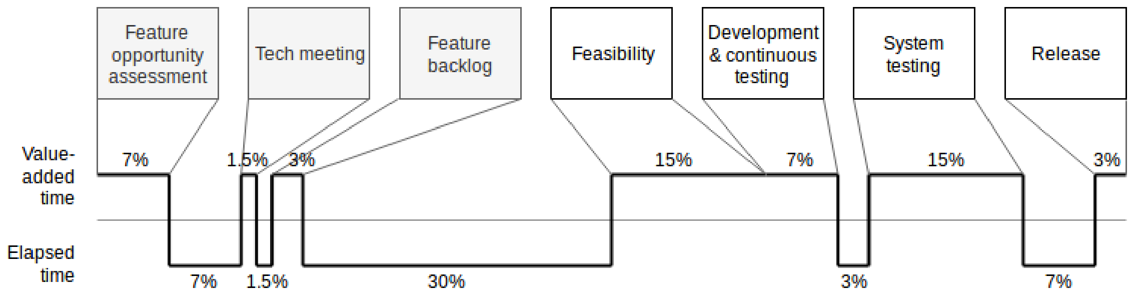


Figure 4: Value stream map before Continuous Delivery

The Figure 4 presents the value stream map made based on the information gotten from the developers of Mentech Innovation responsible for the software development, testing and delivery process as well as the business-decision-related information gotten from the management of the company. The value stream map has been made to depict the process of delivery of a new feature to an application, with the average relative time it takes for all the stages of delivery. As the whole process of delivery has been taken as 100% we can clearly see which stages of delivery take the most time. Additionally, the time has been separated into value-added time and elapsed time. Value-added time is the time when the actual work is being done - business meetings or programming.

Elapsed time is the time spent on waiting for the next stage of the project to start. The stages of the feature delivery are on the top of the Figure 4 - business-related in grey colour and software-development-related in white.

As we can see from the Figure 4, the total time spent on the business-related decisions is much smaller than the time spent on the development of the feature. The biggest percentage of time (30%) is spent between the business and the software stages, but as a communication problem it cannot be fixed by Continuous Delivery. With the Continuous Delivery implementation the author of this thesis aimed to improve the time spent on the stages related to software development. As we can see, due to the fact that Mentech Innovation is still quite a small company and one feature is usually developed, tested and deployed by the same person, the elapsed time between the development stages is not so big. However, the system testing and release stages take 28% (with the value-added and elapsed times included) of the total time. If the time spent on these two stages could be decreased, the new features to the application could be released faster, or the developer could spend this time on the improvement of the feature or development of a new one.

4.4 Deployment pipeline

As mentioned in chapter 2.2 in Continuous Delivery the deployment pipeline is a set of stages that the software has to go through automatically to be released. These stages include building the software, testing it and deploying it. For Mentech Innovation such pipeline had to be drawn to be further implemented later, taking in account the best practices and the requirements gotten from the company management and the lead engineer.

The pipeline created for Mentech Innovation can be seen on Figure 5. It is an extended version of the pipeline that can be seen in chapter 2.2. This pipeline is based on an example pipeline from Humble and Farley (2011, 111).

The steps of the pipeline are in the order they are in because the deployment pipelines are designed to fail fast. In case of failure, the whole pipeline should be terminated as soon as possible. That is why unit tests, which execute fast are ran first. Longer running tests come second. (Chang, 2013.)

The flow of work in the pipeline goes as follows:

1. The software developer commits the code updates to version control
2. This source code goes from version control to the commit stage, where it gets compiled, unit tested, analysed and packaged. The output of this stage gets stored in the Artifact repository, which acts like a storage for the package and the documentation related to it.
3. Next stage is integration. The environment gets configured with the settings from version control, and the package from the Artifact repository is performed integration tests on. The documentation from this process is stored in the Artifact repository.

4. The package is sent to the acceptance stage. Again, the environment gets configured with the settings from version control, and the package from the Artifact repository is performed acceptance tests on. The documentation from this process is stored in the Artifact repository.

5. After that the software goes to several environments - user acceptance and capacity stages - that can be run in parallel. The functioning of these stages is similar to the previous two stages.

6. Then the package is sent to the staging environment. This does not have to happen automatically, it can also be made as a manual step on demand from the management. There the software can be tested, possibly manually, in conditions similar to production.

7. Finally, the software can be manually released to production. The deployment method for it is further described in chapter 4.6.

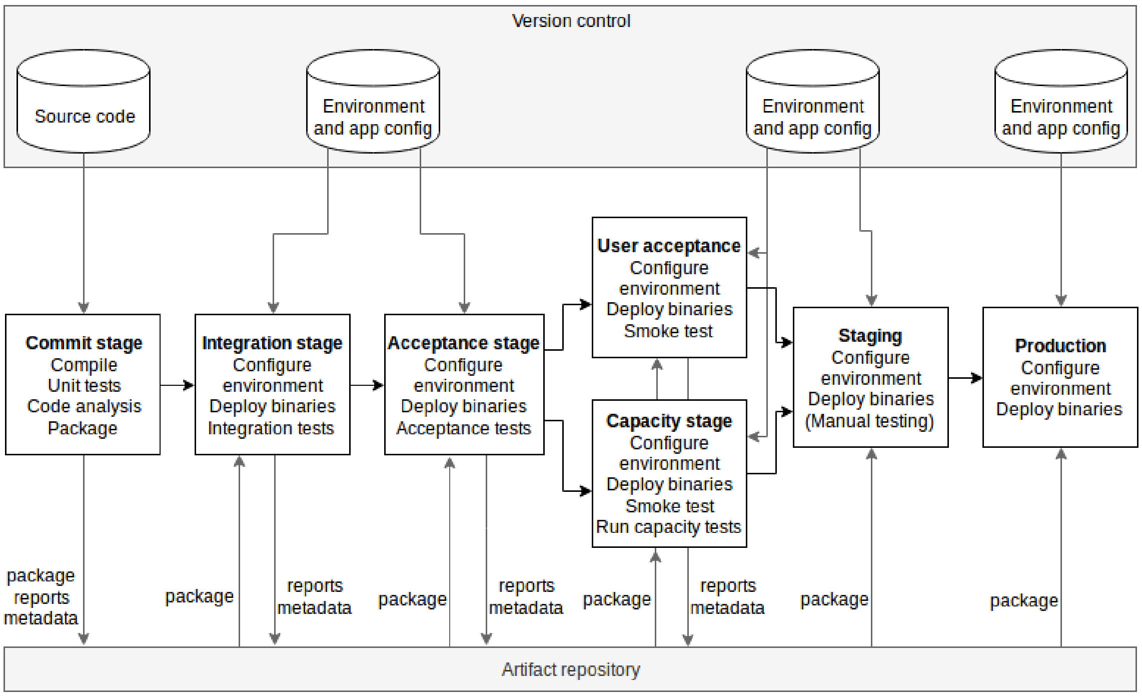


Figure 5: General deployment pipeline for Mentech Innovation

The pipeline created above is a generalised version of a deployment pipeline that can be used for all the software products at Mentech Innovation. However, depending on an application, the pipeline might need to be edited, as some stages of it might not be necessary or applicable. Additionally, the tools that are needed to implement the pipeline and the configurations for these tools can be reused from this project, however some additional configurations or tool choices might be needed depending on a project.

4.4.1 Types of testing

For a full understanding of the methods of software testing, in this chapter the three types of tests mentioned in chapter 4.4 that are needed to ensure the delivery of a high quality application are described.

Unit testing Unit tests test a particular piece of code, a function within the application (Humble & Farley, 2011, 89). For example, an application may contain a method to check the validity of a format of a phone number. Then, an example of a unit test would be to try to automatically input different values (letters, numbers which are too short, etc.) to see whether the aforementioned method works correctly. Unit tests should run independently from any outside sources, such as a database, a filesystem, any external systems, etc.

Integration testing During integration testing the software modules are tested as a group. The way the application communicates with the database, filesystem or any other external systems is tested (Humble & Farley, 2011, 89). An example of an integration test would be sending a request from the application to the database to retrieve a list of the company clients and checking whether the received list equals the predefined value.

Acceptance and capacity testing Acceptance tests ensure that all the criteria for the functionality of the system, its usability and availability, are met (Humble & Farley, 2011, 85). An example of an acceptance test would be checking whether the items are correctly loaded on a page or whether the page is loaded within a certain time. Capacity tests are a subcategory of acceptance tests. Capacity testing is targeted at testing whether the application can handle the amount of traffic it was designed to handle.

Automated testing can provide the confidence for all the people involved in the project that the software product is working as it should. Performing unit testing, integration testing, and acceptance testing on a software product allows the engineers to thoroughly check the functionality of the application, which results in fewer bugs, reduced support costs and satisfaction and trust of clients. (Humble & Farley, 2011, 84.)

4.5 Risk management

Continuous Delivery as a model of work might be challenging to adopt in a company. As mentioned in chapter 4.2, a lot of team collaboration, discipline and time is needed. In addition, Continuous Delivery has various rules that need to be followed (chapter 4.1). In this subchapter the main project risks related to Continuous Delivery are identified and the mitigating strategies are described.

A common model of risk management (DeMarco & Lister, 2003) proposes a way to evaluate the risks by their impact and their likelihood. This model allows to assess each risk's severity. Based on this model Table 2 was created, as an extension of the model not only accessing the individual risks' severity but also calculating the project's risk percentage (Table 3).

In Table 2 all the risks are listed (not in any particular order). For each risk the chance it might happen is given (on a scale from 1 to 5). Additionally, the consequences of risks are described and the factor of impact on the project is given (on a scale from 1 to 5). From that for each risk a score is given which is the likelihood of happening multiplied with the factor of impact. This score can serve as an indicator of the risk's severity of impact on the project.

In Table 3 the end calculation for the risks is provided. The actual score for all the risks is calculated as a sum of the scores of all the risks. Additionally, as there are 13 risks, the total maximum score is 325 (calculated as the maximum likelihood (5) multiplied with the maximum impact (5) multiplied with the number of risks). From that the risk percentage is calculated. As we can see, the resulting risk percentage - the chance that some risk might happen during working with Continuous Delivery - is at a medium level - 35%.

To prevent the risks listed in Table 2 from happening a mitigating strategy for the risks should be put in place. After an analysis of literature (Shahin et al., 2017, 3929-3930), the thesis author has determined several rules to follow for risk mitigation. These rules are listed below with the risks that they can help mitigate mentioned (as a number from Table 2).

- Improve team communication and awareness (mitigates risks 1, 2, 3, 4, 6, 7)
  - Have regular meetings about the project progress and the usage of Continuous Delivery
  - Inform the team members about the outdated branches
  - Meetings with the management to discuss progress
  - Everybody takes responsibility of their code
- Planning and documentation (mitigates risks 1, 2, 6, 12, 13)
  - Implement a status board showing the status of each feature branch and the person responsible
  - Keep metrics of the developers' integrations
- Improve team qualification (mitigates risk 8)
  - Provide the team with necessary literature on the topic of Continuous Delivery
  - Organise trainings and talks on the usage of the Continuous Delivery concept and tools
- Perform thorough research before choosing tools for usage (mitigates risk 4)
  - Whenever a new tool needs to be chosen, a thorough research should be conducted and well documented, preferably using a selection matrix (similar to Tables 4 and 5)
- Pay attention to the testing stage (mitigates risks 5, 8, 9, 10, 11, 13)
  - Practice test-driven development
  - Have a testing workshop with the team
  - Have a manual (or user) testing stage
  - Run the tests in parallel

The rules listed above propose a solution to all the risks shown in Table 2 and it shows that with proper communication within the team, control from the management and lead engineers as well as group effort of the team it is possible to successfully practice Continuous Delivery.

Table 2: Risk analysis

№	Risk	Likeli-hood	Consequences	Impact	Score
1	The progress of the team is slower than expected	2	The features are released slower	3	6
2	Developers do not integrate often enough	4	The team does not practice Continuous Delivery, end product is bad or delayed	4	16
3	It takes a long time for the bugs to be closed	3	The delivery of features is not fast or continuous	4	12
4	Developers complain about the usage of tools	3	Developers work slower	3	9
5	The commit stage breaks	1	The progress slows down, the commit stage needs to be fixed	3	3
6	It takes a long time for the new features to be deployed	2	The features do not get to customer and do not bring value	5	10
7	The team is not collaborating sufficiently	3	Possible problems with the stages of Continuous Delivery process, end product is bad or delayed	4	12
8	The developers or testers do not have sufficient experience developing tests	3	The features are released slower	3	9
9	The developers are working without sufficient test coverage	2	The untested (possibly buggy) code can be released	5	10
10	The developers do not trust tests when they reveal bugs	1	If the developers rewrite tests to match the code, the possibly buggy code can reach production	4	4
11	The tests take too long to run	3	The developers get distracted from programming	2	6
12	Ineffective monitoring of production/staging environment	2	The team does not know if there are bugs to fix	5	10
13	The feedback of the customers takes too long to reach the developers	2	The customers are not satisfied	4	8

Table 3: Risk calculation

Score	115
Total score	325
Risk percentage	35%

4.6 Deployment strategies

The deployment stage is where the product is released to the clients. Therefore it is very important for the best experience of the client to ensure that the application they get is of good quality. And it is essential to be able to rollback a deployment in case some problems arise. This will allow the users to get the working version of an application back while the developers are fixing the bugs in the new version.

There are several ways to deploy an application. The end choice of the deployment strategy affects the way system should be configured, as well as the speed of releasing, impact on users in case of bugs and ways of fixing these bugs. The strategies for deployment are listed and described below.

Recreation deployment The recreation strategy is one of the easiest ways to deploy an application (Humble & Farley, 2011, 260). During this type of deployment the old version of the application is turned off and then the new version of the application is released. Even though this strategy is easy to set up, the shutting down and then turning on the application implies downtime between the turned off and on states.

Blue-green deployment For the blue-green deployment two identical production environments (called Blue and Green) are run. One of them - for example Green - is live, the second one is idle (Figure 6). All the user traffic is in the Green environment.

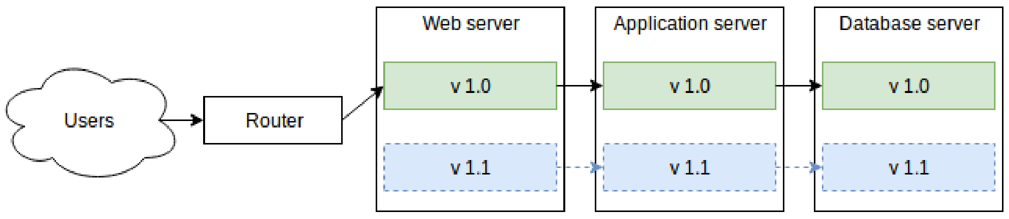


Figure 6: Blue-green deployment - initial state

The new version of the application is released to the Blue environment, where it can be tested. When the testing is done, the router switches the users to the Blue environment which becomes live. The Green environment becomes idle (Figure 7).
One of the benefits of this method is the absence of downtime between the two versions of the application. The users get switched to the new version instantly. Additionally, if there are some bugs in the new version the user traffic can easily be switched back to the old version. However, this strategy is harder to set up (because of database limitations - since both environments use the same database, the database needs to be compatible with both versions of the software) and it is more expensive. (Humble & Farley, 2011, 262.)

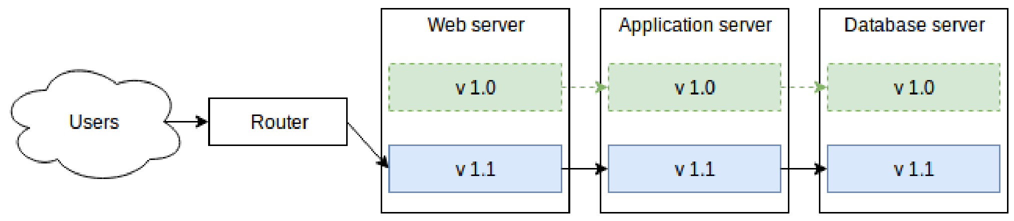


Figure 7: Blue-green deployment - final state

Canary deployment Similarly to the blue-green deployment, this strategy requires two production environments, one live and one idle (Figure 8). The whole user traffic is directed to the live version.

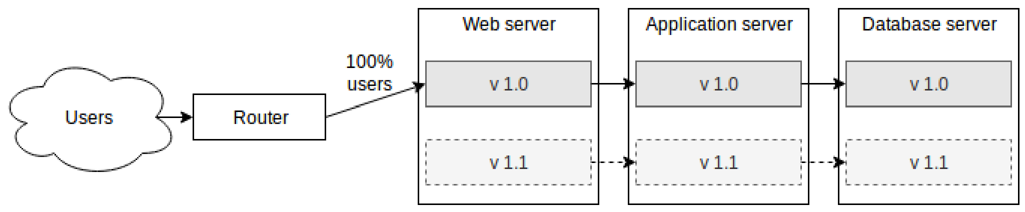


Figure 8: Canary deployment - initial state

To release the new application the developers turn on the second environment and deploy it there. Then, a small part of the user traffic gets routed to the second environment (Figure 9). This way the second environment can be tested in the ”real world” conditions to perform capacity tests of the application and to ensure there are no bugs released to the majority of users. In case the selected users report some problems with the new version, these users can just be routed to the old version of the application and the developers will have time to fix the issues.

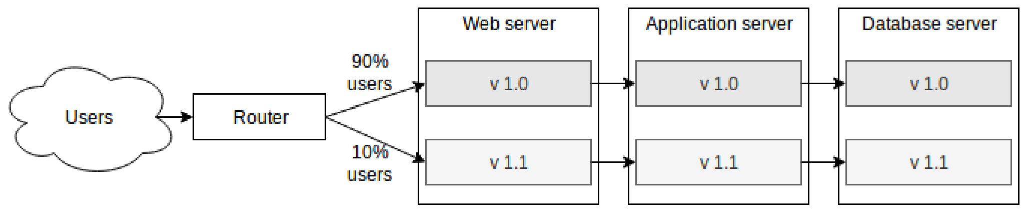


Figure 9: Canary deployment - small user set release

After it has been confirmed that the new version does not contain bugs or issues the rest of the users can be routed to the environment with the new version of the application and the environment with the old version can become idle (the end state will be similar to Figure 7). This way the risk of releasing of a new version of the application can be significantly reduced. (Humble & Farley, 2011, 262-265.)

A/B deployment This deployment strategy is similar to the canary deployment strategy. A/B deployment also requires having two environments, one with the old application and one with the new. While all the users are using the old version of the application a small subset of users gets routed to the new version. Once the user group confirms that the application functions correctly, the rest of the users get routed to the new version.

The difference from the canary release method is that with the A/B deployment strategy the small group of users is chosen based on a certain condition. Some examples of such conditions are geolocation, language or a used technology type (operating system, browser version, screen size, etc.). (Humble & Farley, 2011, 264.)

Shadow deployment With the shadow deployment technique two environments are run, for example A and B. All the user traffic is directed to the environment A. The user requests and actions happening in the environment A get copied and sent to the environment B. This helps to capacity test the environment B. When the environment B passes the tests and proves to be bug-free the user traffic gets routed from the environment A to B.(Tremel, 2017.)

To summarise the information about the deployment strategies Table 4 was created. The most left column features the comparison parameters, the list of which has been created based on the requirements received from Mentech Innovation. These parameters have been chosen as relevant for the way Mentech Innovation deploys software and the impact it has on the end users.

Table 4: Summary of deployment strategies

	Recreation	**Blue-green**	**Canary**	**A/B**	**Shadow**
Zero downtime	No	Yes	Yes	Yes	Yes
Capacity testing	No	No	Yes	Yes	Yes
Targeted users	No	No	No	Yes	No
Complexity of setup	Easy, no change needed in the release configurations	Hard, two environments are needed	Hard, two environments are needed	Very hard, requires two environments and a filter setting	Very hard, requires two environments and settings to redirect the requests
Negative impact on user	Very high	Average	Low	Low	Low
User feedback	Received late. Cannot be handled fast	Received late. Can be handled fast	Received fast. Can be handled fast	Received fast. Can be handled fast	Received fast. Can be handled fast

The comparison of the methods of deployment is presented in a form of a selection matrix (Table 5). The matrix has the same comparison parameters as Table 4. The scores from 0 to 5 (where a higher figure is better) are given to each deployment strategy based on the answers in Table 4. Additionally all the comparison parameters have a certain weight (from 0 to 5, where more is more important) based on the importance of them for Mentech Innovation. In the bottom of the table the end score for each deployment strategy is calculated as a sum of all the parameters multiplied with their weights.

Table 5: Selection matrix for deployment strategies

	Recreation	Blue-green	Canary	A/B	Shadow	Weight
Zero downtime	0	5	5	5	5	5
Capacity testing	0	0	5	5	5	4
Targeted users	0	0	0	5	0	1
Complexity of setup	5	4	3	2	1	4
Negative impact on user	1	3	5	5	5	5
User feedback	1	2	5	5	5	3
Score:	28	62	97	98	89	

The weights for the comparison parameters have a reasoning behind them. The user satisfaction with the product is very valuable for Mentech Innovation. Therefore, Mentech would benefit from a deployment strategy in which the software errors or bugs would impact the user the least and the users would always have a working version of the application. Additionally, the deployment strategy which allows to test the application in the conditions close to real environment with users (capacity test) would make the release of the application more reliable. The author of this thesis work did not have much experience with setting up a deployment strategy, therefore the setup of it should not be complex, however this is not a hard requirement because a good deployment strategy is worth investing time in. The feedback from the users after or during deployment would be appreciated, however not necessary if the released software is properly tested. Finally, the user targeting is an interesting possibility, however for Mentech Innovation it is not relevant because the product is still in its early stages.

As we can see from the end scores the A/B strategy has gotten the highest score of 98. The second best is canary deployment strategy with the score of 97. However Mentech Innovation does not need the functionality that the A/B method offers - releasing the new application only to the users under a certain condition. Therefore since the A/B strategy and the canary strategy scored very close results in the matrix the decision was made to choose the canary deployment as the most suitable deployment method to use at Mentech Innovation.

4.7 Development workflows

So far in this document the flow of the development work has only been described from a business position using a value stream map. Additionally, the development pipeline has been given depicting the flow of a software update from the commit stage to release. But what are the actual steps of a developer when he or she wants to update an application?

To track the changes that are made to the code a version control system, Git, is needed. This way the developers collaborating on the code can see each other's code updates, experiment on new ideas without fearing to break the application and record a message with each change so other collaborators can understand the reason for changing. (Blischak, Davenport, & Wilson, 2016, 1.) Basically, Git works so that the developer has a copy of all the files for the application on his or her computer in a folder called a local repository. The changes the developer makes to these files are tracked by Git. Once the changes to the files are made the developer can commit them - put them to a staging area (in this context, staging area refers to a file, which contains information of what is going to be committed to version control) ready to be sent to the remote (central) repository. This remote repository is usually accessible through a website (e.g. GitHub or GitLab). The developer has to send (push) the changes to the remote repository. Only after that the other collaborators can see the file updates via one of the aforementioned websites.

There are several models of workflow based on the way the developer interacts with the version control system: Centralised workflow, Feature Branch workflow (GitHub flow), Gitflow, Forking workflow and GitLab flow (*Comparing Workflows*, n.d.). There is no standardised process on how to interact with Git, so these workflow models help to ensure the software changes in the version control are handled the same way throughout the team. The models of workflow mentioned before will be further described and compared in this subchapter.

Centralised workflow Centralised workflow uses one repository for all the project files and changes to them. The default development branch is called master, and all the changes are committed to it.

The flow of work goes as follows:

1. Developer clones the central repository
2. Developer makes changes to the files in his/her local repository
3. Developer commits and pushes the changes to the central repository

Feature Branch workflow (GitHub flow) The core idea of the Feature Branch workflow is that the development of each new feature should happen in a designated branch. This way the main branch - master - never contains broken code. When the developer finishes working on an update, he/she can create a pull request so the other developers will be able to check/test the new code and then the head of the project can integrate it in the main master branch.

The workflow of this method is described below:

1. Developer clones the master branch
2. Developer locally creates a new branch (based on master) with the name of a feature he/she is working on
3. Developer makes changes to the files on the feature branch
4. Developer commits and pushes the changes to the central repository and creates a pull request
5. The updated branch gets tested by other developers and the head of the project integrates the feature branch into the master branch

Gitflow This workflow is similar to the Feature Branch workflow - it also involves having a master branch and feature branches. In addition to that, Gitflow allows having separate branches for preparing, maintaining and recording releases. In Figure 10 an example of Gitflow branching is presented. The master branch has the main version of the application and the developers are working on the develop branch, creating feature branches and merging them back to develop. The feature branches never interact with the master. When it is time to release a new version of the application the release branch is used.

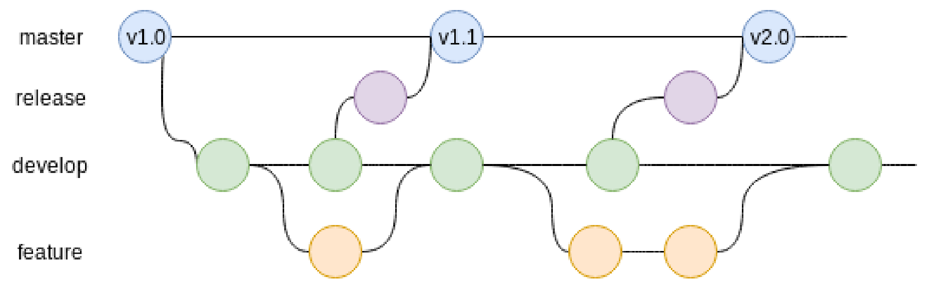


Figure 10: Gitflow branching

The flow of work with the Gitflow method is as follows:

1. The project leader creates the develop branch from the master branch
2. Developer clones the develop branch
3. Developer locally creates a new branch (based on develop) with the name of a feature he/she is working on
4. Developer makes changes to the files on the feature branch
5. Developer commits and pushes the changes to the central repository and creates a pull request
6. The updated branch gets tested by other developers and the head of the project integrates the feature branch into the develop branch
7. When the features on the develop branch are ready for release, a release branch is created from develop. It gets tested and merged with master once it is ready

Forking workflow With the Forking workflow instead of using one central repository, every developer has their own. All the project files are still stored in the main project repository but the development does not happen there. Each developer working on the project should fork (copy) the main project repository to their own account and develop the new features there. Once the development of a feature is done, the developer can file a pull request to the main project repository. After that the project leader can pull the changes to the master branch of the main repository. Forking workflow is very common to use in open source projects.

Below is an example of the Forking workflow:

1. A developer forks the main project repository to his/her own account and clones the project from there

2. Developer makes changes to the files of the project
3. Developer commits and pushes the changes to the repository on his/her account
4. Developer creates a pull request from his repository to the main project repository
5. The project leader checks the changes, approves and merges them into the main project repository

GitLab workflow GitLab is a code hosting platform for version control, as well as a tool for project planning, code management, Continuous Integration and Delivery. GitLab has their own workflow model based on Gitflow and Feature Branch workflow. Similarly to these two workflow models, GitLab workflow proposes to have one master branch which acts as a main releasable version of the application and separate branches for all features that are being worked on.

Additionally to that, GitLab workflow has a set of rules helping to structure and simplify the development process. For releasing created applications or features to public, separate branch called production can be used in addition to master. All the commits to all branches should be tested. The deployments of software should be automated. Generally, a lot of the rules of the GitLab workflow are set to incorporate the ideas of Continuous Delivery.

To compare the workflows and choose the one most suitable for Mentech Innovation the comparison parameters have been defined based on the thesis author's opinion and the requirements received from Mentech Innovation. The parameters are defined based on the author's and company's choice because to the best knowledge of the thesis author, there is no academic research on any of the aforementioned workflows being better than the other workflows. Additionally, the workflow to use is usually determined by the human preferences and team needs.

The comparison parameters were chosen as relevant because they would help to see if a workflow is a good match with Continuous Delivery. Table 6 has been created to summarise the information about the workflow models based on the predefined parameters.

The comparison of the workflow models is presented in Table 7 as a selection matrix with the same comparison parameters as Table 6. As well as in the comparison of deployment strategies, scores from 0 to 5 (where a higher figure is better) were given to each workflow based on the answers in Table 6. Additionally, the comparison parameters have been given the weights from 0 to 5 (where more is more important) based on the importance of them for Mentech Innovation. In the bottom of the table the end score for each workflow is calculated as a sum of all the parameters multiplied with their weights.

The weights have been determined based on the following reasoning. The compatibility of a workflow with the concept of Continuous Delivery is the most important, because in the future Mentech was planning to be using Continuous Delivery. The parameters that go hand in hand with Continuous Delivery - code review, testing and fast fixing of errors, have also been considered important to have in a workflow for Mentech Innovation. Additionally, the workflow model should be simple to encourage the team to use it and to reduce the amount of possible errors and misunderstandings. As Mentech team is still growing, the workflow should scale with its size. Mentech Innovation software developers are working on several fairly complex applications therefore it is important that the workflow model supports that and is typically used for that. The primary repository will be used for deployment therefore it is preferable but not necessary to keep it clean and always in a working state.

The resulting scores of the selection matrix (Table 7) show that the most suitable and beneficial workflow for Mentech Innovation is the GitLab workflow.

Table 6: Summary of workflows

	Centralised	**Feature Branch**	**Gitflow**	**Forking**	**GitLab**
Simple	Yes, only one branch	Only the master branch and feature branches	Similar to Feature Branch but with more branches	Slightly more work because of forking	Only the master branch and feature branches but more branches possible
Typical use	Small projects that don't change often	Large teams or projects	Large teams or projects	Open source projects	Large teams or projects deploying continuously
Code review	Not promoted	Promoted with merge requests	Promoted with merge requests	Promoted with merge requests	Promoted with merge requests
Scaling with team size	Hard to manage in big teams	Easy to manage, team size does not matter	Easy to manage, team size does not matter	Easy to manage, team size does not matter	Easy to manage, team size does not matter
Testing before merging	Not enforced	Promoted with merge requests	Promoted with merge requests	Promoted with merge requests	Promoted as one of the rules
Clean primary repository	No	Yes	Yes	Yes	Yes
Compatible with Continuous Delivery	Possible to set up, however lacks support for some stages	Possible to set up, however lacks support for some stages	Works with release and hotfix stages	Possible to set up, however lacks support for some stages	Yes, takes in account the continuous testing and deploying

Table 7: Selection matrix for workflows

	Centralised	Feature Branch	Gitflow	Forking	GitLab	Weight
Simple	5	4	4	3	4	4
Typical use	2	4	4	3	5	3
Code review	0	5	5	5	5	4
Scaling with team size	1	5	5	5	5	3
Testing before merging	1	4	4	4	5	4
Clean primary repository	0	5	5	5	5	3
Compatible with Continuous Delivery	3	3	4	3	5	5
Score:	48	109	114	102	126	

4.8 Conclusion

As a result of the Literature Research phase of the thesis project the goals that had been set in the beginning of the phase were reached and the research questions for this phase could be answered.

The general structure of the deployment pipeline is presented in Figure 5. It features all the steps a software update has to go through to reach the users. The stages chosen for the deployment pipeline are Commit, Integration, Acceptance, User Acceptance, Capacity, Staging and Production. Additionally, the source code and the environment and application configurations will be stored in version control and the package, reports and metadata - in an Artifact repository.

To create a risk mitigation strategy the main project risks related to Continuous Delivery were identified. To assess their severity, they were evaluated by their impact and likelihood. Based on that, the risk percentage for the whole project was calculated - 35%. To prevent the risks from happening a mitigating rules were offered to help mitigate each of the risks. It was found, that, generally, with the proper communication within the team, control from the management and lead engineers and group effort it is possible to successfully practice Continuous Delivery.

Several ways to deploy an application have been investigated during the Literature Research phase: recreation, blue-green, canary, A/B and shadow deployment strategies. All of them define a way the product is released to the clients, ensuring the quality of it and enabling the developers to rollback a deployment in case some problems arise. After the comparison of these strategies canary deployment was chosen as the most suitable deployment method to use at Mentech Innovation.

There are several development workflows nowadays, based on the way the developer interacts with the version control system: Centralised, Feature Branch, Gitflow, Forking and GitLab workflow. After a research into these models of work it was found that GitLab flow is the most compatible with Continuous Delivery as a lot of its rules are set to incorporate the ideas of Continuous Delivery. Therefore GitLab flow was chosen as the most beneficial workflow for Mentech Innovation.

To sum up, during the Literature Research phase of the thesis project the theoretical basis for an implementation of Continuous Delivery at Mentech was formed. During the next - implementation - stage the chosen deployment strategy, development workflow, risk mitigation strategy and the general pipeline structure were used to implement the Continuous Delivery pipeline for the HUME website of Mentech Innovation.

5 DESIGN AND IMPLEMENTATION

In this chapter the design and implementation of Continuous Delivery for the HUME website of Mentech Innovation are presented. First the deployment pipeline structure for the website was created and the tools were chosen, and the design questions of chapter 2.4.2 were answered. Then the actual implementation of the pipeline for the website was done. This includes the setup for the Continuous Delivery tool, version control flow and the tools for building and testing the software.

5.1 HUME website description

HUME website is the user interface of the Mentech software system (HUME). An image of the website can be seen in the Appendix 1. HUME website gets the data from sensors and outputs it in a visual form. On the website the resources like clients (actual clients of Mentech Innovation) and sessions (measurements taken from the clients) can be managed. To do that, the website has a side menu bar where the lists of clients and running sessions are displayed. When one of the clients or sessions is clicked, the information per client about the measurement sessions or a graph with the running session is displayed.

The information on the HUME website comes from the web service through HTTP requests and web sockets. The website communicates with the web service, which is connected to the database, so for the HUME website to operate both the web service and the database are needed. Additionally, the website access is restricted for security purposes using Keycloak (access management tool), so for integration tests Keycloak needs to be present.

HUME website is written using the following technologies: HTML, SCSS and Vue.js. For version control Git and GitLab are used.

5.2 HUME website deployment pipeline

Based on the general deployment pipeline for Mentech Innovation (Figure 5) a custom pipeline for the HUME website had to be made. Compared to the general deployment pipeline, two of the stages - User Acceptance (manual tests involving users) and Capacity - were removed. Capacity tests are not possible to perform on a website, because these tests are server-side, not client-side. User Acceptance stage was found not necessary, as the pipeline has manual testing in the Staging environment as well.

The flow of work in the pipeline is similar to the one described in chapter 4.4 except for the last steps (due to the removed stages). The changes to the website have to be committed to version control by a developer. Then the rest of the pipeline executes automatically. The website is built, then various kinds of tests are performed on it, then it is released to the staging environment and optionally to production.

Therefore, the first design question of chapter 2.4.2 can now be answered. The structure of the deployment pipeline can be seen in Figure 11 and the description of the steps of the pipeline can be found in chapter 4.4.

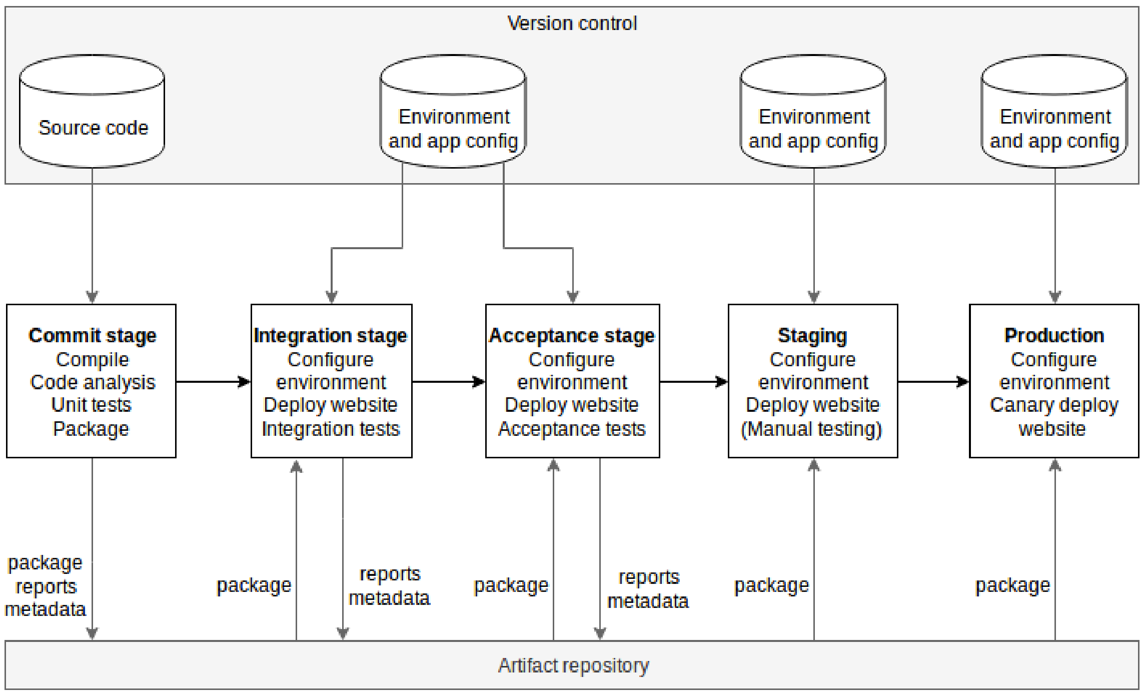


Figure 11: HUME website deployment pipeline

5.3 Selection of tools

The following section lists all the products and tools the author of this thesis chose to use for the implementation of Continuous Delivery for the HUME website of Mentech Innovation. Additionally, the reasoning behind the choice of the tools is provided, as well as the evaluation of the other options, where possible.

5.3.1 Version control

For version control Git and GitLab (*GitLab*, n.d.) were used. These were the technologies of choice of Mentech Innovation and the developers had good experience with them therefore they were the tools used for version control for this project as well.

5.3.2 Artifact repository

For an Artifact repository several existing Artifact repository solutions - Cloudsmith, Packagecloud, JFrog Artifactory, Nexus Repository Pro and GitLab - were investigated. It was decided to use GitLab, because it was already being used for storing artifacts in the other projects of Mentech Innovation. Additionally, the other options did not offer any features that would make the switch from GitLab to the other solution worth it.

5.3.3 Continuous Delivery tool

To orchestrate the whole process of Continuous Delivery, to actually make the software update go through the commit, testing and deployment stages (defined in the Figure 11) a Continuous Delivery tool was needed. To choose the tool that is best suitable for

Mentech Innovation the hard and soft requirements were defined and the most popular tools found on the Internet were compared.

Initially for the comparison the following tools were chosen: Jenkins, GitLab CI, VSTS, Bamboo, Codeship, Codefresh, TeamCity, Travis CI, GoCD, CircleCI and Drone. Mentech Innovation has set as the hard requirements for the Continuous Deivery tool to be availiable as a service (be cloud-hosted) and be compatible with GitLab. TeamCity, Drone and Jenkins are not availiable as a service and GoCD, VSTS, Travis CI and CircleCI are not compatible with GitLab. Additionally, Bamboo was found to have too little information and documentation on the official website to be easy to use. For these reasons the mentioned tools did not participate in the further comparison.

GitLab CI (*GitLab Continuous Integration & Delivery*, n.d.), Codeship (*Continuous Integration, Deployment & Delivery with Codeship*, n.d.) and Codefresh (*Codefresh*, n.d.) were further investigated and their offered features were compared. The gathered information as well as the comparison parameters can be seen in Table 8. The comparison parameters have been defined by the thesis author according to the Mentech Innovation requirements.

Table 8: Summary of Continuous Delivery tools

	GitLab CI	**Codeship**	**Codefresh**
Price ($/month)	95	75	0
Amount of repositories	Unlimited	Unlimited	Unlimited
Amount of users	5	Unlimited	Unlimited
Amount of concurrent jobs	Unlimited	1	2
Docker support	Yes	Yes	Yes
Local testing	Yes with GitLab-runner	Yes, with Jet	Yes
Kubernetes support	Yes	Yes	Yes
Configuration as code	Yes	Yes	Yes
Open Source	Yes	No	No
Community support (according to stackshare.io on 4 january 2019)	4.44K Reddit Points, 1.85K Stack Overflow Questions	1.3K Reddit Points, 206 Stack Overflow Questions	1.04K Reddit Points, 0 Stack Overflow Questions
GitLab compatibility	Yes	Yes	Yes
Cloud hosted	Yes	Yes	Yes
Notifications	Email, web, Slack, etc.	Email, Slack, custom notifications	Email, Slack
GitLab OAuth authorisation	Yes	Yes	Yes

The selection matrix - Table 9 - was created based on Table 8. As well as in the other selection matrices in this document the scores from 0 to 5 (where more is better) were given to each Continuous Delivery tool based on the information in Table 8. Additionally, weights from 0 to 5 (where more is more important) have been assigned to the comparison parameters. These weights have been determined by the thesis author based on importance of them for Mentech and have been approved by the lead engineer of the company. In the bottom of the table the end score for each Continuous Delivery tool is calculated as a sum of all the parameters multiplied with their weights.

Table 9: Selection matrix for Continuous Delivery tools

	GitLab CI	**Codeship**	**Codefresh**	Weight
Price ($/month)	2	3	5	3
Amount of repositories	5	5	5	4
Amount of users	2	5	5	3
Amount of concurrent jobs	5	2	3	2
Docker support	5	5	5	5
Local testing	5	5	5	5
Kubernetes support	5	5	5	3
Configuration as code	5	5	5	5
Open Source	5	0	0	3
Community support	5	4	3	4
Notifications	5	3	2	2
GitLab OAuth authorisation	5	5	5	2
Score:	187	170	197	

The reasoning behind the parameters is as follows: Mentech Innovation uses Docker for containerisation for the applications to run in which eases the environment configurations. There is practically no other as popular tool for this job so it is important for the Continuous Delivery tool to be compatible with Docker. Additionally it is extra important for the Continuous Delivery tool to support local testing (for debugging pipelines) and configuration as code (for version control and reproducibility) as it is one of the rules of Continuous Delivery. Mentech Innovation has a lot of projects ongoing so the amount of repositories offered should be high. Community support of the tool of choice is important because it allows to get the feedback to the arising questions faster and generally allows to find more information about the use of the tool on the internet. The price of the tool per month is not very important (unless its extremely high) but it is of course nicer if it is lower. The Software Development team of Mentech Innovation consists of 5 people so the tool should minimally offer support for 5 users, and more is better. Kubernetes is one more tool which at the time of writing this (7 January 2019) was considered to be used to Mentech Innovation, so it is better if the chosen Continuous Delivery tool supports it. Open Source tools are the preferred tools of Mentech Innovation. The amount of concurrent jobs the tools of choice offers is not very important for Mentech but more is better. It is important to get notifications when the job is finished or there is an error with a pipeline, but the way notifications are managed is not very important. Together with the use of GitLab the tool should support authorisation through it, but other ways of authorisation would also be fine.

From Table 9 it is visible that Codefresh gained the highest score therefore it should used for managing Continuous Delivery at Mentech Innovation.

5.3.4 Commit stage

In the following subsection the tools chosen for each step of the Commit stage of Continuous Delivery are listed and their choice is motivated.

Compile and Package Webpack was chosen as the tool to use for compiling and packaging the code for two reasons. First, Mentech Innovation was using Webpack for their websites so the developers have experience with it and their opinion of it was positive. Second, compared to the other tools for compiling software (Grunt, Browserify), Webpack has better features and easier configurations. Webpack can handle JavaScript, CSS and image files, it can minify these files which is good for website optimisation and split resources into bundles to reduce the website loading time. The configurations of Webpack are also shorter than of other similar tools, so the errors are less likely to occur and less time will be spent on configuration debugging. (Möller, 2018, 11-12.)

Code analysis ESLint and Flow were used for code analysis. ESLint helps the developers to make the code more consistent and to avoid bugs by introducing guidelines for the code writing style (Hautaviita, 2018, 17). ESLint was chosen because it is free and open-source and it is more popular than similar tools - on 6 December 2018 ESLint had 12888 stars on GitHub, compared to a similar tool, JSHint, which had 8066 stars. Stars on GitHub allow the users to mark a project as ”favourite”, so these stars can be an indicator of popularity of a project. ESLint allows the developers to build their own set of rules for code analysis or to use a predefined set of rules, which also can be adjusted. Therefore ESLint is very flexible and easy to use as well. (Paulasaari, 2018, 46-48.)

JavaScript language does not have strong data types, which can cause bugs that are hard to notice (e.g. possibility of inputting a string value in a field for an integer). Flow is an open-source type checker, that is used to prevent these kind of bugs by allowing the developers to enable the enforced use of data types. Flow is the only tool for JavaScript that has this functionality. (Paulasaari, 2018, 50.)

Unit tests In Table 10 the tools for JavaScript unit testing are listed and compared. The tools that were chosen for comparison are Mocha (*Mocha*, n.d.), Jasmine (*Jasmine Documentation*, n.d.), Jest (*Jest*, n.d.) and AVA (*Ava*, n.d.). These are the most widely used tools for unit testing. The parameters for comparison were derived from the features the tools provided. The features that are present in the frameworks are marked with an X.

From the comparison in Table 10 we can see that the Jest unit testing framework is the most feature complete one. For this reason it was decided to use Jest for JavaScript unit testing at Mentech Innovation.

5.3.5 Integration stage

Configure environment Docker Compose (*Docker Documentation*, n.d.) was used to configure the environment for the website to run in. This tool was used in the company on other projects and the developers had good experience with it and, additionally, there are practically no other tools to perform the job that Docker does.

Table 10: Comparison of unit test tools

	Mocha	Jasmine	Jest	AVA
Provides a testing structure	x	x	x	
Integrates well with Vue	x	x	x	x
Provides assertion functions		x	x	
Generates and displays test results	x	x	x	
Snapshots of components possible			x	x
Provides mocks, spies and stubs		x	x	
Code coverage reports			x	
Running tests in parallel			x	x

Deploy website To deploy the website Codefresh was used as it was the tool generally used for the whole automation of the Continuous Delivery processes.

Integration tests It was found that it was possible to use the same tool for the integration stage as for the unit testing stage. Therefore, Jest was used for the integration testing of the software at Mentech Innovation as well.

5.3.6 Acceptance stage

Configure environment, Deploy website For configuring the environment and deploying of the website in this stage the same tools were used as listed and described in the Integration stage section.

Acceptance tests For acceptance tests several most popular user interface testing frameworks were evaluated - Puppeteer, WebdriverIO, Cypress, Nightwatch.js, PhantomJS and TestCafe. For the Mentech Innovation use case it was important to choose a framework that has good documentation, is actively supported and is preferably free to use. For this reason, only Puppeteer, WebdriverIO, Cypress and Nightwatch.js have been further compared via empirical research - the thesis author tried to install and use each of them for writing tests for the HUME website. During the research it was found that the tests in Nightwatch.js were the most readable and easy to write. Additionally, Nightwatch.js has good documentation (unlike Cypress), is easy to install (unlike WebdriverIO) and has cross-browser support (unlike Puppeteer) (*Nightwatch.js*, n.d.). Therefore, Nightwatch.js was used for acceptance testing of the HUME website.

5.3.7 Staging and Production

Configure environment, Deploy website The environment that is used to run the HUME website in the cloud is Amazon Web Services Simple Storage Service (S3) and CloudFront. These tools were chosen by the lead engineer of Mentech Innovation, as the choice of them is out of scope of this project. Amazon S3 is a storage service used to store the files of the website in the cloud and CloudFront delivers these files to the customers.

To configure the environment and deploy the website, similarly to the previous stages, Codefresh was used.

5.3.8 Conclusion

In the previous chapters the tools that were used for the Continuous Delivery pipeline of the HUME website were described and the choice was given a reasoning. Therefore, the second question of chapter 2.4.2 can now be answered. The tools that are used for the deployment pipeline are as follows: GitLab for version control and artifact repository, Codefresh for running the pipeline, Webpack for compiling and packaging the code, ES-Lint and Flow for code analysis, Docker to configure the environments, Jest for unit and integration tests and Nightwatch.js for acceptance tests. The website is deployed to the cloud which works with the Amazon Web Services tools.

5.4 Setting up version control

The GitLab repository has been set up and used at Mentech Innovation before the start of this project. However the new flow of work - GitLab workflow - needed to be established. At the beginning of the project Mentech had several branches on GitLab that were not used, as well as the master branch and the feature branches. The obsolete branches were deleted. With the master branch and the feature branches the GitLab workflow was set up.

GitLab is also used as an artifact repository. It did not need to be set up, as it worked out of the box, no configuration was necessary.

5.5 Setting up the Continuous Delivery tool

To set up the Continuous Delivery tool a configuration file (*codefresh.yml*) was created in the HUME website project. This file contains all the settings for the Codefresh pipeline. The *codefresh.yml* file lists the stages that the software has to go through to be released (such as building, unit testing, etc.) and the setup for these stages.

As it can be seen in Figure 11 there are 5 stages in the HUME website deployment pipeline. However generally they can be divided into four - build (commit), test (code analysis, unit, integration and acceptance), staging and production. Therefore the resulting Codefresh pipeline consists of four stages, and these stages have substages to represent all the steps of the HUME website deployment pipeline. The image of the resulting pipeline steps can be seen in Figure 12. The stages on the image are represented in colours - steps of the build stage are grey, test steps are blue, staging is purple and production is green. Each of the steps of the pipeline is described in this chapter.

For each step of the pipeline several parameters can be specified. For most of the steps the parameters that had to be set were an image, stage and commands. The image setting refers to a Docker image - a file containing libraries, tools, and other files necessary to be able to run the application in a specific environment. The pre-made images can be found on the Docker Hub (library for container images). Additionally, it is possible to create a custom image. In the steps of the pipeline described further a pre-made image was used everywhere where it is not stated otherwise. The stage refers to one of the three stages mentioned before and is needed to visually separate the files into categories. The commands list all the bash scripts to perform the actions that have to happen in the step.

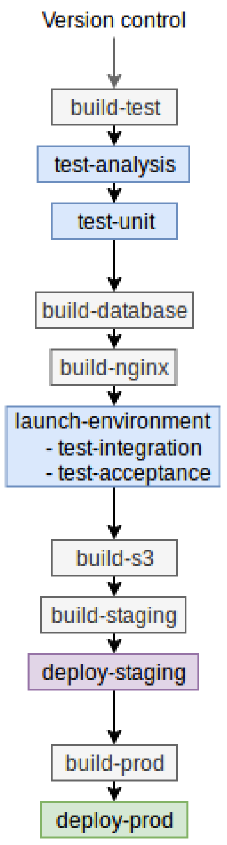


Figure 12: Deployment pipeline steps

The first step of the pipeline is *build-test*. This step belongs to the build stage. The commands set up the access to GitLab for the project, download all the necessary libraries and build the website ready for testing.

The second step is *test-analysis*. This step belongs to the test stage. During this step ESLint is set to check the project files, the setting up and the functionality of ESLint is described further on in chapter 5.6.

The following step is *test-unit*. This step belongs to the test stage. During this step a script is run to execute the unit tests.

Next step is *launch-environment*. It is needed to launch the database, web service, Keycloak, browser for acceptance tests, and the Nginx tool as a server for the website. These things have to be launched in order for the website to be tested to run in a similar environment as it runs in production.

To launch the database an image of it had to be made first as the aforementioned Docker Hub did not have the database image that could be used for this project. This image specifies the management system of the database (PostgresSQL) and lists the data that has to be in this database (columns, rows, entities, etc). Similar image had to be made for Nginx, as an extension of an existing image from Docker Hub, specifying the Nginx image from Docker Hub to be used and the files of the website to be used with it.

The *launch-environment* step uses these images as well as the images from Docker Hub to create the environment for the HUME website to run and be tested in. Due to that the tests of the software will be run during this step and therefore this step belongs to the test stage. This step has two substeps: *test-acceptance* and *test-integration*. During each of them, the scripts for running the tests are executed. The setup of the testing tools and functionality of tests are described further in chapters 5.7 and 5.8.

After the *launch-environment* step, an image had to be made for the deployment to Amazon S3, as an existing pre-made image used different parameters from the ones that were needed. The newly created image was an extension of the pre-made one. Then the website had to be rebuilt with the settings for the staging environment. Next step is *deploy-staging*. It belongs to the staging stage. During this step the website files are pushed to the staging environment in the cloud.

As it is mentioned in the scope of the thesis (chapter 2.2.1), the website should be released to the production-like environment, which is staging in this case. However, for the ease of the future deployments to production, it was decided to add one more step to the *codefresh.yml* file. A conditional step *deploy-prod* was added. This step is executed only after the manual approval of the management/lead engineer of Mentech Innovation. The configurations of this step are essentially the same as the configurations of the *deploy-staging* step. However during this step the website files are pushed to the production environment in the cloud. Staging and production environments are the same in configuration, the difference is that the production version of the website can be used by clients and the staging version can be used by the developers of Mentech for testing.

After creating the *codefresh.yml* file, an account on the Codefresh website was made by the thesis author. Codefresh prompted the user to connect a GitLab repository for which the deployment pipeline should be run, in this case it is the HUME website repository. Then the setup method for the pipeline must be chosen, in this case it is the *codefresh.yml* file which was described earlier. The file is presented for review and then the deployment pipeline is automatically created. After that, with every push of an update to the HUME website to GitLab the pipeline will be automatically run and the developers will be notified by email about the status of it (success or fail). It is also possible to see in real-time on the website of Codefresh how the pipeline is running and view the console log for the status of the pipeline and whether any errors occur and where.

5.6 Setting up the Commit stage

For the Commit stage to be set up for the software of Mentech Innovation all the tools listed in chapter 5.3.4 had to be configured.

Compile and Package For compiling and packaging the website Webpack was added to the website project. For the setting up of it a configuration file was made in the root of the project. The configuration included the information about the project files that need to be exported, the rules for loading the files, etc. Additionally, the script for starting up Webpack was added to the scripts section of the *package.json* file.

Code analysis For configuring the ESLint tool the packages needed for it were added to the *package.json* file of the project (*Configuring ESLint*, n.d.). The configuration file for ESLint was automatically generated. This file defines the rules for ESLint to check the code, such as the amount of spacings, positioning of brackets or variable naming. Two plugins - for the support of Flow and Vue - were added to the file, and some pre-made rules were overwritten.

Unit tests To set up Jest, its package and configurations were added to the *package.json* file of the project (*Jest: Getting Started*, n.d.). The configurations were copied from an already existing project of Mentech Innovation and edited to fit the case. The configuration file enables Jest to be used together with Vue.js, it specifies the tools used to transform the files (for example *vue-jest* tool transforms Vue.js files into HTML, CSS and JavaScript), and lists the directory for the setup file used for getting information from the web service. Unit tests were written to check the functions of the website components. For instance, one of the tests ensures that the component default data is being set correctly when the component loads. The test creates a component instance and expects the value of a certain text field in a component to be equal to the preset value.

5.7 Setting up the Integration stage

In the following chapter the setup of tools listed in chapter 5.3.5 will be described.

Configure environment, Deploy website The setup of the Docker images and the configuration of Codefresh are described in chapter 5.5.

Integration tests As Jest was used for the integration tests as well as for the unit tests, no setting up of the tool was needed. Several tests were written to ensure the website's compatibility with the other elements of the Mentech Innovation system. For instance, the connection to a web service from which the information about the clients and sessions gets delivered to the website was tested. During the test the connection to the web service was established, and the existing website functions were used to try to get the list of clients from the database or to stop a running session for a client.

5.8 Setting up the Acceptance stage

To set up the Acceptance stage the tools described in chapter 5.3.6 had to be configured.

Configure environment, Deploy website The setup of Docker images and the configuration of Codefresh are described in chapter 5.5.

Acceptance tests For the acceptance tests the Nightwatch.js package was added to the HUME website project (*Nightwatch.js*, n.d.). Additionally, a web driver (driver for a browser to be able to run tests in it) had to be installed. The options for web drivers were listed on the Nightwatch.js website. GeckoDriver was chosen for use, as it is the driver for Firefox, which is the browser of choice of the thesis author.

The configuration file for Nightwatch.js had to be created. It specifies the path to the folder, in which the test files are, and the settings for the web driver (e.g. on which port the browser should run). A script for running the acceptance tests was written in the *package.json* file.

With Nightwatch.js it was possible to automatically test the actions of all the items on the HUME website, programming the browser to click on buttons, fill in the input fields in forms and check if the elements of the page loaded correctly. For example, during one of the automated tests the website would be opened, the username and password would be filled in to login and the access to the website would be obtained. Then the functioning of the website would be checked - whether the page has loaded correctly (e.g. all the images and texts on it), whether the lists of clients and sessions have been loaded from the web service correctly and whether the page loads at a reasonable time.

5.9 Setting up Staging and Production environments

The setup of both environments was out of the scope of the project, as the setup of the cloud environment was needed for the whole software system, not only for the website. The setup of the Docker image for Amazon S3 and the configuration of Codefresh are described in chapter 5.5.

5.10 Conclusion

As a result of the Design and Implementation phase the design of the deployment pipeline for the HUME website was made, the research questions for this phase were answered and the Continuous Delivery pipeline was set up.

The structure of the HUME website pipeline is presented in Figure 11. This pipeline is similar to the general one (Figure 5) created during the literature research phase, except for the User Acceptance and Capacity stages that were removed. According to the pipeline structure, after an update to the code of the project is pushed to version control, the website should automatically be built, the code should be analysed, tested (unit, integration and acceptance) and released into a staging environment.

During the Design and Implementation phase the tools to implement the deployment pipeline were chosen. The selected tools are: GitLab for version control and as an artifact repository, Codefresh to run the deployment pipelines, Webpack to compile and package the code, ESLint and Flow for static code analysis, Docker for environment configurations, Jest for unit and integration tests and Nightwatch.js for acceptance tests. The staging and production environments of the website are running in the cloud which works with Amazon Web Service tools.

The tools mentioned above have been configured to work together. A working deployment pipeline was created according to the planned structure of it. The running of the pipeline, including all the stages and the console log printing out the state of the pipeline, can be seen by the developers of Mentech Innovation on the Codefresh website.

6 VALIDATION

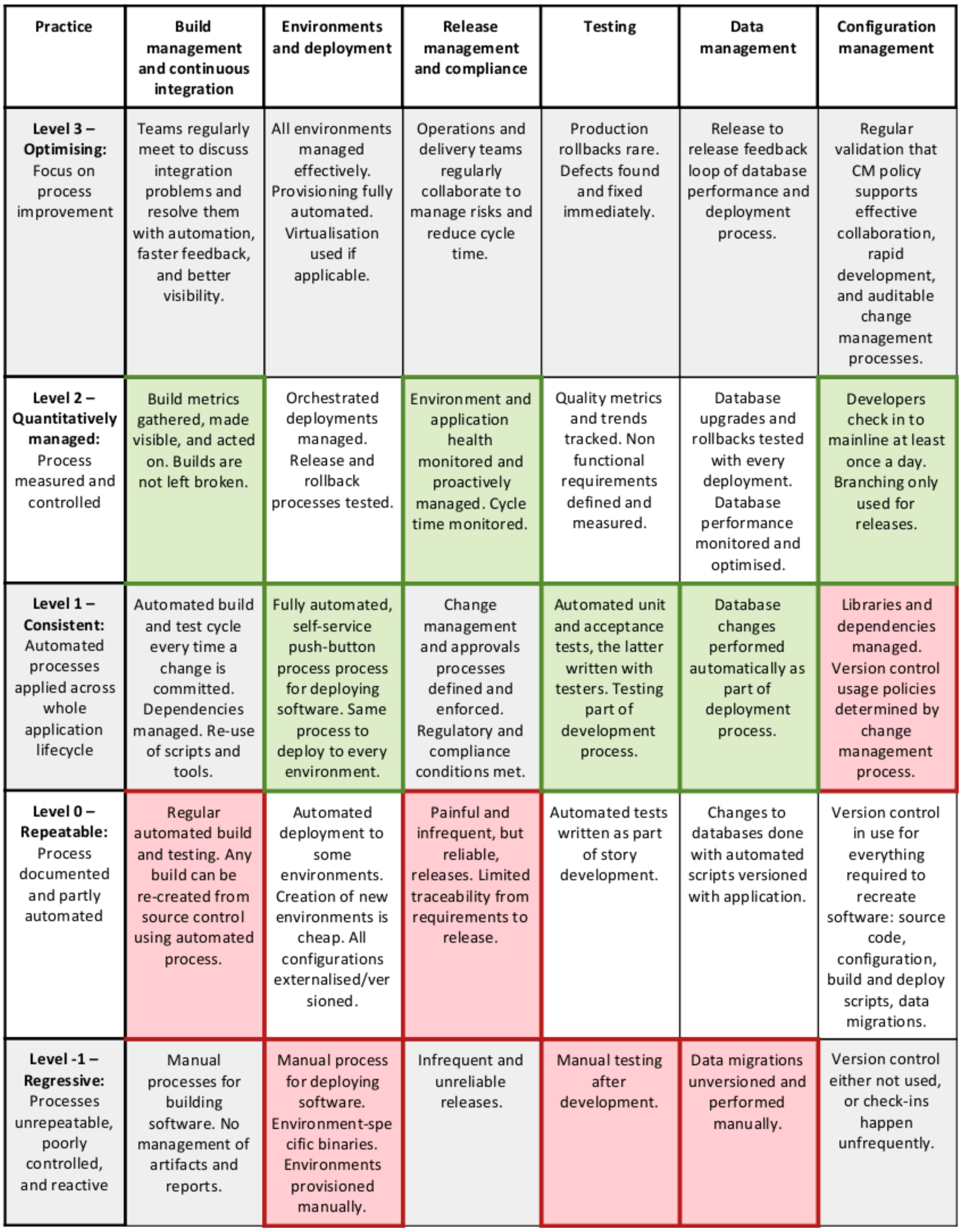
The aim of this project was to examine the concept of Continuous Delivery, evaluate its risks and benefits and, if the concept proves to be theoretically beneficial, to implement Continuous Delivery for the HUME website of Mentech Innovation. The implementation phase included the configurations of the tools and infrastructure of Continuous Delivery. This chapter will demonstrate the validation of the Continuous Delivery concept and its implementation. It includes the theoretical assessment of the concept benefits as of the end of the project, a comparison of the old way of deploying software with the new deployment pipeline and a screenshot and description of the resulting Codefresh pipeline.

For the theoretical assessment the Maturity Model is used. This model can be used to classify the release management maturity of a company, in this case Mentech Innovation. The model uses several parameters, defining the software development processes and practices. These parameters have different levels of maturity, so the company can see how their working practices can improve. (Humble & Farley, 2011, 419.)

The model is presented on the Figure 13. In red the level per each parameter on which the software of the HUME website of Mentech Innovation was before the implementation of Continuous Delivery is marked. The software build was automated, however the tests were not. The environments for each build were created manually. Releases were reliable, however very time consuming. Tests were not written. Data migrations were manual. Version control was set up well, however there was no defined way of how to work with branches, which resulted in a lot of unused branches.

In green the levels per each parameter on which the software of the HUME website of Mentech Innovation was after the implementation of Continuous Delivery are marked. The build-test-release process is automated and is repeatable and reliable. The visual board and the console log of the Codefresh website allow the developers to see the status of the deployment pipeline clearly and act on it. Additionally, all the team members receive emails with the status of the deployment pipeline (success/fail), so any errors can be proactively managed. All the testing is automated as well as the database changes. The Codefresh website displays the time in which the pipeline runs - on average it is 10 mi-nutes. To sum up the level per each parameter moved two levels up on average, and in general the software deployment process can be considered quantitatively managed or consistent.

To visualise the benefit of using Continuous Delivery for the HUME website software releases, a value stream map was created and it is displayed on Figure 14. This map is an updated version of the one that can be seen in the Chapter 4.3. The new value stream map has been approved by the lead engineer of Mentech. Compared to the old version, this value stream map shows a clear improvement in the development & continuous testing, system testing and release stages. The system testing and release value-added time has decreased, as well as the elapsed time between stages. We can see that there is no elapsed time between the development & continuous testing and system testing stages anymore, because the developers can now system test the software by one click of a button whenever they are ready. The only period of elapsed time of the software delivery part of the value stream map (between the system testing and release stages) depends now on a management decision when to release the HUME website to the clients.



Practice	Build management and continuous integration	Environments and deployment	Release management and compliance	Testing	Data management	Configuration management
Level 3 – Optimising: Focus on process improvement	Teams regularly meet to discuss integration problems and resolve them with automation, faster feedback, and better visibility.	All environments managed effectively. Provisioning fully automated. Virtualisation used if applicable.	Operations and delivery teams regularly collaborate to manage risks and reduce cycle time.	Production rollbacks rare. Defects found and fixed immediately.	Release to release feedback loop of database performance and deployment process.	Regular validation that CM policy supports effective collaboration, rapid development, and auditable change management processes.
Level 2 – Quantitatively managed: Process measured and controlled	Build metrics gathered, made visible, and acted on. Builds are not left broken.	Orchestrated deployments managed. Release and rollback processes tested.	Environment and application health monitored and proactively managed. Cycle time monitored.	Quality metrics and trends tracked. Non functional requirements defined and measured.	Database upgrades and rollbacks tested with every deployment. Database performance monitored and optimised.	Developers check in to mainline at least once a day. Branching only used for releases.
Level 1 – Consistent: Automated processes applied across whole application lifecycle	Automated build and test cycle every time a change is committed. Dependencies managed. Re-use of scripts and tools.	Fully automated, self-service push-button process process for deploying software. Same process to deploy to every environment.	Change management and approvals processes defined and enforced. Regulatory and compliance conditions met.	Automated unit and acceptance tests, the latter written with testers. Testing part of development process.	Database changes performed automatically as part of deployment process.	Libraries and dependencies managed. Version control usage policies determined by change management process.
Level 0 – Repeatable: Process documented and partly automated	Regular automated build and testing. Any build can be re-created from source control using automated process.	Automated deployment to some environments. Creation of new environments is cheap. All configurations externalised/versioned.	Painful and infrequent, but reliable, releases. Limited traceability from requirements to release.	Automated tests written as part of story development.	Changes to databases done with automated scripts versioned with application.	Version control in use for everything required to recreate software: source code, configuration, build and deploy scripts, data migrations.
Level -1 – Regressive: Processes unrepeatable, poorly controlled, and reactive	Manual processes for building software. No management of artifacts and reports.	Manual process for deploying software. Environment-specific binaries. Environments provisioned manually.	Infrequent and unreliable releases.	Manual testing after development.	Data migrations unversioned and performed manually.	Version control either not used, or check-ins happen unfrequently.

Figure 13: Maturity Model (Humble & Farley, 2011, 419)

In the old version of the value stream map, the system testing and release stages would in total take 28% of the total time, whereas with the Continuous Delivery enabled these stages take 9% of the total time, and the leftover time can now be spent on the development and continuous testing of the HUME website. Therefore, the enabling of Continuous Delivery has clearly benefited the whole process of software release.

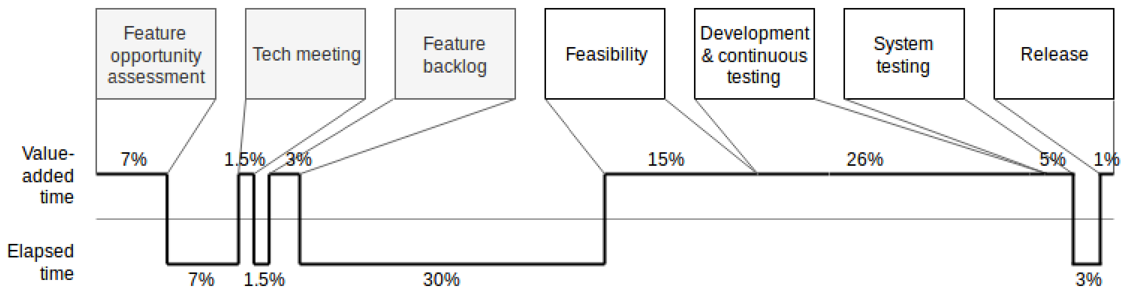
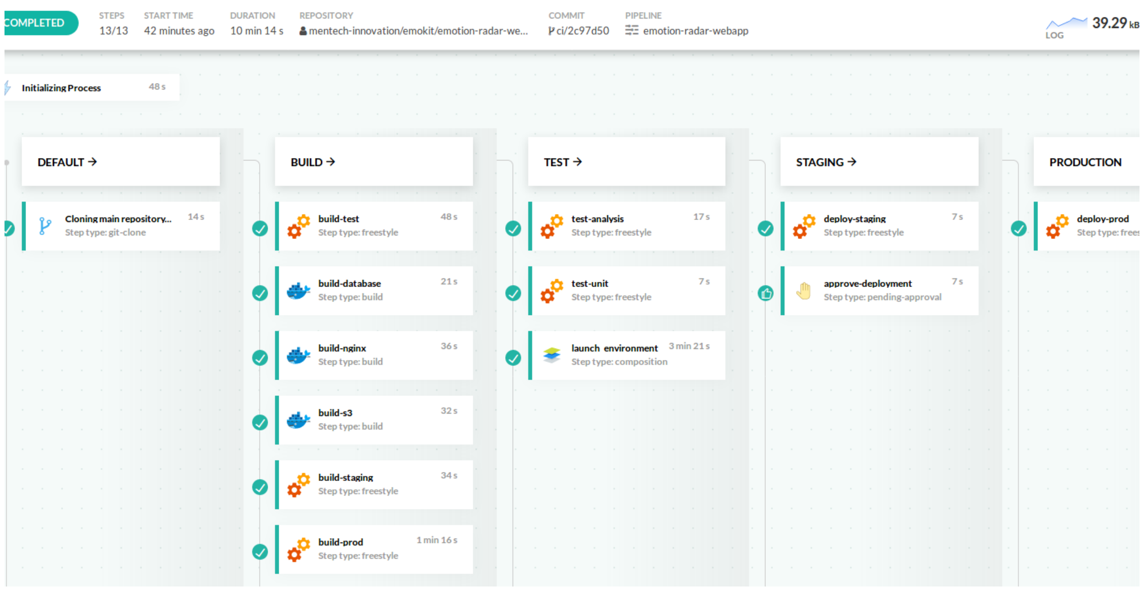


Figure 14: Value stream map with Continuous Delivery

The resulting deployment pipeline created for the HUME website of Mentech Innovation is working correctly as it can be seen on the Figure 15. More images of the Codefresh console can be found in the Appendix 2.

The pipeline is started when any of the developers working on the HUME website push their work to GitLab. If the software has no bugs, all the 13 steps of the pipeline complete without errors, taking in total 10 minutes 14 seconds to run (this time can vary, but on average, it is about 10 minutes). As a result, the software of the HUME website is pushed to the staging environment in the cloud, and, when the management of Mentech wants to release it to their clients, it can be done by just one push of a button.



OMPLETED

STEPS	START TIME	DURATION	REPOSITORY	COMMIT	PIPELINE
13/13	42 minutes ago	10 min 14 s	mentech-innovation/emokit/emotion-radar-we...	ci/2c97d50	emotion-radar-webapp

LOG 39.29 kB

Initializing Process 48 s

DEFAULT →
- Cloning main repository... 14 s — Step type: git-clone

BUILD →
- build-test 48 s — Step type: freestyle
- build-database 21 s — Step type: build
- build-nginx 36 s — Step type: build
- build-s3 32 s — Step type: build
- build-staging 34 s — Step type: freestyle
- build-prod 1 min 16 s — Step type: freestyle

TEST →
- test-analysis 17 s — Step type: freestyle
- test-unit 7 s — Step type: freestyle
- launch environment 3 min 21 s — Step type: composition

STAGING →
- deploy-staging 7 s — Step type: freestyle
- approve-deployment 7 s — Step type: pending-approval

PRODUCTION
- deploy-prod — Step type: frees

Figure 15: Codefresh console screenshot

After the validation of the project and the system the main research question of this thesis can now be answered and the hypothesis can be proved. The concept of Continuous Delivery has improved the process of writing, testing and releasing software at Mentech Innovation. As can be seen from the Figures 13 and 14, Continuous Delivery allows the process of software development and release to be controlled better, as the state of the software being tested and deployed can be seen at all times and if any errors occur the developers are notified of it. As the whole deployment pipeline for the HUME website is automated, the developers can deploy updates to software just by pushing their code to version control, which saves the time on manual testing and deployment, and any human errors while releasing are just not possible. The process of software releasing for the HUME website has become repeatable and reliable, and since the testing and deployment can be now done by just a push of a button, the developers can create and release new features faster, which will benefit Mentech Innovation.

7 CONCLUSION

During this thesis project a research project was conducted on the usage of Continuous Delivery for the software at Mentech Innovation. This topic was important for the company, as in the end of 2019 the company is aiming to turn their emotion sensing platform HUME into production grade software. The software lacked quality assurance and repeatable and reliable releases, and the concept of Continuous Delivery was proposed as a solution to this problem.

The aim of the project was to validate whether the concept of Continuous Delivery would improve the process of writing, testing and releasing software at Mentech Innovation and if so, how? To perform this validation a literature research and an implementation of Continuous Delivery for the HUME website of Mentech Innovation was to be made. A hypothesis to be proved or disproved during the research was: The concept of Continuous Delivery will improve the process of software releasing at Mentech Innovation by making it fast, frequent, repeatable and reliable.

During the literature research the concept of Continuous Delivery, its benefits and disadvantages were evaluated and the general structure of the deployment pipeline for the Mentech software was created. The potential risks when working with Continuous Delivery were listed and mitigating strategies for them were proposed. It was found that with proper communication and planning it is possible to successfully practice Continuous Delivery. Additionally, the deployment strategies for the software were examined and the strategy that was found to suit Mentech the best was canary deployment. The software development workflows were compared. It was found that GitLab workflow was the best compatible with Continuous Delivery as a lot of its rules were set to incorporate the ideas of Continuous Delivery, therefore it was chosen for use at Mentech Innovation.

During the implementation phase of the project the structure of the deployment pipeline for the HUME website of Mentech Innovation was created and the tools for its implementation were chosen. Then, the pipeline was implemented using Codefresh. Using this pipeline, a developer commits their code to version control (GitLab), the code gets automatically compiled and packaged (with Webpack), unit tested (with Jest), analysed (with ESLint and Flow), integration (Jest) and acceptance (Nightwatch.js) tested and released to the staging and production environments (running in the cloud working with Amazon AWS tools).

The deployment pipeline of the HUME website has proved to work correctly. The validation of the implemented concept was conducted in two ways: using a maturity model and using a value stream map. Both of these methods proved that the concept of Continuous Delivery was beneficial to be used with the Mentech Innovation software as it improved the process of software testing and release by making it fully automated, measured, controlled and therefore reliable. Additionally, it saved time for the developers so that instead of manual testing and releasing time can now be spent on developing new features. Thus, the usage of the Continuous Delivery concept was validated and the hypothesis presented at the beginning of the work was proved. It can be concluded that Continuous Delivery has improved the process of writing, testing and releasing software at Mentech Innovation.

8 RECOMMENDATIONS

In this thesis work the implementation and validation of Continuous Delivery on the software of Mentech Innovation has been described. However, for the further successful functioning of Continuous Delivery several recommendations can be given. This chapter lists the work that can be done to support or improve the created system.

As mentioned in chapter 4.5, the Continuous Delivery model can be a challenge to truly adopt in a company due to a big amount of team collaboration, discipline and time that is needed. Therefore, to keep the team committed to the usage of the Continuous Delivery method, the rules listed in chapter 4.5 should be followed. In general, everybody in the team should be informed of the practice and encouraged to continuously deliver the results at the regular meetings. The team members should take responsibility for their code and write tests for it. Additionally, the management of all the teams practising Continuous Delivery should regularly check the status board of the software integrations to see the status of each feature branch and the person responsible for it. With a good collaboration of the team members the highest level of company deployment maturity can be achieved.

In regards to the created deployment pipeline, the Artifact repository in GitLab is set up to store the external libraries needed for the project. To store the package between the stages of the deployment pipeline cache is used. However, as mentioned in chapter 4.4, the package should be stored in the Artifact repository, as well as the documentation from every step of the deployment pipeline execution. This needs to be set up.

The Continuous Delivery system is yet to be implemented on the other elements of the Mentech Innovation system. A lot of the already written code can be reused (for example the configuration files for the deployment tools), however depending on the application different tools might need to be chosen. Since this thesis document has proven Continuous Delivery to be beneficial to be used, both theoretically and practically, Mentech Innovation engineers have to set it up for the code they are working on.

During work with the Continuous Delivery tool - Codefresh - it was noticed that it runs out of memory when running the pipeline. This can lead to failed deployments because the Codefresh memory just cannot handle the amount of code it has to process. The free Codefresh plan that was used only offers 2GB of pipeline memory. Additionally, this plan offers 120 builds/month (so the pipeline can be ran only 120 times/month), which is not an issue when running the pipelines only for the HUME website, but it might be too little if Continuous Delivery with Codefresh will be enabled for all the parts of the Mentech Innovation system. Therefore, to practice Continuous Delivery effectively, it would be beneficial to get the Basic paid Codefresh plan, which enables more pipeline memory (3GB) and more builds/month (220).

Bibliography

Ava. (n.d.). Retrieved 25 January 2019, from https://github.com/avajs/ava

Blischak, J. D., Davenport, E. R., & Wilson, G. (2016). A quick introduction to version control with Git and Github. *PLoS computational biology*, *12*(1), 1–18.

Chang, M. (2013). *Model everything to fail fast.* Retrieved 22 February 2019, from https://www.thoughtworks.com/insights/blog/model-everything-fail-fast

Chen, L. (2015). Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, *32*(2), 50–54.

Codefresh. (n.d.). Retrieved 5 February 2019, from https://codefresh.io/

Comparing workflows. (n.d.). Retrieved 19 February 2019, from https://www.atlassian.com/git/tutorials/comparing-workflows

Configuring ESLint. (n.d.). Retrieved 25 January 2019, from https://eslint.org/docs/user-guide/configuring

Continuous integration, deployment & delivery with Codeship. (n.d.). Retrieved 5 February 2019, from https://codeship.com/

DeMarco, T., & Lister, T. (2003). *Waltzing with bears: Managing risk on software projects*. New York: Dorset House.

Docker documentation. (n.d.). Retrieved 19 February 2019, from https://docs.docker.com/

Farcic, V. (2017). *The ten commandments of continuous delivery.* Retrieved 7 November 2018, from https://technologyconversations.com/2017/03/06/the-ten-commandments-of-continuous-delivery/

GitLab. (n.d.). Retrieved 25 January 2019, from https://about.gitlab.com/stages-devops-lifecycle/

GitLab continuous integration & delivery. (n.d.). Retrieved 5 February 2019, from https://about.gitlab.com/product/continuous-integration/

Halonen, R. (2017). *Improving visibility of test results for continuous integration and delivery pipeline* (Master's Thesis, Degree Programme in Information Technology, Tampere University of Applied Sciences). Retrieved 12 February 2019, from http://urn.fi/URN:NBN:fi:amk-2017112518145

Hautaviita, A. (2018). *Developing a web application on the MEVN stack* (Thesis, Degree Programme in Information Technology, Turku University of Applied Sciences). Retrieved 6 December 2018, from http://urn.fi/URN:NBN:fi:amk-2018120319693

Humble, J., & Farley, D. (2011). *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Boston: Addison-Wesley.

Humble, J., & Molesky, J. (2011). Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal*, *24*(8), 6–12.

Jasmine documentation. (n.d.). Retrieved 25 January 2019, from https://jasmine.github.io/

Jest. (n.d.). Retrieved 25 January 2019, from https://jestjs.io/

Jest: Getting Started. (n.d.). Retrieved 25 January 2019, from https://jestjs.io/docs/en/getting-started

Mocha. (n.d.). Retrieved 25 January 2019, from https://mochajs.org/

Möller, K. (2018). *Developing a graphical user interface for modifying chatbot configurations* (Thesis, Degree Programme in Information and Communications Technology, Metropolia University of Applied Sciences). Retrieved 6 December 2018, from `http://urn.fi/URN:NBN:fi:amk-2018052510356`

Nightwatch.js. (n.d.). Retrieved 19 February 2019, from `http://nightwatchjs.org/`

Nikitina, N., Kajko-Mattsson, M., & Stråle, M. (2012, June). From scrum to scrumban: A case study of a process transition. In *2012 international conference on software and system process (ICSSP) (pp. 140-149)*. IEEE.

Paulasaari, M. (2018). *Tools for code quality in front-end software development* (Master's Thesis, Degree Programme in Information Technology, Metropolia University of Applied Sciences). Retrieved 6 December 2018, from `http://urn.fi/URN:NBN:fi:amk-201804134642`

Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, *5*, 3909–3943.

Sharma, A. (2018). *A brief history of devops, part IV: Continuous delivery and continuous deployment.* Retrieved 6 November 2018, from `https://circleci.com/blog/a-brief-history-of-devops-part-iv-continuous-delivery-and-continuous-deployment/`

Tremel, E. (2017). *Six strategies for application deployment.* Retrieved 8 November 2018, from `https://thenewstack.io/deployment-strategies/`

Appendix 1 HUME website

Below a screenshot from one of the pages of the HUME website is presented. The clients field that can be visible in the side menu of the website is filled with clients (such as ”David Hume”) gotten from the database through the web service. Similarly to this, the sessions field (not visible on the screenshot) is filled with running sessions gotten from the wearable sensor devices through the web service. The items in the clients and sections fields can be clicked, causing a calendar of measurements to appear per client or a graph with a running session to appear per session.

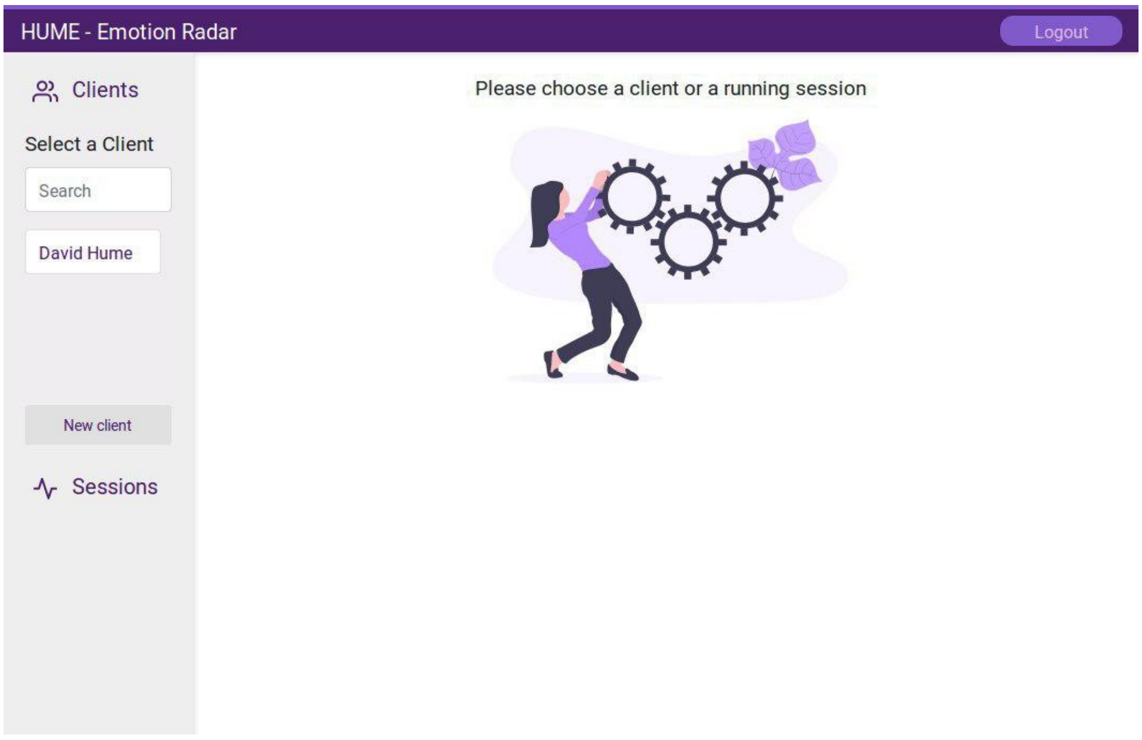
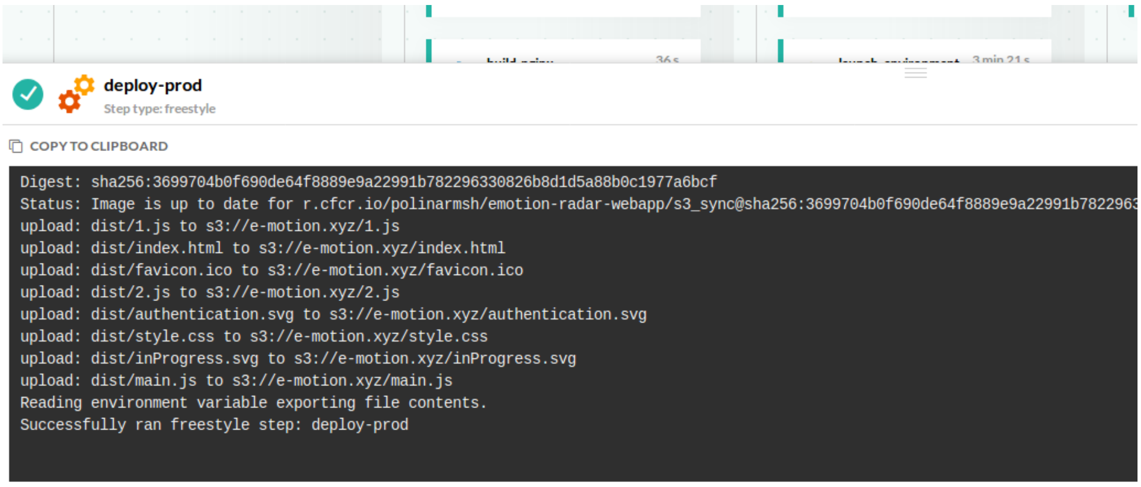


Figure 16: Hume website screenshot

Appendix 2 Codefresh console

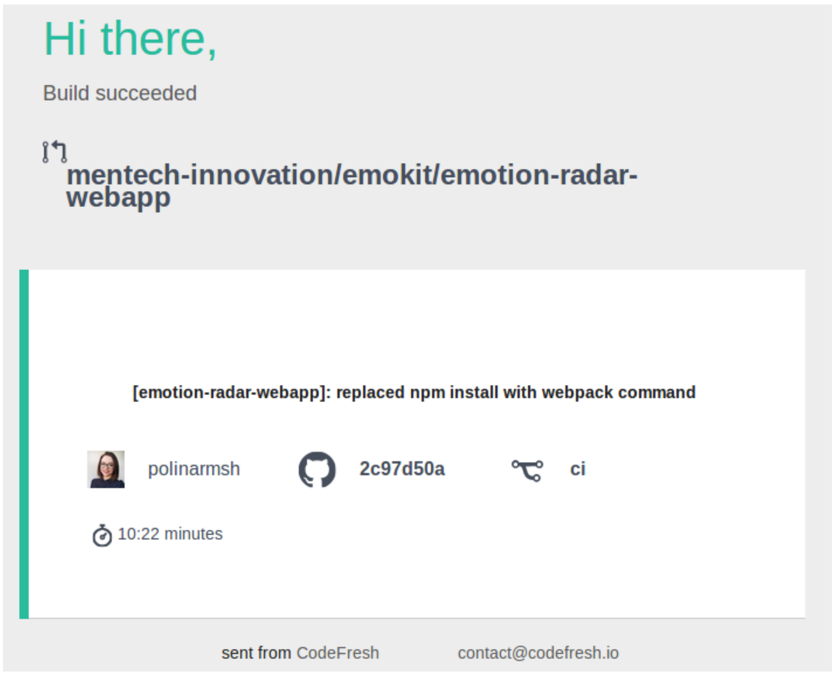
Below on the Figure 17 the log output from running the Codefresh pipeline is shown. When the pipeline is running, it is possible to see for every step of the pipeline what is going on at the moment. All the possible pipeline errors are logged too, so it is convenient for debugging.



```
Digest: sha256:3699704b0f690de64f8889e9a22991b782296330826b8d1d5a88b0c1977a6bcf
Status: Image is up to date for r.cfcr.io/polinarmsh/emotion-radar-webapp/s3_sync@sha256:3699704b0f690de64f8889e9a22991b782296
upload: dist/1.js to s3://e-motion.xyz/1.js
upload: dist/index.html to s3://e-motion.xyz/index.html
upload: dist/favicon.ico to s3://e-motion.xyz/favicon.ico
upload: dist/2.js to s3://e-motion.xyz/2.js
upload: dist/authentication.svg to s3://e-motion.xyz/authentication.svg
upload: dist/style.css to s3://e-motion.xyz/style.css
upload: dist/inProgress.svg to s3://e-motion.xyz/inProgress.svg
upload: dist/main.js to s3://e-motion.xyz/main.js
Reading environment variable exporting file contents.
Successfully ran freestyle step: deploy-prod
```

Figure 17: Codefresh console log example

On the Figure 18 the screenshot of the email that is automatically sent by Codefresh when a pipeline completes is presented. It ensures that the developers are aware of any failed pipelines or know when the pipeline built succeeded. The email is sent to all the developers that are using the Codefresh account of Mentech.



Hi there,

Build succeeded

mentech-innovation/emokit/emotion-radar-webapp

[emotion-radar-webapp]: replaced npm install with webpack command

polinarmsh 2c97d50a ci

10:22 minutes

sent from CodeFresh contact@codefresh.io

Figure 18: Codefresh email example