



Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

**Facultat de Matemàtiques i Informàtica
Universitat de Barcelona**

Continuous Delivery

Bogdan Marcut

Director: Josep Vañó Chic
Realitzat a: Departament de
Matemàtiques i Informàtica

Barcelona, 27 de juny de 2018

Abstract

The project consists in researching, analyzing and implementing a deployment pipeline from scratch. During the implementation, a web-application called *Funny Stories* is created to demonstrate the way this kind of system works.

To develop this system, a series of technologies were used. For the implementation of the application the tools used were Java 8 with the framework Spring Boot for the server side; HTML, jQuery, Bootstrap and CSS for the client side, and MySQL for managing the database.

The web-site allows users to visualize stories posted by other people which are sorted by categories. They are also able to use a voting system to give their feedback for each post. The web-site is public and these features can be accessed by anyone.

When integrating new features into an application and the process is manual, a lot of unforeseen error can happen which can end up delaying the delivery of the software by hours or days. This result in the developer not being able to show his work and delaying the release process which can affect the end users since they cannot use the product. A lot of pressure is put on the responsible for the integration and can end up in a lot of frustration.

These problems are very common inside organizations and have inevitable outcome in the development process. These are indications that something is not right because the delivery should be fast and repeatable.

This project looks at the delivery from another perspective and will present a series of practices which will improve the code integration. This will allow the developers to release their work several times a day while reducing the risks of the process.

The release management of this application is done by using an automated deployment pipeline. This implies that whenever there is a change to the system, the deployment pipeline automatically rebuilds the entire application, tests it for errors and deploys it to the production environment.

Resumen

El proyecto consiste en investigar, analizar e implementar un “pipeline” de desarrollo desde cero. Durante la implementación se creó una aplicación web llamada *Funny Stories* que servirá para demostrar cómo funciona el sistema.

Para desarrollar este sistema, se utilizaron una serie de tecnologías. Para la implementación de la aplicación se utilizaron las herramientas Java 8 con el framework Spring Boot para la parte del servidor; HTML, jQuery, Bootstrap y CSS para la parte del cliente, y MySQL para gestionar la base de datos.

El sitio web permite a los usuarios visualizar cuentos ordenados por categorías y publicados por otras personas. Los usuarios también pueden utilizar un sistema de votos para dar su opinión sobre las publicaciones. Este sitio web es público y todo el mundo puede utilizar las funcionalidades especificadas anteriormente.

Cuanto se integran nuevas funcionalidades en la aplicación y el proceso es manual, puede haber muchos errores imprevistos. Esto resulta a que el desarrollador no pueda enseñar su trabajo y retrasar el proceso de entrega por horas o días. Se pone mucha presión sobre el responsable de la integración de los cambios y eso puede resultar en mucha frustración.

Estos problemas son muy usuales dentro de una organización y tienen resultados inevitables en el proceso de desarrollo. Estos son indicaciones de que algo no está bien porque el proceso de entrega debería ser rápido y repetible.

Este proyecto mira la entrega desde otra perspectiva y presenta una serie de buenas prácticas que pueden mejorar la integración del código. Esto permitirá a los desarrolladores entregar su trabajo varias veces al día reduciendo el riesgo del proceso.

La gestión de la entrega de esta aplicación está hecha utilizando un “pipeline” de desarrollo automatizado. Esto implica que cuando hay un cambio en el sistema, el “pipeline” de desarrollo reconstruirá automáticamente la aplicación, testearla de errores y subirla a un entorno de producción.

Resum

El projecte consisteix en investigar, analitzar i implementar el “pipeline” de desenvolupament des de zero. Durant la implementació es va crear una aplicació web sota el nom de *Funny Stories*, la qual servirà per demostrar com funciona el sistema.

Per desenvolupar aquest sistema, es van utilitzar una sèrie de tecnologies. Per a la implementació de la aplicació es van utilitzar les eines Java 8 amb el Framework Spring Boot per a la part del servidor; HTML, jQuery, Bootstrap i CSS per a la part del client; i MySQL per gestionar la base de dades.

El web permet als usuaris visualitzar contes ordenats per categories i publicats per altres persones. Els usuaris també poden utilitzar un sistema de vots per donar la seva opinió sobre les publicacions. Aquest web és públic i tothom pot utilitzar les funcionalitats especificades anteriorment.

Quan s’integren noves funcionalitats a l’aplicació i el procés és manual, poden haver molts errors imprevisibles. Això fa que el desenvolupador no pugui ensenyar el seu treball i hi hagi un endarreriment en el procés d’entrega de hores o dies. Es posa molt pressió sobre el responsable de la integració dels canvis i això pot resultar molt frustrant.

Aquests problemes són molt usuals dins l’organització i tenen resultats inevitables en el procés de desenvolupament. Aquestes són indicacions de que alguna cosa no està bé, ja que el procés d’entrega hauria de ser ràpid i repetible.

Aquest projecte visualitza l’entrega des d’una altra perspectiva i presenta una sèrie de bones pràctiques que poden millorar la integració del codi. Això permetrà als desenvolupadors entregar el seu treball diverses vegades al dia, reduint el risc del procés.

La gestió d’aquesta aplicació està feta utilitzant un “pipeline” de desenvolupament automatitzat. Això implica que quan hi hagi un canvi en el sistema, el “pipeline” de desenvolupament reconstruirà automàticament l’aplicació, la testearà d’errors i la pujarà a un entorn de producció.

Contents

- Abstract 2
- Resumen..... 3
- Resum 4
- Table of figures..... 7
- 1. Introduction..... 9
 - 1.1. State of the art 9
 - 1.2. Motivation 10
 - 1.3. Objectives..... 10
- 2. Analysis..... 12
 - 2.1. Functional requirements..... 12
 - 2.2. Non-Functional requirements..... 13
 - 2.3. Risk analysis..... 14
- 3. Deployment Pipeline Foundation 16
 - 3.1. The principles of flow and feedback 16
 - 3.2. Continuous delivery and other practices 20
 - 3.3. Low-risk releases 30
 - 3.4. Telemetry 35
 - 3.5. Security..... 40
 - 3.6. The Pipeline in a DevOps culture 41
 - 3.7. Continuous learning and improvement 43
 - 3.8. Maturity model 44
- 4. Implementation, from Theory to Practice 47
 - 4.1. Work on small iterations 47
 - 4.2. Choose the right tools 49

4.3. Start with a walking skeleton	52
4.4. Outside-In driven development	56
4.5. Multiple environments management	58
4.6. Deployment pipeline anatomy.....	60
5. Results vs Objectives	63
6. Conclusions.....	65
7. Bibliography.....	66

Table of figures

Figure 1: Stream map of a product [1]	23
Figure 2: Sequence diagram of the deployment pipeline	24
Figure 3: Extreme programming practices	27
Figure 4: Blue-green deployment.....	33
Figure 5: Canary release	34
Figure 6: Using Stats and Graphite libraries to generate one-line telemetry.....	38
Figure 7: Conflicts between the operations team and the development team	41
Figure 8: DevOps collaboration schema	42
Figure 9: Continuous improvement for DevOps	43
Figure 10: Continuous delivery maturity model	44
Figure 11: Basic application concept.....	47
Figure 12: MVP design.....	48
Figure 13: AWS Elastic Beanstalks flow	52
Figure 14: Branching strategy	54
Figure 15: Funny Stories architecture	54
Figure 16: JSON output for RESTful services	55
Figure 17: Funny Stories database table - category.....	55
Figure 18: Double Loop TDD	57
Figure 19: Funny Stories full database schema.....	58
Figure 20: Multiple environment transition.....	59
Figure 21: Continuous delivery pipeline	60
Figure 22: Travis CI interface.....	61

Figure 23: AWS Elastic Beanstalk blue-green environments before deploying..... 62

Figure 24: AWS Elastic Beanstalk blue-green environments after deploying..... 62

Figure 25: Funny Stories - final product 64

1. Introduction

One of the most important and frequently encountered problems during the lifecycle of an application is deploying a project and reduce its risks of failure. There are different methodologies to approach this which try to increase the success of the process.

Releases should take in count that updating an application comes with various side effects which could affect the user experience. In such cases it is required to develop a release plan so that the clients are not left in the dark about possible failures in the product during or after the release process. A traditional release plan refers to executing a set of stages such as building, deploying, testing and release the application into production. This is a manual and laborious process which is very unreliable since errors might be committed while executing these steps and could result into delaying the deployment for hours or days.

To learn how to improve a release process, an organization should start adopting practices which will help with the automatization of their system and focus on receiving and utilizing feedback. If there is a safe system of work where teams are independent and responsible, the productivity of the developers will be maximized and organizations will be able to provide a quality product.

1.1. State of the art

The product release refers to the launch of a product or a set of features that provide for a user. In more specific terms this refers to a connection between the internal teams of a company and their clients in which the teams focus on delivering new functionalities so that the users can obtain value from them.

Automation is the key to perform tasks which are usually repetitive. Since manual work is unpredictable and causes a lot of risk, automation is the solution to such problems.

Continuous delivery is a type of release which is almost entirely automated in such manner that it would reduce the risks which come with a manual release process. This results in a reliable and repeatable process which provides a quantifiable amount of time for it to be

executed. It also allows for developers and operators to perform such a task at the push of a button. With this the amount of time for releasing the product is lower and it is safer.

1.2. Motivation

During my work experience as a software developer, I always encountered problems with how the organization manages releases. Developers would have to come to work at six in the morning to perform a release which would last around three hours. Whenever there was a bug in the release and the delivery could not be completed, this process would take even more time and it could reach an 8 hours mark. The customers were not happy about it since they could not use the application, the person releasing the software was fed-up with the deployment and the developers would have to spend the rest of the week searching and fixing bugs found in production.

The mentality of the people which organized the release would always be to insert as many changes as possible during a two-week period and just deploy them to production while hoping nothing went wrong. The norm was to encounter lots of bugs after each deployment and the mentality was to fix everything at once and reduce deployments. But since these fixes could end up in being large changes, there would usually be a hotfix deployed which on its own would require other hotfixes.

These problems sparked an interest for me and I chose to research in which ways these processes could be improved.

1.3. Objectives

This project will focus on activities which are usually considered secondary, but require a lot of effort and can take up a long time to execute. Activities such as information flow, automation, telemetry, building, testing and deploying are critical to successfully deliver software. When the risk which come with these activities is not carefully considered, they can cause cost losses.

The objective of this project is to provide a solution which would automatize and speed up the release process of an application while reducing the risks and possible failures of the deployment. This implies that a full application shall be developed during the process which would simulate a real basic production software and would be composed of a frontend, backend and database.

2. Analysis

To comply with the objectives specified in the previous chapter, it is necessary to analyze the requirements for such a pipeline to be implemented and the risks that come with it.

2.1. Functional requirements

A functional requirement defines a function of the system or its components. The functional requirements of the application to be implemented are represented in the following use case diagrams:

Case	Load category
Description	Load posts from a category
Actor	User
Basic flow	<ol style="list-style-type: none">1. The user clicks a category2. The system loads the posts of the clicked category
Pre-conditions	The user entered the page
Post-conditions	The user will obtain the posts

Case	Create post
Description	Create a post in the current category
Actor	User
Basic flow	<ol style="list-style-type: none">1. The user inserts the title and description of the post2. The system stores the post in the system for the current category3. The system loads the created post and the user views it
Pre-conditions	The user has selected a category
Post-conditions	The will see the new post

Case	Upvote a post
Description	Upvote a post from the page
Actor	User

Basic flow	The user clicks the upvote button from a post The system increments the upvotes of the current post The system reloads the upvotes for the current post
Pre-conditions	The user entered the page
Post-conditions	The user will see an increment in the upvotes for the selected post

Case	Downvote a post
Description	Downvote a post from the page
Actor	User
Basic flow	<ol style="list-style-type: none"> 1. The user clicks the downvote button from a post 2. The system increments the downvotes of the current post 3. The system reloads the downvotes for the current post
Pre-conditions	The user entered the page
Post-conditions	The user will see an increment in the downvotes for the selected post

2.2. Non-Functional requirements

It is necessary to develop a walking-skeleton¹ to demonstrate the utility of a continuous delivery pipeline. Since in a real environment there would be several applications that would be deployed together, it is required to develop a product which contains a frontend, backend and database.

The configuration of the system should be secured. All the passwords should be encrypted and not stored on the version control.

The delivery pipeline should be entirely automated and allow the execution of the deployment by pushing a button or executing a command. The application should have at least one end-to-end transaction to fully prove that the automation is working. Each

¹ Proof of concept of a basic architectural concept. It is a tiny implementation of the system which performs a small end-to-end function.

component of the application should include automated tests for the functional requirements.

2.3. Risk analysis

The main risks for developing this project are:

Risk 1	Time management
Description	The project should be finished in approximately 4 months and there are many elements/problems which are still unknown since I miss knowledge about the usage of many technologies required
Impact	Finalization date
Probability	High
Mitigation Action	Reduce the complexity of the simulated application to dedicate time to implement the pipeline and cut edges on the good to have features

Risk 2	Reduced knowledge about some of the technologies involved
Description	It is required to utilize some tools which are unknown to me and they could cause side effects to the original analysis
Impact	The entire project
Probability	Medium
Mitigation Action	Preferably change the technologies if the required time is short or have minor changes to the demo application while still proving the objectives have been accomplished

Risk 3	Payment needs to use certain tools
Description	AWS is a great server renting service which allows free usage with certain restrictions. It is possible that during the development process it would be required to pay for certain services if the limits are surpassed

Impact	The entire project
Probability	Low
Mitigation Action	Pay the required fees or reduce the usage of the servers by utilizing a local environment

3. Deployment Pipeline Foundation

The purpose of this chapter is to show how the various teams inside an organization could work together to help an organization grow. If everyone has a common goal, they can ensure the fast delivery of their work to production while achieving stability, reliability, availability and security.

When teams help each other in the delivery of the software, they start taking more responsibility. On top of implementing features, they will also ensure that their work flows smoothly to everyone involved in the value stream without interfering with the operations team or customers. The operations team will create environments which are better suited for developers which will result in them being more productive. Automation can be introduced into the delivery process to the time it takes to receive upgrades.

3.1. The principles of flow and feedback

The goal of developers is to deliver software as quickly as possible while complying with the requirements and reducing risks. It is essential to be fast because the investment into the software will not produce any income unless it is released. For completing such a task, it is necessary to try and reduce the cycle time² during a release or a bugfix.

Having a quick delivery is important since it allows you to confirm that the change is useful. While the business can suppose that a feature is going to be useful, it is up to the users to make such a decision. Therefore, it is vital that we obtain feedback from the users as quickly as possible, so we could avoid losing time developing useless software.

While we are providing software in a quick manner, it is important to not overlook its quality. It is required to maintain a certain balance while developing because it is possible to lose the precious time that we gained through other means.

² The time it takes from deciding to make a change until it is available to the user

We need to deliver a high-quality product in a quick manner. There are two main things we must do to achieve our goals:

- **Automatize the process:** we must automatize the building, testing and deploying so we can repeat these tasks and quantify the amount of time it takes to complete the cycle. Since it is an automated process, we can avoid errors which come with a manual release. Doing this manually can end up with errors in the deployment which are not controlled or logged, and we can lose track of what we did by changing a simple value in the database and forget about it. This can lead up to hours of frustration searching for a simple problem and lower the quality of the product. During an automated release we ensure that all the followed steps are logged, and we have control of the process. We also ensure that the process is the same every time we go through this cycle.
- **Release frequently:** if the changes to the software are frequent, the time between the releases is short. This reduces the risk of a deployment since there are less things that can go wrong during the process. In case of any kind of error or negative feedback, it is relatively easy to rollback.

While the users can provide feedback, it is not always useful. [1] For such situations it is important to fulfill three criteria for the feedback to be useful in an automated release process:

- The feedback process must be triggered whenever there is any change to the product.
- It must be delivered as quickly as possible.
- The release team must receive feedback and act upon it.

The feedback process must be triggered whenever there is any change to the product

A usual application is normally made of four components: host, configuration, code and data. During any changes in these components the application behavior is affected, and we must ensure it is verified.

Whenever there are changes to the host environment the whole system must be tested to check if any of the other components are affected. A simple update in the operative system can end up making the application unusable.

The configuration of an environment can be different if there are multiples environments for our application. For example, the developers tend to use their local machine with a local configuration to develop the product. The production environment is usually different because we can make our product point towards a real data storage. Therefore, the software should be tested for any changes in any of the environments.

The source code is recompiled whenever there is a change to it and therefore a new binary must be built and tested. This process should be automated to gain control over it since it is one of the most repetitive process in the development of the application. Automating this step is also known as continuous integration which is described in Chapter 3.2.

The structure of the data is usually changed. Whenever there is a new feature to be implemented, there are usually data manipulations. In such cases we must automatize such changes and test them accordingly.

All being said, the feedback process involves testing any changes in a fully automated manner. The tests could be different, but they usually include the following:

- The source code must be compliable. Having a code that is wrong from a syntax point of view cannot be compiled into a binary.
- The unit tests for the software must pass. We must assure that at a very basic level our code behaves like it is supposed to.
- The software should have a certain quality by being covered with tests which should surpass a certain percentage of checked code.
- The acceptance tests for the product must pass. We must ensure that our software is able to support a certain capacity, availability, security etc. so it would meet the user's needs.
- The product should go through a demonstration phase. This means that we should restrict the new product to a certain number of users to make sure that there are no bugs or missing features. A product owner could take on this job since he is the most qualified in knowing the business requirements.

These tests should be executed in an environment which is very similar to the production environment, so we would not encounter unexpected errors.

Feedback must be delivered as quickly as possible

To receive fast feedback, it is needed to automate the process. This way the process becomes scalable with the hardware and problems that come with manually building, testing and deploying are avoided. Moreover, people can end up being fed up with doing such repetitive work manually. They also give up precious learning time and developing by performing such tasks.

Automation of the deployment pipeline comes with resource costs if the entire system is automated. Even so, the main objective is to free the human resources from doing repetitive tasks and have them dedicate their time to interesting work.

It is necessary, during the commit stage of the pipeline, to have tests with the following characteristics:

- They are quick.
- The tests must cover a very large part of the code. It should surpass a 75% coverage rate for them to be reliable and ensure the application works.
- Failed tests should stop the deploying and not release anything. Whenever a test fails it must indicate that a critical fault occurred, and the application would not have been able to run properly.
- They should not depend on an environment when executing them. Doing so will avoid having to duplicate the entire production system and have cost losses.

In later stages of the pipeline the test should have the following characteristics:

- They can run slower and should support parallelization.
- If any test fails, we should still be able to choose if the release can be completed. An example would be if during the release an external service is down, and we cannot fully test it, we might still want to release the product.

The first set of tests of our application should give us a high level of confidence in our software and the ability to run them on cheap machines. Doing so we can detect errors quickly

and stop the execution of the process so we would not dedicate resources to something we know would fail.

One of the main goals is to receive fast feedback on changes. Ensuring fast feedback requires us to dedicate time developing the software. The main thing we need to focus on is to organize our software and how to use version controls. The developers should commit often to the version control systems and separate the code in different components to manage large teams. The branching system should be simple and, in most cases, avoided.

The release team must receive feedback and act upon it

Everyone involved with the process of delivering software should also be involved in the feedback process. People involved in a project such as developers, testers, operations, database admins, infrastructure specialists and managers should work together on improving the delivering process of the software. Continuous improvement plays an important role in the swiftness of the delivery quality software. There must be a retrospective meeting held after every iteration to improve the delivery process of the next iteration.

The feedback should be also broadcasted for the team by using physical dashboards or other methods to make sure that everyone is aware of it. Every team should be responsible of broadcasting the feedback.

Lastly, the feedback should be always acted upon. Whenever there is a problem the whole team should be responsible and prioritize what course of action should be taken. After a decision has been made, the team can return to their previous tasks.

3.2. Continuous delivery and other practices

During the developing process teams tend to spend most of the time developing on a product that is in an unusable state. The reason for that is that nobody runs the entire application until it has been finished. While developers might be running unit tests, most of them tend to not actually run the software on an environment that is like the production environment.

There are many cases where branches tend to have a long lifespan and acceptance tests are not ran on them. This causes to the integration phase to be a lengthy and buggy stage until it can be tested. In some extreme cases, it can be found that when they get to this phase, the software is not implemented as intended. This results in an inestimable and very time-consuming integration period.

A solution to this problem is to automatize the integration and deployment processes. There are several practices which allow automation which are described below in an incremental order.

Continuous Integration

Continuous integration builds the application and make it go through an extensive automated testing whenever somebody make a change to the software. If there are any errors during the building or testing, the teams prioritize solving the problem immediately. This ensures that during continuous integration the software is always in a working condition.

Continuous integration changes the way we think about an application. Without it our software is broken unless someone proves otherwise. With it, if there are enough automated tests, the software is proven to work with any changes made to it and in case of failure it can be fixed right away. This allows for the product to be delivered faster and with less bugs which are easier to fix.

There are several things we need before implementing continuous integration:

1. Version Control

Everything in the project should be stored in the same version control repository. This refers to code, tests, database scripts, building and deploying scripts, any file that is required to create, install, run and test the application. This is important because we want everything to be stored in one place, so any version of our project is executable at any point in time. It also facilitates rolling back the application in case of a failed deploy.

2. Automated Build

It should be possible to start the building of the application from the command line. While an IDE provides tools to execute builds or test without touching the command line it should still be possible to do so. The reasons for this are:

- The build process needs to be executed in an automated way from the continuous integration environment, so it would be controlled when anything goes wrong.
- The build scripts should be treated as code. They should be comprehensive and refactored, so anyone can read them. It is not possible to version the IDE generated processes.
- It is easier to understand, maintain and debug.

3. Agreement of the team

Continuous integration is a practice. It requires the development team to be committed for such a change to be adapted. Everyone must agree to integrate small changes and give maximum priority to fixing the application whenever it fails.

To utilize continuous integration, a continuous integration software is not required. While such a practice can be implemented by using only a version control system, it is recommended to utilize a CI tool since they are relatively easy to install. Some of the tools that can be utilized are TravisCI, Bamboo, TeamCity, Hudson etc.

Continuous Delivery

Continuous integration provides an increase in quality and productivity for the projects that utilize it. It focuses primarily on development teams by providing fast feedback whenever there is a problem with any change. It focuses on compiling the code and executing a series of unit and acceptance test. At the end of this process there will be a stage of manual tests and a release process. This causes time loss from the software development through testing and operations. Some of the generated problems are:

- Build and operation teams waiting for fixes.
- Testers waiting for a green build of the application.
- Developers receiving bug reports long after the team has already moved on to another version of the software.

- Encounter that the architecture does not support system non-functional requirements at the end of the development process.

Continuous delivery is an extension of continuous integration to ensure that the changes to the application get to the users in a quicker and more sustainable way. This implies that on top of the automatization of the building and testing phase, we also automate the release process and make the deploy possible with the push of a single button. This creates a feedback loop since releasing the product to production is easy and we can receive quick feedback from the software and the deployment process.

In this process we end up with a pull system. Every team becomes more independent. The testing team deploy the software into their environments by pushing a button. Operations teams deploy the software in production environments by pushing a button. Developers monitor through which stages of the release process their builds have been and their problems. Managers monitor key metrics (cycle time, code quality). As a result, every team gets access to the things they need and visibility of the release process to improve it. This results in a faster, safer and repeatable deployment.

To implement an end to end automation of the build, test, release and deploy processes, we require a deployment pipeline. A deployment pipeline is an automated process which gets the software from the version control to the customers. The process consists of building the software, having it go through a series of testing stages and then deploying it.

The deployment pipeline focuses on creating a process which would deliver features needed by the users. The process of creating a product can be represented through a stream map as shown in Figure 1.

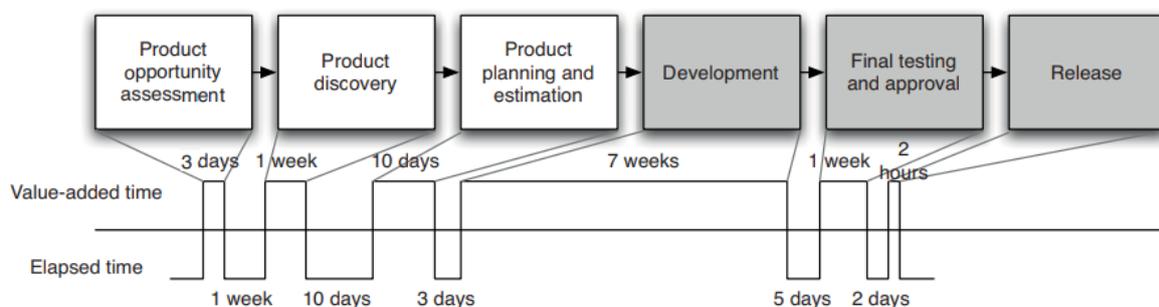


Figure 1: Stream map of a product [1]

The stream map shows how the planning of developing a software is going to be and it can be extended by assigning teams to each stage or specify requirements and resources. The focus in continuous delivery start from development to releasing the product. The difference of this part is that builds pass many times through these stages. A better way to understand this diagram is by examining it as a sequence diagram as shown in Figure 2.

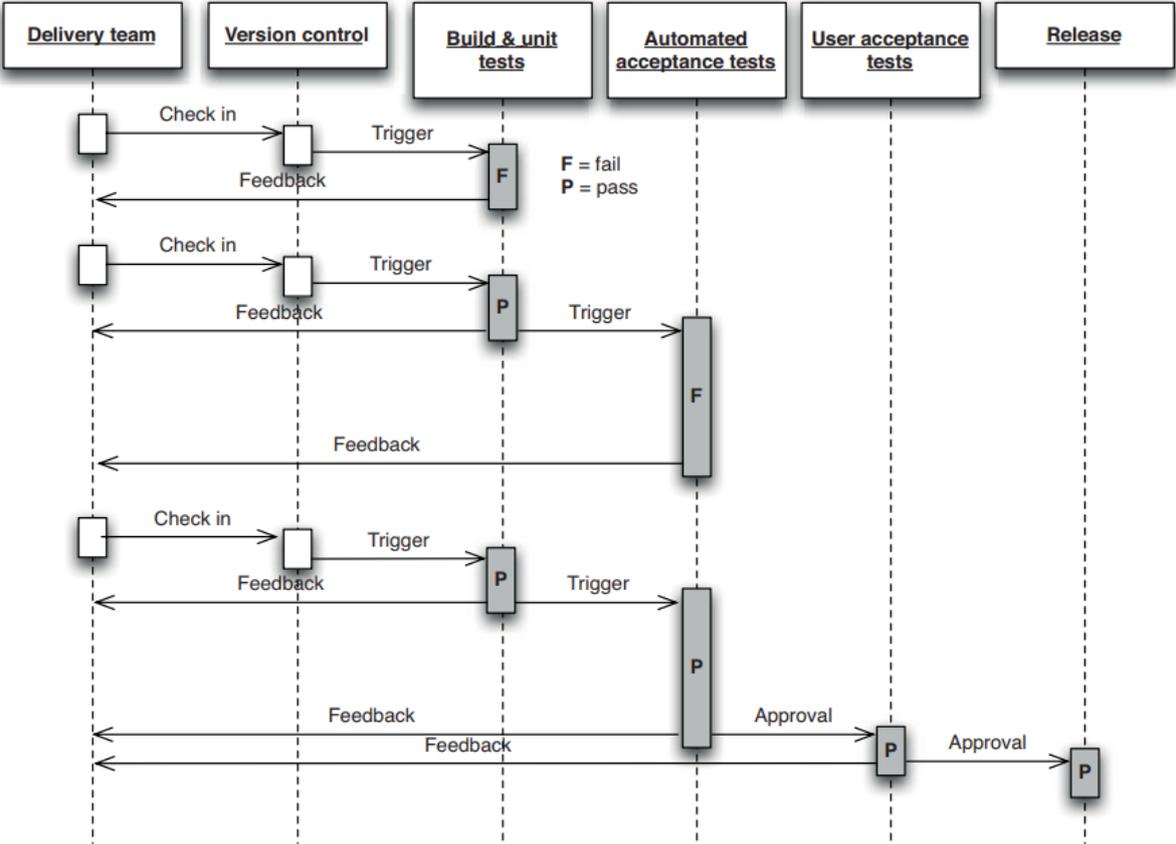


Figure 2: Sequence diagram of the deployment pipeline

In this case we can see that everything starts with the version control. This triggers the build and the first series of tests (unit tests) which are the quickest and most extensive tests the product has. This evaluates the compiled code and it is where continuous integration stops.

At this point we can trust that the code was written in a manner that the developer intended, but we cannot stop here. Now we should dedicate more resources to the pipeline and make the environment of the build more production-like with each stage.

The advantage of this pattern becomes very important. It is now possible to detect most bugs because of the intensive testing of each build. Regression bugs³ are avoided since the entire application is tested and in case of failure the product does not reach production which means that the changes do not affect users. This means that the developers can focus on fixing the actual problem.

Since the deployment is automated, it becomes a quick, repeatable and reliable process. This results in releases being a normal custom in the team and they would not fear the failure of the deploy. Teams become less anxious when releasing software since there is no risk and therefore release more frequently. The worst-case scenario is releasing a bug which causes the team to revert the software to a previous version and fix the problem on their local machines.

To achieve such a state, it is required to automate a series of tests to decide if a release can become a release candidate. Depending on the project there might be slight changes in the testing suite or in the release stages but generally they should contain the following:

- **The commit stage:** makes sure the system works at low level. It should compile, pass unit tests and run a code analysis.
- **Acceptance tests stages:** makes sure the product works at a functional and nonfunctional level and meets the requirements of the customers.
- **Manual test stages:** these stages should check for any error that might have passed through the previous stages. They might include exploratory testing environments, integration environments and user acceptance testing.
- **Release stage:** delivers the product to the users. The software should be deployed in a production environment in this stage.

The previously described stages compose the deployment pipeline. This represents an automated software delivery process. This does not mean that there would be no human interaction, but it rather ensures that the complex steps are automated, reliable and

³ A feature that used to work but no longer does

repeatable. The interaction with the deployment pipeline is increased since the teams can deploy the application by pushing a button.

Continuous Deployment

Following the principles of **Extreme Programming**, the extreme would be to deploy the product to production whenever the automated tests have been passed. This results in a new practice called **Continuous Deployment**.

The main idea is to take the already existing continuous delivery pipeline and automatize the deploy stage instead of pushing a button to do so. This way every time a change is made to the software, the application will be deployed to production if every stage is green. To avoid any possible errors/bugs it is necessary to have very extensive and reliable automated tests covering the entire application.

Continuous Deployment can be used with a canary release so the new features that come out would be available only to a percentage of the users to ensure that the new version is working properly. This will allow for the release to be even less risky.

Continuous deployment is not a practice that would work in any project. In some cases, the teams might not want to release a new feature to production. A company could have certain restriction and need approvals to release the product. In contrast, there are other cases where this could potentially increase productivity.

The first reaction upon hearing about continuous deployment is that it is too risky to release in such a manner. But as specified before, we take a lot of measures to ensure that we have a very low risk level. Since we are integrating frequently, the changes would be relatively small. Also, when changing the code, every test is relaunched and ensures that nothing is broken. Therefore, the risk will be reduced to only the change that has been made. While it might not seem so, continuous deployment is a practice which greatly reduces the risk of a release when compared with a traditional release process.

One of the most important things is that the developers are required to be more cautious when they program. It also forces them to do the right thing by automatizing the entire deployment pipeline. They are also required to develop reliable tests which could run on production environments.

Extreme Programming

“Extreme Programming (XP) is an agile software development framework that aims to produce higher quality software, and higher quality of life for the development team. XP is the most specific of the agile frameworks regarding appropriate engineering practices for software development. [2]”

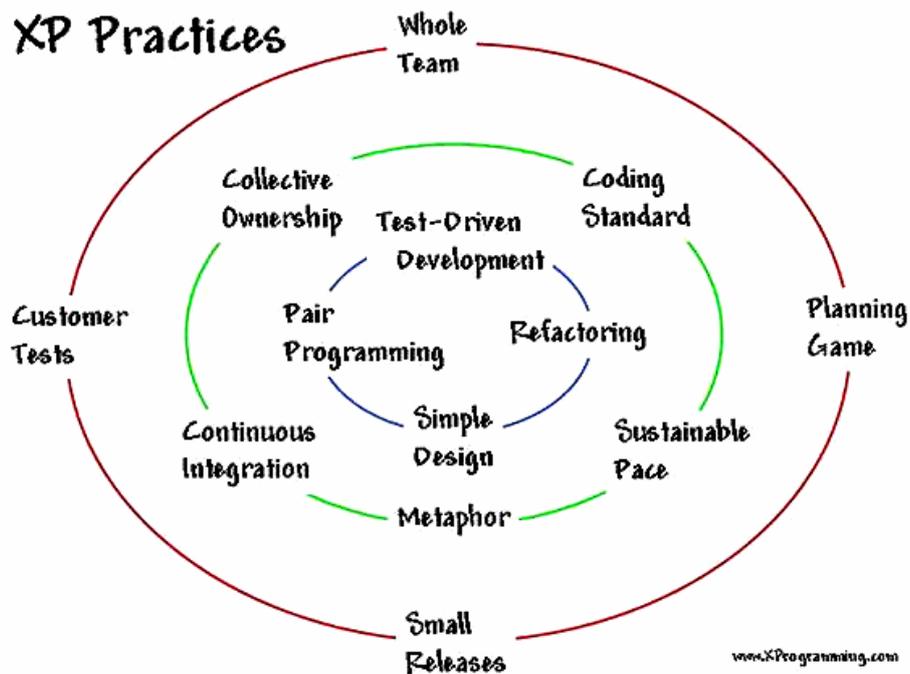


Figure 3: Extreme programming practices

While using extreme programming in continuous integration, delivery and deployment practices, it is recommended to consider its usage.

Extreme programming practices are usually appropriate when working with projects that contain the following characteristics:

- Sudden changes in the software requirements.
- When there are risks involved while using new technologies in time limited projects.
- Small and local development teams.
- In projects where, automated unit and functional tests could be utilized.

There are five values that extreme programming brings to a project:

- **Communication:** when developing a software communication is the key to proportionate information to everyone else. Extreme programming is a practice which

forces communication inside the team which can be done by talking, using white boards etc.

- **Simplicity:** the focus here is to do the only the necessary things such as keeping the design of the application simple to make it easier to maintain, support or revise. It also focuses on the things needed in the present instead of trying to predict any future needs.
- **Feedback:** it gives the team hints of which areas must be improved and revise their practices. Gathering feedback is important after building something because the team can continuously improve the product in the next iteration.
- **Courage:** it refers to acting in case of fear. Courage is needed to face organizational issues which lower productivity, to stop doing things that you used to do, to accept feedback and act upon it.
- **Respect:** another important value is to have mutual respect inside the team to communicate with each other. It is also needed to accept feedback of fellow teammates to improve together.

The core of extreme programming is found in its practices. These practices can be applied inside a local team, but it can lower the risk even more if everyone is adapting to them. There are thirteen practices:

1. **Sit together:** communication is key and the best way to do it is to sit face to face with the team in the same space without any form of barriers.
2. **Whole team:** everyone involved in the development of a product forms a team. This means that everyone should be working together daily to accomplish a specific goal.
3. **Informative workspace:** the workspace should be set up in such manner which allows face to face communication but also gives each individual privacy when they need it. To communicate information, information radiators ⁴should be used.

⁴ A large display of critical team information that is continuously updated and located in a spot where everyone can see it

4. **Energized work:** the team is effective when developing software when it is free of any distraction. This concept refers to taking measures to assure that the team can get into a focused state.
5. **Pair programming:** it refers to a group of two people which works on the same machine. The idea is that two people can come up with solutions faster and they can constantly give feedback to each other. It also helps when a person is stuck on a certain thing. It is proven that this practice improves quality and quickness because both come up with solutions.
6. **Stories:** the product should be described properly to understand which features are of importance to the customers/users. The stories should include short descriptions of the features which can be used for planning purposes and remind the team what they must focus on.
7. **Weekly Cycle**⁵: the team should meet on the first day of the week to hold a retrospective of their progress. Afterwards, the customer picks the stories he would like to have delivered during the current iteration and the team plans how to approach it. The goal of the iteration is to complete the planned work and test the features. T
8. **Quarterly cycle:** this represents the release. The customer presents the plan of the desired features for a certain quarter to have an idea, for both the team and the customer, of when these features would be available. This is a high-level planning which can end up being manipulated during the cycle. Every weekly cycle the plan is revised so that everyone is aware of the changes.
9. **Slack:** the idea is to add low priority stories to the weekly and quarterly cycles to drop them if the team is not able to reach the goal. Since everything is just an estimation, it is good to plan which things can be dropped.
10. **Ten-minute build:** the idea is to have the application built and tested within ten minutes. Since it is a process which should be ran frequently, it is important to reduce this time to less than 10 minutes. This practice also enforces the concept of

⁵ It refers to an iteration with a one-week period

automatization of the build and test processes. This results in utilizing continuous integration.

11. **Continuous integration:** most teams do not apply this practice since they encounter many problems when trying to do so. In these cases, the teams have a mentality of “if it hurts, avoid it as long as possible”. This is the opposite of extreme programming way of thinking which suggests “if it hurts, do it more often”. The reason for this approach is that whenever a problem is encountered while integrating the code, a lot of problems surface and in most cases integrating more often makes encountering the problem easier.
12. **Test-first programming:** in a normal environment the process is to develop the code, write the tests and then run them. This practice focuses on writing automated tests which would fail, run the failing tests, develop the code to make the tests pass and run the test. This reduces the feedback cycle and helps the developers encounter problems easily and induce less bugs in production.
13. **Incremental design:** this practice proposes to dedicate time to understand the proper breadth-wise perspective of the system design and define the details of the design when developing features. This reduces the risk of having to make changes to the system since the requirements are not known at the beginning of the development. Refactoring is also an important practice which helps the design to be simple and remove any duplicated functionalities.

3.3. Low-risk releases

One of the most important processes while releasing a software is the ability to roll back to the previous version in case of failure. Debugging the application in a production environment is a very complex and difficult process since any mistake can directly affect the users. There should be a system which allows the team to revert quickly during the working hours when the application is mostly used. There are several techniques which allow releases and rollbacks with zero downtime such as blue green deployments and canary release.

The main difficulties in rollbacking the application are the data and the other systems the application is integrated with. If the release process includes changes to the data, it can

be difficult to restore to a previous version. With releases which include multiple systems, the process becomes complex and rolling back can end up being a pain since everything must be restored.

There are two basic principles which need to be followed to ensure a smooth rollback after a failed release. The first is to store the state of the production environment in a backup before releasing the application. The second is to execute the rollback plan prior to every release to ensure that it works.

The simplest way of rolling back an application is to restore it to the previous version. If the delivery process is automated, the easiest way to rollback is to just execute the deployment pipeline of the previous working version. This includes restoring the environment to the previous configuration so everything is the same as before.

Recreating the environment during each deploy is a way to ensure that rollbacking is easy. The reasons are as follows:

- If there is no automated rollback process, but an automated release process, the redeploy is a fixed-time operation and there is less risk involved.
- The process was executed several times successfully before. Since rollbacking is not as common, it is more likely to contain bugs.

There are several disadvantages when creating such a system and these are:

- The time to redeploy is fixed and different than zero. This leads to the application being unavailable during this period.
- The redeploy overrides the new version therefore debugging becomes impossible. This can be lightened if there is a virtual production environment. Simple applications allow deployments of several versions at the same time with a production URL pointing to the desired version.
- Rollbacking the database implies that the changes that happened between a release and its rollback to be lost.

Zero-downtime releases

A hot deployment⁶ is a process of switching application versions almost instantly. In this case the rollback should work in the same way if anything goes wrong.

The key element to zero-downtime releases is to separate the parts of a release so they can be executed independently. It should be possible to release shared resources before a deploy. Static resources are easy to version and manage since there could be various versions available at the same time.

Databases are quite difficult to manage in a rollback. There are two important requirements for rolling back the application:

- All the transactions that happened between a release and a rollback should still be maintained.
- Maintain the application available.

The rollback is easy if the changes to the schema are scripted. However, rollbacking is impossible through scripting if it implies adding data from temporary tables because the integrity is affected or if it involves deleting new critical transactions.

In such cases there are several solutions to roll back to previous versions of the application. When upgrading the database and the application, we should make a copy of the new transactions. This allows us to review these events and execute them into the redeployed system. Such preventions should be taken only if there are sensitive transactions during the rollback since it is a costly process.

Another option is to implement a blue-green deployment. Since there are two environments with the old and new version, we can easily check for differences. At the beginning of a blue-green deployment, a backup must be scheduled and before the backup change the database to a read-only mode. After this process is done, the database is restored in the green environment and then the users are switched back to it. If a rollback is needed, the users are switched back to the blue environment and then the transactions from the green environment are transferred to the blue one.

⁶ A zero-downtime release

Blue-Green Deployments

The idea behind this type of deployment is to have two identical production environments called blue and green. This practice is one of the most powerful techniques used to manage releases.

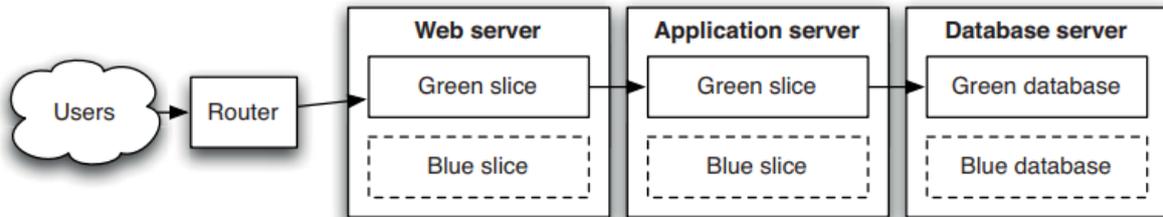


Figure 4: Blue-green deployment

As seen in Figure 4, users are redirected to the green environment which is the current version of the application. Whenever a new release is deployed, a blue environment is created with the new version without affecting the other environment. In the blue environment we can run smoke tests to ensure it is running correctly. At this point, if we want the users to go to the new version, we just reconfigure the router to point new transactions to the blue environment. This causes the green environment to become a blue environment and vice versa. If anything goes wrong, the only thing we must do is reconfigure the router to switch back.

While this improves the redeploying of the product, it can bring some problems. The database cannot be switched instantly since it takes time to migrate all the data from the green environment to the blue environment. One way to solve this problem is to put the database into a read-only mode so there would be no changes made while deploying. Then the database can return to its normal state. If a rollback is needed while the database is in read-only mode, there are no other actions required to complete the process. However, if there are changes, we need to implement a system which allows us to bring those changes back to the blue environment.

Canary Release

Canary release aims to deploy a new version of the application to only a part of the production servers as shown in Figure 5 to get fast feedback from the users. This implies that

only a small set of users will be able to see the new release while the others would still utilize the previous version. If there is a problem with the new version, only a small portion of the users will be affected and therefore reduce the risk of the release.

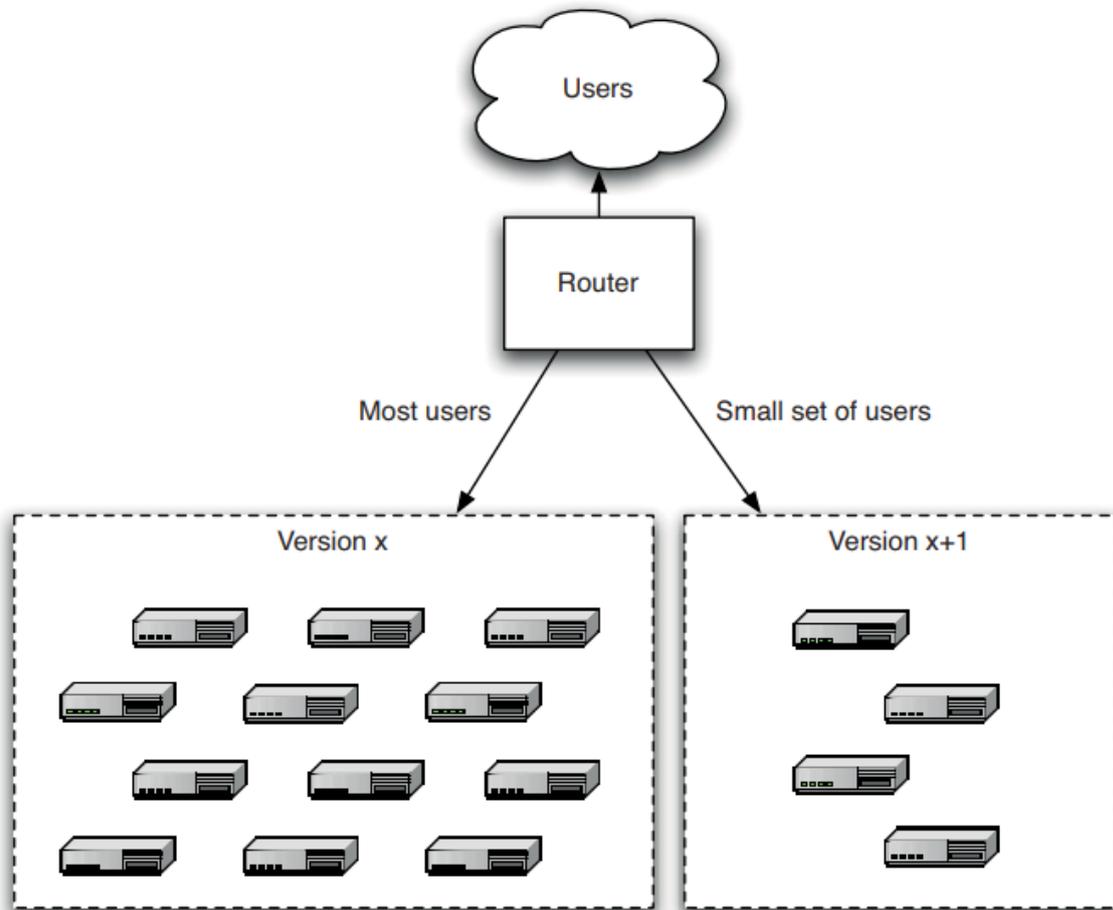


Figure 5: Canary release

Just like in a blue-green deployment, the new version is deployed to an environment which is not routed to smoke test it and check if everything is working as intended. Finally, we select a group of users and give them visibility to the new application. This process is not necessarily being limited to only one new version. There are cases where we might want to obtain feedback about different approaches of a certain feature and select the one the users like the most. The benefits of a canary release are:

- Rolling back is relatively easy
- A/B testing might be used to obtain feedback about different versions of a feature and select the version which is most liked by the users.

- It is easy to verify if a new version supports a large capacity of users since the number of customers who can access the new application can be gradually increased.

Canary release is not a deployment for any type of application. It can also be expensive to maintain a cluster of servers running. There are problems with database and resource upgrades. A way of avoiding this problem is to use a share-nothing architecture where each node runs independently without sharing any service.

3.4. Telemetry

An important part of maintaining the software is to monitor it properly. The team must have an insight about how the application is running in the production environment. The business needs feedback to see if their strategies are profitable. When things go wrong, the operations team must be properly informed about the problem and have the necessary tools to find the cause of the problem to fix it. Historical data must be stored for future purposes such as noting down the reasons why the system behaved improperly to continuously improve the infrastructure.

Using production telemetry⁷ allows the teams to understand the cause of a problem in a production environment and focus on solving the actual problem. To adopt this behavior, the system must continuously create telemetry. The goal is to create telemetry in all environments and in the delivery pipeline to ensure that all the services are working correctly. Whenever something goes wrong, we must be able to determine the exact cause of the problem and focus on fixing the bug.

Creating a centralized telemetry infrastructure

Usually the developers are used to logging operations, but they do it for the events that interest them, and the operations team ends up monitoring only the state of the environments. This results in cases where an unusual behavior appears, and no one can

⁷ An automated communications process by which measurements and other data are collected at remote points and are subsequently transmitted to receiving equipment for monitoring [3].

specifically determine the exact cause of the problem, creating risks and increasing the time in which the system is in a red state.

The solution to finding problems is to design the environments and the application in a way they can generate enough telemetry, so we would understand how the entire system is behaving. If the system has a proper logging and monitoring system, we can start developing dashboards to portray the information to everyone in the team. A modern monitoring strategy should consist of the following components:

- **Data collection:** we should create telemetry a log form for the business logic, application and environments layer to better understand how the system works and how it behaves. Usually this is stored locally on each server, but it is recommended to send the data to a common service to facilitate its reading.
- **A router which stores the events and metrics:** by collecting telemetry we can analyze it and perform health checks on the system.

Centralizing the logs helps us in turning them into metrics. This way we can count errors and summarize them to produce a single metric for the entire infrastructure. We can also perform statistical operations on them to see if there is an increase in the error count and alert the teams.

Telemetry should be collected from the important events from the deployment pipeline as well to see if there is a time difference between executions or to see the status of the tests when deploying to an environment. This allows us to detect any signs of possible problems and correct them before the product reaches the production environment.

The telemetry we create should be able to tell us exactly when, where and how important events happen. These logs should be able to be analyzed manually or through an automated analysis independently from the system that produced them. Preferably, the access should be done through an API to avoid having to wait until another team proportionate the logs.

Create telemetry which helps production

Once we have a telemetry infrastructure, we must make sure we are creating enough telemetry. To accomplish this, the teams must adopt telemetry as a day to day practice in the

new and existing services. Each team should implement telemetry which are mostly dedicated towards their needs.

There are several logging levels which can trigger alerts:

- **Debug level:** information which is of the lowest importance. Usually it is used for debugging the program in case of errors. These logs are normally disabled in the production environment and they are enabled during troubleshooting.
- **Info level:** consists of the information about the user or system transactions.
- **Warn level:** information which could potentially be an error. The logs usually contain long time responses and could trigger alerts.
- **Error level:** information which contains failures such as failed service calls or internal errors. It usually triggers an alert.
- **Fatal level:** information which tells us that the application must be terminated. It triggers alerts.

All significant transaction should be logged to keep the information reliable. These transactions could be [3]:

- Authentication/authorization events.
- System and data access operations.
- System and application changes.
- Data changes such as additions, eliminations or editions.
- Invalid inputs
- Resources
- Health and availability
- Startups and shutdowns
- Faults and errors
- Circuit breaker trips
- Delays
- Backups success or failure

To make it easier to manage all the data, it is recommended to group it in functional and non-functional attributes.

Using telemetry to help with bug fixes

The culture of the company plays a very important role in how outages and problems are fixed. Usually there is a need for blame to prove one team's innocence. This can lead into groups which avoid documenting changes or not publish telemetry, so they would not be blamed.

Telemetry helps us formulate hypotheses about what things went wrong and formulate a cause and an effect to solve it. This results in a faster MTTR⁸ and reinforce connections between the development and operations teams.

To help the teams to find and fix bugs, we must first proportionate the right tools to them and enable the creation of metrics as a daily work. We must develop an infrastructure and a series of libraries which help the development and operations teams to create, display and analyze telemetry.

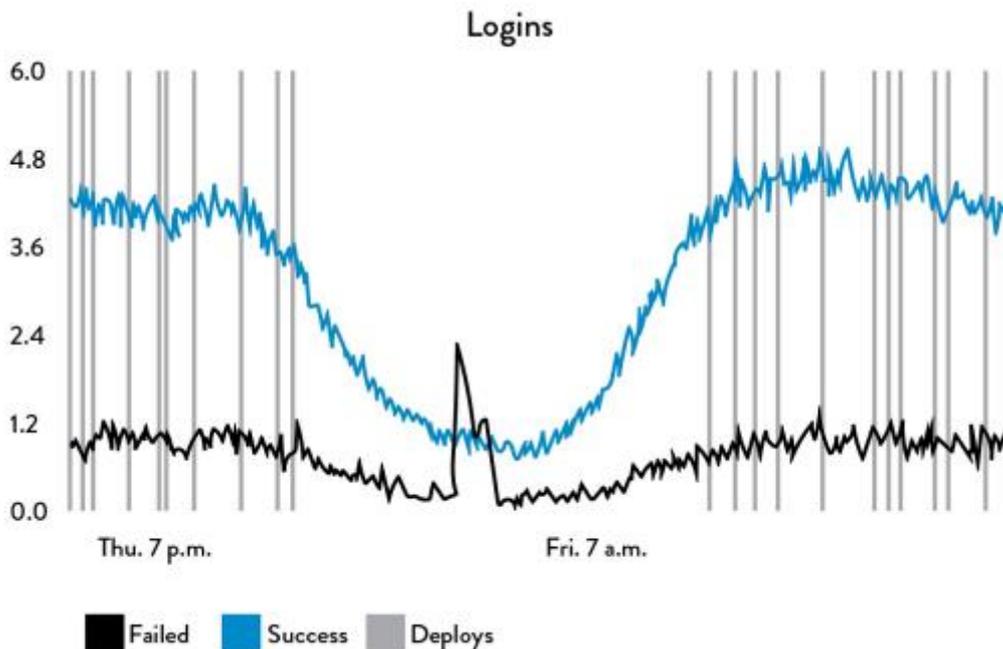


Figure 6: Using Stats and Graphite libraries to generate one-line telemetry

⁸ Mean time to repair

Figure 6 displays how login events create telemetry using one line of code. The graph shows the deployments with gray vertical lines and the amount of successful and failed logins per minute.

Generating telemetry daily helps us continuously improve our application as problems occur and develop in a way which enables us to discover problems easily. This helps us create even more metrics and therefore, helps us study our application even better.

Enable access to telemetry and information sources

The next step in utilizing telemetry is to dispatch the obtained information to the rest of the teams, so everyone gets the data that they are interested in without having to wait for days to obtain it.

If the telemetry is centralized, everyone can analyze how the system is doing overall. The monitoring server should utilize data analyzation frameworks or tools to portray the information in a user-friendly manner. If we put the production telemetry in the areas where the developers and operations work, everyone can see how the system is behaving. It is recommended that everyone in the value stream⁹ can access this information.

This is often referred to as an information radiator, defined by the Agile Alliance as “the generic term for any of a number of handwritten, drawn, printed or electronic displays which a team places in a highly visible location, so that all team members as well as passers-by can see the latest information at a glance: count of automated tests, velocity, incident reports, continuous integration status, and so on. [4]

Once we can radiate our telemetry, we can share this information with the customers. We might want to create a webpage which specifies what services are available in that moment and inform the users of their status. This can also be radiated to the external customers to increase the trust and give transparency to not leave them in the dark.

There should be enough telemetry for every level of the application, the infrastructure and the environments. The metrics we can obtain are:

⁹ Series of steps that an organization uses to build solutions and provide a continuous flow to a customer [8]

- **Business level:** amount of sales, revenue gained from the sales, registration rate, login rate, A/B testing results, etc.
- **Application level:** transaction measurements, response times, application faults, services usage, etc.
- **Infrastructure level:** web traffic, CPU load, disk usage, ram usage, etc.
- **Client software level:** application errors, crashes, transaction times, etc.
- **Deployment pipeline level:** build status, build time measurements, deployment frequency, test status rates, environment status, etc.

By having such metrics, we can determine the health of the services using concrete data instead of pointlessly point fingers at each other. We can also determine if there are security faults and see where the errors originate from. If we correct these errors at an early stage, we can do it in a relatively small amount of time instead of leaving the problem coexist and become more complicated with each iteration.

3.5. Security

DevOps practices help with the delivery of the software by making the feature releasing faster. These practices rely on the automatization of the deployment pipeline since changes are happening at a fast pace. The security of the system becomes affected since everything is so dynamic and there are many things which need protection. The solution to this problem is to automatize security efforts.

Any change to the system can bring new vulnerabilities in the network which renders the previous security system useless. Our goal is to reduce or eliminate the risk which comes with new changes.

While a slow deployment pace seems like the way to go about this problem, it is not necessarily the case. Slow developments cause a lot of time loss and can end up breaking many other things which adds work towards testing the application

These issues are addressed by the DevOps culture which focuses on a rapid development and deployment. Rollbacks become relatively easy to do and with automation everything can be executed by the push of a button.

The sensation of faults in security during continuous delivery is misplaced since creating small patches to the production application can be easily understood and acted upon. This way we can develop security strategies easily since the change is minor. The big difference is that everything just happens at a faster pace, which means that vulnerabilities should be acted upon quickly. These compromises are solved so quickly that there is no time for an attacker to act upon it which turns out in a strategic advantage for us.

3.6. The Pipeline in a DevOps culture

“Agile software development has broken down some of the silos between requirements analysis, testing and development. Deployment, operations and maintenance are other activities which have suffered a similar separation from the rest of the software development process. The DevOps movement is aimed at removing these silos and encouraging collaboration between development and operations.” [5]

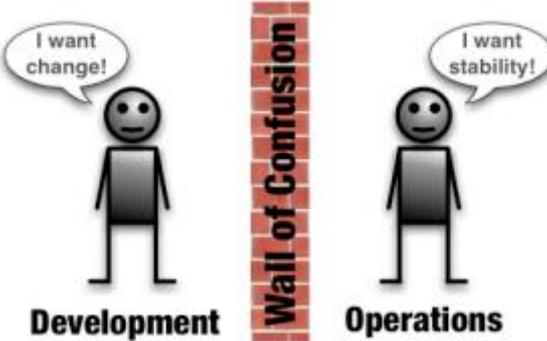


Figure 7: Conflicts between the operations team and the development team

The DevOps culture encourages collaboration between the development team and the operations team where the responsibility is shared. If the maintenance of the system is handed over to the operations team by the development team, the developers lose interest in ways to improve the system since “it is not their job”. However, if the responsibility is shared, both teams come to an understanding of each other’s needs and work together on ways to simplify deployments and maintenance. The practices of continuous delivery and continuous deployment surfaced because of this communication. By working together, the

operations team comes to an understanding of the business logic through the development team.



Figure 8: DevOps collaboration schema

When trying to adopt the DevOps culture, there should be no silos between operations and development. Documentations are usually a bad communication sign because the teams should be working together on finding solutions and implementing them. This should be done early on to encourage this kind of behavior. The documentation ends up as a reason to blame each other when instead they should be taking responsibility together for the system. Another antipattern is to have a designated DevOps position or team inside a firm. This results into creating silos in the organization and this goes completely against what DevOps really is.

DevOps culture also supports autonomous teams. The organization should put their trust in their staff and encourage this practice. The teams should be able to make decisions and changes as they see fit without going through complicated processes. The risk ends up being managed differently and it eliminates fear within the team.

One of the biggest changes that the DevOps culture brings is that it becomes easier to deploy code to production. To reduce the risks of failure when deploying, the team should value the quality of the code when developing. The techniques to achieve a low risk deployment are Continuous Delivery [3.2] and self-tested code.

Feedback starts being of importance to the team because it allows to continuously improve the relation between the development and operations teams. Using the feedback to monitor the production environment becomes very useful since health checks and improvements can be performed.

The DevOps movement encourages automation to improve collaboration and reduce risks which come with manual processes. By having an automated build, test and deployment, the teams end up being freed of the burden of repetition and can focus on feature developing and improvement. Since automation implies having it written as code, we can easily version these changes and document them. By automating the server configuration, the config becomes available to anyone. Doing so, the server configuration becomes easy to understand and both teams can end up changing it.

3.7. Continuous learning and improvement

The first release of a software is the first stage of its life and its first time stepping into the real world. Since the application will continue evolving, there are going to be more releases. Therefore, it is important that the deployment pipeline evolves with these changes.

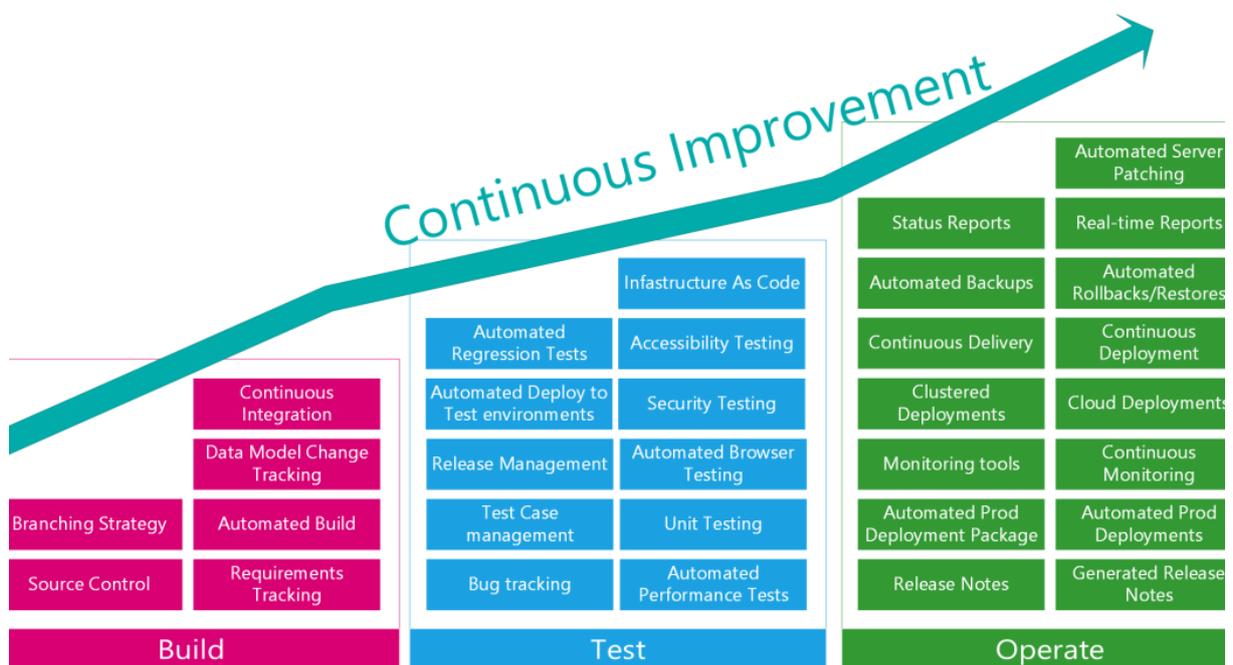


Figure 9: Continuous improvement for DevOps

The team should focus on regularly hold retrospectives on the delivery process. Each team must reflect upon the green and red deliveries and plan future improvements. There should be a member in the team which makes sure that the ideas implemented. On the next iteration, the selected member should inform the team of the changes that happened or didn't happen. This process is known as the Deming cycle¹⁰.

Everyone in the organization should be involved in the continuous improvement process. If feedback happens within silos it can lead to blaming each other, local optimization instead of global optimization. This can be avoided by radiating the feedback within the organization as explained by the DevOps practices.

3.8. Maturity model

	Initial	Managed	Defined	Quantitatively Managed	Optimizing
Culture & Organization	<ul style="list-style-type: none"> Teams organized based on platform/technology Defined and documented processes 	<ul style="list-style-type: none"> One backlog per team Adopt agile methodologies Remove team boundaries 	<ul style="list-style-type: none"> Extended team collaboration Remove boundary dev/ops Common process for all changes 	<ul style="list-style-type: none"> Cross-team continuous improvement Teams responsible all the way to production 	<ul style="list-style-type: none"> Cross functional teams
Build & Deploy	<ul style="list-style-type: none"> Centralized version control Automated build scripts No management of artifacts Manual deployment Environments are manually provisioned 	<ul style="list-style-type: none"> Polling CI builds Any build can be re-created from source control Management of build artifacts Automated deployment scripts Automated provisioning of environments 	<ul style="list-style-type: none"> Commit hook CI builds Build fails if quality is not met (code analysis, performance, etc.) Push button deployment and release of any releasable artifact to any environment Standard deployment process for all environments 	<ul style="list-style-type: none"> Team priorities keeping codebase deployable over doing new work Builds are not left broken Orchestrated deployments Blue Green Deployments 	<ul style="list-style-type: none"> Zero touch Continuous Deployments
Release	<ul style="list-style-type: none"> Infrequent and unreliable releases Manual process 	<ul style="list-style-type: none"> Painful infrequent but reliable releases 	<ul style="list-style-type: none"> Infrequent but fully automated and reliable releases in any environment 	<ul style="list-style-type: none"> Frequent fully automated releases Deployment disconnected from release Canary releases 	<ul style="list-style-type: none"> No rollbacks, always roll forward
Data Management	<ul style="list-style-type: none"> Data migrations are performed manually, no scripts 	<ul style="list-style-type: none"> Data migrations using versioned scripts, performed manually 	<ul style="list-style-type: none"> Automated and versioned changes to datastores 	<ul style="list-style-type: none"> Changes to datastores automatically performed as part of the deployment process 	<ul style="list-style-type: none"> Automatic datastore changes and rollbacks tested with every deployment
Test & Verification	<ul style="list-style-type: none"> Automated unit tests Separate test environment 	<ul style="list-style-type: none"> Automatic Integration Tests Static code analysis Test coverage analysis 	<ul style="list-style-type: none"> Automatic functional tests Manual performance/security tests 	<ul style="list-style-type: none"> Fully automatic acceptance tests Automatic performance/security tests Manual exploratory testing based on risk based testing analysis 	<ul style="list-style-type: none"> Verify expected business value Defects found and fixed immediately (roll forward)
Information & Reporting	<ul style="list-style-type: none"> Baseline process metrics Manual reporting Visible to report runner 	<ul style="list-style-type: none"> Measure the process Automatic reporting Visible to team 	<ul style="list-style-type: none"> Automatic generation of release notes Pipeline traceability Reporting history Visible to cross-silo 	<ul style="list-style-type: none"> Report trend analysis Real time graphs on deployment pipeline metrics 	<ul style="list-style-type: none"> Dynamic self-service of information Customizable dashboards Cross-reference across organizational boundaries

Figure 10: Continuous delivery maturity model

¹⁰ It is a four-part management method which preaches continuous improvement. It is made up of: plan, do, study, act

To evaluate an organization's practices and working practices, a maturity model should be used. This model helps the organization to identify where they stand in terms of maturity and where it could improve. All the roles involved in the process of delivering software and how they work together must be studied to obtain a proper view of the organization's standing. This model can be seen in Figure 10: Continuous delivery maturity model.

The main goal is to improve the organization. What we are looking for is:

- Cycle time reduction. We need to deliver the software fast, so the organization could improve its profits.
- Reduced bugs. This improves efficiency and cuts time from offering support.
- Planning should be effective by increasing the predictability of the software delivery cycle.
- The ability to follow certain practices.
- To effectively determine and solve which risks come with a delivery.
- Reduced costs by reducing risks that come with software delivery.

The maturity plan will help improve the organization in these cases. To achieve these improvements, we should apply the Deming cycle.

1. Determine the current configuration and release management maturity.
2. Chose the area which has the lowest maturity level. In this case, we need to decide what improvements are going to be implemented and estimate its costs, benefits and priority.
3. Implement the changes planned in the previous stage. In most cases a conceptual model can be created. If the organization is doing bad in a certain area, people will be motivated to bring these changes.
4. After the changes have been implemented, we need to measure if they are truly working through acceptance criteria. By consulting the teams, we can learn about the execution of the implementations and where they could be improved.
5. Repeat the process in base of the knowledge obtained from the previous iterations.

The improvements should be done incrementally, one level at a time. There should be no jumps since they can cause a lot of impact and will result in failure. These changes can take up to several years to implement and we should look for changes which bring value to us.

Every step is a learning process and we should not give up when a certain step fails; we should just try again in a different way.

4. Implementation, from Theory to Practice

In this chapter I will focus on presenting how I implemented the entire system which is automated by using a continuous delivery pipeline. This will serve as a demonstration about how the theory which was explained in the previous chapter can be applied in a real environment.

The idea is to implement a continuous delivery pipeline which will release the software to production automatically. There are several elements required to implement a delivery pipeline.

First, we need an application to release in a production environment. It should contain end-to-end transactions to be able to test its behavior in a real environment. The application to be implemented is called Funny Stories which will be a simple application with basic end-to-end operations.

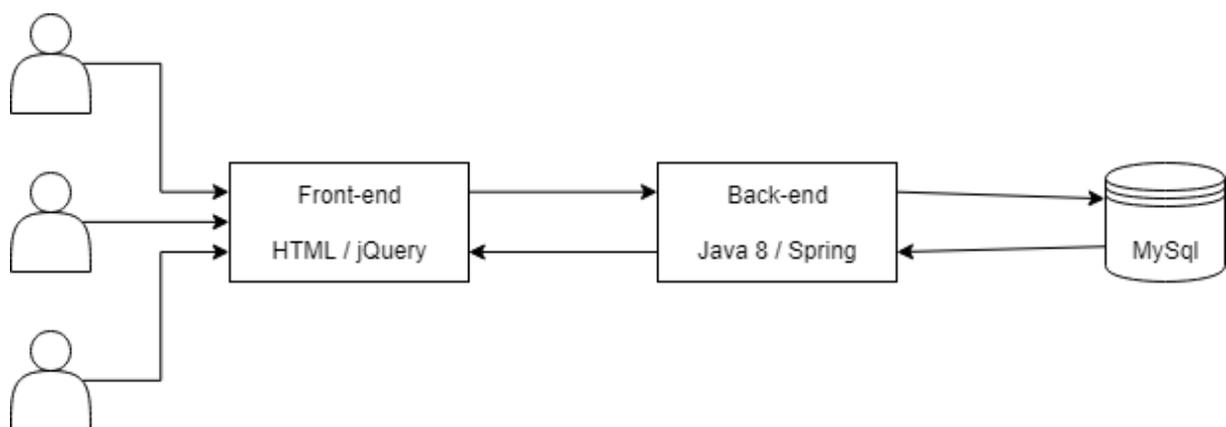


Figure 11: Basic application concept

Second, we need a hosting service which will serve as the production environment and will hold the application.

4.1. Work on small iterations

The goal is to design a minimum viable product (MVP) which is a technique used to design a new product or website with enough features to satisfy the customers. It is a deployable version which has the following characteristics:

- It has enough value for people to buy it.
- It proves enough benefit to maintain customers.
- It provides a feedback loop to guide future development.

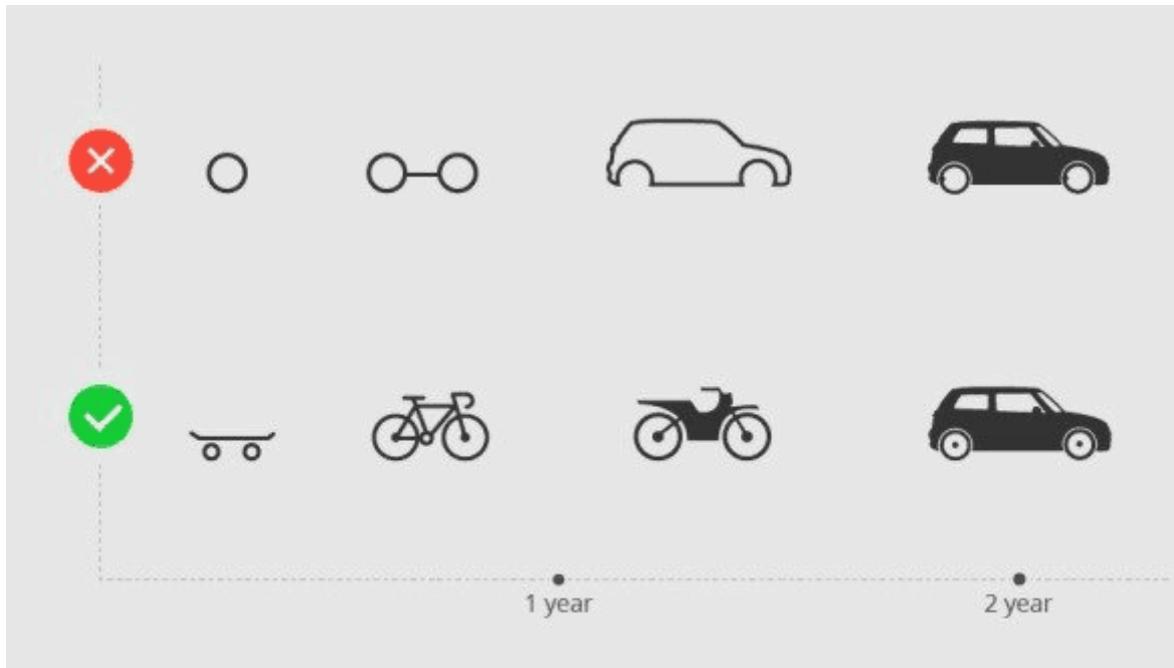


Figure 12: MVP design

By following the MVP model, I designed the first version of the product which is called Funny Stories with the following characteristics:

- Funny Stories will be a website which will contain posts published by users. There should be multiple post categories which the user can select and visualize. The users should be allowed to create new posts in the category they visualize. The posts will have a ranking system which will allow the users to upvote or downvote them.
- The website should be public and accessible by anyone.
- The interface should be minimal, but user friendly.
- The system should use open source tools which will have zero costs.
- There website should be available by 27th of June 2018.

The idea is to have small iterations while developing this application. The features should be integrated as often as possible to have a working product in production as soon as possible. This will bring a lot of benefits since it is less likely for anything to go wrong when performing a change. Also, if the changes are small, reverting to a previous version will not

have much impact. It will help preserve the application's behavior and ensure a more disciplined way of working.

4.2. Choose the right tools

An important step in the development of a software is to choose the tools which are most suited for the demanded product and the technologies which the development team feels most comfortable with.

Version control

Version control systems are tools which help a software team keep track of all the changes inside a repository. If any mistake is made, the developers can compare the files in the repository to previous versions or perform a rollback.

The version control used in this project is GitHub since it is a free repository hosting service. It provides a console and a web-based interfaces where developers can store their projects.

GitHub provides project management tools (backlog and Kanban) which will be used to organize the project and keep track of the realized and pending work. This way we can keep track of all the iterations and estimations of the project.

Application

The application will have three main components: a back-end, a front-end and a database.

The back-end represents the data access layer and the business logic from the server side. It should be a simple and easy to test component. For such purposes and possibility of future implementations we will create RESTful web services. To achieve a simple and easy way to implement backend, we will use the Spring REST framework with Java 8.

The front-end will be implemented using HTML and jQuery since it is the most basic way to access the RESTful services of the back-end. The UI layer of the application will be

personalized by using Bootstrap and CSS to change its appearance and have a user-friendly interface.

For the database we will use MySQL since it is an open source relational database management system. It is the most common and easy to use database and it will work great with this project.

Project management

For packaging the application, we will use Maven with Node.js for the back-end and front-end respectively. These tools are great for managing external libraries and internal structure of the project.

Maven is a build automation tool which is primarily used for Java projects. This tool provides a standard way to build a project and manage its dependencies. Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period. To attain this goal there are several areas of concern that Maven attempts to deal with [6]:

- Making the build process easy.
- Providing a uniform build system.
- Providing quality project information.
- Providing guidelines for best practices development.
- Allowing transparent migration to new features.

Maven will be used to package the back-end into a JAR which will contain all its dependencies (Spring, Jackson, etc.).

Node.js is a JavaScript run-time environment which includes everything a program written in JavaScript needs. It provides a package ecosystem of open source libraries which is the largest in the world.

To avoid any environment changes when deploying the application, we will use Docker containers to virtualize the system it runs on. Containers allow the developer to package an application with all its dependencies and deploy it as a package. This ensures that the application will run on any Linux machine and it will not be affected by any customization of

the machine. This works similarly to a virtual machine, with the difference that this virtual container will utilize the Linux kernel it runs on and the applications inside it. By using this tool, we can ship the front-end and the back-end as an entire package and ensure that it will run on any environment.

Pipeline

Since we need to implement a continuous delivery pipeline, we need a platform which will automatically build, test and deploy the application. A way to do this is to have a hosted platform which will detect any commits we make in the project's repository and execute a series of steps which we specify.

"Travis CI is a hosted continuous integration platform that is free for all open source projects hosted on GitHub. With just a file called ".travis.yml" containing some information about our project, we can trigger automated builds with every change to our code base in the master branch, other branches or even a pull request." [7]

Travis CI is a tool which integrates very well with GitHub and will link any commit to a Travis build. It will show notifications directly on GitHub's interface and allow a better understanding of the project's status.

The pipeline should notify the developers of the status of an execution once it is completed. To check the notifications, Travis CI provides a notification system which sends a message to a Slack server. Slack is a very useful tool to communicate with your team and using Travis, we can keep track of the builds.

Infrastructure

To reproduce a production environment, we will need a server which will hold our application. With AWS Elastic Beanstalk, we can create environments and deploy or manage applications without worrying about the infrastructure that runs those applications.

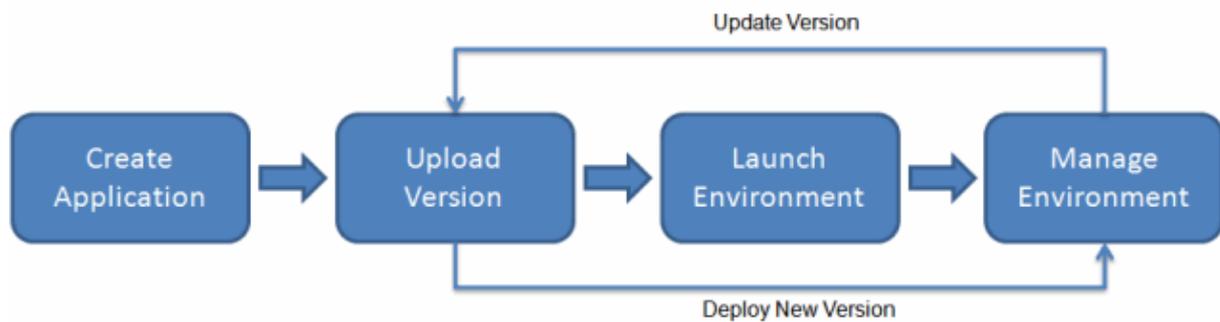


Figure 13: AWS Elastic Beanstalks flow

The database will be hosted on an Amazon Relational Database Service (RDS). It is a service which makes it easy to set up, operate and scale a relational database on the cloud. It is a great service since it will allow us to host the entire infrastructure on the same provider.

Since one of the objectives of this project is to keep everything cost-free, we will be using one environment for production and the local machines of the developers.

4.3. Start with a walking skeleton

A walking skeleton is a proof of concept of the basic architectural concept. It is a minimal implementation of the system which performs an end-to-end function.

For this project a good walking skeleton could be a simple functionality which will prove that all the components could be tied together. For this purpose, we can implement a feature which allows the users to view the categories from the database. This implies that the front-end must call a RESTful service from the back-end and then the back-end accesses the database to obtain the categories. Afterwards, the back-end will return the data to the front-end and the information will be displayed to the user.

To implement this feature, we will use an architecture (Figure 15) which is based on the MVC pattern. The front-end is a consumer of the back-end RESTful services made up of a single HTML page which uses Ajax petitions to get the information.

RESTful web services are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web.

In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

- **Resource identification through URI:** A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery.
- **Uniform interface:** Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource.
- **Self-descriptive messages:** Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.
- **Stateful interactions through hyperlinks:** Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.

Before we can start writing any code, we need to configure and establish a branching strategy for our version control. Since our final goal is to have a continuous delivery pipeline, we can employ a basic branch-per-issue workflow. This refers to having a single master branch which contains our production code and should always be green and feature branches which extend from the master branch and are used to implement features.

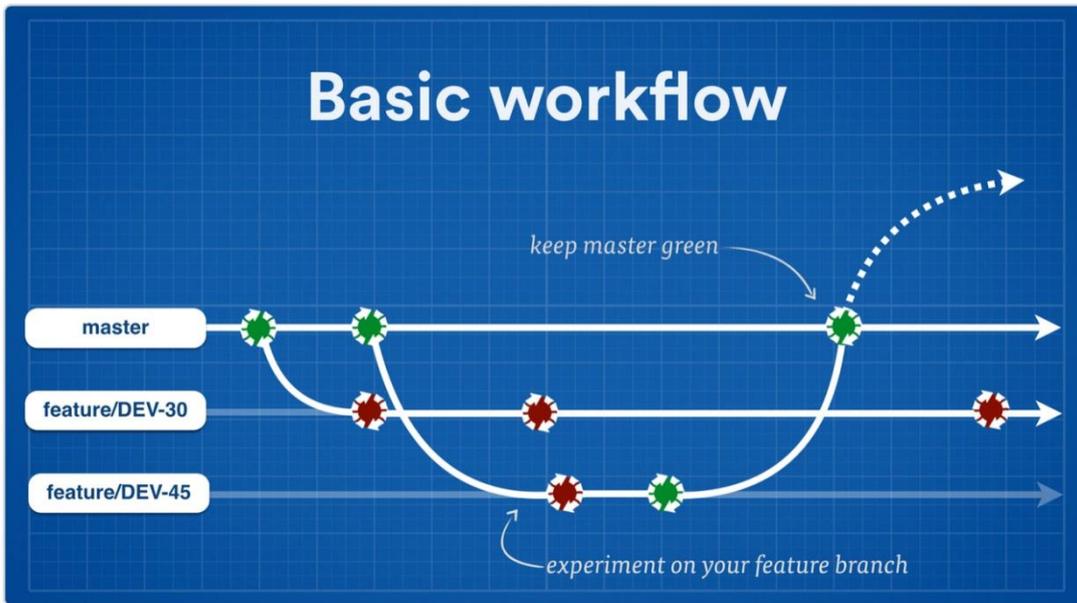


Figure 14: Branching strategy

The back-end of the application is made of three layers (Figure 15):

- **REST Controller:** here is where we define the HTTP endpoints of the RESTful services and control all the errors of the application. The response of the service will return standard HTTP status codes (100x, 200x, 300x, 400x, 500x) and any data defined by the service.
- **Service:** this is the layer which holds all the business logic which is related to the application.
- **Repository:** here is where the data logic is located. From here we can access different data sources to obtain information.

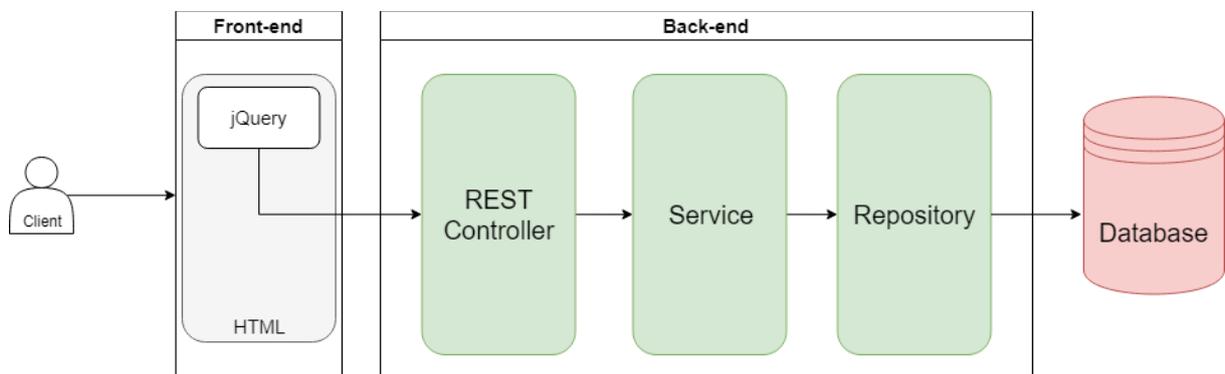


Figure 15: Funny Stories architecture

The category service from our walking skeleton will be “/category” which will be a GET operation with the endpoint “/category”. Since we are going to implement RESTful services,

we will use JSON data outputs for this service. The data model for the category service should be as following:

```
1 {
2   "statusCode": 200,
3   "errorMessage": null,
4   "data": [
5     {
6       "id": 1,
7       "name": "School",
8       "description": "School Posts"
9     }
10  ]
11 }
```

Figure 16: JSON output for RESTful services

At the point we can define the database model for this feature. We will need a table which simply stores the name of the category and a short description of it. It should be defined as following:

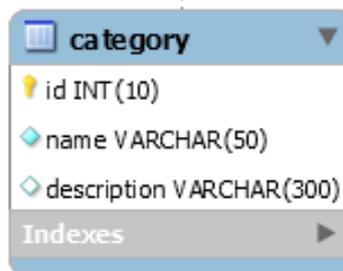


Figure 17: Funny Stories database table - category

Since we are still in the testing phase of our implementation, we will create a continuous integration pipeline to prove that the implemented walking skeleton could be automatized.

First, we need to be able to execute all the steps required by a continuous integration pipeline in a local environment. To do so, we need to create bash scripts which will execute unit tests, build the application and create a docker image with the binary of the walking skeleton so it can be later deployed to any environment. To accomplish this, we need to create a docker repository through docker's web interface and use it to store our images.

By utilizing Travis CI, we can now implement the continuous integration pipeline by executing the script in the order we want. The configuration of the pipeline for our application should be as following:

1. Execute the unit tests using maven. The command is: `mvn verify`.
2. Build the binary using maven. The command used is: `mvn build`.
3. Build a docker image with the constructed binaries.
4. Version the image by using the unique build number from Travis.
5. Upload the image to our docker repository.
6. Notify the status of the finished build on Slack.

If the process was successful, we can run this image by simply downloading and executing it.

4.4. Outside-In driven development

Now that we have proved that our concept is functional, we can move on and define how the rest of the product should be implemented.

While we implemented the display of the categories, there are still many features left. At this point we can decide which of the remaining features we want to implement. These features are:

- Show the posts from a category.
- Like or dislike a post.
- Create a post in the current category.

One of the principles of Extreme Programming is to utilize Test Driven Development. This means that we, as developers, should prioritize writing tests first and then code.

Outside-In TDD lends itself well to having a definable route through the system from the very start, even if some parts are initially hardcoded. The tests are based upon user-requested scenarios, and entities are wired together from the beginning. This allows a fluent API to emerge and integration is proved from the start of development.

By focusing on a complete flow through the system from the start, knowledge of how different parts of the system interact with each other is required. As entities emerge, they are mocked or stubbed out, which allows their detail to be deferred until later. This approach implies the developer needs to know how to test interactions up front, either through a mocking framework or by writing their own test doubles. The developer will then loop back, providing the real implementation of the mocked or stubbed entities through new unit tests.

When using Outside-In TDD, the developer needs to have prior knowledge of how the entities in the system will communicate. The focus is on how the entities interact rather than their internal details, hence the use of high level acceptance tests. Outside In solutions often apply the Law of Demeter¹¹. This adheres well to the “Tell Don't Ask” principle, and results in state only being exposed if required by other entities.

With Outside-In TDD the implementation details of the design are in the tests. A change of design usually results in the tests also being changed. This may add more risk, or take more confidence to implement.

A good technique to put in practice the Outside-In Development is the Double Loop TDD which consists in starting with an acceptance test written from the perspective of the user. The next step is to write a unit test for each entity and collaborator that we need for this implementation. When the feature will be done, the acceptance test will be green.

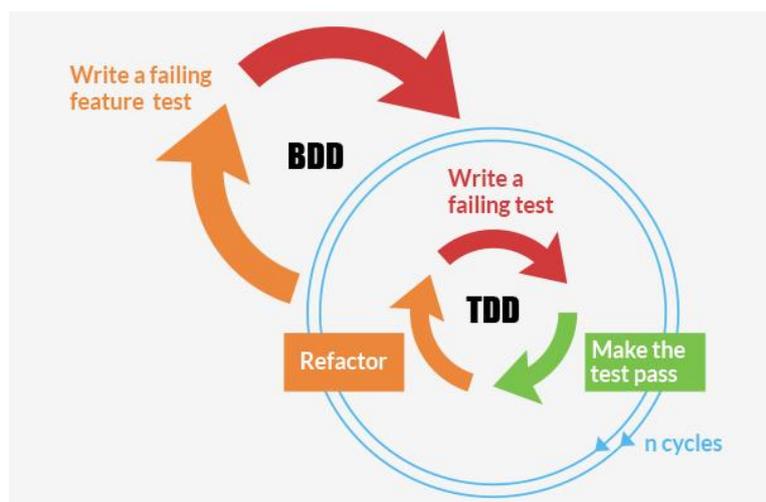


Figure 18: Double Loop TDD

¹¹ A new entity is never created without the primary entity asking what it wants from it

Taking in count these practices, we continue to develop our application. Since we are using outside-in practices, we start by creating the acceptance tests from the front-end side. The rest of the implementation is done in iterations by designing acceptance tests with Mocha.js which is a JavaScript testing framework that can run in the browser or in the console. The tested front-end features can now be implemented and we can move on to the back-end.

In the back-end we use junit and Mockito to design unit and integration tests, and then we write the code just as in the walking skeleton case. To access the data from the database, we use native queries. The final database schema is displayed in the Figure 19.

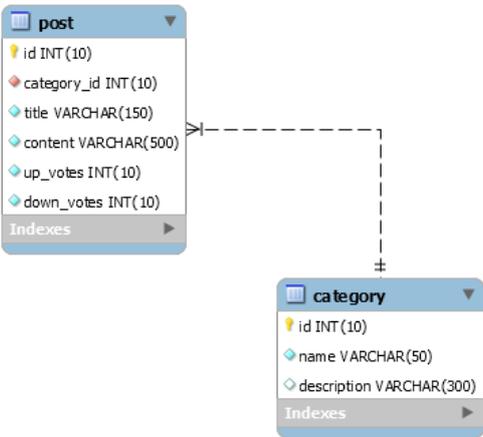


Figure 19: Funny Stories full database schema

If the tests pass in the local environment, we can commit them to the feature branch and let the pipeline do the work.

4.5. Multiple environments management

The reason for having multiples environments is to test the application before deploying to production, where real users will have access to the new changes. The simplest environment is the localhost, where the developer can test or debug the code in the fastest way. The number of environments in a company may depend on the size of the company, team's organization, architecture, costs etc...

The challenge that we have when we are dealing with multiple environments is how we manage this environment to accomplish the needs for testing the application. First, you need to build these environments on top of a series of principles such as:



- The environments need to be as similar as possible to the production.
- Each member of the team should be able to spin up a new environment to test the application.
- To spin up a new environment you should only press a button or launch a command.
- The application needs to be immutable when is deployed to different environments.
- The environments should be independent, without sharing resources or data.
- The deployment mechanism should be the same for all the environments.
- Try to automate everything.

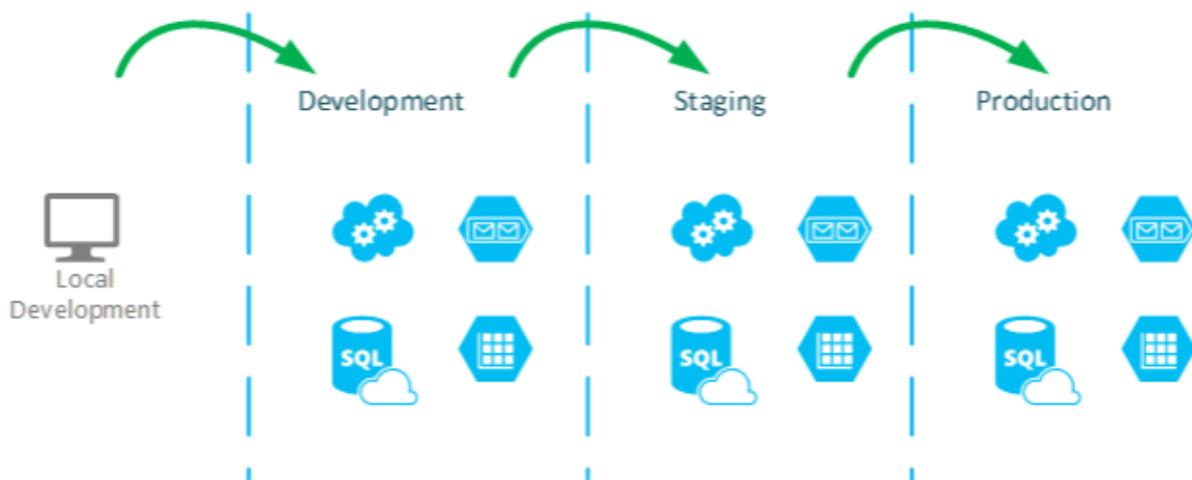


Figure 20: Multiple environment transition

On this project the approach was to start with one environment to maintain a zero cost on our cloud provider. After we finish the automation for the infrastructure with Terraform, we could create or destroy each environment in terms of minutes.

Creating or destroying environments with one command gives you the power to manage multiple environments in the same time, without making an extra cost in your cloud provider.

We wanted to practice deployment techniques like Blue-Green or promoting the application between different environments from the beginning. This way we avoid having any surprises after the entire implementation of the application is done.

We package the application inside a docker image with all his dependencies. This way we obtain an immutable application that can be deployed in any environment. This image can be uploaded to a repository, versioned and can be run also in the localhost.

There are some details that need to be different between environments, like the database URL where the application connects to save the data. In this case we inject this information with environment variables. This way we keep the immutability and we obtain flexible application that can be ran in new environments only by adding new values for the environments variables.

4.6. Deployment pipeline anatomy

As explained in Chapter 3, the deployment pipeline is an automated manifestation of the process from getting the software from a repository to the users. This process consists of building, testing and deploying the software to the production environment. By utilizing Travis CI, we can implement a fully automatized deployment.



Figure 21: Continuous delivery pipeline

A Travis CI build stage is a way of grouping jobs and running them after another stage has completed. This is a sequential process in which each stage runs on a different virtual machine.

Since we have a project with multiple branches, we should define what strategy we employ for each one of them. Since the master branch is the only one that should end up in the production environment, we separate the pipeline in two parts: building and deploying Figure 22.

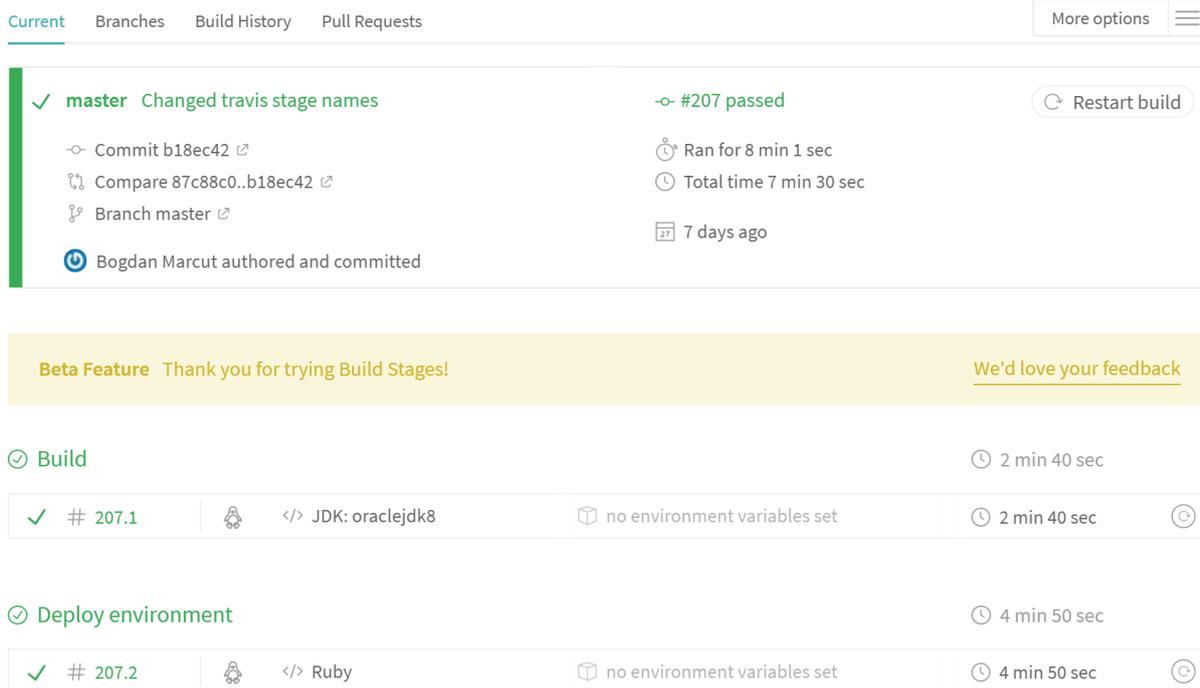


Figure 22: Travis CI interface

In this case the building stage is a process which we would execute for any commit made on any branch. This would produce a docker image for every commit so it could be tested on any machine. The process is the same as the continuous integration process which was implemented for the walking skeleton. The only difference is that maven is configured differently so it would execute the front-end and back-end tests.

The deployment stage is where we change the version in the production environment. This step is restricted to be executed only when committing to the master branch. In this case we would have a stable version which could be presented to our customers.

The steps to deploying the application to production are:

1. We must download the docker image from the repository since we are running on a new virtualized machine (stage).
2. We must upload the downloaded image into the AWS system. This is done by using a AWS S3 repository which is a data storage location.
3. We must use a Blue-Green Deployment to release the new version. Therefore, we must create a new environment (called blue) with the configuration which was

previously defined and the version which was uploaded to S3. We obtain the URL of the created environment which should have the following format: *pro-funny-stories-blue-<version>.eu-west-3.elasticbeanstalk.com*.

Funny Stories App

env-207 Environment tier: Web Server Platform: Docker running on 64bit Amazon Linux/2.9.2 Running versions: 207 Last modified: 2018-06-27 20:38:05 UTC+0200 URL: pro-funny-stories-green.eu-west-3.elasticbeanstalk.com	env-208 Environment tier: Web Server Platform: Docker running on 64bit Amazon Linux/2.9.2 Running versions: 208 Last modified: 2018-06-27 21:41:48 UTC+0200 URL: pro-funny-stories-blue-208.eu-west-3.elasticbeanstalk.com
---	--

Figure 23: AWS Elastic Beanstalk blue-green environments before deploying

4. We smoke test the environment to check if the uploaded version is correct.
5. We exchange the URLs of the green environment (*pro-funny-stories-green.eu-west-3.elasticbeanstalk.com*) and the blue environment (*pro-funny-stories-blue-<version>.eu-west-3.elasticbeanstalk.com*).

Funny Stories App

env-207 Environment tier: Web Server Platform: Docker running on 64bit Amazon Linux/2.9.2 Running versions: 207 Last modified: 2018-06-27 21:43:37 UTC+0200 URL: pro-funny-stories-blue-208.eu-west-3.elasticbeanstalk.com	env-208 Environment tier: Web Server Platform: Docker running on 64bit Amazon Linux/2.9.2 Running versions: 208 Last modified: 2018-06-27 21:42:12 UTC+0200 URL: pro-funny-stories-green.eu-west-3.elasticbeanstalk.com
--	---

Figure 24: AWS Elastic Beanstalk blue-green environments after deploying

6. We smoke test the new green environment to make sure that the version is correct.
7. (optional) We rollback the application to the previous version by using the AWS interface.

5. Results vs Objectives

This chapter focuses on portraying the results once the development process ended and the application was ready to deliver to production.

The final product has satisfied the initial objectives. The pipeline can automatically build, test and deploy the application while executing tests and ensuring that there is no error in the process. Here are the main accomplishments of this system:

- Fully automatized deployment pipeline.
- Fully automatized infrastructure.
- Can easily rollback to a previous version.
- Environment independent.
- Can be deployed on any system and ensure that it works.
- Produces telemetry.
- The application is simple and usable with all the planned features.
- Everything can be executed from a local environment.

Future improvements:

- Implement functional tests.
- Extend the application's functionality.

During the implementation of this system I encountered several problems which were caused by the chosen tools or because of time constraints. These problems did not affect negatively the final product, but they did stray a bit from the theory.

Travis CI is a great software which allows developers to implement automatized pipelines. During my implementation, a continuous delivery pipeline was supposed to deploy the application by pushing a button. This is not possible to implement by using Travis since it does not support non-commit executions. This forced me to automatize the deployment process and deploy the application to production each time there was a commit to the master branch which resulted in a more advanced pipeline. It is possible to have the deploy done manually, but it would imply the usage of another application and this would unnecessarily complicate the deployment process.

The walking skeleton was based on one of the main application. In a real organization, the walking skeleton would be a mock application which would have nothing to do with the main product. This was done to save time and because the application was so simple that it allowed the implementation of a very basic end-to-end feature.

The rollback is done by pushing two buttons instead of one. This is because Travis CI does not permit the implementation of non-commit executions and a new application should be introduced to the system.

The application is currently in production and it can be accessed by any public user. The final product can be seen in Figure 25. The website where it can be accessed is: <http://pro-funny-stories-green.eu-west-3.elasticbeanstalk.com/>

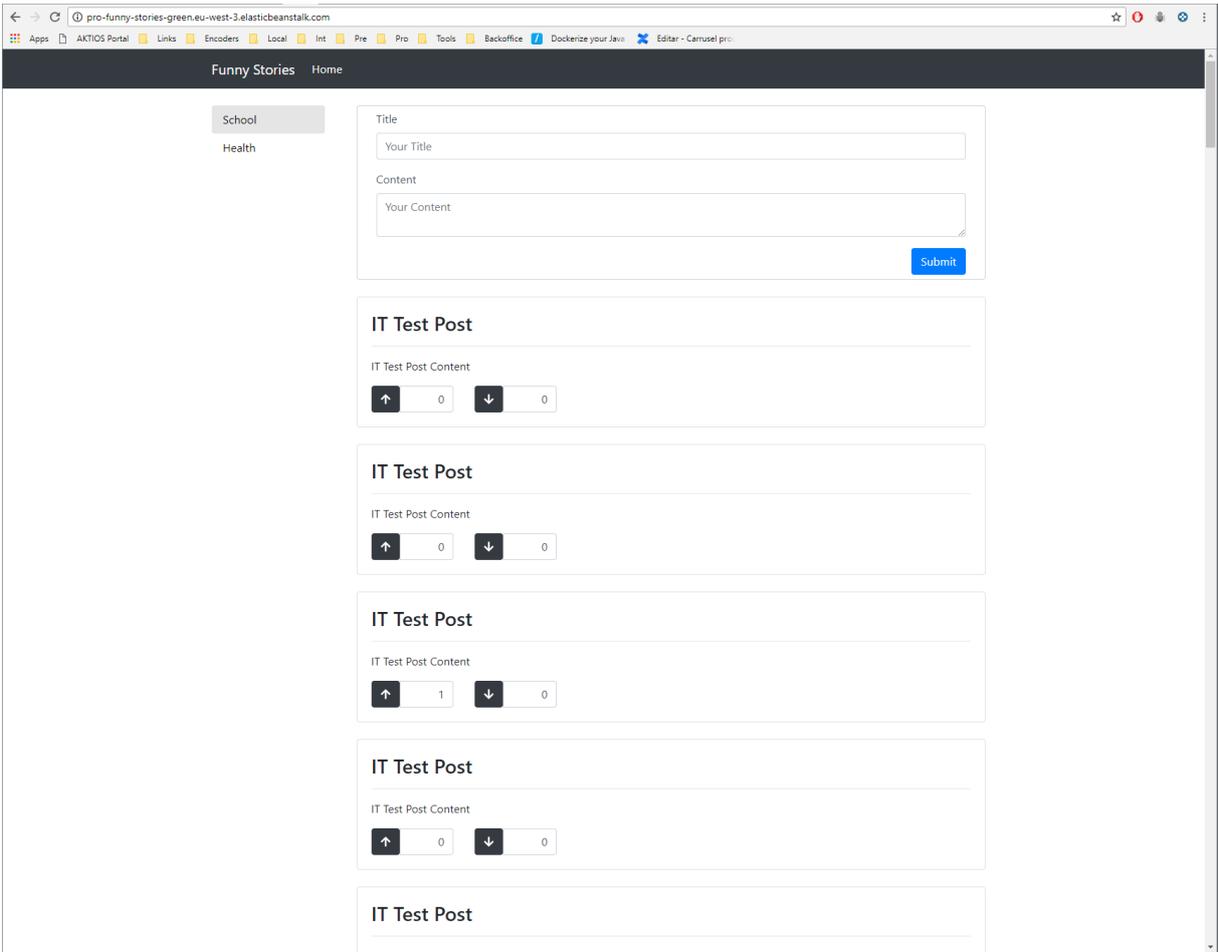


Figure 25: Funny Stories - final product

6. Conclusions

After three and a half months of work I have fulfilled the initial objectives. To achieve this, a good planning, previous requirements analysis, modern tools and technologies, and a qualitative test system were needed.

I learned a lot from the bibliography that I read and the new tools that I used. I also found some limitations in some cases, which are almost impossible to detect when you choose them at the beginning of the project, but in my opinion, it is a common fact that it is present during a developer's career. The most important thing is to react fast and to take good decisions as soon as possible.

The Continuous Delivery practice and DevOps culture has grown a lot in the last years and more and more companies figure out that this is the right way to develop their applications. There are a lot of tools available in the market and sometimes it is a matter of preference to decide which ones to use.

In a world with so many options I think that the most important parts of my work are the practices that I used, the principles, and methodologies because the tools are changing constantly and a good organization is always prepared for the change.

The project can be extended with new features such as:

- Invest money into the servers to maintain two environments available at the same time.
- Test A/B deployments to receive feedback about the new features from the users.
- A more advanced monitoring system.
- Transform the application from a monolith to a microservice application.
- Implementation of more types of testing.
- Implementation of new features to the software such as: a login system, limitations to the upvotes and downvotes, etc.

7. Bibliography

- [1] J. Humble and D. Farley, *Continuous Delivery*, Upper Saddle River, NJ: Addison-Wesley, 2010.
- [2] Agile Alliance, "Extreme Programming," Agile Alliance, [Online]. Available: <https://www.agilealliance.org/glossary/xp/>. [Accessed 01 06 2018].
- [3] G. Kim, J. Humble, P. Debois and J. Willis, *The DevOps Handbook - How to Create World-Class Agility, Reliability, & Security in Technology Organizations*, Portland, United States of America: IT Revolution Press, 2016.
- [4] Agile Alliance, "Information Radiators," Agile Alliance, [Online]. Available: <https://www.agilealliance.org/glossary/information-radiators>. [Accessed 04 06 2018].
- [5] M. Fowler, "DevOpsCulture," ThoughtWorks, 09 07 2015. [Online]. Available: <https://martinfowler.com/bliki/DevOpsCulture.html>. [Accessed 22 02 2018].
- [6] The Apache Software Foundation, "Apache Maven Project," The Apache Software Foundation, 03 02 2018. [Online]. Available: <https://maven.apache.org/what-is-maven.html>. [Accessed 03 02 2018].
- [7] S. Basu, "evatotuts+," Envato Tuts+, 18 09 2013. [Online]. Available: <https://code.tutsplus.com/tutorials/travis-ci-what-why-how--net-34771>. [Accessed 05 03 2018].
- [8] Scaled Agile, Inc., "Value Streams," SCALED AGILE, INC, 02 11 2017. [Online]. Available: <https://www.scaledagileframework.com/value-streams/>. [Accessed 06 06 2018].

- [9] TechBeacon, "Streamline security with DevOps and continuous delivery," TechBeacon, [Online]. Available: <https://techbeacon.com/automate-security-for-continuous-delivery>. [Accessed 10 06 2018].
- [10] J. St-Cyr, "Continuous Improvement for sitecore DevOps," Agile and ALM, 14 10 2017. [Online]. Available: <https://theagilecoder.wordpress.com/2017/10/14/continuous-improvement-for-sitecore-devops/>. [Accessed 15 06 2018].