Benefits of continuous delivery for Sigma IT Consulting
Abstract

Manual deploys and testing of code can be both time-consuming and error-prone. Many repetitive manual steps can lead to critical tests or necessary configuration being forgotten or being skipped due time-constraints resulting in software that doesn’t work as intended when deployed to production.

The purpose of this report is to examine whether continuous delivery (CD) could lead to any positive effects and if there are any obstacles for CD in an Episerver project at Sigma ITC. The study was done by implementing a CD pipeline for a project similar to a real project at Sigma then letting the developers work with it and interviewing them about their current workflow, their attitude towards the different steps involved and if they saw any problems with CD for their project. Even if the developers, in general, where positive to CD they had some reservations about it in their current projects. The main obstacle the developers saw where the time/cost which would affect the customer and also some uncertainty around the complexity in testing Episerver.

The results show that there could be positive effects of CD even if the project type is not optimal for reaping all the CD benefits, it all depends on people involved seeing a value in testing and the questions around testing in Episerver are straightened out.

**Keywords**: Continuous delivery, continuous integration, DevOps, testing, automated testing, UI-testing.
1 Introduction

1.1 Background

1.2 Related work

1.3 Problem formulation

1.4 Motivation

1.5 Objectives

1.6 Scope/Limitation

1.7 Target group

1.8 Outline

2 Method

2.1 Data

2.2 Tools

2.3 Selection for interviews

2.4 Reliability and Validity

2.5 Ethical Considerations

3 Implementation

3.1 Infrastructure

3.2 Continuous Integration

3.2.1 Step 1 - Running Unit tests

3.2.2 Step 2 - Deploy to QA & UI tests

3.2.3 Step 3 - Deploy to staging & Manual tests

3.2.4 Step 4 - Deploy to production, smoketests & rollback.

3.3 UI testing framework

4 Results

4.2 Value of continuous deployment

5 Analysis & Discussion

5.1 Testing

5.1.1 Testing today

5.1.2 Value of testing

5.1.3 Implementing testing

5.2 Value of CD
1 Introduction

The introductory chapter gives a background to our subject, related work, problem formulation, motivation, objectives, scope/limitations and target group of the project.

This report goes through the obstacles and effects of the implementation of continuous delivery (CD) pipeline for a web development project at Sigma IT Consulting in Växjö. The Project is built on the .NET framework and Episerver, which is a content management server to make it easier for the customer to edit content on their site. Hopefully, this could lead to the project we are working on getting a DevOps culture around it, and ignite a focus toward DevOps in other projects at Sigma.

Sigma ITC is a consulting company, and the office in Växjö mainly works with web applications like the one this thesis revolves around. A lot of the projects have a similar structure due to several of them being built on .NET and Episerver which is the largest platform in the Växjö office. Making it easier for the developers to create and run tests, automatically running the tests in production like environments and being able automatically to deploy to production would give benefits both in higher confidence in the code and being able to do it faster and more consistently for the developers at Sigma.

1.1 Background

During the past eight years, there has been a lot of discussion about DevOps, an expression coined in 2009 by Patrick Debois when he organized a conference in Ghent called the DevOpsdays [1]. The point of the meeting was to bridge the differences between development and operations. Historically development and production have been working towards opposing goals where operations want the environment as stable as possible while the developers continuously are expected to introduce new features and changes. Those who have a connection to software development and operations have probably heard the phrase: "It works on my machine." This is one of the essential things DevOps tries to solve by sharing the responsibility for the product working where it adds value to the customer, in production between development and operations. A mindset in DevOps is that nothing is ever done until it works in production [2].

The flow from development to operations is sometimes visualized by the wall of confusion (see figure 1.1) where developers work separated from
operations. When the developer thinks they are done they throw the software over the wall to operations, even though there is no evidence that the software works other than on the developer's computer [3].

![Wall of confusion, image borrowed from Microsoft Developer Blog][1]

The primary objective of DevOps is to get development and operations to work towards the same goal, and this should be the same goal as the overall goal for the company which ideally would mean both stable operations and the ability to release new changes/features fast and reliable. It is not limited to dev and ops but also security, testing and anything else that is a part of the value stream. The idea is that instead of first developing all features, then adding tests and security and lastly get everything to run in production you instead make sure that all these parts are integrated from an early stage by a team that has competencies in all the mentioned areas or close collaboration between them [4].

An essential concept of DevOps is visualizing and speed up of the flow from development to operations (for example with the help of a Kanban board). This is where work going from left to right can be seen, and problems or bottlenecks can be found. Another important concept is the feedback that goes from right to left where one as fast as possible want feedback if anything
does not work as expected. This is done by running unit tests, integration
tests, UI tests, monitoring and so on. This should be done continuously
during the development [5].

Different types of tests are run in different parts of the pipeline. The
first tests to be run are unit tests which test the smallest part of the code in
isolation. Dependencies the code have to other parts like methods/functions,
databases or API’s should be mocked to make sure that a failing test is due to
the tested code and not a failure in something the code is depending on. Agile
testing can be visualized by the test pyramid where the unit tests make out the
foundation of the pyramid (see figure 1.2). The idea is that the unit tests are
the first tests to run since they should be fast and can give instant feedback
and if they are not passing, there is no reason to continue executing the
test-suite at all.

At the other end of the test spectrum, we have the end to end (e2e)
and UI testing. Compared to unit tests where the code is tested in isolation it
is the opposite of e2e and UI testing. Instead of having small tests that test a
small piece of the code there is now fewer test that instead touch on as many
parts of the system as possible.

In between unit tests and the end to end/UI tests, there can be several
levels where the dependencies between different components, modules, and
databases of the system gradually increase as one goes up in the pyramid.

Figure: 1.2: Software Testing Pyramid, an image inspired
by James Willets Blogg [6].
The relationship between Continuous integration (CI)/Continuous delivery (CD) and testing is that the CI server is responsible for building and testing the code [7], and as it has finished one level of tests from the pyramid it goes up the next level. Also as we get further up the pyramid the tests will span over larger parts and incorporate more dependencies of the system, testing the integration between different parts of the application. The higher we get in the pyramid the more important it is that the environment should be as similar to the production environment as possible. Continuous Delivery means that at every commit should result in that the code is tested extensively enough regarding both the quality of the tests and that the tests are run in appropriate environments so that the code potentially could be pushed straight to production [8].

Configuration management (CM) tool is used to set up the different environments needed for the project automatically. An important part is being able to build all the environments from one list of dependencies which makes the various environments as identical as possible regarding their dependencies. A CM tool can also be used in the deployment step where it would be responsible for moving an artifact to the server and make sure that any configuration is done [9].

1.2 Related work
Chen, Lianping have made several studies where he studies the effects and problems of continuous delivery, how to overcome the obstacles and how to architect towards continuous delivery [12], [13], [14]. Most of the problems found by Chen seems to stem from the companies not following what already is considered best practice in books like The DevOps Handbook [5], Continuous Delivery [8] and Infrastructure as Code [9]. Where the companies seem to have chosen to implement parts of a DevOps culture or continuous delivery but even though not all parts where implemented they expected all the benefits. Chen's work is also based on studying the implementation made by others, not doing the implementation. For the most part, though Chen has found that continuous delivery has had a positive effect on the companies in his studies like faster time to market, higher quality and happier customers. Several studies are pointing to the positive impact where the authors have something to gain on DevOps/CD, like sales of books or software. The following articles DevOps: Profiles in ITSM Performance and Contributing Factors [10] and DORA Platform: DevOps Assessment and Benchmarking
1.3 Problem formulation

Many companies today begin to more and more transition into a DevOps culture and the use of continuous integration (CI) and continuous delivery. This is done both to build confidence in the code, but also to be able to deliver value to the customer in less time, taking the idea of agile development all the way into production. Currently, Sigma does not use a continuous integration tool more than as a build step for the creation of an artifact which then can be manually deployed.

In this project, we will investigate how Sigma could benefit from implementing continuous delivery in their development process and if there are any obstacles in doing so. We aim to do this by evaluating and implementing test frameworks and verify that the tests run with a suitable CI tool. The next step is to automate the steps to deploy the software in different environments and run proper tests in QA/staging to build confidence in the release candidate before going into production.

1.4 Motivation

The software development industry is continuously evolving, which means that companies must be in the forefront to be competitive. The motivation for implementing continuous delivery for Sigma is primarily to bring faster value to the customer, but at the same time to create more confidence in the product. According to Chan who has written several studies focused on continuous delivery/deployment, there can be a significant improvement in time to market and in the quality of the product by using continuous delivery [12], [13], [14].

In the long run, the hope is that it can be used as a tool for moving towards a DevOps culture. Kim et al. estimate that the total value of the waste from not following DevOps principles is approximately the same as France
yearly GDP [5]. Another selling point of DevOps is that instead of trying retrofit testing and security it should be built in from the start which results in both more efficient development and better quality software [15].

1.5 Objectives

<table>
<thead>
<tr>
<th>O1</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Find suitable testing frameworks for the project</td>
</tr>
<tr>
<td>1.2</td>
<td>Write tests for the project</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O2</th>
<th>Continuous Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Find suitable CI tool</td>
</tr>
<tr>
<td>2.2</td>
<td>Use the CI tool to run the tests continuously</td>
</tr>
<tr>
<td>2.3</td>
<td>Implement instant feedback on failures</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O3</th>
<th>Continuous Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Automate deployment</td>
</tr>
<tr>
<td>3.2</td>
<td>Implement automatic rollback on the failing smoke test</td>
</tr>
</tbody>
</table>

This report will address three overall objectives, which have been divided into sub-objectives.
The first objective expects to find a user interface test framework that is suited for the development project and the pipeline.

For objective one, the expectation is to have tests to run in the continuous delivery pipeline, since there is no way of evaluating the pipeline without tests to build confidence in the code.

The second objective is to find the right tool to implement continuous integration. The expectations are to find a tool that is suited for the pipeline and fits into the toolchain at Sigma. The tests should run every commit/pull request and give the development team instant feedback in case of failures through a suitable communications channel.

For the last objective, the expectation is to automate deployment to pre-production and production to get the code to move between environments as fast as possible with high confidence due to the automated testing from previous objectives. The pipeline should be able to bring code to production fast and reliable where it can bring value to the customer. The expectation is also to automatically be able to roll back to an earlier version of the application to minimize downtime in production in case of failing smoke tests.

1.6 Scope/Limitation
Since DevOps is a broad subject which could affect many parts of a company the scope is limited to some prerequisites for DevOps like testing, continuous integration, continuous delivery, and continuous feedback. For both UI testing and CI-servers where our goal is to find suitable frameworks or tools, we will need to limit the number of products we will investigate. The thesis limited to one Episerver projects and the main focus will lay on a specific development project at Sigma.

1.7 Target group
The target group for this project is primarily developers and system administrators, but also customers and companies that would benefit or have an interest in continuous delivery or DevOps. Developers will benefit from this pipeline through their everyday work to build confidence in the code. The product owner will get higher quality software faster and cheaper compared to traditional development.
In the next chapter, the methods for implementing the various objectives will be presented. The Implementation chapter will go into more detail on how we solved the different objectives, here is a description of the pipeline. In the next chapter, the results of the project will be presented. Analysis and Discussion chapter will analyze and discuss the results as well as conclusions from data collected in the project. In the final chapter, a summary of the results will be presented, and recommendations for future work will also be made.
2 Method

This report is based on a case study to analyze the effect of implementing continuous delivery on Sigma IT Consulting. Through observation and interviews, Sigma's current workflow has been analyzed to find where improvements could be made, observations on the current way of working and the number of manual steps parts of the workflow have.

Sigma is not responsible for any post-development servers, and it is part of the customer. Since we had no access to the production servers for this project, we had to simulate this by setting up our own. This meant that we could not even use the live project that the developers worked with. Instead, we used a template from Episerver as it is very similar to what the project developers are working with today and that does not change the way the pipeline works.

2.1 Data

Qualitative data has been used to capture the feel of the developers about the confidence in the code, which is difficult to get by a quantitative study since there are a limited number of developers to work with; therefore a qualitative method is required.

This was done through interviews after the continuous delivery pipeline was implemented, and after the developers have tested the pipeline. There were no questions for the interviews, but an open discussion was held on the basis of a checklist that was followed to get the answers we were looking for. This to capture as much of the feeling as possible without leading the person to an answer.

All developers got to work towards the pipeline, so everyone had the same level of knowledge. They were given a brief introduction to be able to use the pipeline, but short enough to provide no influence about their perception of it. The introduction highlighted how the workflow looked and how they would work with it, without putting any weight on what was happening in the different stages.

By trying to ask open questions as possible and let the developer explain what's positive and negative, we try to get the developer to respond to his own experiences. The questions were objectively stated not to influence the developer.

To direct the developers in the same direction and to measure and
analyze the data from the interviews, we used support words see Appendix 1. This meant that we could find a common denominator among the developers, but it also helped us to compile the results.

Because it is difficult to memorize interviews and to keep notes, all interviews were recorded. After the interviews were done, we listened to the interviews and brought down everything on paper, and sorted them under the correct heading in the questionnaire.

2.2 Tools
The choice in CI server was Jenkins, initially, Appveyor was used, but it was found to be a hindrance later in the pipeline where the flow and feedback needed to be controlled in a way that could not be done with AppVeyor. Several of the CI servers were excluded due to not being able to work with .NET Framework projects since it needs to run in a Windows environment. Other tools used was MSBuild for compiling code and MSDeploy for deploying the code.

As a UI test framework, Selenium was chosen, mainly because a lot of the alternatives, in the end, was built on Selenium anyway but took some of the integration with other tools and configuration away. End test was fascinating as well due ease of setup and use but could unfortunately not be used since it does not support Internet Explorer 10 which is a target browser for the project at Sigma.

2.3 Selection for interviews
The sample selection for this report was made by developers who work with the product on a daily basis. This was done to gain an idea of the perceived sense and confidence created for the code by making use of continuous delivery. The interviewees all have a development service within the company. To be able to compare the different interviews for a more in-depth analysis of these, three people were selected to answer the interrogation questions.

2.4 Reliability and Validity
Since the project will use interviews with people in the business, there are some reliability issues. This may be that the person interviewed has had a bad day and therefore respond more negatively to the questions asked than it would be done on another day. There are also other soft parameters that could
challenge the reliability, for example, the relationship between interviewers and respondents, where the respondent can answer the questions positively or negatively based on what he thinks about the interviewer. It is also crucial for the respondent to take the questions seriously to get relevant data.

The time the developers have to test the pipeline and work with the pipeline also plays a role in reliability. It takes time for developers to get full confidence in the continuous delivery pipeline, which for this project is adversely affected, as developers only are able to test the pipeline for a shorter period. This makes specific data misleading because they have not yet built up to full confidence. Bias can also be a reason why reliability can decrease, developers may feel fear of testing something new and because of it having a negative attitude towards the pipeline. Certain bias may arise from the fact that they think their current workflow is good enough so that they become blind to any improvements that would be brought with the pipeline. There is also a risk that there is bias in the opposite direction, and the developer has created a positive opinion.

Something that may also be a problem for reliability is a misinterpretation in communication, where the answer means something else than the interviewer has perceived.

The report is based only on a project on Sigma, which makes this report specific to the Sigma project itself. For validation, therefore, this continuous delivery pipeline is not based on a genetic perspective but specific to Sigma.

2.5 Ethical Considerations

All collective information about people interviewed has for ethical reasons been deleted to protect personal information.

Sensitive information that may damage Sigma as a company or information from Sigma's orders has also been deleted from the report in order not to be published.
3 Implementation

This chapter presents the implementation of the continuous delivery pipeline which starts with describing the whole environment needed, like the infrastructure, web-servers and the database. Then the integration between the chosen UI-test framework, CI-server and test environments and finally a way of controlling the flow to the different stages depending on the results of the tests run on/from the CI-server. The various stages have been split up into chapters below. Figure 3.1 shows the desired flow from a commit or pull request to the application running in production.

![Figure 3.1 Showing the desired flow of the CI/CD pipeline.](image)

3.1 Infrastructure

It was not feasible for us to do implementation directly on the customer's servers in this project due to it being used for testing by the customer and being hosted on-premise by the customer. Instead, OpenStack was used to simulate an environment like the one used by the customer. This gave a lot more control over the environment and a lot less risk of causing problems in real environments, which in hindsight was a good choice due to the many issues that arose during the implementation that surely would not have been appreciated in the customer's test and production environments. Automated provisioning of a network, LAN and router, Ansible control machine, database and web-servers was made to OpenStack using Ansible.

The different servers can be seen in figure 3.2, and an overview of the most
critical software on the various servers can be seen in table 3.1.

![Overview of the infrastructure on Openstack](image)

**Figure 3.2: Overview of the infrastructure on Openstack**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous Integration server</td>
<td>Windows Server 2012</td>
<td>Jenkins, Katalon Studios, MSBuild, MSDeploy</td>
</tr>
<tr>
<td>QA</td>
<td>Windows Server 2012</td>
<td>IIS, .NET</td>
</tr>
<tr>
<td>Staging</td>
<td>Windows Server 2012</td>
<td>IIS, .NET</td>
</tr>
<tr>
<td>Production</td>
<td>Windows Server 2012</td>
<td>IIS, .NET</td>
</tr>
<tr>
<td>Database</td>
<td>Windows Server 2012</td>
<td>MSSQL</td>
</tr>
</tbody>
</table>

Table 3.1 List of main software on the different servers

3.2 Continuous Integration

After the infrastructure and environments were set up a CI tool was chosen to automate the tests and give the user control over the paths over the possible options for failing/passing tests. In the image above Jenkins are
already added, but in reality, this came later in the process. A couple of options for CI servers where tested but Jenkins was chosen for several reasons, the main reason was that it is mature and have plugins for many different scenarios and other tools. We also had previous experience with Jenkins and knew that it had the necessary functionality and it is open source so there is no cost or limitations with it.

After implementation of the CI server Github was connected to Jenkins, and on a commit, a Jenkins gets a notification that something has changed in the repo. Jenkins then pull the repo do the necessary steps to build the software like downloading any dependencies if there been any changes to them. It then runs the unit tests, and in case they pass it deploys it to QA.

3.2.1 Step 1 - Running Unit tests
The first step after a commit or pull-request on the repo is that Jenkins receives a webhook from Github. The repo gets pulled by the Jenkins server, and any dependencies are downloaded and the code compiled with MSBuild and packaged as an artifact which is stored on the Jenkins server. The unit tests are then executed using MSTest and depending on the outcome of the tests the pipeline will either notify the developers about failing tests on Slack with a link to the test report or automatically deploy to QA which is done in step 2.

3.2.2 Step 2 - Deploy to QA & UI tests
In this stage the code is deployed to the QA server with MSDeploy where it replaces last deployed version, some files that are not meant to be replaced by a deploy like licensing files and BLOBS are kept between deploys. Everything needed for MSDeploy to work is already in place after the setup of the infrastructure. Once the application has been deployed Jenkins initiates the UI tests by running a command executing the tests in Katalon Studios. This is the most time-consuming part of the tests and since the tests need to be performed in the same manner as one would do them manually, so even if they are done automatically different browsers need to be opened and the different tests executed. An example of the script for a UI test can be found below.

```csharp
WebUI.click(findTestObject('Page_CuraHomepage/btn_MakeAppointment'))
```
figure 3.3 Script for testing login on a webpage in Katalon Studios.

Once the tests are finished, Jenkins will yet again post a message to the appointed Slack channel. On failing tests, a notification with a link to the test report will be sent and on success a message containing a link to a page where the developer can choose to continue the flow and deploy to staging (step 3). The reason it does not automatically deploy to staging is that there might be ongoing manual tests conducted in this environment either by another developer or by the customers. The feedback sent to the developers in Slack for a failing unit or UI test can be seen in figure 3.4. The developer can then follow the link to see a test report on the Jenkins server (see figure 3.5). If all tests pass a message regarding manual approval will be sent to the developers (see figure 3.7).

FAILURE Template 186 (Open) --- Ops something went wring!

figure 3.4 Shows the feedback sent to the project developer on failed unit or UI tests.
3.2.3 Step 3 - Deploy to staging & Manual tests

Once the developer has chosen to deploy to staging after successful tests in step 2 the application will be deployed using MSDeploy to staging, in the same manner as it was deployed to QA in stage 2. No automated tests are run in this environment at all and will it will await a manual approval for being deployed to production.

3.2.4 Step 4 - Deploy to production, smoketests & rollback.

In this step, the application will be deployed to production and where a smoke test will be run. The smoke test can be something as simple as a curl getting the start page or a more elaborate e2e test where several parts of the system are tested using a UI test. In case of a failing smoke test, Jenkins will redeploy the last successful build so that there always is something working in production. The feedback sent to the developers in Slack for a failing smoke test can be seen in figure 3.6.
3.3 UI testing framework

A couple of options for UI testing frameworks where evaluated. Katalon Studios chosen due to was able to test the different browser versions needed where Internet Explorer 10 set the target. Katalon Studios are built on top of Selenium but was easier to work with which made it easier for people with different backgrounds to work with the tests due being able to create tests both by a recorder which anybody could do or by writing code. This makes it possible for people not involved in writing the code to write test and run the test suite, it could add value that they also could add or edit tests even if they do not know the project language. Katalon Studios also made it easier to separate the testing from the .NET solution which also made it more accessible to people not running the entire development environment on their local machine and instead uses it on post dev environment. A test report was generated with Katalon Studios and Jenkins would act accordingly depending on the results from the tests and either deploy to production or notify the developers in case of failed tests. Currently, the feedback is sent by Slack, but this could be changed to whatever type of communication needed.
4 Results

After letting the developers work with the pipeline and UI testing tools interviews were performed regarding the developers' experiences from working with the pipeline, their current workflow, and their opinion on how well continuous delivery would work in their project. The interviews were done with three developers.

Developer 1 did at the moment not work in an Episerver project but is usually working on the main project studied. Developer 2 currently work on the main project and Developer 3 works on a similar project where the underlying structure and infrastructure are resembling the main project.

All three developers have been deploying code to the two environments, user acceptance test (UAT) and production in their respective project. In both projects, the deploys are done to UAT at the end of each sprint where the customer tests the code. After approval from the customer, the code gets deployed to the production server. For both projects, several developers are doing deploys. For at least for project 1, the deployment is documented well enough that any developer at Sigma working with similar projects should be able to make an deploy. For project two the documentation is not fully updated since the project went from development to the maintenance phase.

According to the developers the sprint's length varies from one to four weeks depending on the stage of the project. During development, the sprint times are three to four weeks, and for development, during the maintenance phase, they are generally between one to two weeks. On occasion, there is a need for a hotfix which warrants an extra deploy, but bugs can usually wait for the next scheduled deploy.

A deploy takes between 15 minutes and an hour depending on if there are any manual changes to make to the servers. Developer 1 and 2 who works at the same projects both estimate 30 minutes to 1 hour. And developer 3 estimates 15 minutes to an hour.

None of the developers feel worried or nervous about deploying, but developer 3 thinks that the larger a deploy is, the more things can go wrong, and he tries not to wait too long between deployments. Two of the developers feel that the deployments are more of a distraction.

- “A developer should develop as much as possible and not do other
things, which happens to often, sometimes working with requirements and sometimes deploying. If it was possible to eliminate those steps one could be more effective”

Developer 1

- “It messes with the rhythm of the job when you have to deploy manually.”

Developer 2

None of the projects the developers currently work on have any unit or UI tests. The main reason according to the developers seems to be the cost/time involved, Another reason is uncertainty around testing Episerver or a CMS and how time consuming it would be.

- “In the end it is about money, since we charge the customer, in the end it is all about the money. Many developers would probably want to write and have tests, the tests are for the developers after all, the customers would probably not want to pay for it”

Developer 1

- “It is hard as a consultant when you know that all the time one spend will cost the customer a lot of money, the project I am working on was very messy in the beginning since customer did not have any requirements. We did not have the time create structured unit tests.

When working with a large CMS it is more focus on integration tests and a lot of mocking i necessary, it is doable but very complicated.”

Developer 2

- It is probably partly due to Episerver which makes it harder to setup testing, it is a lot more configuration. A part is that we have not felt it would be worth it, there is not the time. We could estimate a higher cost to make time, which would have had to done a the start of the project and the customer would have to agree to it. The customer does not care that much about testing because they do not have enough knowledge about testing. They would probably appreciate it if they knew what it could give.
4.2 Value of continuous deployment
When it comes to value added by CD, the three developers have a somewhat different view. All the developers see problems around motivating the extra cost of extensive testing to the customers. Developer 2 and 3 think that the value of testing will come after development has finished and the project is in the maintenance stage where new developers can make changes to the code with higher confidence that their code has not affected something it should not. All three developers agree that having automated tests would give higher confidence in the code. There is, however, some uncertainty among them around the complexity of testing code that is built on Episerver due to it being a CMS which adds some complexity to the project and where parts of the layouts are generated from the database rather than the code.

- “UI-testing could be harder with a CMS, and there is not one layout that is correct, maybe it is possible to solve by using a database for testing. Unit testing could also be problematic, a lot of the backend code comes from the framework[episerver], which makes it more complex.”
  
  Developer 1

- “I do not think it is viable to UI test in a project like this, it would be possible to write the tests, but it would be cumbersome to maintain them. It would not be cost-effective”
  
  Developer 2

- “A problem with UI testing could be the pages are generated dynamically, perhaps testing the “frames” rather than the content is possible. However, I believe there would be great benefits to testing.
- I think the benefits could be seen in the long run. The developers in a project can change, and I think that having tests makes it harder to make changes that break the project. I think it adds value in the long run.”

  Developer 3

There have seldom been any problems in deploying, and all three developers have a hard time recollecting any problem with manual deploys that could have been solved with automated deploy. They all agree that continuous
delivery could prevent some of the errors that arise from the human factor. The developers describe how a human error can occur. However, generally, the developers do not see any significant risks with manual deploys.

- “One example is that we deploy remote, where it may be easy to turn off the remote computer, and it has happened before.”
  
  Developer 1

- “With the way we do it [deploying], there are extremely few things that can go wrong, and the only thing is that in some sprints we have to create folders with specific permissions or make changes to machine.config which gives the server a specific setup. These are changes done directly to the code which could go wrong. The reason everything has been working well for us is that we are meticulous with documenting any changes, but there have been some problems, but it is very unusual.”
  
  Developer 2

- “It is has happened that the wrong version has been deployed, that is the biggest problem I have come across, so there is a risk of a human error in the deploy”
  
  Developer 3

Developer 3 has also had some problems with code from the previous deploy causing problems and continues that this is probably something that could be solved by automated steps that removed the old deployment and replaced it with the new code. Developer 3 also thinks that continuous delivery provides better documentation for deployment.

- “I think that automated deploys provide better documentation than if you would do it manually because the script will be the documentation.”
  
  Developer 3

Developer 3 saw value in being able to do fast deploy to an environment where the customer could try out a feature, instead of sending print screens and questions to the customer. Especially for projects with unclear requirements where the requirements are worked out in collaboration with the customer during development rather than before the development
starts. None of the developers thought that the frequency of deploys would go up by implementing a CD pipeline (apart from deploying to where UI tests are run) since the current workflow does not support it. Due to there being a time for UAT where the customer tests the project and then deploy to production is planned and then executed when the customer has approved the UAT.

All three developers saw a problem with having three post development environments due to licensing costs from Episerver for each environment. No prices were discussed, but according to the developers they are high enough that it is unlikely that the customer would want to spend the extra money.

When asked if there are any risks of loss of knowledge with automating none of the developers thought that it was a risk. Developer 1 said the following:

- “There is a risk that the one responsible leaves so that the competency disappears. However, the pipeline does not feel any riskier than anything else”

  Developer 1
5 Analysis & Discussion

The results show that continuous delivery on Sigma could potentially provide improvements in their flow and the developers have a high interest in it. All the developers agree that removing the manual steps and replacing them with a pipeline where the code is automatically tested and after approval automatically deployed to production would give higher confidence in the code, save time and minimize context switching. However, continuous delivery might not provide the desired increase in value due to the developers being doubtful in if the customers are willing to spend more money on testing and that they today use Episerver which the developers think will add much complexity to testing.

5.1 Testing

One advantage of continuous delivery should according to the theory be that the developers gain confidence in the code, as it is always tested and changes can be made with less risk for unintended changes to other parts of the code and the developers get instant feedback as something goes wrong [5],[8]. As the result shows, the developers describe that they believe it would give confidence in continuously testing the code. However, all three developers have expressed doubts about the viability of testing their projects, all think it would be technically possible, but due to complexities around Episerver and it being a CMS and time and cost constraints, they are doubtful if it would be possible.

5.1.1 Testing today

When the developers were asked about the state of testing in their projects, it turned out that they did not have any tests today. The main reason for this is that it is not something that the customer requests or value but it is something that they would have to pay for, and even if it has a value, in the long run, it would be hard to motivate initial cost to the customer. The fact that the projects today have no testing at all, which is a cornerstone of continuous delivery makes the implementation of continuous delivery complicated due to the scope of the tests needed.

To get a well-functioning continuous delivery pipeline for a project which helps to build confidence in the code, one would have to change the mindset of mainly the customers but perhaps also the developers.
5.1.2 Value of testing

The value/cost of testing is hard to discuss since all of the developers had some hesitation both around unit tests and UI tests with Episerver. Since all the interviewed developers have somewhat limited experience in working with tests in Episerver both regarding unit tests and UI tests, it is hard to know if it is the uncertainty and initial learning curve of testing or if it is that much more complex to test Episerver than a regular .NET project. Since Episerver is proprietary software and they work with IT consultants as partners, it is hard to find any open repository on Git to review how other projects have been tested.

According to the theory on CD, the testing is an essential part, from the result the testing is what the developers see as the most significant problem [5],[8]. The reason for this is in part that there are no tests today, which would take a considerable amount of work to implement on projects of this size. This is partly due to the complexities of using a framework like Episerver and partly due to the developers not being used to testing. There is also a problem with getting the customer to understand the benefit of testing and just not seeing it as a cost, but more like an investment. The developers do not think the customer will be interested in something that will cost them more money, make the development longer but potentially could make it easier for the developers in the long run.

5.1.3 Implementing testing

The three developers all saw problems with implementing UI tests since the layouts are dynamically generated from the database and that they all have local databases which would make one developer’s UI tests fail on a post-development environment or when to run on another developers machine. A solution discussed by the developers could be writing UI tests that were focused on the code that builds the skeleton around the content that is provided by the customer. This would however not solve UI testing between different browser which one of the developers saw as a welcome feature.

Another solution for keeping the database synced between the different development environments and the environment where the UI tests runs could be to have all the local development environments to connect to the same database so that all the content any developer adds with the
Episerver interface would be added to the same database. The same database would then be used on the QA server when the UI tests are running. It is time-consuming to go from no testing at all too extensively testing the project, and since none of the developers are used to testing with Episerver or testing at all the initial cost of testing the entire project would be high.

We agree with the developers that it would be easier to implement continuous delivery for a new project where tests can be created from the start, and the developers get a chance to perform testing from the beginning where the code can be written in a testable way instead of trying to retrofit a whole test-suite.

It is also essential that the customer sees the value in testing and continuous delivery but at the same time has an indulgence with the initial cost.

As the developers describe, continuous delivery and testing are not something that we notice from the start of a project. The project is continually changing, and the staff is replaced with time, the project would greatly benefit from continuous delivery during the maintenance phase when the system has often reached a high level of complexity.

5.2 Value of CD

All the developers saw value in continuous delivery in that it gives confidence that extensive testing was continuously performed and the code gets automatically deployed to the different environments. All the developers saw even higher value in the maintenance phase where the developers go over to work on new projects while another developer takes over the project.

None of the developers experience that the deploys have been especially troublesome, but in some instances, there have been problems that could have been avoided by minimizing manual steps. One of the developers mentions that they do manual configuration on each server when needed and document it thoroughly, which is a typical antipattern according to the literature surrounding DevOps and CD [5],[8],[9]. However, on the other hand, the literature often talk about more complex systems that are operated on multiple servers, with integrations between several services and legacy systems. The projects studied in this report does in contrary to a lot of the examples in the literature not have more than two servers and a database server in total for both test and production environments.
An advantage that one developer mentioned was that the documentation sometimes suffers. Here, instead of having to document all actions, a self-documented script should serve as documentation as well as code for the pipeline.

When it comes to the automation of deploys all the developers are confident to the potential time savings that could be made, two of the developers mostly saw the deploys like a distraction that took focus from other tasks and felt that it would be valuable if it could be done automatically. We have not made a time study on the deploys, but there is no question about that time could be saved by using automated deploy. The developers estimate that a manual deploy takes 15 minutes to one hour per environment. A best-case scenario where no changes are done to the server, database or build dependencies could with automation be done in less than one minute, not counting running unit or UI tests (which currently are not run in the manual deploy either).

5.3 Environments

In the workflow we presented to the developers we used three environments, all the developers said that the customer probably was not willing to spend the license costs for an extra Episerver license. An additional server would of course also result in other costs like Windows Server licensing and the operating costs of the server. A solution to this problem could be that there is only a short window for UAT where all automated deploys to the test server would be halted. It would not be optimal since there would be a time where not all commits or pull requests would be UI tested, but it would be better than not using UI tests at all.

5.4 Implementation

Instead of using the real website used in the project, we used a template. This meant that the lead time for the pipeline where the developers tested the pipeline was much faster than they would have been on the live website. The results may have been affected because of this, and the developers may have experienced the continuous delivery as very fast and agile and made a comparison with their real projects.

It also made it very difficult to compare how much faster the continuous delivery pipeline is compared to manual work. Because it does not provide accurate data to match lead times between the two since they do
not contain the same amount of data. A possible solution too in this could be to use the real project to ensure that the developers get the correct view of the pipeline as possible.

5.5 Related studies

Most examples of successful DevOps/CD implementations initially had very complex deploys that even in best case scenarios have takes several hours and in worst case several days, involving several people from multiple departments and several different integrations to other systems. In the projects we have been looking at deploys have taken 15-60 minutes and only one developer. It is hard to compare the continuous development value on a company like Netflix, Facebook or Etsy to a Swedish municipality website.

There could, of course, be value in adding CD to a project like this but one can not expect the same value as it would give in more complex application since the deploy is not that complex, problem-prone or time consuming, to begin with. The fact that it is a web application run from one server also makes viable, and according to the developers it has worked out well thus far. At least as long as it only is done every one to four weeks. In the literature, there is talk about everything from daily deploys to hundreds of deploys per day. It is easy to see that when there are multiple deploys each day, it is time-consuming and laborious to test everything after each deploys manually.
6 Conclusion

This report addresses the benefits and obstacles of continuous delivery on an Episerver project at Sigma IT consulting. After interviewing three developers who work with similar projects, all of them are positive to CD after working with the pipeline and feel that continuous delivery would give greater confidence in continuously running the code through a pipeline, and testing it. However, all the developers have doubts on whether it is viable to test the application.

As the pipeline requires good tests to create confidence in the code, it is essential that the application can be tested. The developers are in general positive to testing. However, they do show some skepticism about writing unit and UI tests in Episerver projects and they are unsure if the complexities of writing tests in Episerver will make it too costly. The developers currently have little experience in testing Episerver project, and UI testing, which can lead to the integration for continuous delivery is somewhat steeper.

We believe that there is an initial hurdle for all people involved in the project implementing CD, the type of projects is not necessarily ideal to maximize the value added by CD but we still think it would have its merits. We believe a good start would be starting up a new project with a suitable CD pipeline implemented and try to write tests throughout the project.

6.1 Future work

The project focuses on how developers feel about continuous delivery, which means that the focus has not been on optimizing the pipeline. We have only used a temple to demonstrate how continuous delivery could work in practice. This means that we did not write comprehensive tests that would be needed in a real project.

Further research could be done on how Sigma's live project could be run in the pipeline. This would involve more extensive testing to see how the lead times in the pipeline would stand against a manual deployment. Further research can also be done how much benefit continuous delivery would have for the company, as this work only addresses developer perspectives. To increase the validity of the interviews, further research could also tell whether data from more interviews could be distinguished from the results of this report.
References


Appendix 1

Checkboxes used for the open interview to ensure that no important topic were missed during the interview.

- **CD general**
  - feeling?
  - want to work with?

- **Project specific**
  - lead dev
  - who is deploying
  - deploy documentation
  - environments
  - testenv lockdown

- **Current workflow**
  - workflow today
  - deploy times
  - deploy frequency
  - feeling
  - testing
    - automated?
    - if not why?
    - type of application?
  - Other automation?
  - Problems deploying
    - human/automatable?

- **Effects of CD**
  - testing
    - feeling?
    - confidence?
  - project related problems(episerver/CMS)
    - your project
    - pros/cons
    - investment vs value
    - more deploys?
      - pro/cons

- **Pipeline test**
  - effects