An Investigation of Automating Software Deployment Using Continuous Delivery Tools
- A cost-benefit study in the case of multiple system instances

En undersökning av automatisering av mjukvaruleverans med hjälp av verktyg för Continuous Delivery - En kostnad-nytta-studie i fallet med multipla systeminstanser

Yousif Touma

Supervisor : Anders Fröberg
Examiner : Erik Berglund
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Yousif Touma
Abstract

Manual deployments of software is a tedious, repetitive and non-scaling method of deploying software. Continuous Delivery is a practice that enables automated deployment of software in a rapid fashion at the click of a button. When deciding whether to start using a new practice, software companies need to make an assessment from a cost-benefit perspective. This thesis compares automated deployments through Continuous Delivery with manual deployments from a cost perspective. The comparison is done at a small software company where two tools for Continuous Delivery are chosen based on requirements imposed by the company. The tools, Octopus Deploy and Azure DevOps, are cost efficient to different degrees. Octopus is cost efficient if several deployments per week are necessary, particularly if many deployment targets are involved. Azure DevOps is quickly cost efficient in most cases due to its pricing scheme, only needing roughly one deployment per week for few deployment targets, and a couple of deployments per year for many deployment targets. The initial cost of having a paid employee set up the tool needs to be paid off, but is easily done within a year using weekly deployments with a small number of deployment targets.
Acknowledgments

A big thank you to Anders Fröberg, my supervisor for this thesis. Our biweekly meetings greatly helped me keep up the pace, stay on the right track, and stay confident during the work. Always nice to share a coffee and talk some floorball as well. A thank you to my examiner, Erik Berglund, as well.

A special thank you to my colleagues at Agricam. It has been a pleasure doing my thesis with you. Thank you to Ellinor, my friend, mentor, and manager. A big thank you to Martin, who helped me structure my ideas, provided feedback, and had great patience with me during the project. A thanks to Johan, Erika, Zen, and Elin who were all involved in one way or another during the work on this thesis, be it with discussions about writing or any other interesting topic, it was always appreciated.

An extra thank you to my dear friends and class mates: George Yildiz, Fredrik Wallström, Sebastian Lindmark, Andreas Järvelä, and Daniel Proos. Five years with labs, exams, lunch, and hanging out have been memorable. This past semester, writing our theses, I have much appreciated eating lunch and discussing our projects. It has been an honor to be your class mate and will continue to be an honor to be your friend. A special thanks to George Yildiz and Fredrik Wallström for providing feedback on the thesis.

Finally, the biggest of thanks to my mother Iman and my father Joseph. You are my biggest supporters and without you, none of this would have been possible. I am forever grateful.
## Contents

**Abstract** iii  
**Acknowledgments** iv  
**Contents** v  
**List of Figures** vii  
**List of Tables** viii  

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Glossary</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>1.1</td>
<td>Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Aim</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Research Questions</td>
<td>3</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Research Objectives</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Delimitations</td>
<td>4</td>
</tr>
<tr>
<td>1.5</td>
<td>Expected Results</td>
<td>4</td>
</tr>
<tr>
<td>1.6</td>
<td>Background</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Theory &amp; Related Work</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Terminology</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Continuous Integration</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Continuous Delivery</td>
<td>7</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Continuous Deployment</td>
<td>7</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Continuous Release</td>
<td>8</td>
</tr>
<tr>
<td>2.1.5</td>
<td>DevOps</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Implementing CDL - Benefits and Challenges</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Important Factors for CDL</td>
<td>9</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Benefits</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Challenges</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Cloud Migration</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Types of Migration</td>
<td>11</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Methods of Migrating Software Applications to the Cloud</td>
<td>11</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Effects of Moving an On-Premise Application to the Cloud</td>
<td>12</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Case Studies</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Method</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Pre-Study</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Implementation</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Evaluation</td>
<td>16</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Calculating the Cost of an Engineer</td>
<td>16</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Baseline Cost</td>
<td>16</td>
</tr>
</tbody>
</table>
4 Results

4.1 Pre-Study

4.1.1 Chef

4.1.2 Puppet

4.1.3 Ansible

4.1.4 SaltStack

4.1.5 Buddy

4.1.6 GoCD

4.1.7 Octopus Deploy

4.1.8 Azure DevOps

4.1.9 Choice of Tools to Evaluate

4.2 Implementation

4.2.1 Octopus Deploy

4.2.2 Azure DevOps

4.3 Evaluation

4.3.1 Baseline Cost

4.3.2 Using CDL Tools

4.3.3 Summary

5 Discussion

5.1 Results

5.1.1 Pre-Study & Implementation

5.1.2 Evaluation

5.2 Method

5.2.1 Pre-Study & Implementation

5.2.2 Evaluation

5.2.3 Replicability, Reliability, and Validity

5.2.4 Sources

5.2.5 Alternate Method - Cloud Migration

5.3 The work in a wider context

6 Conclusion

6.1 Future Work

Bibliography
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Average monthly salaries of Swedish engineers in software industry.</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Script to install Azure DevOps agent and connect to a Deployment Group.</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Command sent to NuGet to install a .NET application from a NuGet package.</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>.nuspec file to tell OctoPack how to package the .NET web application.</td>
<td>27</td>
</tr>
<tr>
<td>4.4</td>
<td>Average cost of deployment in relation to number of machines for sequential manual deployments.</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Average cost of deployment in relation to number of machines for parallel manual deployments.</td>
<td>30</td>
</tr>
<tr>
<td>4.6</td>
<td>Yearly cost of using Octopus Deploy based on number of machines to deploy to.</td>
<td>31</td>
</tr>
<tr>
<td>4.7</td>
<td>Yearly cost of using Octopus Deploy based on number of machines to deploy to (zoomed in on up to 100 machines).</td>
<td>31</td>
</tr>
<tr>
<td>4.8</td>
<td>Yearly cost of using Azure DevOps based on number of parallel jobs.</td>
<td>33</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Octopus Deploy pricing.</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>Overview of tool evaluation.</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Time spent on sequential manual deployments.</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>Cost per machine when deploying sequentially.</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Time spent on parallel manual deployments.</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>Cost per machine when deploying in parallel.</td>
<td>29</td>
</tr>
<tr>
<td>4.7</td>
<td>Time elapsed during automated deployments with Octopus Deploy.</td>
<td>30</td>
</tr>
<tr>
<td>4.8</td>
<td>Time elapsed during automated deployment with Azure DevOps.</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Number of required deployments to make up for initial setup cost using Octopus Deploy.</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Number of required deployments to make up for licensing cost using Octopus Deploy.</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>Number of required deployments to make up for initial setup cost using Azure DevOps.</td>
<td>38</td>
</tr>
<tr>
<td>5.4</td>
<td>Number of required deployments to make up for licensing cost using Azure DevOps.</td>
<td>39</td>
</tr>
</tbody>
</table>
0.1 Glossary

This section defines and describes some of the acronyms, words and terms used in this report. The purpose of the glossary is to help the reader follow the report with less disturbance. The list is alphabetized. Some terms are further elaborated upon in the report since they are more important to the study at hand.

**CDL**
Continuous Delivery. The practice of maintaining releasable versions of your software at all times, ready to be released to a production environment at any time through the press of a button or similar \[1\].

**CDP**
Continuous Deployement. The practice of continuously releasing every integrated piece of software to a production environment \[1\].

**CI**
Continuous Integration. Fowler \[2\] defines CI as the practice of integrating your software changes on a regular basis, as often as every day. The integration includes automated build and test tasks to verify successful integration.

**DevOps**
A conjoined acronym of the terms Development and Operations. Although harder to define consistently, it can be described as a set of values, principles, methods, practices and tools, used holistically to achieve quality in products and processes in a conjoined effort between engineers in development and operations \[3,4\].

**IaasC**
Infrastructure-as-a-Code. Configuration of software environments through well-defined, structured files. The concept is used to be able to version control environment definitions and keep environments reliable.

**NuGet & NuGet Packages**
NuGet is a package manager for .NET \[1\]. A NuGet package is a compressed folder containing the files necessary to use a .NET application or library.

**On-premise (solution)**
Meaning that a software system is installed, hosted and commissioned at the location where it is being used, typically at the physical premises of a customer.

**PR**
Pull Request. When a developer asks for their changes to be accepted into a SCM-repository. Merge Request is a different term with the same meaning.

**SaaS**
Software-as-a-Service. A software system that is made available to use for customers over the internet, through a web browser.

**SCM**
Software Configuration Management. The task of keeping track of and control software changes, which is usually done by a SCM-system. Examples of such systems are Git, Mercurial and Subversion.

\[1\]https://www.nuget.org/
Making changes to your deployed software will often require both testing and planning, as well as coordination between the different involved departments \[5\]. It is indeed a time consuming task to develop, assure quality of, and deploy new software and it is not desirable to add even more costly overhead to this process. One such overhead is to manually perform software updates when there are multiple customers that use different instances of the same software system. This is particularly the case when the time spent is regardless of the size of the software update. A small bug fix takes the same amount of time to deliver to your customers as the work produced by several weeks of development. This is not a sustainable solution in the long run as the customer base grows and is in dire need of a more agile and time-efficient method.

DevOps is an emerging term within software development that builds upon agile principles but expands across several new areas. DevOps can be seen as a concept that combines values, principles, methods, practices and tools concerning the gap between software development and software operations. Two of these practices are Continuous Delivery (CDL) and Continuous Deployment (CDP). CDL aims to make every integrated change in a software system a potential release candidate, ready for automatic deployment, whereas CDP aims to put each of these candidates in a production environment automatically \[3, 6\].

Besides from automating tedious and repetitive deployment and installation tasks, there are other potential benefits to gain from the more rapid development-to-release cycle. Companies can respond quicker to customer feedback to improve both their product and customer satisfaction as well as spend more time on building the right features by learning what customers want \[7\].

This thesis studies the effects of implementing CDL at the company Agricam AB. The company develops a software system that analyzes data produced by a thermal camera. The system is delivered as an on-premise solution to their customers, and is manually updated by the company’s employees through remote administration of servers.

1.1 Motivation

The benefits of implementing automation in more areas of software development have begun to emerge and are several. Companies are choosing to invest more and more money in their automation infrastructure as the benefits are becoming more and more clear \[8\]. Both studies
as well as practitioner reports show benefits in areas such as customer satisfaction, time to market, effectiveness, process consistency and product quality [6, 7, 8].

Software development through agile principles and methods is more or less the de facto standard in the software development industry today. In the widely cited Agile manifesto by Beck et al., the highest priority is stated to be “to satisfy the customer through early and continuous delivery of valuable software.” [9]. It would thus be a natural step for an organization that strives to be agile to also implement CDL as part of their software development process.

Another aspect to take into consideration is the cost of having engineers spend their valuable time performing the repetitive task of deploying software manually. Engineers cost a lot of money for a company and while this is compensated by producing value for the company in the form of competence and work, it is important that time is spent efficiently. As the number of instances to deploy to grows, performing these manual deployments will take up a disproportionately large share of the workload. The introduction of automatic deployments ought to make the deployment cost more predictable and scalable in the long run.

There are several tools on the market that enable automation of software deployment and there is a variation in how they function, on what platforms they prosper and how they are priced. It is thus important to take multiple factors into consideration when choosing which tool to use at a company to make it beneficial. Examples of such tools are Chef[1], Puppet[2], Ansible[3], SaltStack[4], Octopus Deploy[5] and Azure DevOps[6]. These tools highlight different aspects of themselves, ranging from full configuration management through Infrastructure-as-a-Code (IAAC) (which in turn can be achieved either through a declarative or procedural approach, depending on tool), to complex deployments made easy.

1.2 Aim

The aim of this thesis is to implement CDL using two of the tools available on the market and evaluate the cost-related effects of doing so. The tools are chosen based on a set of requirements provided by the company, which forms the context in which the study is applicable. To be able to evaluate cost-related effects, a baseline is determined by analyzing the process of deployment at the company before the implementation of CDL. This baseline is then compared to deployments using the CDL tools. The comparison is done in regards to the allocation of resources such as time and money. Furthermore, it is taken into account how scalable the deployment process is in regards to the number of instances to deploy to. It is also interesting to discuss ease-of-use when it comes to implementing CDL with the evaluated tools. The result of this thesis can be used by software practitioners in similar situations to decide whether to implement CDL and which tool to use to do so.

1.3 Research Questions

1. What are the cost-related effects of implementing Continuous Delivery at a small software company, in relation to the number of deployments made per year by the company?

A small software company is defined as a company with 1-10 employees where the company’s primary business is related to software development.

---

[1]https://www.chef.io/
Research objectives will help in the endeavor to answer the research question by clearlyformulating sub-questions that need to be answered during the project’s progression. Theobjectives are reached and answered during different stages of the thesis, depending on theobjective. They are formulated in the following subsection.

1.3.1 Research Objectives

1. What are the requirements on the CDL tools to be evaluated, from the company atwhich this project is conducted?
2. Which CDL tools can be implemented with regards to the requirements defined by thecompany where this project is conducted?
3. What pricing schemes do the chosen CDL tools use?
4. What is the subjective experience of implementing the CDL tools chosen in this project,as a software developer with limited experience in similar tools?

1.4 Delimitations

The study will be conducted at a small Swedish company that installs their software sys-
tem as independent instances on-premise for their customers. The company develops theirsystem in the .NET environment[7] for Windows machines only and uses TeamCity[8] as theirContinuous Integration (CI) server. While the results of this study may be useful for soft-
ware practitioners and companies with different contexts, it is not the purpose of this studyto produce a one-size-fits-all solution to the software deployment problem.

1.5 Expected Results

The expected findings are that there will be cost benefits of using a CDL tool instead of manu-
ally updating every instance. It is however not certain that it will be beneficial for any numberof instances or with any tool. Since different pricing schemes apply and the time spent de-
ploying increases with the number of instances, it is likely that there will be either a startingpoint or different regions where it is beneficial to use CDL. This is particularly the case whenthe time spent to deploy becomes a disproportionately large part of the development-and-
release cycle of a software system.

1.6 Background

Agricam AB is a software company that was established in 2010. The company developssoftware systems to be used in the agricultural sector, primarily for detecting costly problemsusing sensors such as thermal imaging cameras. The company currently distributes theirprimary software system as an on-premise solution to their customers. The customer uses thesoftware in a Software-as-a-Service (SaaS) fashion, meaning that the customer experiences that the software is automatically distributed to them over the internet, when it in fact is anon-premise solution, administered by the employees of Agricam. The price model is also one that is commonly used in SaaS solutions, where the customer pays for the actual use of thesystem, rather than in a large sum after which they own the right to use the software.

The primary software system is built using the .NET framework and consists of anASP.NET website and a couple of modules in the form of .NET applications (in the range5-10). The system is run on independent Windows servers and uses the built in Windows
server manager Internet Information Services (IIS) to run the ASP.NET website. GitHub is used for Software Configuration Management (SCM) and TeamCity is used for CI. The current number of system instances is in the range 15-25 and the instances are both domestic as well as international.
2 Theory & Related Work

This chapter presents the concepts that are of interest for this thesis, and research that has been conducted on the area. The focus lies on CI, CDP, CDL and different methods and tools used to implement these techniques and workflows.

Another approach to the problem of updating on-premise software is also presented. Focus here lies on cloud migration of applications. Different methods and tools for doing this are presented, along with rationales for why this alternate approach may be desirable.

2.1 Terminology

In research on the field of automation and continuous practices within software development, there is no clear consensus on the terminology to use \[10\]. Terms are used interchangeably and with different definitions. Stahl et al. \[3\] conducted a systematic mapping study on research in the field where they came to these insights and then contributed two-fold to the area. Besides providing a suggestion on definitions of the terms, they also encourage fellow researchers to be explicit on what is included as well as not included in each used term to avoid the issue of ambiguity. This report will follow both guidelines provided by Stahl et al. by using their definitions and being explicit with what is included in said definitions. The following subsections will define the different terms, and for the purpose of this report, elaborate on practical use cases and realizations of the terms.

2.1.1 Continuous Integration

To expand upon the definition provided in section 0.1, Stahl et al. \[3\] define CI to be a developer practice, meaning it is the result of how developers behave and not how the development process is set up. In the end, developers are still required to actually integrate their changes often, preferably daily, to practice CI.

There are several software products on the market, as well as established workflows, to help facilitate the use of CI. Software products include for example TeamCity and Jenkins\[1\]. They can be installed on a server to keep track of changes in a SCM-repository and automatically fetch, build and test your software when new changes are pushed to the repository.

\[1\]https://jenkins.io/
Such products may have support for CDL and CDP as well. A typical workflow that takes advantage of such products is based on doing a Pull Request (PR) with the suggested change to the source code. When the PR is submitted to your SCM host, the changes are fetched to the CI server and then built and tested. If all is clear, the changes are integrated with the rest of the source code. The CI server can provide an environment which is easier to control and replicate compared to running the same tasks on for example a developers machine, making it a more reliable approach.

2.1.2 Continuous Delivery

Like described in section 0.1 Stahl et al. [3] define CDL as about being able to put software in a production environment at any time. A descriptive term that is used by the authors is that each software change should be a “potential release candidate”. A discrepancy is also made between developer and development practice. Where CI, as described in subsection 2.1.1 is dependant on developers actually integrating often (developer practice), CDL is implemented as a part of a functional pipeline following a change to the software (development practice). This means that the CDL practice is triggered automatically when it is due. Stahl et al. also emphasize that it is outside of the scope of CDL to select which changes that are put in production. It is rather a business decision of sorts, for example depending on practicality or customer requirements.

CDL is typically implemented by at least using SCM, setting up automated testing and setting up automated deployment [11]. The step from CI to CDL is not necessarily big. If the number of instances for the product being developed scales less-than-linearly with number of customers, for example a web service or a web site that typically is not one instance per customer, then the procedure can be rather simple. It might be enough to write a script that stops a server from running, copies files from one location to another and then restarts the server. The complexity of the script can of course vary, for example if there is a requirement of no downtime for the service.

If, like in this study, there are multiple remote and on-premise instances that need to be updated, there are other aspects that need to be considered. For example, there needs to be access to the remote machines that run the software. It is however likely that they are protected by firewalls. There also needs to be a protocol that defines how communication and transmission of files is done between the remote machines that are to be updated and the server that is to distribute the new software version. Many tools that enable CDL use some sort of agent that is installed on the remote machine and then connected to a master server [2,3,4]. These agents typically either use a polling mechanism or listen to a pushing mechanism from the master server to receive updates that are then installed on the remote machine following a pre-defined procedure. Such a procedure can consist of for example updating installed services, replacing binaries and restarting web servers.

2.1.3 Continuous Deployment

In line with the definition in section 0.1 Stahl et al. [3] suggest that CDP is basically CDL but where the automated deployment is triggered automatically when software changes have successfully finished the prerequisite stages, placing all the release candidates in production. It is also defined by the authors as an operations practice, implying that it goes beyond what developers do and how development is practiced. It integrates with the operating of software products in a much closer manner than the previously explained practices.

2 https://octopus.com/docs/infrastructure/deployment-targets/windows-targets
4 https://docs.chef.io/chef_client_overview.html
2.2. Implementing CDL - Benefits and Challenges

In comparison to CDL, CDP does not necessarily require many extra steps to be achieved since the automated deployment mechanism that is used in liaison with CDL can often simply be triggered automatically as well. It is however not uncommon to use automated configuration management through IaaS to automatically set up and provision production environments [6, 7]. This might be required since deploying automatically potentially creates risks in not being ready to handle failed deployment. However, a well developed and functioning deployment pipeline could also eliminate these risks that are also present when not using CDP, generally making releases more reliable [7, 8].

Another potential approach is to use so called canary deployments. This means that new releases are deployed to a smaller subset of users first, allowing the company to evaluate how the new version works in practice before deploying it to all users [10]. It is however subject to discussion if this approach really constitutes CDP or only CDL.

2.1.4 Continuous Release

Another term defined by Stahl et al. [3] is Continuous Release (CR). CR is, like CDP, a subsequent practice to CDL and can even be the same thing in some circumstances, depending on the context. The authors define it as a business practice and differ it from CDP by stating that it is about making release candidates available to stakeholders. They exemplify with software that is installed by the user, where a deployment is conceptually odd and a release probably fits better as a description.

CR can for example be realized by building software installers for the targeted platforms and uploading them to a publicly available website where users can download the installer and install the product.

2.1.5 DevOps

The term DevOps is, like already presented in section 0.1, one that is harder to define in precise terms. Stahl et al. [3] even consciously decide to not provide a complete definition because of the disparity in the field. They do however recommend explicitness where it is used. In this paper, to further clarify the previously provided definition from section 0.1, the definition is that DevOps is built from adhering to and using a set of values, principles, methods, practices and tools, which makes DevOps a superset of the previously mentioned terms [3]. Stahl et al. [3] provide examples to the different parts of DevOps. Values are basically agile prioritizational values that can be found in [9], such as “responding to change over following a plan”. Principles include “simplicity” and “automation”. Methods and practices such as CDL, CDP and CI are included and tools refer to tools that enable and support such methods and practices.

DevOps can be realized in an organization in different manners. It is increasingly common to describe the role of an engineer that is employed or sought after as a DevOps engineer [12]. Typical skills that such engineers hold are communicative skills, process automation skills, scripting skills and business understanding [12]. In a smaller company, there might not be need for a person working solely with DevOps, and the responsibility is instead shared between developers or simply as part of one developers job description. In a larger organization, there might be an entire department for DevOps, perhaps distributed throughout the different development teams.

2.2 Implementing CDL - Benefits and Challenges

This section presents how CDL can be implemented in an organization and the benefits and challenges that have been found in previous research in the area.
2.2. Implementing CDL - Benefits and Challenges

2.2.1 Important Factors for CDL

Forsgren and Humble [13] found that the most important factors for successful CDL are SCM, automated testing and CI. In line with this, Rahman et al. [11] conducted a study on the practices used in combination with CDP at 19 different companies and found that all of them were using SCM, automated testing and automated deployment. Olsson et al. [14] created a stairway model to visualize the natural steps for an organization to evolve from traditional waterfall development to CDP. The model implies that the prerequisite steps to CDP are an agile development organization and CI as an implemented practice. To implement a functioning deployment pipeline, it might also be necessary to re-architect existing systems [10]. Monolithic or otherwise dependant software systems might prove troublesome to integrate in an automated deployment practice since features will need to go through more instances, thus hindering them from being developed and delivered continuously.

2.2.2 Benefits

Forsgren and Humble [13] found that CDL helps reduce perceived deployment pain (fear of deployment or the view that they are disruptive) and burnout (as a result of stress), making it beneficial for employees working in a technical department.

Leppänen et al. [15] interviewed 15 technology companies about perceived benefits and obstacles. They found six categories of benefits. The most agreed upon benefit was faster feedback to development. Developers could receive feedback from customers regarding what features the customers really needed and should thus be improved upon and what features should be discontinued. In line with this, improved customer satisfaction was another category. Customer feedback could be responded to in a faster manner. A third category was more frequent releases, which is also in line with the previous categories. Added value in form of faster time-to-market and more informed stakeholders is mentioned. The final categories were improved quality and productivity, effort savings and closer development-operations connection. All of these emphasize that automation helps increase robustness and quality in both product, communication and delivery.

These findings are not unique. Rodríguez et al. [7] conducted a systematic mapping study on 50 primary studies in the field and found similar categories of benefits. The categories are, in order of frequency of finding, as follows: shorter time-to-market, continuous feedback, improved release reliability, increased customer satisfaction, improved developer productivity, rapid innovation and narrower test focus.

Chen [8] provided an experience report of implementing CDL at a large company. The reported benefits include accelerated time to market (release frequency), ability to build the right product, improved productivity and efficiency, release reliability, improved product quality and improved customer satisfaction. The time from having an idea of a new feature to having the idea implemented and in production has dramatically decreased. The frequent delivery and feedback cycle lets the team solicit input from users in a rapid manner, letting developers know which features to continue to work on and which to stop developing. The time spent on setting up environments and deploying software to production has also decreased, in some cases from days to instantaneous. The automation and streamlining of the release process makes it more reliable and also naturally tested since it is repeated. It is mentioned that engineers feel less stress when it comes to releases. Product quality in terms of unfixed bugs in the system used by customers has also dramatically decreased. Errors conceived from manual configuration and bad practices have also been erased.

None of these benefits are, however, focused on direct cost-related benefits. This further motivates this thesis.

Klepper et al. [16] conducted a case study on CDL usage in mobile applications. They found a reduction in the number of delivery steps from ten to one in complex projects and from five to one in other projects. They also found a reduction in the number of team mem-
bers that needed to be involved. With CDL, only a release manager was needed in most cases. The time spent on delivery ranged from an hour up to an entire day before CDL implementation. After the implementation of CDL, the time spent could be reduced down to five minutes.

2.2.3 Challenges

Leppänen et al. [15] identified obstacles categorized under eight themes from their interviews with 15 technology companies. The first is resistance to change. The resistance can come both from management as well as from the established culture in the company. The second theme is customer preference, which falls close to the third, domain constraints. The customer can impose constraints on the delivery of systems, for example if the customer cannot handle new features too often. Constraints like this can come straight from the domain in which the company operates. If the company must follow rigorous restrictions and manual testing protocols for example, CDL may become impractical. It can also be the case that the customer operates in a domain where updates require a halt of production, making the cost of updates too expensive to be performed too often. The remaining themes relate much to developers, development and the software architecture. The development team needs to be confident enough to trust their automated deployment, since it is their responsibility to deliver working software. This in turn puts great importance in substantial automated testing, testing environments that can emulate production environments and setting up potentially complex delivery pipelines that can be trusted. If the software itself is not designed to easily implement CDL, it may require substantial rewriting of code to even be able to deploy automatically. It is also a potential problem if automated tests take hours to execute due to the sheer size of a system and its tests. A potential issue is also found in testing that is difficult to automate. The examples brought up relate to exploratory testing and testing of performance under different workloads.

Shahin et al. [10] conducted a systematic literature review which included 69 papers and contributed with, amongst other things, a classification of reported challenges in implementing continuous practices, including CDL. Out of these 20 classes in total, 13 fall under either the category of challenges in adopting CDL or the joint category of challenges in adopting CDL, CDP, and CI. The classes are also thematically grouped under the following themes: team awareness and communication; lack of investment; change resistance; organizational processes, structure, and policies; lack of suitable architecture; and team dependencies. Many reported findings are similar to those in [15], but new types of findings are also reported by the authors. They mention challenges such as infrastructure and resource cost; technical and soft skill requirements; lack of technical tools with sufficient security, reliability and maturity for different tasks; and frequent changes in database schemas. Other reported challenges that are categorized under CDP but might arguably still apply to CDL are for example hardware and network dependencies.

Rodríguez et al. [7] also found similar challenges in their systematic mapping study. They point out some cognitive aspects found in literature, that developers and organizations need to adjust to faster release cycles and that it might be fearsome to put new features in production at a rapid pace for the unaccustomed. They also found customer unwillingness as a factor to consider. Partly because of the fear of receiving unfinished features but also because of changes to the user interface that may be difficult to get used to if they arrive at a high rate. Increased effort in quality assurance and testing is also highlighted as a big challenge.

The experience report provided by Chen [8] highlights organizational challenges as the biggest ones. The author discusses the difficulty in organizational changes in general, but while there is research on implementing such general changes, research on changes related to implementing CDL is scarce and thus requested by the author. The requirement of collaboration between teams and departments is brought up as a challenge, as well as the release process that needs to be better adopted for CDL.
2.3 Cloud Migration

In the stairway to heaven model, provided by Olsson et al. [14], the barriers encountered when climbing up the stairway is presented from their multiple-case study. In the step from CI to CDL (CDP is the term used in the study but it falls under the definition of CDL from subsection 2.1.2), a barrier is the variation of network configurations on customer sites. It is generally a problem when customization is necessary for a system to work. Another aspect that is brought up is the need to keep the internal status reporting of projects frequent and of high quality to maintain high transparency in the organization.

Just like with the benefits, none of these challenges are related to direct costs in using CDL, which also motivates this study even more.

2.3 Cloud Migration

This section presents an alternate approach to solve the challenge of achieving CDL of software. It is possible to achieve similar end results by migrating your on-premise solutions to a centralized cloud service, fully embracing the SaaS way of providing your system to your customers. Doing so, there should only be a very small and limited number of instances to update, mitigating the issue of spending resources on manual installation. Below, different types of migrations are explained, followed by methods of migrating, expected effects of migrating and some case studies on the subject.

2.3.1 Types of Migration

Andrikopoulos et al. [17] have identified four types of cloud-migration, in order to distinguish different approaches.

1. Replace: This approach is done by replacing one or more architectural components of the software with cloud services. This may require rewriting the parts of the code that interact with these components. Using a cloud database instead of a local database would be an example of this type of migration.

2. Partially migrate: This type entails migrating some of the software’s functionality, by moving a set of architectural components, from one or more layers, to the cloud. For example, using a cloud database in combination with cloud functions.

3. Migrate the whole software stack: This is done by moving the entire application to the cloud. This is the easiest way to migrate and is often done by encapsulating the application in virtual machines and then run them in the cloud. However, this only works for some types of applications.

4. Cloudify: Cloudifying an application means a complete migration to the cloud, thus converting the application to a fully cloud-native system. This requires larger changes to both the data and business logic of the application.

Throughout this section, these types of migration will be referred to as Type 1, Type 2 etc.

2.3.2 Methods of Migrating Software Applications to the Cloud

This section describes what can be found in previous research regarding different methods or tools that may be used to migrate software to the cloud.

Peddigari [18] presents a unified framework for streamlining the migration strategizing procedure, and recommends a factory-based approach for executing the migration. The method is very thorough, making sure a proper examination of existing application is done, so that the migrated application fulfills the same requirements. If the application needs re-engineering to be executable in the cloud environment, the process to determine and execute this is also very thorough. The factory-based approach to perform the migration is also very
2.3. Cloud Migration

detailed. Teams are compiled to perform the different steps, which for example consist of information gathering and analysis, development of reusable components and assessment and release.

Another method, presented by Cai et al. [19], proposes the use of a software tool for modifying code related to storage and security. The motivation is that a lot of manual rework of the target software is needed to enable migration to a cloud environment in the first place.

Khajeh-Hosseini et al. [20] present two different decision support tools, one aiming to help model the cost of cloud migration and one aiming to assess benefits and risks. The cost modeling tool provides a means to analyze the application in its current state to find out what it would cost to run the application on some of the public cloud platform vendors. The benefit/risk assessment tool can be used to identify potential risks that the authors have identified in related work. It also provides means to mitigate the risks. The case study performed by the authors indicated that it may be useful in a pre-study stage when a cloud migration project is planned.

2.3.3 Effects of Moving an On-Premise Application to the Cloud

Research of the benefits and risks by migrating on-premise applications to the cloud, has been made by Bibi et al. [5]. They conducted a case-study where they compared the costs of maintaining the company’s software, to their cost assessment of what a cloud solution would cost. Their conclusion was that the studied company would, using an SaaS, in total need to spend almost half of what an in-house solution costs. However, before deciding on migrating to the cloud, there are many aspects of the costs and benefits that needs consideration.

The main draws of cloud-solutions are, according to Jamshidi et al., that they reduce operational costs, increase the applications scalability and that they create efficient utilization of resources [21]. Assuming an available SaaS-solution exists, the provider will set up and maintain the overall infrastructure. This saves both money and time for the company adopting this solution. The total benefit, compared to the cost, is usually better for type 4 migration when compared to the other types [17]. Type 4 will however require a greater initial effort compared to the other types, which is why it is often not considered by stakeholders.

The main concerns of adopting cloud-solutions are privacy and security, and many companies choose not to migrate their core application to the cloud because of these factors [21]. The larger part of your software that is put in the cloud, the higher the security risk becomes. The overall reliance of the software might also decrease, with having to rely on external factors such as network and providers, especially for Type 3 migration [17]. The amount of customizability and configuration is also limited when relying on a provider [5]. The performance of the system will of course suffer because of the added latency, especially for types 3 and 4 migration where the entire application is moved [17].

2.3.4 Case Studies

A study by Khajeh-Hosseini et al. [22] analyzed how the migration of a company’s IT system from on-premise to the cloud affected the company financially and socio-technically. The paper also identifies a lack of case studies in the subject area and contributes to the field by simply conducting the study.

Their methodology consisted of three steps. The first was to identify the costs of hosting the system on-premise and compare them to a similar solution using a cloud platform. The second step involved analyzing the company’s support and maintenance database to identify what issues would be affected by the studied migration. Finally, semi-structured interviews were held with the employees to further understand what stakeholders exist and how the migration would affect their work. This was done in a methodological fashion using stakeholder impact analysis.
2.3. Cloud Migration

A stakeholder impact analysis has the goal of identifying key stakeholders, what changes to their tasks would result as a consequence of the change and how these changes would affect the stakeholders. After doing this, analysis is done on how these changes affect stakeholders in a wider context such as how their professional relations are impacted. Finally, it is analyzed whether the change will be perceived as just or not.

The result of the study was that while the cost analysis showed that the migration would save money over a five year period, there were issues for the identified stakeholders that made it hard to motivate the migration. The cost analysis needs to be more thorough and include migration cost, cost of infrastructure support and effects on support and maintenance. The stakeholder impact analysis proved useful and can be used even further in similar cases.

This study was also followed up by a paper presenting a toolkit called Cloud Adoption Toolkit which can be used to support the process of assessing cloud migration feasibility [23]. The paper presents the toolkit and also the results of a case study that uses the toolkit. The case study provided the insights that despite what cloud providers try to advertise, cost saving estimates cannot easily be generalized. It depends on the system and how resources are used. They also found that it is important to analyze the organizational impact of migrating a system.
This chapter describes the method used to conduct the study. First, the pre-study phase is described, where the focus was to solicit requirements from the company where the project was carried out and then use the requirements to evaluate the available CDL tools on the market. Then, the implementation phase is described. This phase includes a description of how the CDL tools are integrated in the existing pipeline and how they are configured and implemented. Finally, the evaluation phase is described. This includes how the experiments are set up, what is measured, how it is measured and how the cost-benefit analysis is conducted. The research methodology should, according to the guidelines by Runeson and Höst [24], be characterized as action research. The methodology is closely related to a case study but with the difference that action research is not only about observing, but also about being involved in the process. There is a thin line between simply studying the effects of a change, in which case the methodology should be classed as a case study, and also focusing on the change itself. Since implementation is a substantial part of this project, action research is a better characterization of the conducted work.

3.1 Pre-Study

This section describes how the pre-study phase was conducted. The aim of this phase was to solicit and interpret requirements on the choice of CDL tool, from the company at which the project was conducted. Furthermore, the aim was to use these requirements to search the market for available CDL tools. When a tool was deemed a feasible candidate, the final aim of the phase was to verify compatibility of the tool by implementing a simple proof-of-concept before going through with the tool to the implementation phase. The search procedure itself was searching in a semi-structured manner for tools on the search engine Google using terms such as ‘Continuous Delivery’, ‘Continuous Deployment’, ‘Automated Deployment’, and similar terms.

To define the requirements, informal discussions were held with relevant stakeholders within the company. The result of these discussions were:

- The tool should have some sort of free tier or evaluation license so it can be evaluated for free.
• The tool should not impose any requirements on new infrastructure or platforms that are not currently available or being used in the company.
  - For example, any software must run on Windows machines.
• The tool should support integration with TeamCity, or at least be able to interact with artifacts produced from TeamCity.
• The tool should support deployment of .NET websites and applications.
  - In particular, deploying .NET websites to IIS on Windows.
• The tool should be well documented and actively maintained.
  - Documentation is deemed good enough if basic to moderate use cases are documented.
  - The tool is deemed actively maintained if it has been updated at least during the end of 2018 or has planned updates for early-to-mid 2019.
• The tool should be easy enough to get started with and use for a developer with little experience in similar tools.
  - This represents the experience level of the author of this thesis and the development team in the company where the study is conducted.
• The tool should have support for deploying different versions of software to different customers depending on the customer configuration.

If any of the requirements were discovered to be unfulfilled during the examination of a tool, the examination was aborted and the tool was deemed unfit. The goal was to find at least two tools with which a proof-of-concept could successfully be implemented so a comparison could be made between different alternatives.

The proof-of-concept was chosen to include deployment of one .NET website to IIS on a Windows machine and deployment of one .NET application to the same Windows machine. This choice was made due to the composition of a full deployment of the entire product, which consists of a quantity of .NET websites and applications. At first, the proof-of-concept deployments were made to a developers Windows machine within the company’s local network. Since this does not reflect upon the real use case, this was later modified to a deployment to a Windows machine at a customer site.

Since there are requirements that depend on the skills of the person attempting to use the tool, as well as subjective understanding of for example documentation, a maximum timeframe was allotted to each tool. If a proof-of-concept could not be achieved within 16 hours of effective work, the requirements related to ease-of-use are deemed unmet, at least for a first iteration. If no other alternatives can be found, subsequent iterations of attempting to implement the tool was allowed, until two tools were implemented successfully.

### 3.2 Implementation

This section describes the implementation phase. The goal of this phase was to implement the chosen tools to mirror an actual deployment. A delimitation in the implementations is that it is seen as sufficient if the deployments are made up to a point where the software may require updated configuration regarding for example file paths to function correctly. The number of deployment targets was chosen from the maximum number of allowed deployment targets in the free licenses used in this project. The lowest such number was used to accommodate all the evaluated tools. The same deployment targets were also used in all cases. The deployment targets were chosen to give a realistic variation between machines in
3.3 Evaluation

terms of geographic location and perceived quality of internet connection. To meet the goal
of doing a full deployment, the following goals needed to be achieved:

- Fetch the relevant packages from TeamCity to the deployment target
- Install all applications, eight in total, on the deployment target by unpacking them
- Install both websites on the deployment target by doing the following for both:
  - Unpack the package at a location on the deployment target
  - Configure IIS by setting up a custom port for the website, starting an application
    pool, allowing anonymous authentication and pointing the website to run from
    the location the package was unpacked to

The deployment process should also be able to be started by performing a simple task such
as running a command or pressing a button.

3.3 Evaluation

This section describes the method used to evaluate the CDL tools against using manual de-
ployments from a cost-related perspective.

3.3.1 Calculating the Cost of an Engineer

The cost of an engineer can, of course, vary a lot. Since the study is based in Sweden, relevant
data and measures are applied. The average salary of an engineer in the software industry in
Sweden is retrieved from the Swedish Association of Graduate Engineers\footnote{https://arkiv.sverigesingenjorer.se/About-us/ionestat_eng/} [25]. They do an
annual survey of the salaries of their members. The average salaries, using the available data
at the time of writing this thesis, is shown in Figure 3.1. The job descriptions that are included
can be seen in the figure. The average is then calculated from all of these job descriptions.
Any monetary values that are in the Swedish currency SEK are converted to USD using the
formula \( 1 \text{ SEK} = 0.1080 \text{ USD} \) as retrieved from the live exchange rate at Oanda\footnote{https://www.oanda.com/currency/live-exchange-rates/}. This
conversion is made since all the CDL tools have their pricing in USD.
The monthly cost of an engineer is set as their monthly salary. The monthly cost is then used
to calculate the hourly cost by dividing the monthly cost with the average number of working
hours in a month. A decent approximation of the number of working hours per month is that
a month consists of four and a half weeks and that each week consists of 40 hours of work,
which totals 180 hours of work in a month. If we also assume that each month on average
has one national holiday that incurs on a working day, we instead get 172 hours of work
in a month. To summarize: the average hourly cost of a Swedish engineer is calculated as
\( \frac{\text{average monthly salary}}{172} \). Given the average salary for all job descriptions that is shown
in Figure 3.1, the average hourly cost becomes \( \frac{5610}{172} = 32,6 \text{ USD per hour} \) when rounded
to one decimal point.

3.3.2 Baseline Cost

To be able to understand the cost-related effects of implementing CDL, a baseline cost must
first be established. This baseline cost is the time spent by an engineer performing manual
updates. To calculate the baseline cost, the average time spent on performing manual updates
is measured. The time is measured in two different ways. The same five deployment targets
that were selected to be used with the CDL tools were chosen here. First, manual updates
are performed five different times per machine, for the five selected deployment targets. The

\[1 \text{ SEK} = 0.1080 \text{ USD} \]
updates are done sequentially at this stage. This is done to get an average measure on how long it takes to deploy an update to a single machine if effort is put into the task. Then, deployments are run in parallel for all five machines, to better reflect upon how updates are really performed. This is also repeated five times to get an average time on updating five machines if effort is put into optimizing the task of updating many machines simultaneously. It also provides a measure of an average time spent per machine that can be compared to the previous measure where updates were done sequentially. For completeness sake, it is also measured how long an individual update takes during the parallelized process. This time is measured using the same start and stop steps as in the sequential updates.

A manual update consists of a couple of steps. The prerequisite of a deployment is that a release is made in SCM, which triggers a release build in TeamCity, after which the necessary artifacts are transferred to an FTP server where they await manual download from a customer machine. The artifacts in this process primarily consist of an executable installer that installs all applications and websites on the machine. The installer and the other relevant files are packed in an archived folder on the FTP server. The manual update then begins by connecting to the customer machine using the remote desktop software TeamViewer. This is when the time is started. FTP software is then used to download the archived folder. The folder is then unpacked on the customer machine and the installer file is then executed. Once the installer finishes, the process of updating is finished and the time is stopped. Any faced obstacles during the process that affect the time spent were noted during the process. The process was parallelized by doing the same process on multiple machines simultaneously, for example by starting the download from the FTP server and then proceeding to connect to another machine in the meantime, jumping back and forward between machines to maximize efficiency. Again, any time wasting obstacles that were faced were noted. These measurements were made during office hours, between eight in the morning and five in the afternoon. Some of the attempts were made right after one another whilst other attempts were made on different days.

3https://www.teamviewer.com/
3.3.3 Using CDL Tools

To measure the cost of using CDL tools to accomplish deployment of software updates, there are several aspects that are relevant. First, it is necessary to estimate a start-up cost of implementing it in existing environments. To do this estimation, the time spent on achieving a proof-of-concept as well as the full implementation is measured and the cost is calculated using the hourly cost of an engineer from subsection 3.3.1. Then, it is necessary to calculate the cost of using the tool. Here it is interesting to project how the cost scales with the number of customers since it is the cost of updating at a customer site that is measured in the baseline. In relation to this, it is interesting to include how cost scales with frequency of updates, as this was a prominent beneficial factor as presented in subsection 2.2.2. Finally, it is relevant to measure how long it takes to do deployments corresponding to the ones manually performed. Here it is relevant to measure time spent by an engineer doing any necessary task as well as how long the deployments actually take, both per individual target, but also in total. These measurements were also made during office hours, between eight in the morning and five in the afternoon. Some of the attempts were made right after one another whilst other attempts were made on different days.
4 Results

This chapter presents the results from the pre-study phase, as well as the results from implementing and evaluating the CDL tools, using the method presented in chapter 3.

4.1 Pre-Study

The following sections describe the CDL tools that were included in the pre-study. Each section elaborates on some of their attributes in relation to the requirements presented in section 3.1 and the subjective experience from trying to implement a proof-of-concept with the tool, if applicable. The sections are given with the tools that did not meet the requirements first.

4.1.1 Chef

Chef is a Continuous Automation platform with a couple of different products aiming at solving slightly different problems, all of which relate to DevOps. Habitat is one of the products, and it is advertised to be open source and support deployment and management of applications. Since neither any information nor any plugins were found that enabled integration with TeamCity, Chef and its related products was deemed an unfitting candidate.

4.1.2 Puppet

Puppet is also a platform aiming at providing means to automate infrastructure and software management and delivery. Puppet Pipelines is the part of Puppet that is meant to handle CDL. There is a self-hosted version, which requires Unix-based operating systems to operate for the server software. There is also a SaaS version but no integration to use artifacts from TeamCity is available. For these reasons, Puppet was considered an inappropriate candidate for the project.

---

1https://docs.chef.io/platform_overview.html
2https://www.habitat.sh/
3https://puppet.com/products/how-puppet-works
4https://puppet.com/products/puppet-pipelines
4.1.3 Ansible

Ansible is an IT automation engine that enables provisioning, management and deployment of resources, configuration and applications. Ansible has a product called Ansible Tower which only runs on Unix-based operating systems. Because of this, Ansible was classified as an unfit candidate.

4.1.4 SaltStack

SaltStack is a platform for orchestrating, managing and automating infrastructure. To get access to SaltStack software, there is a form to fill on their website, after which contact is supposed to be made by the company behind SaltStack. This contact was never made by the company, making it impossible to further examine SaltStack as an alternative.

4.1.5 Buddy

Buddy is a CI and CDL platform aiming at providing a simple user interface for automating delivery of applications. There is a self-hosted version called Buddy Enterprise, but it requires Unix-based operating systems. There is also a SaaS version but no way to integrate with TeamCity to use artifacts that are ready to deploy. There is also no simple way to deploy to Windows machines. For these reasons, Buddy was ruled out as a candidate.

4.1.6 GoCD

GoCD is a free and open source CDL server. The central server component of GoCD can be installed on Windows machine and configured as a web server to receive requests. Agents can be installed on deployment targets and can also be on Windows machines. These agents are configured as polling agents by default, making it possible to use them under varying network configurations from customer premises. It is also possible to configure agents from the server to be under different environments which enables deployment of different versions to different customers. There is a plugin available to enable fetching of NuGet packages from a NuGet repository, which is sufficient for collecting artifacts from TeamCity. The project is updated with new releases every month and there are active discussions on GitHub where the source code is hosted. There is also a community forum that is active with users.

An attempt was made to implement a proof-of-concept, but obstacles were faced when trying to take artifacts from TeamCity and actually deploying them. The documentation is extensive but does not provide any examples on how to achieve a simple deployment. Attempts to implement a proof-of-concept were stopped due to exceeding the time limit of 16 hours. Because of this difficulty to succeed in deploying, GoCD was regarded a moderately fitting but poor candidate for this project.

4.1.7 Octopus Deploy

Octopus Deploy is a deployment automation server that can be downloaded and installed as self-hosted software on Windows machines, or be used as a SaaS solution. Deployments can be made by installing agents called Tentacles on Windows machines. Octopus has an unadvertised free tier which allows you to run deployments on up to five different deployment targets. This free tier is only for the self-hosted version, however. For this thesis, Octopus is tested on a self-hosted instance using this free tier. Octopus also has a plugin developed for

---

5 [https://www.ansible.com/overview/how-ansible-works](https://www.ansible.com/overview/how-ansible-works)
7 [https://buddy.works/](https://buddy.works/)
8 [https://www.gocd.org/](https://www.gocd.org/)
integration with TeamCity. The product is updated with new releases frequently and documentation is extensive. There is an active support forum and also an active community on the messaging application Slack. It is possible to deploy different versions by using roles which you assign to different customers. Octopus has well developed support for .NET deployments, for example with predefined steps for deploying to IIS.

The tool has different pricing schemes depending on if you are using the self-hosted version or the SaaS solution. For the self-hosted variant, you pay a yearly fee where you get access to a pre-determined number of deployment targets depending on how much you pay. There are two types of self-hosted variants as well, normal and Data Center. With Data Center, an even larger number of targets is possible, as well as other features such as clustered databases and several server nodes to maintain high performance and availability. For more than 75 deployment targets, Data Center license is required. The prices can be seen in Table 4.1. For the SaaS variant, you pay 45 USD per month as a base for five deployment targets, with an added nine USD per month for each additional target. For comparison, this totals to 108 USD per year, per deployment target. Both pricing schemes include an unlimited number of users of the software.

Table 4.1: Octopus Deploy pricing.

<table>
<thead>
<tr>
<th>Deployment Targets</th>
<th>Self-hosted yearly cost (USD)</th>
<th>SaaS yearly cost (USD)</th>
<th>Self-hosted yearly cost per target (USD)</th>
<th>SaaS yearly cost per target (USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>420</td>
<td>540</td>
<td>84</td>
<td>108</td>
</tr>
<tr>
<td>10</td>
<td>840</td>
<td>1080</td>
<td>84</td>
<td>108</td>
</tr>
<tr>
<td>15</td>
<td>1260</td>
<td>1620</td>
<td>84</td>
<td>108</td>
</tr>
<tr>
<td>20</td>
<td>1680</td>
<td>2160</td>
<td>84</td>
<td>108</td>
</tr>
<tr>
<td>25</td>
<td>2100</td>
<td>2700</td>
<td>84</td>
<td>108</td>
</tr>
<tr>
<td>40</td>
<td>3280</td>
<td>4320</td>
<td>82</td>
<td>108</td>
</tr>
<tr>
<td>50</td>
<td>4100</td>
<td>5400</td>
<td>82</td>
<td>108</td>
</tr>
<tr>
<td>75</td>
<td>6150</td>
<td>8100</td>
<td>82</td>
<td>108</td>
</tr>
<tr>
<td>100</td>
<td>7200</td>
<td>10800</td>
<td>72</td>
<td>108</td>
</tr>
<tr>
<td>150</td>
<td>10500</td>
<td>16200</td>
<td>70</td>
<td>108</td>
</tr>
<tr>
<td>200</td>
<td>14000</td>
<td>21600</td>
<td>70</td>
<td>108</td>
</tr>
<tr>
<td>300</td>
<td>20400</td>
<td>32400</td>
<td>68</td>
<td>108</td>
</tr>
<tr>
<td>400</td>
<td>27200</td>
<td>43200</td>
<td>68</td>
<td>108</td>
</tr>
<tr>
<td>500</td>
<td>34000</td>
<td>54000</td>
<td>68</td>
<td>108</td>
</tr>
<tr>
<td>750</td>
<td>51000</td>
<td>81000</td>
<td>68</td>
<td>108</td>
</tr>
<tr>
<td>1000</td>
<td>63000</td>
<td>108000</td>
<td>63</td>
<td>108</td>
</tr>
<tr>
<td>1500</td>
<td>93000</td>
<td>162000</td>
<td>62</td>
<td>108</td>
</tr>
<tr>
<td>2000</td>
<td>116000</td>
<td>216000</td>
<td>58</td>
<td>108</td>
</tr>
<tr>
<td>2500</td>
<td>140000</td>
<td>270000</td>
<td>56</td>
<td>108</td>
</tr>
<tr>
<td>3000</td>
<td>162000</td>
<td>324000</td>
<td>54</td>
<td>108</td>
</tr>
<tr>
<td>4000</td>
<td>208000</td>
<td>432000</td>
<td>52</td>
<td>108</td>
</tr>
<tr>
<td>5000</td>
<td>250000</td>
<td>540000</td>
<td>50</td>
<td>108</td>
</tr>
<tr>
<td>7500</td>
<td>337500</td>
<td>810000</td>
<td>45</td>
<td>108</td>
</tr>
<tr>
<td>10000</td>
<td>420000</td>
<td>1080000</td>
<td>42</td>
<td>108</td>
</tr>
</tbody>
</table>

To implement a proof-of-concept, the Octopus Deploy self-hosted server software was installed on a central Windows machine at the company’s premises. This server will be referred to as the central server in the rest of this section. A basic configuration for managing deployment targets and deployments was set up. At first, a client-side agent, called a tentacle, was installed on a development machine within the company’s network. This agent was connected to the central server by following simple steps through the graphical interfaces provided by the agent software and the central server software. This agent was set up as a listening agent, which basically means that it is configured as a web server that the central server can send requests to, for example containing information that there are new deploy-
ments to receive. Since the central server and the agent are both installed within the same network, this configuration is feasible. This is not the case with the machines at customer sites. Since there is no control over the local network configuration, it is difficult or even impossible to reach the agent from outside the network. It is therefore not a feasible approach to set up the agents to be listening. The agent was instead re-installed and set up to be a polling agent, which means that the agent periodically sends requests to the central server, asking if there are any new tasks to perform. Since the central server is either installed within the company’s network, or on a public cloud service, it is possible to configure it to be reachable from any customer site.

After establishing communication between the central server and the agent, next step was to configure deployments of one .NET website to IIS and one .NET application. First, an integration with TeamCity was established. To do this, new steps were added in the build process in TeamCity, which packaged the relevant applications to NuGet packages using the Octopus TeamCity plugin and then published these to a NuGet package feed on the central server. To accomplish this, a NuGet package called OctoPack, also developed by Octopus, needed to be installed in the projects that were going to be packaged as applications or websites. In some cases, it is also necessary to manually describe the content of the projects that are to be packaged by writing a specification file. This is the case when OctoPack is not capable of automatically detecting the necessary files to include in a package. These are written in the .nuspec-format. Publishing packages to the feed makes sure that any software that has passed the build process in TeamCity is made available in an internal repository on the Octopus central server, ready to be included in a deployment.

Next, a project was created on the central server, in which a process could be defined. This process contains different steps which make up a deployment. To perform the required deployments for the proof-of-concepts, two step templates provided by Octopus could be used. One called Deploy to IIS and one called Deploy a Package. The step for deploying to IIS requires some basic configuration of how the related IIS website should be composed and it is done intuitively for someone with basic experience in using IIS. It is also necessary to choose which package to deploy to IIS, as well as to what deployment target roles to deploy to. The package deployment step only requires choosing which package to deploy and to what deployment target roles to deploy to.

Finally, an attempt to deploy could be made, and the IIS website was successfully deployed and installed, as was the application.

Overall, the procedure of setting up a proof-of-concept deployment with Octopus Deploy was rather straightforward with few if any obstacles. The total time spent implementing the proof-of-concept, including initial pre-study, was eight hours.

### 4.1.8 Azure DevOps

Azure DevOps is a platform that provides collaboration tools for SCM, CI/CD, artifact management and testing. The platform is made up of different products that are integrated into the platform. The CI/CD tool that is particularly in focus here is called Azure Pipelines. Azure DevOps comes in two tastes, one self-hosted version called Azure DevOps Server and one SaaS version called Azure DevOps Services. For this project, the SaaS version is tested. This due to it definitely having a free tier, while the self-hosted version appears to require other subscriptions to be free in the sense that it is included. This SaaS software can be seen as a central server from which deployments are administered and is therefore referred to in this section as the central server. Since the platform is developed by Microsoft, Windows compatibility is present in all aspects. There is a plugin that enables integration with TeamCity.

---

10https://octopus.com/docs/api-and-integration/teamcity
11https://octopus.com/docs/packaging-applications/creating-packages/nuget-packages/using-octopack
12https://docs.microsoft.com/en-us/nuget/reference/nuspec
13https://azure.microsoft.com/en-us/services/devops/
and the product is frequently updated. There is extensive documentation on the product and some documentation to cover basic use cases. There is a small community on an unofficial Slack workspace and an official user and support forum. Deployment of different versions to different targets is made available by the use of deployment groups. Deployments are made possible by installing agents on physical or virtual Windows machines and registering them with the central server.

There is a free tier available for evaluation, but the pricing scheme is not as linear as with many other tools. For starters, since the part being evaluated in this project is CDL, focus is limited to costs related to using only Azure Pipelines. It is not fully clear how exactly pricing is calculated but discussions with Microsoft support and documentation provides the following information: For the SaaS version, one Microsoft-hosted parallel job that can run for up to 60 minutes each time, with a maximum of 1,800 minutes per month, is included for free. If one pays 40 USD per month to upgrade, the job can run for 360 minutes each time and an unlimited number of minutes per month. One can then buy extra Microsoft-hosted parallel jobs for 40 USD per month, per job. A parallel job can constitute either a build job or a release job. In the context of this project, such a job constitutes a release job. One can also use self-hosted parallel jobs. Such jobs have no time limitations, it only dictates how many jobs can be run in parallel. One self-hosted job is included for free, each additional one costs 15 USD per month. What is unclear is what actually constitutes a self-hosted job. The implemented proof-of-concept, which is described below, does not seem to consume any execution time, so deploying to agents on remote machines can be assumed to be self-hosted jobs. This means that the tool is actually free to use, independent of number of deployment targets. The only thing that can incur costs is assigning more than five users to administrate the central server. Since this is easily avoided, it is not included as a potential cost.

To implement the proof-of-concept, an organization was first set up on Azure DevOps Services. The organization then had a project created in it. In the project, next step was to create a deployment group. This in turn automatically assigned the group to an auto-generated organization-level deployment pool. Using the overview of the deployment group, a script can be generated. This script should then be executed on the deployment target machine. The script downloads and installs the agent, as well as connects it to the deployment group. The script can be seen in Figure 4.1. It contains some placeholders related to the naming of the agent and which project and deployment group to connect it to, as well as a authentication token, which are taken out of the example for security purposes. The agent is automatically set up to be an agent of polling type, even if this is not explicitly stated anywhere. This means no particular network configuration needs to be configured.

After successfully installing the agent and connecting it to the central server, next step was to configure deployments of one .NET website to IIS and one .NET application. First, an extension to allow connection to TeamCity for artifact downloads was installed in the organization. Then, a service connection was set up in the project which allows connection to the TeamCity server. To generate the necessary packages and make them available, the OctoPack tool and Octopus TeamCity Plugin that was used for Octopus Deploy was once again utilized. The tool and plugin is free and does the job of packaging .NET applications regardless of Octopus Deploy and therefore serves a purpose here as well. The same steps for using OctoPack and the plugin as in subsection 4.1.7 were applied, with the difference that the packages did not need to be published to any NuGet feed. The build step used also publishes the packages to the artifact store in TeamCity automatically. Since the connection to Azure DevOps uses the artifact store directly, this method is used instead. The difference compared to the other method is that packages are not persisted in the CDL tool itself. It becomes dependant on TeamCity storing artifacts for later usage if such usage is required.

Next step was to define the deployment process. To do so, a release pipeline was created in the project. The required artifacts were chosen and a stage was created. In the stage, a job was created, and then populated with three tasks. The job was configured to run on deployment groups, and the deployment group where the agent had been installed was chosen. The
 config .cmd deploymentgroup deploymentgroupname "DEPLOYMENT GROUP NAME HERE" —agent AGENT_NAME_HERE —runasservice —work '_work' —url 'PROJECT URL HERE' —projectname 'PROJECT NAME HERE' —auth PAT —token AUTH_TOKEN_HERE; Remove-Item $agentZip;

Figure 4.1: Script to install Azure DevOps agent and connect to a Deployment Group.

first task that was created took advantage of a NuGet task template. It allows for execution of commands using binaries for the NuGet package manager that are included with the installed agent. The template allows for entering what type of command to use with NuGet. Since no command was fitting for the job, a custom command was written instead and passed to NuGet. The command that was sent to NuGet can be seen in [Figure 4.2]. It takes advantage of configurable variables that are available to the agent, all of which are enclosed in parentheses prefixed with '$'.


Figure 4.2: Command sent to NuGet to install a .NET application from a NuGet package.
The second and third tasks were created to be used together using two step templates to configure and deploy to IIS. The task to configure IIS was straightforward for someone with basic knowledge of IIS. It requires choosing a name for the website, choosing authentication type, what ports to listen on and point out which physical path on the machine the website runs from. The deployment task requires choosing which IIS website to deploy to and which package or folder that should be installed on the physical location pointed out in the previous step.

After all of these steps were set up, a deployment could be made by creating a release of this release job. Both the website and the application were installed successfully.

The overall procedure of doing the proof-of-concept deployment was rather straightforward and no precluding obstacles were faced. The total time spent implementing the proof-of-concept, including initial pre-study, was twelve hours.

### 4.1.9 Choice of Tools to Evaluate

The evaluated tools are summarized in Table 4.2. Each row summarizes the evaluation of a tool, and which tool the row summarizes is seen in the left-most column. Each data-containing column corresponds to the requirements stated in section 3.1, except for the right-most column which contains any miscellaneous but relevant data. Each cell can either contain an affirmative post which means that the requirement is met, a dissenting post which means that the requirement is not met or no post at all which means that the requirement was not evaluated due to another requirement not being met. There are, however, a few special cases. Puppet is considered partially infrastructure compliant. This because only parts of the software requires new infrastructure, making it require less effort to adopt. GoCD might possibly support .NET deployments. It is unclear since the documentation was difficult to grasp and there were no templates or similar to be found to confirm support. All of the evaluations are described more in depth in each row-corresponding subsection in section 4.1.

Since there were only two tools that met every requirement, they were also chosen. The chosen tools were as such Octopus Deploy and Azure DevOps.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Free tier/evaluation license</th>
<th>Infrastructure compliant</th>
<th>TeamCity integration</th>
<th>.NET deployments</th>
<th>Active project</th>
<th>Ease of use</th>
<th>Flexible deployments</th>
<th>Misc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Octopus Deploy</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>Chef</td>
<td>-</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Puppet</td>
<td>-</td>
<td>Partially</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Ansible</td>
<td>-</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Saltstack</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Could not get access</td>
</tr>
<tr>
<td>Buddy</td>
<td>-</td>
<td>No</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GoCD</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Possibly</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>Azure DevOps</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
</tbody>
</table>

### 4.2 Implementation

The following sections present the results of implementing the chosen tools from subsection 4.1.9. They describe how full scale deployments were configured for each tool and any problems that were faced. The implementations are extensions of the proof-of-concepts, and where applicable, the results in section 4.1 are referred to. The total time spent on implementing deployments with the tool, including the proof-of-concept, is also presented since this information was required to do estimations of initial costs of using the tool. The number of deployment targets was set to five, since Octopus Deploy has this limitation in its free license.
4.2.1 Octopus Deploy

This section describes the full implementation of Octopus Deploy. The implementation builds upon the proof-of-concept described in subsection 4.1.7. No alterations were made to the initial configuration.

The first step was to install the polling agents on the remaining machines. Four more machines were installed and configured in the same manner as the machine used in the proof-of-concept.

The next step was to add more steps in the process that was defined in the proof-of-concept. Seven more .NET applications deployments were added using the step template Deploy a Package, which takes an artifact in the form of a NuGet package and unpacks it at the deployment target. Another IIS website deployment step was also added using the Deploy to IIS template. It was configured similarly as the IIS website from the proof-of-concept, but using a different port number, application pool and execution folder. This website application has a different folder structure than the one from the proof-of-concept and issues were faced getting OctoPack to package it correctly. A .nuspec-file was written to manually include all necessary files. The content of the file can be seen in Figure 4.3. After successfully doing this, all steps were successful and a full deployment could be made.

The total time spent, including the eight hours of work on the proof-of-concept, was 20 hours.

4.2.2 Azure DevOps

This section describes the full implementation of Azure DevOps. The implementation builds upon the proof-of-concept described in subsection 4.1.8. The first step was to install the remaining agents and connect them to the existing deployment group. The same script as in the proof-of-concept was used again on each machine, successfully installing the remaining targets.

When moving on to configuring the remaining steps for a full deployment, it was discovered that Azure DevOps has a feature for grouping steps in so called task groups. This can facilitate reuse of deployments of modules that are needed in different types of deployments. The first task group to be created was to deploy all the standard .NET applications. The first task clears the installation directory. The subsequent eight tasks use the script in Figure 4.2 together with the existing NuGet task template to unpack NuGet packages. One of the tasks was a task copied from the proof-of-concept. The next task group was used to deploy one of the .NET websites to IIS. This task group simply contains the two remaining tasks from the proof-of-concept. The final task group was a copy of the second, with modified configuration of naming of the website, folder to execute from, port number and application pool. Furthermore, variables was found as a feature in the release pipeline, enabling defining variables that can be used in the deployment process. The added variables were used to define the installation directory and set whether to output log messages during deployment. Worth noting is that the issue with packaging of the second website was also relevant here but since this issue was already solved during implementation of Octopus, it was not necessary to solve the issue again.

The total time spent, including the twelve hours for the proof-of-concept, was 24 hours.

4.3 Evaluation

This section presents the results of doing the time measurements and cost calculations presented in section 3.3. The results are divided into subsections, the first presenting the results of doing time measurements to calculate the baseline cost. The results of the cost calculation is also presented. The second subsection presents the results of the time measurements of
4.3. Evaluation

<?xml version="1.0" encoding="utf-8"?>
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
  <metadata>
    <!-- Required elements -->
    <id>Agricam.CaDDi.WebServer</id>
    <version>1.0.0</version>
    <description>None</description>
    <authors>Agricam</authors>
  </metadata>
  <!-- Optional elements -->
  <!-- ... -->
</metadata>
<!-- Optional 'files' node -->
<files>
  <!-- This assumes that some files (see marked below) that are not in this project are already copied/moved to the project root folder -->
  <file src="favicon.ico" target="favicon.ico" />
  <file src="Global.asax" target="Global.asax" />
  <file src="Web.config" target="Web.config" />
  <!-- START files/folders that are not included in project but need to exist -->
  <file src="asset-manifest.json" target="asset-manifest.json" />
  <file src="index.html" target="index.html" />
  <file src="manifest.json" target="manifest.json" />
  <file src="static\js\*" target="static\js" />
  <file src="static\css\*" target="static\css" />
  <file src="static\media\*" target="static\media" />
  <!-- END -->
  <file src="bin\*.dll" target="bin" />
  <file src="bin\roslyn\*" target="bin\roslyn" />
  <file src="bin\i386\*" target="i386" />
  <file src="bin\amd64\*" target="amd64" />
</files>
</package>

Figure 4.3: .nuspec file to tell OctoPack how to package the .NET web application.

doing deployments with the CDL tools and their respective cost of usage. The final subsection summarizes the results presented in this section. During this entire section, the machines are named Machine A through E. Each machine has the same name assigned to it during all trials.

4.3.1 Baseline Cost

This section presents the measurements and calculations related to establishing the baseline cost.

Sequential Deployments

First, the results of measuring time spent on doing manual updates in a sequential manner is presented in Table 4.3.
Table 4.3: Time spent on sequential manual deployments.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0:02:46</td>
<td>0:09:35</td>
<td>0:07:30</td>
<td>0:03:35</td>
<td>0:03:34</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>0:02:35</td>
<td>0:05:37</td>
<td>0:05:32</td>
<td>0:02:36</td>
<td>0:02:56</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>0:02:03</td>
<td>0:04:12</td>
<td>0:05:35</td>
<td>0:02:14</td>
<td>0:02:17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>0:02:39</td>
<td>0:04:16</td>
<td>0:05:32</td>
<td>0:02:23</td>
<td>0:02:13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>0:03:12</td>
<td>0:02:53</td>
<td>0:06:37</td>
<td>0:02:10</td>
<td>0:01:59</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Average</td>
<td>0:02:39</td>
<td>0:05:19</td>
<td>0:06:13</td>
<td>0:02:36</td>
<td>0:02:39</td>
<td>0:03:33</td>
<td>0:19:26</td>
</tr>
<tr>
<td>Best attempt</td>
<td>0:02:03</td>
<td>0:02:53</td>
<td>0:05:32</td>
<td>0:02:10</td>
<td>0:01:59</td>
<td>0:02:55</td>
<td>0:14:37</td>
</tr>
<tr>
<td>Worst attempt</td>
<td>0:03:12</td>
<td>0:09:35</td>
<td>0:07:50</td>
<td>0:03:35</td>
<td>0:03:52</td>
<td>0:05:37</td>
<td>0:28:04</td>
</tr>
</tbody>
</table>

Any cells annotated with ‘*’-symbols were attempts that encountered issues. The meaning of the symbols is explained below:

- * Problems updating IIS website, had to manually shut down IIS during installation and then restart.
- ** Received urgent phone call.
- *** Used a different, non-working FTP application at first.

Note that there is no relation between the machines in an attempt. This explains the empty Average and Total columns for attempts one through five. What is instead calculated is the average time spent per machine in the average case, the best case and the worst case. The best and worst cases are added for completeness sake, adding a range on the cost. The total time spent in each case is also put in the table for comparison with parallelized attempts, both manual and with CDL tools.

Using the average time per machine in the three cases, the cost per machine is shown in Table 4.4. The time spent is converted to hours and then multiplied with the cost per hour to get cost per machine. A graph depicting how this cost scales with number of machines to deploy to can be seen in Figure 4.4. The cost assumes all machines are updated and is per such full deployment to all machines.

Table 4.4: Cost per machine when deploying sequentially.

<table>
<thead>
<tr>
<th>Time spent per machine</th>
<th>Number of hours</th>
<th>Cost per hour (USD)</th>
<th>Cost per machine (USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best case</td>
<td>0:02:55</td>
<td>0:049</td>
<td>32,6</td>
</tr>
<tr>
<td>Average</td>
<td>0:03:53</td>
<td>0:065</td>
<td>32,6</td>
</tr>
<tr>
<td>Worst case</td>
<td>0:05:37</td>
<td>0:094</td>
<td>32,6</td>
</tr>
</tbody>
</table>

Parallel Deployments

Here, the time spent on parallelized manual updates can be seen in Table 4.5. The rightmost column depicts the total time spent on updating all five machines within an attempt. Note that this is, as expected, less than the sum of the time it took to update each individual machine. The average of these times, divided with the number of machines, constitutes the average time spent per machine, which is also depicted in the bottom row of that column. The Average-column depicts the average time it took to update a machine within a particular attempt. The Average-row depicts the average time it took to update a particular machine over all five attempts. The cell where the aforementioned row and column meet depicts the time it took to update a machine on average over five attempts.

Any cells annotated with ‘*’-symbols were attempts that encountered issues. The meaning of the symbols is explained below:

- * Forgot about which steps had been done, had to look over this.
4.3. Evaluation

Machines

Cost per deployment (USD)

0

100

200

300

400

20 40 60 80 100

Best case cost

Average case cost

Worst case cost

Figure 4.4: Average cost of deployment in relation to number of machines for sequential manual deployments.

Table 4.5: Time spent on parallel manual deployments.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Attempt 1</th>
<th>Attempt 2</th>
<th>Attempt 3</th>
<th>Attempt 4</th>
<th>Attempt 5</th>
<th>Average</th>
<th>Parallel total</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0:02:24</td>
<td>0:02:02</td>
<td>0:02:24</td>
<td>0:02:32</td>
<td>0:02:10</td>
<td>0:02:18</td>
<td>0:09:28</td>
</tr>
<tr>
<td>B</td>
<td>0:03:27</td>
<td>0:03:43</td>
<td>0:03:43</td>
<td>0:03:49</td>
<td>0:04:37</td>
<td>0:04:40</td>
<td>0:09:13</td>
</tr>
<tr>
<td>C</td>
<td>0:06:00</td>
<td>0:05:43</td>
<td>0:06:38</td>
<td>0:05:34</td>
<td>0:05:36</td>
<td>0:05:49</td>
<td>0:08:33</td>
</tr>
<tr>
<td>D</td>
<td>0:05:53</td>
<td>0:05:53</td>
<td>0:02:33</td>
<td>0:02:50</td>
<td>0:02:16</td>
<td>0:02:49</td>
<td>0:08:15</td>
</tr>
<tr>
<td>E</td>
<td>0:01:33</td>
<td>0:02:26</td>
<td>0:01:31</td>
<td>0:03:56</td>
<td>0:01:35</td>
<td>0:02:32</td>
<td>0:07:13</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

** Used a different, non-working FTP application at first.

The cost calculation for the parallel attempts can be seen in Table 4.6 and the corresponding graph depicting how the cost scales with number of machines to deploy to can be seen in Figure 4.5. It is also assumed here that all machines are updated and the cost is as such per full deployment to all machines.

Table 4.6: Cost per machine when deploying in parallel.

<table>
<thead>
<tr>
<th>Time spent per machine</th>
<th>Number of hours</th>
<th>Cost per hour (USD)</th>
<th>Cost per machine (USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0:01:43</td>
<td>0:029</td>
<td>32,6</td>
</tr>
</tbody>
</table>

4.3.2 Using CDL Tools

This section presents the results of the timed trials and cost calculations for the evaluated CDL tools.

Octopus Deploy

First, the time it took to deploy using Octopus is displayed in Table 4.7. The right-most column depicts the total time spent on updating all five machines within an attempt. Note
that in this case, this is pretty much the same as the slowest machine, as all deployments have
finished when the last one is finished. The average of these times, divided with the number of
machines, constitutes the average time it takes to update a machine, which is also depicted in
the bottom row of that column. The Average-column depicts the average time it took to update
a machine within a particular attempt. The Average-row depicts the average time it took to
update a particular machine over all five attempts. The cell where the aforementioned row
and column meet depicts the time it took to update a machine on average over five attempts.
The times displayed in this table are, however, not time spent by an engineer. The time spent
by an engineer is as little as mere seconds and is therefore disregarded. What is necessary to
note, however, is that two attempts failed, one of which failed twice. This induced extra time
spent on investigation of the cause and restarting the deployment. The cause, in all cases, was
that polling agents did not poll on time, which caused the entire deployment to be marked as
failed, including the ones that worked as expected. The polling agents were restarted on the
machines, solving the issue, at least temporarily. The extra time spent was not measured and
is therefore also disregarded here but it is discussed in [Chapter 5]. Failed attempts are noted
in the table using "*"-symbols, one for each failed attempt. The failing machines are noted in
the same manner, one symbol per fail.

The cost here is, since time spent deploying is negligible, the cost of setting up and starting
to use the tool and the cost of the tool itself. The start cost is, using the measure of time spent
in subsection 4.2.1

\[20 \text{ hours} \times 32.6 \text{ USD/hour} = 652 \text{ USD}\]

The cost of using Octopus as the number of machines to deploy to grows is seen in Figure 4.6. A zoomed in version showing the cost up to 100 machines can be seen in Figure 4.7. The figures are based on Table 4.1.

![Figure 4.6: Yearly cost of using Octopus Deploy based on number of machines to deploy to.](image1)

![Figure 4.7: Yearly cost of using Octopus Deploy based on number of machines to deploy to (zoomed in on up to 100 machines).](image2)
4.3 Evaluation

Azure DevOps

First, the time it took to deploy using Azure DevOps is displayed in Table 4.8. The rightmost column depicts the total time spent on updating all five machines within an attempt. Note that in this case just as in the case with Octopus, this is pretty much the same as the slowest machine, as all deployments have finished when the last one is finished. The average of these times, divided with the number of machines, constitutes the average time it takes to update a machine, which is also depicted in the bottom row of that column. The average column depicts the average time it took to update a machine within a particular attempt. The average row depicts the average time it took to update a particular machine over all five attempts. The cell where the aforementioned row and column meet depicts the time it took to update a machine on average over five attempts.

The time spent by an engineer is in this case also as little as mere seconds and is therefore disregarded here as well. Note worthy, again, is that one attempt failed. This induced extra time spent on investigation of the cause and restarting the deployment. The cause was that downloading artifacts did not complete fully, making it impossible to unpack one of the artifacts in one of the steps, causing the deployment on the particular machine to fail. The other machines succeeded, regardless of the status of the failed machine. For the sake of the time measurements, this attempt had to be restarted. The extra time spent was not measured and is therefore also disregarded here but it is discussed in chapter 5. Failed attempts are noted in the table using ‘*’-symbols, one for each failed attempt. The failing machine is noted similarly.

Table 4.8: Time elapsed during automated deployment with Azure DevOps.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Attempt 1 *</td>
<td>0.02.32 *</td>
<td>0.07.09</td>
<td>0.08.38</td>
<td>0.03.18</td>
<td>0.01.48</td>
<td>0.04.40</td>
</tr>
<tr>
<td>Attempt 2</td>
<td>0.02.16</td>
<td>0.09.07</td>
<td>0.07.55</td>
<td>0.03.15</td>
<td>0.01.43</td>
<td>0.04.31</td>
</tr>
<tr>
<td>Attempt 3</td>
<td>0.02.43</td>
<td>0.06.33</td>
<td>0.08.29</td>
<td>0.03.15</td>
<td>0.02.15</td>
<td>0.04.39</td>
</tr>
<tr>
<td>Attempt 4</td>
<td>0.03.19</td>
<td>0.04.25</td>
<td>0.08.30</td>
<td>0.04.16</td>
<td>0.01.44</td>
<td>0.04.27</td>
</tr>
<tr>
<td>Attempt 5</td>
<td>0.01.30</td>
<td>0.04.36</td>
<td>0.08.36</td>
<td>0.04.06</td>
<td>0.01.38</td>
<td>0.04.09</td>
</tr>
<tr>
<td>Average</td>
<td>0.02.32</td>
<td>0.06.21</td>
<td>0.08.26</td>
<td>0.03.38</td>
<td>0.01.50</td>
<td>0.04.33</td>
</tr>
<tr>
<td>Average per machine</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.01.45</td>
</tr>
</tbody>
</table>

The cost here is, just like with Octopus, since time spent deploying is negligible, the cost of setting up and starting to use the tool and the cost of the tool itself. The start cost is, using the measure of time spent in subsection 4.2.2:

$$24 \text{ hours} \times 32,6 \text{ USD/hour} = 782,4 \text{ USD}$$

The cost of using Azure DevOps is, as explained in subsection 4.1.8, not dependant on the number of machines to deploy to, but rather on the number of parallel jobs. A graph depicting the yearly cost, based on number of such jobs, is seen in Figure 4.8.

4.3.3 Summary

This section summarizes the results from the evaluation phase.

Using sequential manual deployments, the average time spent per machine to update was 0:03:53, at a cost of 1,6 USD per machine per deployment. When doing deployments in parallel, the time was reduced to 0:01:43, which corresponds to a cost of 0,9 USD per machine per deployment. In both cases, problems were encountered that delayed the deployment.

When using Octopus Deploy to deploy, the average time spent per machine to finish deploying was 0:01:25. With Azure DevOps, this time was 0:01:45. This is not the time spent by an engineer, however, which is negligible since it only requires pressing a button. Deployments did fail, however, in both cases, requiring an engineer to spend time to solve the problem, thereby inducing a cost. This time was not measured and is therefore omitted in the results and only discussed in chapter 5.
4.3. Evaluation

The cost is instead calculated from the time it took to set up the tools and the licensing cost of using the tool. Octopus Deploy costs 652 USD to set up and Azure DevOps costs 782.4 USD, both measured in time spent by an engineer setting them up. The licensing cost differs between the two tools, Octopus costing per deployment target, whereas Azure DevOps has a pricing scheme based on number of parallel jobs, making it difficult to make a side-by-side comparison of the two. Octopus Deploy has a linear growth of cost in relation to number of deployment targets if using the SaaS version, and a less-than-linear growth of cost in relation to number of deployment targets if using self-hosted. Azure DevOps is free if one parallel job is sufficient, followed by a linear growth of cost in relation to number of parallel jobs, not restricted by number of deployment targets.

Figure 4.8: Yearly cost of using Azure DevOps based on number of parallel jobs.
This chapter discusses the results and the method used. It also discusses the work in a wider context, such as ethical and societal aspects.

5.1 Results

This section discusses the results from chapter 4. The discussion is divided into subsections discussing the different phases of the study in chronological order. Referrals are also made to related subjects from chapter 2 to support the discussion.

5.1.1 Pre-Study & Implementation

A noticeable result from the examination of the field of CDL tools is that tools specifically designed to handle .NET applications in a Windows environment are also the tools that ultimately were successful. Octopus Deploy, while also applicable on other platforms, is built using .NET and highlights itself being suited for .NET deployments. All applications from Octopus are also only for Windows machines. There are also plenty of templates to facilitate such deployments. Azure DevOps, on the other hand, is a Microsoft product, which naturally helps any tasks designed for the Windows platform. Useful templates for such .NET deployments that were conducted were also provided in Azure DevOps. What is important to point out regarding the other alternatives that were explored is that while they were disregarded for not meeting some requirement, it is not necessarily so that they do not meet the requirement. That which is the result of the pre-study is what was discovered during the study. It might be possible to develop integrations with TeamCity, or find them somewhere that was not discovered at the time of the study. In the case of GoCD, it seemed possible to perform the tasks required in the project but it was simply too hard to succeed in the time frame of the study. A developer with more experience might have been able to succeed, however.

There were also some tools that were not examined further because of requirements of Unix platforms to install software. Such candidates may be excellent but would also require installation of, management of, and learning to use new environments. An advantage, however, is that such operating systems come in free and open source variants such as Ubuntu and Linux, which are also often supported in many cloud platforms.
Another thing that can be discussed regarding implementing CDL is that it requires a solid foundation of for example CI in the organization. Some of the studies from subsection 2.2.1 also indicate this, stating the importance of CI, automated testing, automated deployment and SCM to facilitate CDL (and CDP). The results of this study also indicates that it must be considered which tools that function well with the current environment and architecture. It might be required to change or modify the current setup to enable CDL. It might also be required to re-architect existing systems to be able to deploy them, which was also presented in subsection 2.2.1.

There is also a noticeable difference in pricing schemes between the two selected tools. Octopus is simply priced based on how many agents you install and register with the central server, while Azure DevOps is priced more on your usage. This might make different tools more suitable for different types of software systems. A company that mainly develops applications where all users use the same system, for example a web service, might benefit more from using Octopus since there are potentially few instances that will require deploying to. Depending on the use case, this might be more cost-efficient compared to using jobs in Azure DevOps, which might start costing money if many parallel jobs are required, which might be the case if there are many applications that are developed in parallel by multiple teams (regardless of the number of instances). In the case of few applications with many instances per application, the other way around might be more cost-efficient. The comparison of the cost of the tools is discussed further in subsection 5.1.2.

5.1.2 Evaluation

This section discusses the results from section 4.3. The discussion is conducted chronologically in relation to that section before broadening the discussion with comparisons between different results and other applicable theories.

Baseline Cost

From subsection 4.3.1, we find the results from doing manual updates. These manual updates were done in two ways, providing a good measure on how much time it may take. First, there were some issues encountered when doing updates, which was noted in Table 4.3 and Table 4.5. These were allowed in the study to better display life-like scenarios. This also highlights a disadvantage with manual deployments, the disturbances recorded in these attempts would in at least two of the cases be avoided with automated deployments. It is also remarkable that the deployment times vary a lot, both with a certain machine but also between machines, particularly in the sequential attempts. The variation between machines is probably since they are located at different geographical locations with different stability and bandwidth on their internet connection. The variation within attempts on a certain machine is probably also due to stability of internet connection varying between attempts. In the sequential deployments, the best case, worst case and average case were reported. It is however, as can be seen, not very probably that these cases will occur due to the uncontrollable nature of the variations in the deployment times. They are still useful for providing a range on the cost, perhaps mostly the worst case, since they allow for budgeting, but the average case is the best measure since it includes the variations.

Looking at the table of parallel deployments, we can see that the average time spent per machine (0:01:43) is even better than the average time spent in the best case when doing sequential deployments (0:02:55). This should indicate it to be the superior method. One of the issues encountered when doing parallel deployments, however, was that which step that had been accomplished was forgotten. It is possible that this issue is something that scales with number of instances to update, making it harder to keep track of what is done where when doing more and more things in parallel. The cost per machine, on the other hand, is more than twice as high when comparing the feasible average case cost from sequential
deployments with the cost of parallel deployment. If this extra cost is significantly higher in absolute terms, the parallel deployment strategy is the only feasible strategy for manual deployments and the one that should primarily be used as baseline for comparison with the automated ones (at a 0.9 USD cost per machine per deployment). When doing this comparison, it is important to remember that this issue of forgetting steps exists.

There is also a need to consider the feasibility of updating as many as hundreds or even thousands of systems manually. The most efficient method, parallel deployments, would require more than a day and night in manhours to finish for 1000 machines. Even if this is arguably feasible, it is unlikely that engineers would accept such a tedious and repetitive task.

Using CDL Tools

From subsection 4.3.2, the first part shows the results of deploying using Octopus Deploy. The results show a small improvement in average deployment time per machine, from 0:01:43 in the manual case to 0:01:25 with Octopus. The individual deployments are not even faster in all cases, for example Machine C goes from 0:05:48 on average to 0:07:02 on average. The internet connection is once again the factor that probably influences these times the most, since transfer of the files to deploy is still a required step. The reason the average per machine is lower is most likely due to being able to parallelize fully. When using parallel manual deployments, there is still time lost in commencing all deployments. As already mentioned in the results, the time spent is not equal to the cost here, since the engineer only presses a button to start the deployments. The cost is instead incurred in setting up the tool and using the tool. There is, however, another possible cost that surfaces when a deployment fails, as was the case three times during the attempts with Octopus. In all three cases, it was due to the polling agent not polling when required. The error required restarting the agents on the affected machines. This task may be of varying complication, depending on the environment. Since the error lies with the software developed by Octopus, and appears unpredictable, this may of course affect any company aiming to move towards CDL. What needs to be considered, however, is that automating deployments with CDL tools should enable more control over the deployment procedure. As reported by Chen [8], a CDL pipeline is repeatedly tested and as such more reliable. It also provides means for easier rollback of faulty updates, as well as easier ways of finding where and why deployments failed. The role of failed deployments should as such not be overplayed.

To go back to the cost, it is presented as a start-up cost and cost of usage. The start-up cost is, of course, dependant on who sets up the tool. The result in this study can be seen as a decent measure of the setup cost for an engineer with little previous experience in similar tools, if the system to deploy is of similar size and complexity as the one used in this study. The cost of using the tool is, as can be seen in Figure 4.6 and Figure 4.7, independent of number of deployments. It is only related to the number of machines. It is also obvious that it is cheaper to use the software on your own machines, but it is necessary to remember that using your own machines requires purchasing hardware and administering the machines running the software.

Using the cheapest way to deploy manually, in parallel, the cost per deployment per machine is 0.9 USD. A table showing how many deployments that need to be done to make up for the cost of setting up Octopus (up until only one deployment is required) can be seen in Table 5.1. If you have one machine, you need to deploy twice per day for an entire year to make up the start cost. If you have 30 machines, you only need to update once every two weeks for a year to make up the cost. With as many as 100 machines or more, the setup cost is quickly reached and can basically be considered negligible. At 1000 machines, even though probably unfeasible to update manually to begin with, only one deployment is required to make up for the cost.
Table 5.1: Number of required deployments to make up for initial setup cost using Octopus Deploy.

<table>
<thead>
<tr>
<th>Machines</th>
<th>Average cost per manual deployment (USD)</th>
<th>Setup cost Octopus (USD)</th>
<th>Deployments required</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0,9</td>
<td>652</td>
<td>725</td>
</tr>
<tr>
<td>5</td>
<td>4,5</td>
<td>652</td>
<td>145</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>652</td>
<td>73</td>
</tr>
<tr>
<td>20</td>
<td>18</td>
<td>652</td>
<td>37</td>
</tr>
<tr>
<td>30</td>
<td>27</td>
<td>652</td>
<td>25</td>
</tr>
<tr>
<td>40</td>
<td>36</td>
<td>652</td>
<td>19</td>
</tr>
<tr>
<td>50</td>
<td>45</td>
<td>652</td>
<td>15</td>
</tr>
<tr>
<td>75</td>
<td>67,5</td>
<td>652</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>90</td>
<td>652</td>
<td>8</td>
</tr>
<tr>
<td>150</td>
<td>135</td>
<td>652</td>
<td>5</td>
</tr>
<tr>
<td>300</td>
<td>270</td>
<td>652</td>
<td>3</td>
</tr>
<tr>
<td>1000</td>
<td>900</td>
<td>652</td>
<td>1</td>
</tr>
</tbody>
</table>

Besides this setup cost, there is also the cost of using the tool. A table showing the number of deployments required to make up for the cost of using the tool can be seen in Table 5.2. The number of machines included in the table are the ones where the cost changes for self-hosted (based on Table 4.1). Reading from the table, it can be seen that to make up for the hard cost of using and setting up Octopus, one to two deployments per week, depending on how many machines are involved, is required.

Table 5.2: Number of required deployments to make up for licensing cost using Octopus Deploy.

<table>
<thead>
<tr>
<th>Machines</th>
<th>Self-hosted yearly cost per target (USD)</th>
<th>Deployments per year required for self-hosted</th>
<th>SaaS yearly cost per target (USD)</th>
<th>Deployments per year required for SaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>84</td>
<td>94</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>40</td>
<td>82</td>
<td>92</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>100</td>
<td>72</td>
<td>80</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>150</td>
<td>70</td>
<td>78</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>300</td>
<td>68</td>
<td>76</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>1000</td>
<td>63</td>
<td>70</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>1500</td>
<td>62</td>
<td>69</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>2000</td>
<td>58</td>
<td>65</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>2500</td>
<td>56</td>
<td>63</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>3000</td>
<td>54</td>
<td>60</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>4000</td>
<td>52</td>
<td>58</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>5000</td>
<td>50</td>
<td>56</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>7500</td>
<td>45</td>
<td>50</td>
<td>108</td>
<td>120</td>
</tr>
<tr>
<td>10000</td>
<td>42</td>
<td>47</td>
<td>108</td>
<td>120</td>
</tr>
</tbody>
</table>

The second part of subsection 4.3.2 shows the results of deploying using Azure DevOps. Remarkably, the average time per machine is two seconds slower compared to deploying manually. Just like the case with Octopus, however, the time it takes to deploy is not under examination here since the engineer does not need to interact beyond the press of a button. The results in using Azure DevOps is, otherwise, very similar to that of Octopus. The difference here lies in the failed attempts. The failed attempts, as explained in the results, were due to the transfer of the artifacts partly failing, making it impossible to unpack the packages. The same arguments applied when discussing the attempts with Octopus can be applied here as well, that the control of and feedback regarding deployments that is made possible should compensate for the occasional failed deployment due to these factors.
Once again, the cost is presented as the start-up cost and cost of usage. The start-up cost is again dependant on who does the work and the complexity of it, but can be seen as a decent measure of the cost for an engineer with little experience in doing such work. Figure 4.8 shows the cost of using Azure DevOps, and is also independent of the number of deployments. It is, however, not like Octopus, as it grows with how many parallel jobs you want to run, not to how many targets you want to deploy.

First, a table showing how many deployments that needs to be done to make up for the start-up cost is seen in Table 5.3. Since the start-up cost is a little higher here, more deployments are required. With a sufficient number of machines, however, it is also rather negligible and the number of required deployments are also very similar.

Table 5.3: Number of required deployments to make up for initial setup cost using Azure DevOps.

<table>
<thead>
<tr>
<th>Machines</th>
<th>Average cost per manual deployment (USD)</th>
<th>Setup cost Azure DevOps (USD)</th>
<th>Deployments required</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.9</td>
<td>782.4</td>
<td>870</td>
</tr>
<tr>
<td>5</td>
<td>4.5</td>
<td>782.4</td>
<td>174</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>782.4</td>
<td>87</td>
</tr>
<tr>
<td>20</td>
<td>18</td>
<td>782.4</td>
<td>44</td>
</tr>
<tr>
<td>30</td>
<td>27</td>
<td>782.4</td>
<td>29</td>
</tr>
<tr>
<td>40</td>
<td>36</td>
<td>782.4</td>
<td>22</td>
</tr>
<tr>
<td>50</td>
<td>45</td>
<td>782.4</td>
<td>18</td>
</tr>
<tr>
<td>75</td>
<td>67.5</td>
<td>782.4</td>
<td>12</td>
</tr>
<tr>
<td>100</td>
<td>90</td>
<td>782.4</td>
<td>9</td>
</tr>
<tr>
<td>150</td>
<td>135</td>
<td>782.4</td>
<td>6</td>
</tr>
<tr>
<td>300</td>
<td>270</td>
<td>782.4</td>
<td>3</td>
</tr>
<tr>
<td>1000</td>
<td>900</td>
<td>782.4</td>
<td>1</td>
</tr>
</tbody>
</table>

The number of deployments required to compensate for the cost of using the tool is dependent on how many parallel jobs you need as well as the number of deployment targets. A table with these numbers can be seen in Table 5.4. The calculations are based on the costs shown in Figure 4.8 together with the 0.9 USD cost per manual deployment per machine. For example, using two jobs (costing 180 USD per year) with 5 machines (0.9 * 5) requires 40 deployments to make up for the cost of using the tool. Since the cost is not dependent on the number of machines in this case, it starts to get rather costly to use the tool if you pay for many parallel jobs to few machines. This is probably not a relevant use case anyhow and the tool therefore scales well starting with rather few machines.

Manual vs Octopus Deploy vs Azure DevOps

First, comparing the methods of deployment to each other, we can see that Octopus finishes the task fastest, followed by doing it manually, with Azure DevOps being the slowest. When taken from the perspective of cost, Azure DevOps pays of within a year if deploying once a week to 20 or more machines, then being free when using only one parallel job, as is probably feasible in a small company. Octopus, on the other hand, requires deploying more than twice per week to 40 or more machines to pay of the initial cost, but also requiring to uphold this rate to continue paying off. Both CDL tools experienced issues with failing deployments, but manual deployment also experienced issues with for example IIS that the automated ones did not. Furthermore, a machine does not forget to perform steps that might be mandatory but easy to miss and hard to detect. A human might do.

Another aspect is the feasibility of updating many machines. It is already a question whether it would be accepted by engineers to take on the task of manually updating more than 20-30 systems. With this comes also the feasibility of updating many systems often.
Table 5.4: Number of required deployments to make up for licensing cost using Azure DevOps.

<table>
<thead>
<tr>
<th>Machines</th>
<th>Deployments per year required with 1 job</th>
<th>Deployments per year required with 2 jobs</th>
<th>Deployments per year required with 3 jobs</th>
<th>Deployments per year required with 10 jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>40</td>
<td>80</td>
<td>360</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>45</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>150</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>300</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1500</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2500</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7500</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

If it takes two full working days to update all systems, it is practically impossible to be both developing the systems and deploying them on a weekly basis. This is an argument that using CDL forces companies to be more agile, releasing frequently and using the learned lessons to improve their products and methods. This is also presented as a benefit by studies brought up in subsection 2.2.2. It is important to, besides looking at the measurable costs, also look at other benefits that come with CDL. Effects such as reduced stress for employees that handle deployments, higher customer satisfaction from faster feedback cycles, higher reliability in deployments, all these helping increase product quality, must be considered when deciding whether to implement CDL.

One must, however, also consider the challenges that may come with moving towards CDL. Some of the challenges brought up in subsection 2.2.3 were encountered in this thesis work. For example, network dependencies and configuration had to be considered. If a tool is not designed with on-premise installations in mind, and there is no control over the routing of requests within that network, the agent will not be reachable. This is solved with the two evaluated tools in this project with the polling mechanism. Octopus also had the possibility to have listening agents, which were tested at first, but they were not suited for the environment under test. There are many other challenges that are not technical in nature: resistance from management; customer unwillingness; requirements on the organization and developers to change or evolve. The results of the study might give an incentive to pilot a project to at least implement and evaluate CDL on a subset of the company, since the start-up cost is surmountable for most companies.

5.2 Method

This section discusses the method used and the study itself as a whole. The first part discusses the three phases of the study, including choice of tools, the setting of the study and the evaluation. Some discussions on the replicability, reliability and validity of the study is also included. The second part discusses the sources used in the study.

5.2.1 Pre-Study & Implementation

There are several things that may impact the results. First, the subjective experience of the person conducting the study will affect how the tools are conceived during the pre-study. What tools that are deemed too hard to use will of course depend on who is using it. The time limit that was set to get a reasonable way to know when to stop trying may be plenty for some developers and close to nothing for other developers. It is also not guaranteed that
all requirements that were deemed unmet were in fact impossible to meet for the tool. It may instead be a matter of skills and experience in the user of the tool. Another aspect is that the study may have missed some tools due to lack of knowledge in the field. Extensive searches on the internet were made, but it is always possible that some tools are not the most advertised, particularly if they have a smaller market share, which may not mean that the tool is unfit for the task.

The informal discussions held to get the list of requirements, even if sufficient in this case, may be too imprecise to allow for easy replication of the study. A more formal approach may be preferred, using a checklist-based structured or semi-structured interview for example, to allow streamlining of requirement solicitation.

More strict requirements on the actual deployment process may also be imposed. For example, if downtime is an important factor in the decision on whether to adopt CDL, the deployment should probably constitute the entire process, including any shutting down and starting of all software processes.

5.2.2 Evaluation

There are several things that can be discussed here. First, the cost of an engineer can be different in reality compared to the cost used in this study. Regional aspects must be considered, since engineers cost differently in different parts of the world. There are also other costs included, for example taxes paid by the employer. Since these costs vary a lot, the choice was made to not include any of these, but the cost can due to these factors be as much as fifty percent higher.

Since the person doing the work in this thesis can be considered a junior developer, it can be discussed whether junior salaries should be used in the study. It is, however, not guaranteed that the person doing manual deployments is a junior developer. The task might be placed upon any engineer, with any type of salary. It is also not necessarily so that a person with little experience in implementing CDL is also a junior developer. A developer that has worked for 20 years in the industry might simply have worked with other things, but still need to do the work done in this project. All of this makes using the average salaries of all engineers a reasonable choice.

When doing measurements on manual updates, there are also factors that can affect the measurement. First, there is an inherent risk that the person performing the manual updates is not representative for an average updater of the system. This is mitigated in this study since the person updating is the same as in the real case. Another problem is the learning factor when doing repeated measurements. It is noticeable that the manual deployments become faster after the first or second attempt. This is partly representative. In the real case of updating, the first couple of machines may meet obstacles in the procedure, while the rest of the machines are subject to this same learning factor. Since it may matter which machines are treated to which type of update, this can be seen as a disturbance in the study.

When the measurements were made might also have been a factor and could have been considered more. Since internet traffic might affect transfer speeds, and as such also deployments, both manual and automated. Since the nature of an unstable internet connection means that it varies not only over longer time spans such as a day, but also within minutes, this should not have affected the results that much. Doing the required work during the night might have had some effect since overall internet traffic is less during that time. This is not really a normal use case, however, since manual deployments are made during working hours, not during the night.

The start-up cost of using the tools is also, of course, a very subjective measure, highly dependent on the person performing the setup. This should be repeated using different persons to get a better measure on the average cost.

With better free-tiers or resources to purchase licenses, the study could also be performed on more machines, providing a better average measure in all measurements that were made.
5.2.3 Replicability, Reliability, and Validity

The replicability of the method could be higher. There are things that keep the replicability rather low. Mainly it is the fact that the context is very specific. Different software companies with other types of systems may make the list of requirements different, make the start-up cost much higher or lower and ultimately have completely different outcomes. Another consequence would be that the measurements would potentially contain completely different contents in terms of steps to perform and how the automated deployments look like. The involvement of a person doing many of the parts of the study is also a factor that reduce the replicability.

The reliability is, following the discussion on replicability, also potentially low. In the circumstance that the context can be replicated, the reliability should be high, given that the person involved in the study is replaced with a person of similar skills. Exactly the same results may not be possible to achieve, but results within a reasonable error of margin would be expected in the case of replicated context. Since this is nothing that can be fully expected in the case of repeated study, the reliability as a whole must be considered moderate at best.

The validity can be discussed in relation to different parts of the study. The measurements of time spent on updates measure exactly that, and used in coherence with the cost of spending time, they provide a measure of the cost of performing updates. Since the full measure of benefit in the cost-benefit analysis depends on more factors than just the cost, it is explicitly stated that this time measure is not sufficient to be decisive. This helps the validity of the study since it does not try to state conclusions that are unsupported. The modularity of the different parts of the study, consisting of separate measures that hold their own, and when brought together all make part in describing a whole, also makes the study more credible. There are the time measures, the calculations of the cost of engineers performing deployment related work, the cost of engineers setting up tools, the cost of tools themselves and theory from other studies on challenges and benefits outside of these measures. All of these components, each valid on their own, help support the validity of this thesis as a whole.

5.2.4 Sources

There are several types of sources used in this thesis. First, we have published, peer-reviewed, scientific papers. Among these, there are both primary studies (e.g. [13, 14]), as well as secondary studies (e.g. [7, 10]). The primary studies bring forth interesting results on their own, such as how CDL impacts performance in an organization. The secondary studies instead synthesize many other studies to provide an overview, for example on the most prominent benefits in adopting CDL. The papers are mainly conference papers, providing confidence in their content. Some sources are published in journals. Two of them ([8, 15]) come from a journal that may more or less be considered popular science journals. In the case of these two articles, one provides a practitioners report on the consequences of implementing CDL while the others provide an overview from interviews with companies that adopt CDL to some extent. While not claiming to be particularly scientific, they still provide good insights that add value as sources in this thesis as practical and subjective cases are also important to understand the effects of CDL.

There are also some articles, published simply on web pages, that are used in the thesis ([1, 2, 9]) as these are considered foundational in their fields, proved by their many citations in numerous published papers.

There are also blog posts used ([4, 12]), and they provide meaningful help in defining hard-to-define terms and roles.

Finally, the source on the salaries of Swedish engineers in software industry ([25]) uses the results of a survey that is only available to the members of the Swedish Association of Graduate Engineers. The source requires logging in as a member to get access to the results. The results are also dependant on the time of the study and subject to change every year.
5.2.5 Alternate Method - Cloud Migration

The alternate method discussed in section 2.3, cloud migration of on-premise solutions, is also a feasible approach to be considered. Depending on the system, it might be a sound thing to do regardless, given the benefits in subsection 2.3.3. It is important that the architecture of the system is fit for the cloud. If it is not, there might be need for major re-architecting of the system, which would cost a lot. If cloud migration is not a major undertaking, it facilitates CDL by providing control over the systems and resources. Depending on the system, it might require only updating few instances. Depending on that, which type of CDL tool that best fits the company, might be different compared to before cloud migration. If required deployments are few, CDL might not be necessary at all. In the context of this project, cloud migration was never a feasible approach. The system is dependant on large quantities of data to be solicited at the customer site through thermal cameras. To stream this data to the cloud would be both very costly, but also impractical, since the internet connections of the customer sites are of varying quality.

5.3 The work in a wider context

This thesis contributes by evaluating the use of CDL tools to enable automated frequent deployments of software. One aspect to this is the energy consumption that comes with more frequent deployments. In larger scales, the energy used to deploy software products may be substantial. It may at least be substantial in relation to not enabling automated deployments. A company may go from deploying once a month to several times per day, potentially to hundreds or thousands of machines. The resources used by the machines performing the deployments is something that should be considered when planning for implementations as the ones in this study. Perhaps a cost-benefit study in the context of energy consumption should be recommended to evaluate the footprints of a company on the planet.

Another aspect that may need consideration is the concept of canary deployments. Canary deployments makes new versions of software available to selected subsets of users before making them available to all users. The ethical consideration that can be discussed is whether it is fair that paying customers experience potentially inferior software as some sort of test subjects. With reasonable mitigation, this may not be an issue, particularly if such procedures are described in contracts and licenses. It is, however, something that should not be done without consideration.
This thesis aimed at evaluating the cost-related effects of implementing CDL with the use of CDL tools. The research question to be answered was:

• What are the cost-related effects of implementing Continuous Delivery at a small software company, in relation to the number of deployments made per year by the company?

Compared to the expected results in section 1.5, the actual results were similar. First of all, with some tools, the licensing cost induces a requirement to deploy often to be cost efficient compared to deploying manually. If there are few deployments, and the number of machines is feasible to deploy to manually, the pure cost benefit of using CDL does not exist. If there is a need to deploy at least one or two times per week or more, the cost of the tool is paid off, even for few machines. With other tools, the pricing allows cost pay off rather easily. If one or two parallel jobs is sufficient, the licensing is paid of either immediately or with weekly deployments, even for rather few machines. If more jobs are required, this probably means you have many machines as well, and the case is not worsened.

In both cases, other benefits that are not measured concretely in terms of cost, are probably included and also make up for any risk of failure during automated deployment. Such benefits are: reliable deployments with good rollback procedures; faster feedback from users and customers; better working environment for engineers; and higher product quality.

Other challenges that must be considered when choosing whether to use CDL and which tool to use are: varying network dependencies and configuration; how to meet resistance from management and existing work culture; and requirements on infrastructure and environments.

6.1 Future Work

There are a couple of things that would be interesting to look at for future work. First, the study could be conducted as a multiple case study, conducted at several companies, to better study the effect of moving to CDL in different contexts, perhaps providing a more generalizable result. It could provide a better help for practitioners when considering to implement CDL. Another reason for studying different contexts is that it allows for more CDL tools to
be implemented and evaluated, since different contexts come with different types of systems on different platforms. Such a study can also be done from a different perspective. If time is an essential factor, for example if downtime must be reduced, a similar study can be done to compare which tools are the most efficient to minimize downtime. Since many systems are hosted in public cloud platforms today, it might also be interesting to study which platforms best support CDL, or which tools work best for a particular public cloud platform in terms of cost, time or some other measure.
Bibliography


