Aalto University
School of Electrical Engineering
Degree Programme in Automation and Information Technology

Antti Paloposki

# Enabling Continuous Integration through deployment automation

## Case Study: Property transaction system of Finnish National Land Survey

Master's Thesis
Espoo, January 17, 2018

**January 17, 2018**

Supervisors: Professor Valeriy Vyatkin, Aalto University
Advisor: Tatu Kairi M.Sc. (Tech.)

Aalto University
School of Electrical Engineering
Degree Programme in Automation and Information Technology

ABSTRACT OF MASTER'S THESIS

**Author:** Antti Paloposki

**Title:**
Enabling Continuous Integration through deployment automation Case Study: Property transaction system of Finnish National Land Survey

| | | | |
|---|---|---|---|
| **Date:** | January 17, 2018 | **Pages:** | 42 |
| **Major:** | Information and Computer Systems in Automation | **Code:** | T-110 |
| **Supervisors:** | Professor Valeriy Vyatkin | | |
| **Advisor:** | Tatu Kairi M.Sc. (Tech.) | | |

Finnish National Land Survey has commissioned Eficode to develop a service called Property Transaction Service for handling property issues electronically. As part of the development work, Eficode set up automated acceptance testing in a dedicated acceptance test environment but there was no deployment automation that is required for proper Continuous Integration and only time when new versions were deployed was the dedicated acceptance test period for release candidate.

Scope of this thesis was to implement a deployment automation which would enable frequent and effortless deployments to the acceptance testing and development environment using an IT automation tool called Ansible. The result was an automated deployment process that released new versions for acceptance testing with very little input from the developers. Research and industry consensus both support the assertion that frequent automated deployments with automated testing improve software quality and increase predictability in software projects. In this thesis it is demonstrated that implementing a deployment process that is as far automated as possible will significantly increase the frequency of deployments without comparable investment in workload.

**Keywords:** Continuous Integration, Software Development, Ansible, Devops

**Language:** English

# Acknowledgements

I wish to thank both Eficode and the National Finnish Land Survey for the opportunity to write my thesis for them. I would like to thank my instructor Tatu Kairi for his extensive feedback and my supervisor Valeriy Vyatkin for directing the thesis work.

I would also like thank Mei, my friends, colleagues and family for the encouragement and support they have given me throughout this final stretch of my studies.

Helsinki, January 17, 2018

Antti Paloposki

# Abbreviations and Acronyms

| | |
|---|---|
| NLS | National Land Survey |
| PTS | Property Transaction Service |
| CI | Continuous Integration |
| DNS | Domain Name System |
| NFS | Network File System |
| WAR | Web Application Archive |

# Contents

# Chapter 1

# Introduction

Eficode Oy (later Eficode) has been developing a property transaction service for the Finnish National Land Survey in order to fulfill the Finnish Land Code requirement for an electronic property transaction that is open for every Finnish resident wishing to conduct their real estate business. Purpose of this thesis is to measure how application deployment automation enables and affects the level of Continuous Integration and what are the implications in terms of software quality and time savings.

## 1.1 Problem statement

Changing the architecture of the product under development from monolithic to microservices complicated the deployments of the service and created a need to review the deployment process that had become a time consuming task. Since product is running on six different environments and consisted of eight different services, releasing new version used up a lot of time that was away from development work. Complicated release also meant that new versions would be released to acceptance testing environments at relatively slow intervals and automated tests in user acceptance testing environments would be running old versions of the software, making them redundant.

In the optimal case, every time new commits are added to development branch of the version control, new version of the product would be packaged and shipped for final quality assurance on the condition that automated acceptance tests developed by Eficode would pass. By automating the deployment process as far as possible, development team would have more time to develop new features, extend test coverage and implement other technical improvements. Also any problems that might have gone unnoticed when specifying new features would be visible faster with fast release cycle.

Aim of this thesis is to give an overlook on the development practices of a complicated, modern web software product, identify potential improvements and present one example of a practical technical improvement to decrease the manual workload of developers and improve the development process. The research question of the thesis can be summarized as:

**How does deployment automation affect the software development process**

To measure the effects of deployment automation we observe two metrics

**Metric 1:** How much faster does the cycle of deployments get?

**Metric 2:** How much working time is saved with automated deployments?

## 1.2 Structure of the Thesis

In Chapter 2, an outlook of Software Development methods, Continuous Integration and Deployment as well as Configuration management is presented to give some background information as to why deployment automation matters and why automatically triggered deployments are useful. Following that, there is a more accurate technical description of this particular project and current infrastructure setup which has partly created the demand for deployment automation. In this section there is an overview on the software development practices to get the reader introduced on the processes that guide software development and thus prepare them for the practical section as well as evaluation. An IT automation tool called Ansible is also presented to give information on how the new release automation works and what kind of things it enables us to do.

Chapter 3, the Environment gives a detailed description of the system currently under development, it's infrastructure, architecture and information about motivations for National Land Survey to develop the product and weaknesses in the current process that the practical work in this thesis attempts to fix.

The practical part in Chapter 4 of the thesis will be presented in implementation section, which presents the previous process of releasing a new software and what kind of changes were made to it. Practical part outlines how the previously presented product and it's configurations can be automated to provide faster and easier deployments.

In the Chapter 6 , discussion goes through the additional improvements that could be made to reach the goals that National Land Survey wants to

achieve with this product, including the development process from specifications to release as well as technical and cultural improvements. Finally in Chapter 7 there is a conclusion giving an outlook on what was done, why it's important for this type of software project and how did it change the process of releasing new software.

# Chapter 2

# Background

Development of a large application with multiple developers requires practices and ways to organize work in order to maintain control over the complexity of the project. In the software industry, these methods typically borrow from Agile movement and are referred to as frameworks, which project managers and teams can apply to retain control over their products and conflicting demands for its development from different stakeholders. Typical large software project follows some of these frameworks, such as Scrum, which is an iterative and incremental framework designed to optimize predictability and control risk [21]. Agile itself is a mindset that was summarized in Agile manifesto [2] as following principles:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over comprehensive documentation
- Responding to change over following the plan

Agile methods were created to mitigate the real and perceived weaknesses in conventional software engineering and waterfall method, but are not applicable to all projects [20]. The motivation with adopting Agile methods was that predicting the future developments of a complex software product is hard and sometimes impossible, therefore it should be Agile in order respond to changing business environment and requirements. Agility in this case means ability to effectively respond to changes, by trying to deliver operational intermediate versions of working software in short intervals. It also recognizes that human planning is prone to error and should therefore not be trusted too much. Any long-term plan will probably change before the project is completed, so project plan should be flexible from the start. A

number of topics related to modern application development is listed under this topic to give the reader an introduction to the practices relevant to the project related to this thesis. In this kind of methodology, it is recognized that making small, frequent, incremental changes and quickly fixing the failing build and tests during every change is crucial to achieving high quality and fast delivery of desired features. [26]

Applying Agile methodology into practice requires several practical improvements that have developed over time as the industry best practices for managing complexity and implementing the Agile software development ideas into practice and ways that software development teams build new software [27]. The most important ones of these practical improvements are listed as their own separate sections in this Chapter to give an idea of the overall infrastructure and tools needed for a team developing complex software.

## 2.1 Continuous Integration and Delivery

Continuous Integration was developed to counter the problems of sequential software development model. After the requirements, analysis, design and implementation stages were finished, substantial problems were often met at the integration stage of software project [23]. When large amount of changes is brought into a complex system, the more likely it becomes that parts of the system either will not work together as intended, or cause unexpected behavior. These problems are difficult to fix and often require large rewrites of code that was already supposed to be finished. As the number of changes and interdependent services grow, the more difficult it gets to integrate them to work as intended.

The process that eliminates a dedicated integration stage in software development is usually referred to as pipeline. This is the process of doing integration work continuously where all changes in the version control will be submitted into a process referred to as deployment pipeline. When the deployment pipeline detects new commits in the development or any other user defined repository, automated testing and deployment process will begin to ensure feedback is received swiftly and problems with integrating the related systems will become visible as early as possible, preventing problems. There is no standard pipeline, but typical pipeline will contain automation for build and deployment automation with automated testing. [12]

Continuous Integration an Agile method that increases the responsiveness of project since the functionality of its core parts is maintained at all times. Even if the integration testing succeeds, some functionality might still be missing and product is not necessarily ready for release before acceptance

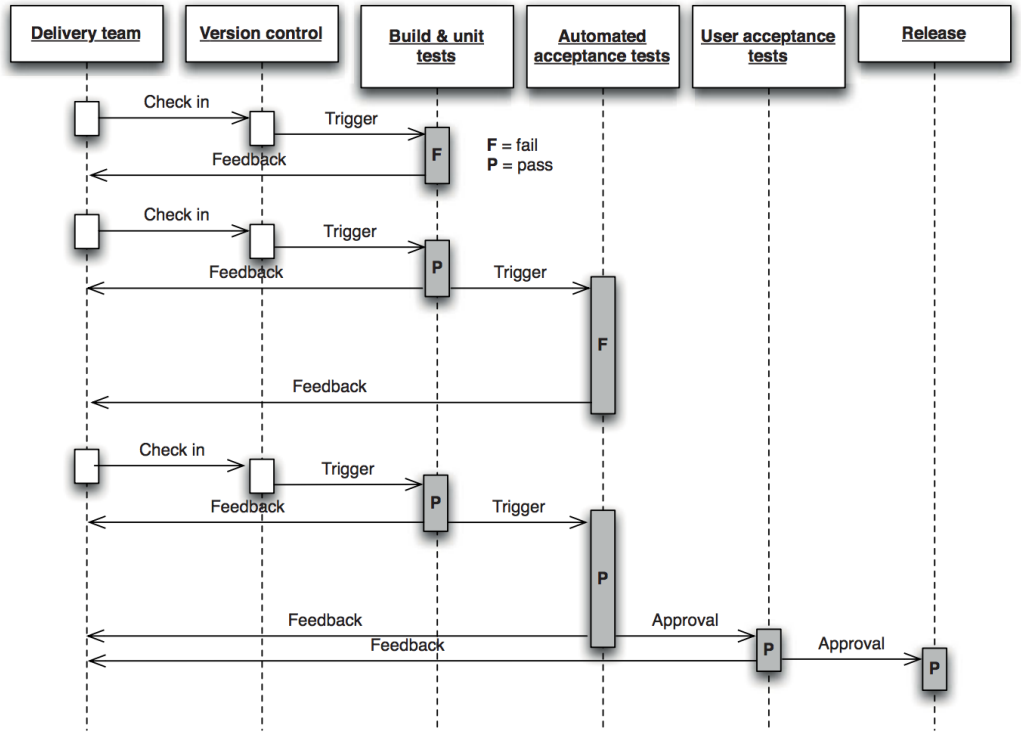



Figure 2.1: Visualization of the process of getting software from version control to release known as Continuous Integration/Delivery pipeline [18]. Feedback at every stage determines whether changes flow downstream. Failure gives immediate feedback for developers, stops the process and waits for for new changes to restart the process.

testing.

Continuous Delivery takes a more extreme attitude towards automating the development and release processes by enabling new changes to be published in production whenever the owner of the product wishes so. The idea here is to automate everything related to releasing software as far as possible so that the human error is eliminated from the process. Continuous Delivery still leaves the option open for human involvement in the quality assurance instead of relying on purely automated tests. In Continuous Deployment, human involvement has been completely eliminated and the deployment process happens based on the feedback of automated steps in the release pipeline.

Both Continuous Integration and Delivery are ultimately about providing value faster for the product owner, improving communication between developers and eliminating the human error related to a new release.

### 2.1.1 Jenkins CI server

When software is built and ran, there are typically multiple steps involved, which can include cloning the right version from version control, building it, provisioning the machines that run the program and deploying the built software in these machines. When implementing Continuous Integration in practice, a Continuous Integration server is needed to coordinate, schedule and trigger all of these tasks. Jenkins is one of the currently used open source automation server in the industry. It enables the implementation of all the steps required for a CI pipeline and provides a simple user interface for modifying the existing CI setup. Once software has been deployed, automated tests can also be triggered from Jenkins server, in order to gather test result and allows users to monitor the statuses of different test suites. [30]

Usually tasks automated by Jenkins are referred to as jobs in the Jenkins user interface. One job is simply a collection of steps on the path to building, releasing and testing software.. A job might also be just something that user wants to easily and repeatedly trigger without manually going through every step required.

Overall a deployment probably consists of multiple jobs, which can be set to trigger based on status of 'downstream' jobs and other conditions, such as time, or change in version control system. This will create what is usually called a pipeline

### 2.1.2 Software Testing and Test Automation

Continuous Integration and Delivery naturally increases the demand for integration and acceptance testing, since new, deployable versions of the software are expected in very short intervals. Since Continuous Delivery should be independent of human action, it requires a highly automated quality assurance, where automated tests should be executed at unit, integration, system and acceptance level.

First layer of automated testing is unit tests to make sure that the modified function or class is still working as intended. When code under unit tests needs to access some other functionality they are provided as mocks or stubs which are explained later in the chapter. This is the most basic level of testing and is usually done before any other type of testing occurs.

Integration testing is done in order to make sure that different software components work together as intended and is performed after unit testing and before acceptance testing.

Acceptance testing is the process of comparing the software functionality against the initial requirements from project stakeholders. It is usually done

when new features are being added to make sure that changes didn't destroy any other required functionality and that new features fulfill the requirements of software.

CI is essential to integration and acceptance testing since the premise of it is to provide feedback from these levels continuously, rather than in a dedicated integration stage. Therefore automation testing at CI-server mostly focuses on the integration and acceptance level testing, while lower level testing is done before publishing the work to other developers.

#### 2.1.2.1 Stubbs and Mocks

Often in development work, developers have to rely on external services to ensure the full functionality of their software. For example, a website might need to access the population registry to ensure the identity of the person who just signed as enterprise user from the national enterprise register and testing this feature requires the developers to be able to use either a test version of the register or a real person tied to a certain company and his social security ID.

Relying on external services can result in impediments if they have service breaks or extra work to compensate for features that are not yet delivered, so instead of relying on external service, it's possible to develop a 'Stub' that mimics the functionality of desired service in the development and test stages and use the real system in the QA stage and once product goes live with actual customers[14]. Mock is a slightly smarter version of Stub that does not only define a predetermined response, but also mimics the behaviour of the object it's replacing in a controlled way. [13]

## 2.2 Microservice architecture

Microservices developed to counter problems related to monolithic architecture, where modules of software cannot be executed independently, but rather as a part of larger subset with other modules as dependencies. This makes monolithic software function poorly in distributed systems, which in turn are required to achieve scalability. These attributes and some other general problems related to monolithic architecture were listed in paper 'Microservices: yesterday, today, and tomorrow' [7]:

- Large-size monoliths are difficult to maintain and evolve due to their complexity. Tracking down bugs requires long perusals through their code base.

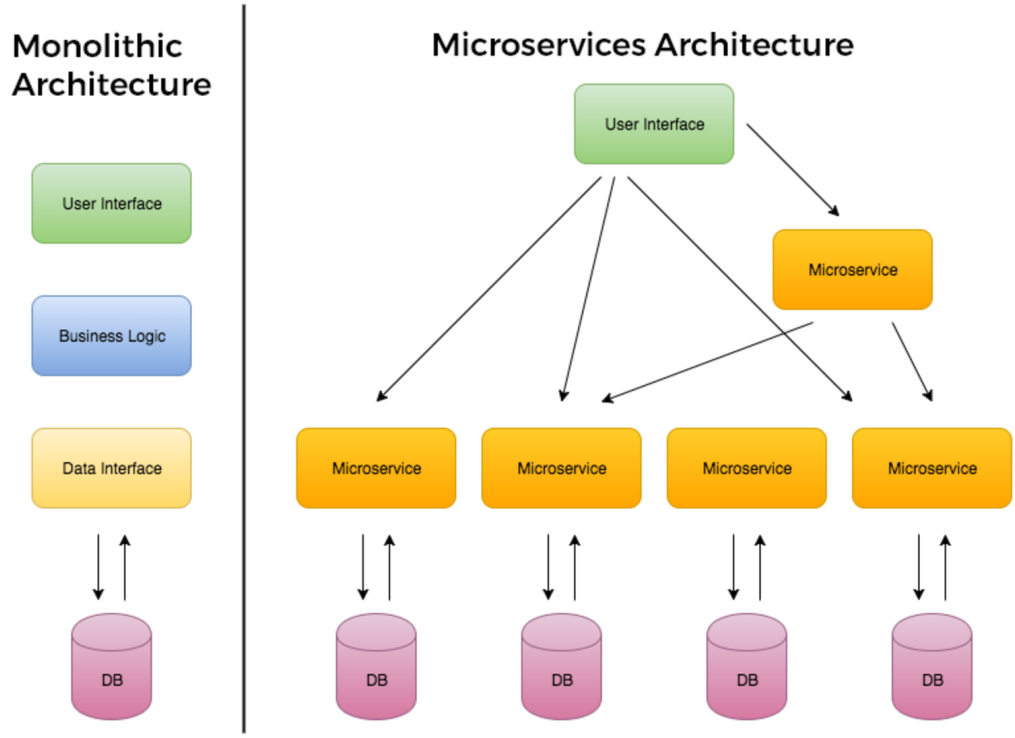
- Problematic dependencies where updating or adding new modules results in inconsistent systems that do not compile, run or misbehaves
- Any change in the system requires rebooting the entire application for changes to take place making it cumbersome to develop new features in an Agile way.
- Lack of scalability. The usual strategy for handling increments of inbound requests is to create new instances of the same application and to split the load among said instances. However, it could be the case that the increased traffic stresses only a subset of the modules, making the allocation of the new resources for the other components inconvenient
- Technology lock in, where all the new features have to be developed using the same language, framework and version as the rest of the application resulting in complex legacy solutions.

Microservice architecture has been an industry trend during the past years to mitigate the problems listed above [8]. A microservice is an independent process that interacts with other processes via some kind of messaging. For example, a food delivery app might have a separate service that keeps track of available restaurants and returns lists of those matching the queries coming from main application.

The way microservices try to answer problems rising from monolithic architecture are [8]:

- A more limited amount of functionality limits the scope where a bug might be originated from.
- Bringing up gradual transitions where a new, improved service can be brought up along the old service.
- Changing a single module does not require rebooting the entire system, but rather just a small part of functionality that can go offline for a limited time. Because of the small size of microservices, reboot time is also decreased, thus increasing availability for end users.
- Microservices are independent of each other and if one service becomes a bottleneck, more instances of it can be deployed to handle the traffic.

These days microservice architecture is a trending topic in software engineering and projects with service-oriented architecture being increasingly migrated to microservice architecture and distributed systems to achieve the benefits listed above [8]. However, in addition to their strengths, microservices have also weaknesses, greatest of these being the communications overhead [9]. When services have to communicate with each other over network,




Example 2.2: Microservice vs. Monolithic architecture [25]

it might cause significant overhead compared to monolithic architecture. Another issue is ensuring interoperability between services, when product gets very complicated with many services communicating with each other over network, a substantial amount of infrastructure is needed just to run tests for the product. Network problems such as latency might create extra cases to take into account in when programming a service, this creates extra complexity into the product. A team using microservices might also run into trouble if the project becomes very heterogenous with each developer preferring to work with his preferred stack and, thus, making the project harder to approach for new developers.

When introducing microservices into a product the importance of automated testing, continuous integration, configuration management and other software engineering processes are highlighted as the integration of different services together becomes more complicated.

## 2.3 Software Configuration Management

Software configuration management is the task of tracking and controlling changes in the software and is part of the larger cross-disciplinary field of configuration management [28].

Automating tasks related to the server configurations and application deployment is a mechanism to ensure that servers reach the state defined by developer consistently and that documentation of the application setup and infrastructure persists in the . The most important benefit of configuration management is to avoid creating setups that are difficult to reproduce [16]. An industry term for this type of problem is 'Snowflake Server', which means that over time after numerous manual updates and configuration changes, user has difficulties in reproducing the server where application is running and there is uncertainty of which configurations are important and why.

One example countermeasure to the configuration drift of Snowflake Servers is using an IT-automation tool to re-sync any used servers to match the configurations any configuration that the automation tool has been set to modify will be consistent with other servers in the same environment, stopping the drift towards Snowflake servers. [15]

In large systems, infrastructure automation also helps to decrease the workload of developers and system administrators as well as eliminates human error from configuration management. In addition to automated tests, automated configuration management is also essential to achieving shorter release cycle and faster feedback from the development work. [10]

Manual releases have been identified as one of the common design problems in software development [17]. The symptoms of this design problem include extensive and detailed documentation for deployment steps where there are multiple chances for mistakes, reliance on manual testing to verify product is working, different configurations for environments, such as connection pool settings, releases take longer than few minutes to perform, frequent rollbacks to older versions because of problems [19]. Some sources have established that manual deployments are sometimes suitable when the service to be deployed is very simple and is deployed to only few nodes in one or two environments [32]. Large software projects consisting of several services that are deployed into multiple different environments benefit from automated deployment system in order to save time and minimize mistakes made during the deployment. [32]

Another design problem is the manual configuration management of the environments.

### 2.3.1 Ansible

Ansible is an open source IT automation tool for configuration management, deployment and orchestration released in 2012. It was designed to be minimalistic, consistent, secure and highly reliable with as low learning curve as possible. [5]

The problem which Ansible was created to address was the lack of robust and easy to manage solution for solving the orchestration, deployment and configuration management problems at the same time. The problem with other solution was the need for configuration management agents in target machines and the desire to create a configuration management tool that would not require maintenance on its own. This solution eliminates the problem of managing the management that persists in tools that use agentful architecture. Like other configuration management tools Ansible differentiates servers into control machine and nodes, but due to its agentless architecture, only the control node needs to have Ansible installed. [29]

State driven resource model of Ansible means that it is goal oriented instead of scripted and thus allows repeatable and reliable IT infrastructure configuration. The declarative play only states the desired end state of node instead of the paths needed to reach the state. This makes Ansible tasks very minimalistic and human-readable. Ansible uses declarative YAML-language [6] to list its tasks and sorts them in different 'plays'. One play typically consists of a large set of tasks associated with some activity and can be broken down to smaller 'roles' which further isolate tasks to their own groups [**?** ]. One task with Ansible might be to stop httpd service, another one to update its configuration file and third one to restart httpd service. Together these tasks could form a role, eg. 'update configurations and restart httpd', that is part of larger play that makes sure all the configurations on application servers are consistent with the latest release. A playbook is always matched with a set of hosts that it's run again and which are defined in Ansibles inventory file. Inventory file is where all the names of relevant applications servers, or hosts, are listed and where they can be grouped as the developer sees fit.

Ansible can be extended with custom modules that allow user to define their own tasks, written in Python, Ruby, Perl, or similar scripting language. Users can create their own custom modules and share them with other Ansible users for increased functionality. At the moment of writing this thesis, modules are grouped as Core, Extra and third-party modules, where Core modules are the ones associated with basic functionality, Extras contain lots of other product-related modules, such as cloud service provider modules and third-party is community generated content.

Currently Ansible is released as an open source software, but it's owned by Red Hat which sells subscriptions to Ansible Tower that includes GUI for monitoring, scheduling and inventory management. [29]

```
inventories/
   production/
      hosts              # inventory file for production servers
      group_vars/
         group1          # here we assign variables to particular groups
         group2          # ""
      host_vars/
         hostname1       # if systems need specific variables, put them here
         hostname2       # ""

   staging/
      hosts              # inventory file for staging environment
      group_vars/
         group1          # here we assign variables to particular groups
         group2          # ""
      host_vars/
         stagehost1      # if systems need specific variables, put them here
         stagehost2      # ""

library/
module_utils/
filter_plugins/

site.yml
webservers.yml
dbservers.yml

roles/
    common/
    webtier/
    monitoring/
    fooapp/
```

Example 2.3: Example of Ansible project structure [4] Group variables and hosts are in their separate inventory directories

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running
    service:
      name: httpd
      state: started
  handlers:
    - name: restart apache
      service:
        name: httpd
        state: restarted
```

Example 2.4: Example of Ansible Playbook structure [6]. The purpose of this playbook is to update the Apache web server, and restart it immediately afterwards. It should be noted that the Ansible handler will not restart Apache web server if it was already in the latest version as there is no change to trigger handler.

# Chapter 3

# Environment

## 3.1 Property Transaction Service in National Land Survey

National Land Survey of Finland (NLS) is an official body that deals with cartography and cadastre issues subordinated by Finnish Ministry of Agriculture and Forestry. The service being developed is one of the electronic services offered by National Land Survey for cadastre issues and aligns with the overall trend of digitizing public services for easier access, lighter bureaucracy and faster process.

Property Transaction Service was created by National Land Survey of Finland to handle property issues electronically in an easier and faster way [22]. Electronic handling of property issues serves the future aim of NLS to resolve property issues automatically and without delay and currently provides multiple services to Finnish land owners, sellers, buyers, as well as real estate agents and banks such as:

- Authorize another person, such as a real estate agent or a bank, to act on persons behalf
- Make a preliminary purchase agreement
- Write and sign a bill of sale or a deed of gift
- Give consent to transfer real estate property
- Apply for a mortgage on a property you own or a property when registration of title in your name is pending
- Supplement an incomplete application of registration of title

- Transfer electronic mortgage deeds
- In conjunction with a conveyance, give a written commitment to transfer mortgage deeds of which you are the holder
- Register a Leasehold or Special Right
- To write and sign a bill of sale, apply for a mortgage, or sign a preliminary purchase agreement etc.

Currently the Property Transaction Service (PTS) is being developed by Eficode Oy and this thesis is delivered as a part of development work from Eficode. The requirements engineering has included close cooperation with the Central Federation of Finnish Real Estate Agencies, Federation of Finnish Financial Services and the National Land Survey. [22]

The service is accessible via a browser or HTTP API consisting of several different services that communicate between each other and other public sector services such as authentication services for private and company users and property register. The end vision of the product is to handle property issues easier, faster and enable electronic services defined by Finnish Code of Real Estate. [24]

## 3.2 Software architecture of the product

Product has been broken down into seven parts in order to separate different functionality into their own services and follow the microservice design principles. One of these services is only related to the development stage as it is a Stub for external services which the service needs in order to function as specified. Therefore only six of the services are running in production. The services product consists of include: The HTTP API service that takes in incoming documents from integrating users such as banks or real estate agents, Issue list service that keeps record of the ongoing issues that users has in the service, pdf service that generates pdf documents from the structured data for user to view, admin service for NLS personnel to perform managerial tasks, ownership service that forwards the ownership registration applications to Property Register and the main application where applications are received and signed via browser interface by the customers, excluding the ownership registrations.

The Property Transaction Service integrates to other public services using an enterprise service bus. Integrated services include internal services within the NLS, such as Property Register, which keeps track of all the real

estate ownerships and their mortgages in Finland as well as other external public services that are required for authentication, electronic payments and determining user rights within the service, such as role as company representative.

## 3.3 Current environments

Currently the product is deployed to six different environments, each consisting several application servers running either the entire product or subset of the services:

- Local development environment
- Internal development environment
- Acceptance testing environment
- External development environment
- External training environment
- Production test environment
- Production environment

This means that certain versions of the software have to be maintained and updated in six separate environments at all the time. In this list, local development environment means developers personal computers, while others are publicly available either in the internal network of NLS or public internet. Increasing workload related to the release of new versions and hotfixes was the motivation for work that led to this thesis.

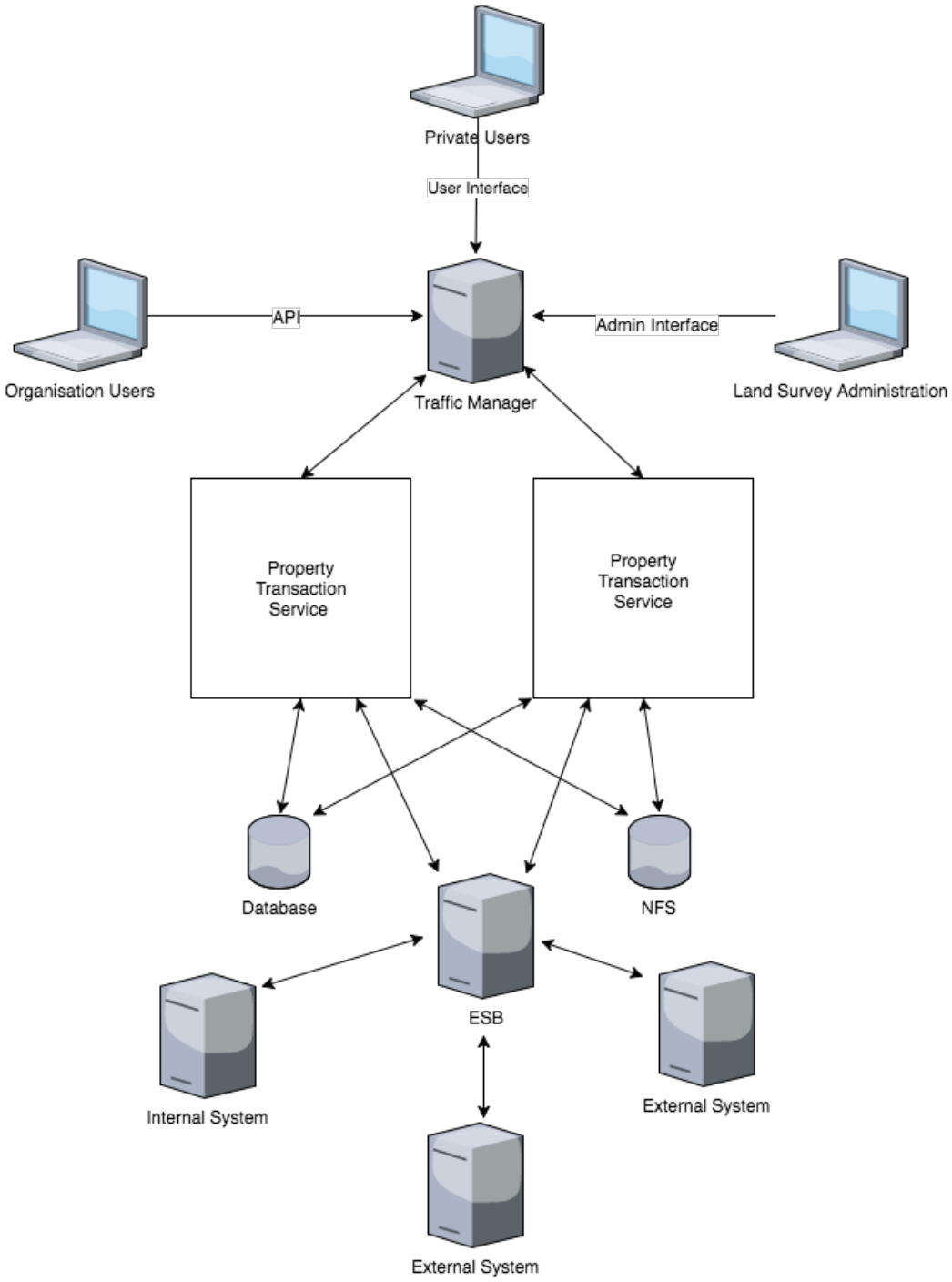
One of the challenges in these environments is that production environment and production test are classified [3] as elevated security networks. This means that only authorized people with very high level security clearance have access to the application servers where the production version is running and root access is denied for developers. This complicates the creation of proper release pipeline significantly since updates have to still be launched manually by the select people from NLS. In practice, an automated release pipeline can only be implemented to the acceptance test environment because of the IT policy restrictions, but given the dependency of other NLS background systems, shrinking the release cycle is so far out of reach. At the moment when practical implementation of deployment automation was

about to begin, the first objectives were to create at least capability for continuous delivery and for this purpose, the acceptance test environment is more than enough.

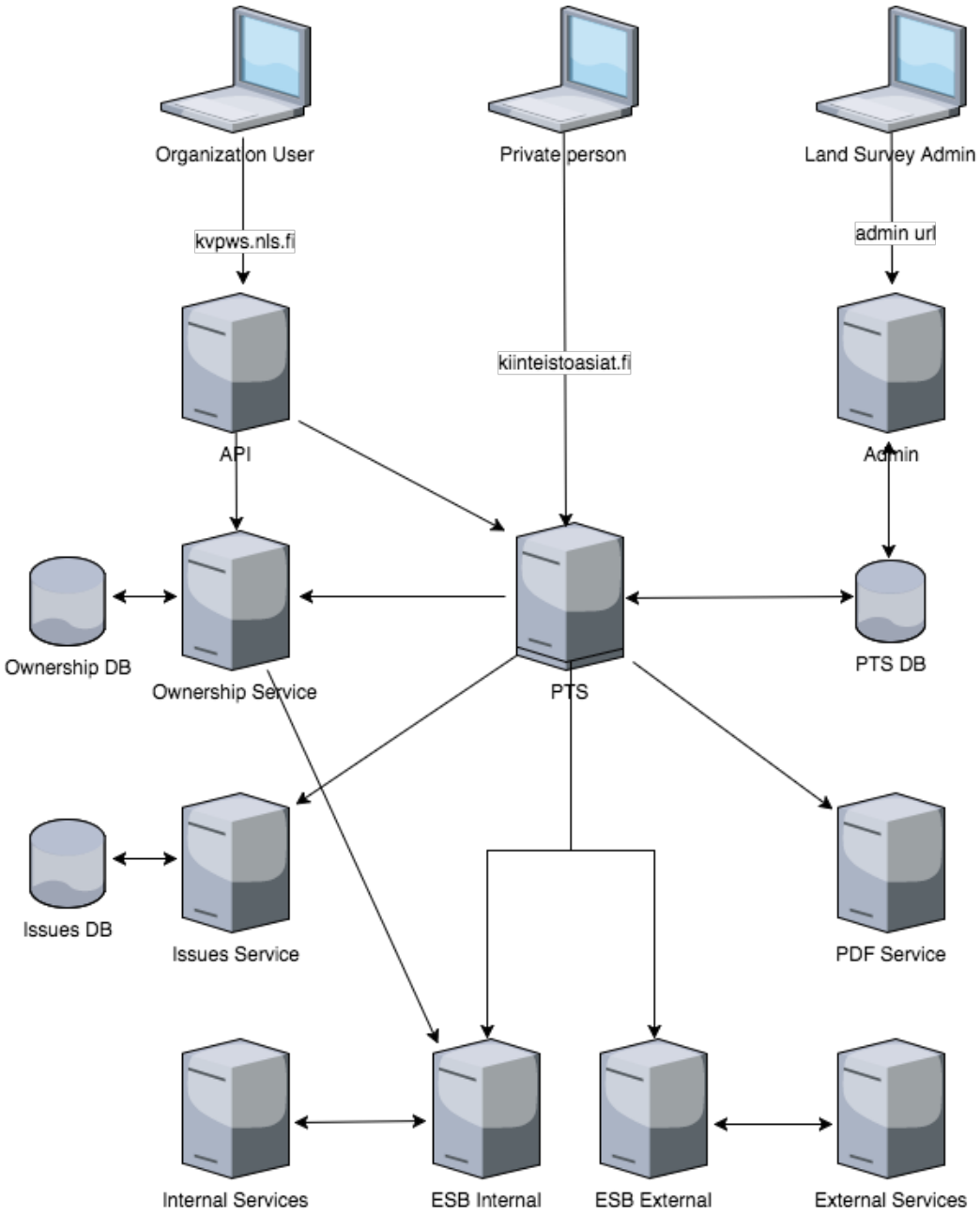
## 3.4 Previous deployment automation

Deploying the services was already partly automated with shell scripts, but the project containing scripts was poorly maintained and difficulties had started to rise when deploying new versions of the software for acceptance testing or production. As there are multiple services running on multiple application servers, the old deployment scripts that had been developed for monolithic application started to become a serious impediment and took time from development work with all the manual steps required. There was also a demand for storing the configurations of web servers, HTTP proxies, etc in Git version control system in case there would be any changes to the server configurations.

Another flaw in the shell script approach was that it's significantly more difficult to reach idempotent solutions i.e. operations can be repeated without getting a different outcome, in changing the configuration of application server on multiple nodes since shell scripting is not declarative, unlike YAML. It also added to the challenge, that maintenance of these application servers was outsourced to another stakeholder, which maintains with little communication or documentation, creating a real demand for idempotent configuration management solution.




Example 3.1: Network architecture of Property Transaction Service




Example 3.2: Application level architecture of Property Transaction Service

# Chapter 4

# Implementation

## 4.1 Automating deployments with Ansible

First part about deployment is packaging the correct software version for future deployment. With the correct software version and configurations related to the target environment. Once package is created it can be stored for later use in case there is a need for reverting to older version or deployed immediately. Local actions that take place on the Ansible control node where the package creation takes place include:

1. Cloning the source code from version control system to match the user-defined version
2. Installing all dependency libraries for the application using dedicated dependency management tool
3. Creating the WAR-package which contains the compiled application in binary format for the Java Servlet Container
4. Schema migrations file in SQL format for possible changes to database schema
5. Conditional notification to the instant messaging application channel used by the development team
6. Adding all the necessary Ansible resources such as: playbook, inventory, roles, migration files and the WAR-package into an archive that can be transferred to elevated security network if package is going to be deployed to production

After all the local tasks are finished and installation package is complete, Ansible opens SSH connections to all the application servers that have been

defined in the Ansible inventory file. Once setup is done, Ansible starts running the remote tasks. Remote tasks can also be run without first creating a package. These actions are:

1. Stopping the application and HTTP servers
2. Removing old deployment, copying new WAR-files to Java Servlet Container applications and updating relevant configurations
3. Running database schema migrations (only once regardless of the number of application servers)
4. Archiving logs from the previous version in tar archive
5. Restart application and HTTP servers and check that service is responding normally
6. Destroying and re-creating the job queue for scheduled tasks, such as sending pending applications to property register
7. Notify the instant messaging application channel used by the development team that new version has been released

Like the local tasks, remote tasks can also be ran any time separately by giving the installation package as a parameter. Services can also be deployed individually or all at once depending on the need. As program runs in an elevated security network, installation package in transfered to NLS personel who can start an update with a single command. When installing from elevated security network, a special installer script is created that gives Ansible all the parameters it needs, meaning that NLS personel don't need to worry about giving them.

Another set of tasks was to update the all the user interface texts as a separate part unrelated to the normal database migrations. Instruction, message and rule texts that are available in the user interface can thus be modified in the administration view by NLS personnel in any environment and exported to the desired environment using the text update task. The normal process for this is that Eficode writes and exports tasks from development to acceptance testing environment where NLS verifies them and fixes potential mistakes. After NLS has approved texts in the acceptance testing, they are exported to other environments.

### 4.1.1 Rolling updates

One of the built-in features in Ansible enables software to be updated without maintenance breaks. By shutting down one application at the time, load

balancer will automatically redirect the incoming requests to other application servers. When update is finished and web server starts, load balancer will start to redirect traffic to that server and perform the identical steps with the next application server. This type of 'rotation' eliminates the need for maintenance breaks, since one or more of the application servers will always provide a working implementation of the product.

In case of this particular project, there is option to disable rolling updates and release to production environment is done with a maintenance break if there are changes to the database schema or HTTP API. Reasoning here is to make sure that the applications that service receives are in same version. Since software is updated together with a variety of other products developed by other teams, an organizational decision was made not to develop the pipeline into continuous delivery since receiving different kinds of documents from customers requires extensive integration with other NLS internal systems, such as property register.

At the moment whenever a new release is scheduled, there is a maintenance break for updating all the other related systems as well to support new features. However, in case of a hotfix, the service can be updated in a zero-downtime fashion since changes only affect the service developed by Eficode.

## 4.2 Scheduling deployments from Jenkins CI

Once the manual work and shell scripts had been rewritten with Ansible, another step was to identify the conditions that would require new deployment of the software. A decision was made by the team to update internal development as a rolling update whenever new changes were detected in master- and development-branches in the version control. This way the internal development environment would reflect the latest status of the product and publish the changes for the entire team for quality assurance phase.

Since the external development and training environments still had a long release cycle, it was decided to run the deployments on the internal development environment as well as external acceptance testing environments on the condition that unit- and integration-level tests were successful. Then the nightly job at Jenkins CI machine would run automated acceptance test suites for the document interface would add the final part of the automated quality assurance and acceptance testing that was performed by Eficode before handing the software over to National Land Survey for their quality assurance.

## 4.3 Automating deployments in elevated security network

Since extending the Jenkins CI machine to continue the delivery pipeline to production test and production environments is not possible due to information security policies at NLS, the the built version of software is simply packaged into compressed package with all relevant configuration files and transferred to the elevated security network for NLS personel to inspect and deploy. Once the installation package is ready, Ansible playbooks for performing the installation can be launched manually from the control machine in elevated security network to finish the release.

# Chapter 5

# Evaluation

In this Chapter the state of new deployment process is presented and the results of new deployment automation are reviewed by using two different metrics. As it has been previously established from literature review, adopting continuous integration increases the project predictability, developer productivity and communication between teams [31]. The old deployment process had long intervals between deployments to acceptance testing environments and the difference between intervals of deployments to the system which is running the automated acceptance tests is examined.

## 5.1 Current status of CI in Property Transaction Service

Development team utilizes a Jenkins CI server for controlling it's deployment process for development and user acceptance testing environments. Normally changes are tested and deployed to shared development environment automatically multiple times per day and nightly to the acceptance testing environment if all unit and integration tests are passing for the development branch. Real-time reporting for team is helpful in explaining why some application servers might not be unresponsive at the moment and give the team feedback on the number of new changes being merged into version control. Around midnight if build is stable and unit and integration tests are passing, new version will be deployed to the acceptance testing environment and automated acceptance tests are triggered to provide additional on the functionality of the application on a system level. Automated acceptance tests in this environment are not only testing the internal functionality of the system, but also that the product is functioning with other Land Survey systems as expected.

Deploying new versions of software is done with Ansible, by first building new version of the software with desired dependencies, copying these packages to the target environment application servers and updating configuration files related to the infrastructure and application at the target environment. When all of these tasks are finished, web servers are restarted and application deploys for the end users.

## 5.2 Deployment frequency

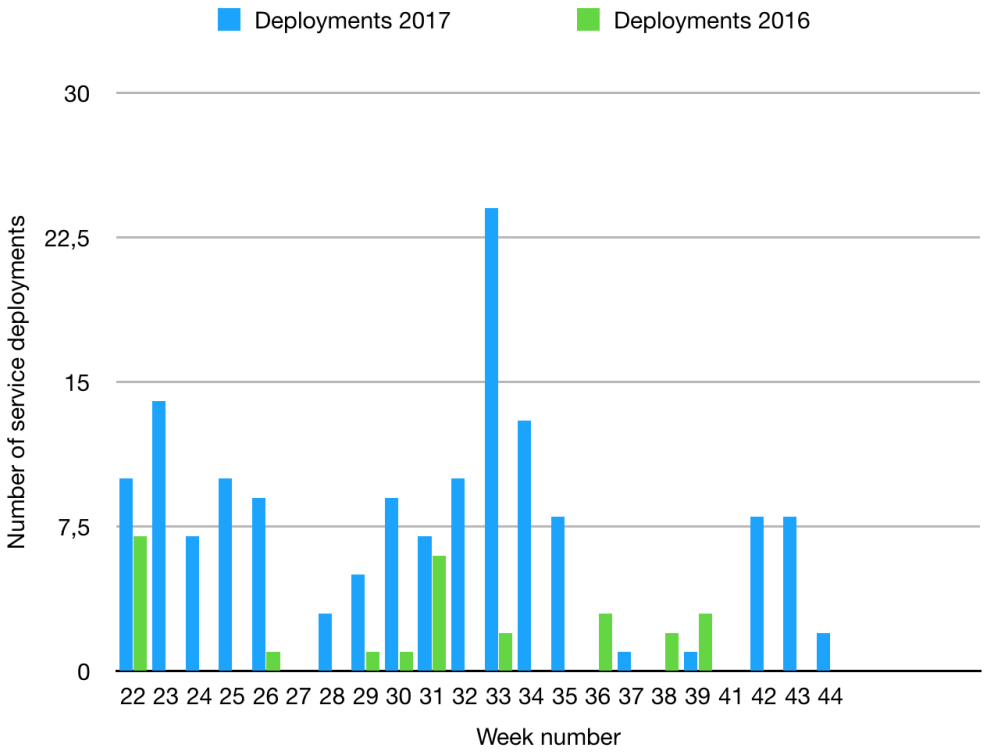
In previous setup, deployments to the user acceptance testing environment were performed approximately once a month. History of deployments in to the User Acceptance Testing environment is presented in list format in accordance to the Land Survey internal documentation [11] Table 5.1 lists dates and components installed into user acceptance testing environment before November 2016 when deployment process was remodeled.

Over a period of four months, June 2016 to October 2016, there was total number of 26 deployments to user acceptance testing environment. In 2017, during a similar time frame of June to October (109 workdays), there were 149 deployments to user acceptance testing environment. During the old deployment process, intervals between deployments were substantially longer, meaning that automated acceptance tests would be testing the same version for extensive periods of time. On an average, there was one deployment per 4,19 workdays. In the new setup, it is uncommon to have even delay of one week in deployments and in an average there was new deployment per 0,73 workdays. Even though the benefits of implementing continuous integration have been widely accepted in the field of software development, the improvements in quality are difficult to measure quantitatively. As deployment cycle has shrunk to approximately less than one fifth of the previous cycle, a conclusion can be drawn that significant improvement has been achieved.

## 5.3 Deployment workload

Previous deployment process required one of the developers to run series of multi-stage deployment scripts that was reported as a problematic and time consuming process which caused lots of errors.

Considering the frequency of deployments that new architecture requires to achieve Continuous Delivery, it could be estimated that for a developer to manually run deployments in the same pace as current process, one manual deployment taking approximately an hour, it would take approximately 40




Example 5.1: Visualization of the deployment history of PTS to user acceptance test environment, a substantial increase in deployments can be observed

hours per month to achieve the same result as automated process. In the old process, every deployment meant that one of the developers was required to be physically present an set correct configurations for the environment, thus taking these 40 hours away from developing features, writing tests, etc. slowing down the work of development team. As the previous process can immediately be judged as very labor intensive, it can be concluded that automating deployment process also saves developers working time on top of other benefits. The time that was previously spent on preparing for deployments and performing them has been directly transferred into development, testing and support for the end users, enhancing the productivity of the team. It is difficult to estimate the exact increase in productivity, but as just before a new release has typically been the most busy part of the release life cycle, it is helpful that the most routine parts of the work are automated.

# Chapter 6

# Discussion

As the application deployment and parts of infrastructure were automated, a great deal of work still remains in the product development. In its current state, it would seem that the bottleneck for development speed is not the deployment automation, but rather the multiple dependencies to other internal systems inside NLS. For example, receiving new types of documents through the property transaction service interface requires property register to be able to receive them after application parties have given their signatures. When releasing a new version, all these integrated services must be updated as well to support new functionality, making every new release complex and difficult to test on the system level. In the discussion part there is a review of some other topics in this project related to the deployment and development process.

## 6.1 Other metrics

In addition to the two metrics that were presented in the evaluation chapters,, there are other benefits that are more difficult to quantify as precisely as the two main metrics. First possible improvement is the higher availability of up to date User acceptance testing environment and internal development environment. Since latest development version is installed on both environments at high frequency with rolling update, doing quality assurance work is faster since these environments are available under all normal conditions. Manually deploying to these environments would be extremely time consuming and standard update without the zero downtime feature would repeatedly take these environments offline

Since server configurations are also checked or verified by the automated deployment process, the risk of accidentally changing them decreases. Cor-

rect configurations are stored in version control together with the deployment scripts and deployment process checks whether or not the target configurations are matching to the ones found from version control. If there are conflicts, deployment process automatically overwrites the configurations to match the version defined in the beginning of new deployment. This decreases the risk for configuration errors for the configurations that the deployment process includes. As Eficode only delivers the software and does not have full access to the infrastructure, some of the infrastructure configurations are out of Eficodes control and therefore not in the scope of this thesis.

## 6.2 Future work

### 6.2.1 Infrastructure automation

Martin Fowler introduced a vision of 'phoenix' server [15] which means regularly burning down and provisioning new instances to run software on periodically to avoid 'snowflake servers' which are difficult and time-consuming to get up and running. Configuration drift in the application servers becomes problematic since it gets more and more difficult to understand which parts of the configuration affect what. When developers have limited access to production environments it makes solving issues and bugs from production more complicated from developer perspective because of the restricted access to view the configurations, logs and database that is used in production. [16] It is also more difficult to practice deployments or try to replicate production system for debugging if the complete configurations of production servers are not known to developers.

To speed up the deployments of new versions, new virtual machines could be provisioned with Ansible, containing the latest software version and then Load Balancer or DNS name server could start redirecting traffic to new application servers. Currently, this kind of automation was not possible to implement due to constraints from IT service provider used by NLS but in an advanced setup, both the product developed by Eficode and its integrated services could be provisioned into internal network for testing. After quality assurance verifies that systems are working as intended, similar process could be replicated in the production environment by replacing one by one the virtual machines hosting previous version with the new version.

### 6.2.2 Developer access

Since developers are also responsible for supporting the application, doing their work effectively requires them to have some kind of access to production servers in order to read the logs and determine what has caused the issue they are trying to solve. As developers are providing the software that runs in production it would be consistent with the DevOps culture that they can also have some visibility to the logs and configurations that exist in production back ends. If the information security requirements of NLS restrtict access from non-essential persons, maybe at least one member of the development team should have read access to the production servers in order to be able to assist properly with the issues customers may have experienced.

### 6.2.3 Accelerated test automation

As ability to do Continuous Delivery depends heavily on the length of the feedback cycle, it's important to get fast feedback on fresh changes. While working on the feature branch, changes are tested with unit and integration tests before merging into development branch. On the largest service, running the integration tests takes about 32 minutes, being thus the slowest feedback cycle on before merging the changes into development branch. When the entire principle of DevOps focuses on capability to introduce changes fast and reliably, getting the tests to run as fast as possible is a high priority..

API test suites are even more problematic than unit- and integration-level tests. Those are ran nightly on the user acceptance testing environment and running them the entire test set on a single API version takes around four hours. Typically two versions of the API have to be maintained meaning that one night is barely enough to run all the tests. If new features of issue types are developed for the product, one night won't be enough to comprehensively test the development branch. To ensure faster test results, suites should be ran in parallel and rewritten to support parallel runs of the test suites.

As the automated acceptance tests take a long time, a smaller subset of critical test cases could also be created for verifying some of the core functionality and most common use cases. For example, document signatures and state could be tested only for once for each type, and receiving a new document through API in XML format would be tested in multiple different ways, including negative cases,.

## 6.3 Other possible deployment strategies

One particularly interesting case of automated deployments would be blue-green deployment, which achieves zero-downtime update to a service built for receiving data continuously from Internet of Things devices where high availability is critical to prevent data loss [1]. In this case the developed service enjoyed higher degree of flexibility as the operations side of the project was using a cloud service provider for provisioning virtual machines and was able to implement blue-green deployments by changing the DNS configurations. The process of blue-green deployment means that the deployment process would initialize new server instances in cloud providers infrastructure with newer versions of software and change DNS configurations to point to a new load balancer once instances were ready for use and running the latest release. Persistent data still existed in the database during deployment.

As a result of the zero-downtime update features, the project was able to release with higher frequency and save time from the manual work [1]. It also offered a backup plan in case a deployment went wrong since the old instances were still running and DNS configurations could be reverted to older version.

# Chapter 7

# Conclusions

Configuration management and automated application deployment are prerequisites for minimizing feedback cycle and feature lead time that are primary goal of Continuous Delivery.

Especially in a case where the application actually consists of multiple microservices, the downsides of manual deployments are particularly highlighted. With automated deployments, the risks and downsides of manual deployments can be alleviated, as there is an easy, repeatable and reliable process that maintains documentation of the infrastructure and dependencies in itself.

The solution for automatic deployments that was created as the practical work of this thesis and delivered to the National Land Survey helped the team to release software faster and focus on developing new features instead of doing the manual work related to the stages of release cycle. It also made releasing hotfixes more Agile, as no maintenance break needs to be scheduled and communicated to the development team, acceptance testers, integrating services and in the future - hopefully - end users of the application in production server.

As Eficode is not the product owner, this work also provided value for the National Land Survey in the form of automating some of the development process and making them more independent of the domain knowledge that has built up in the Eficode team during the past years. Since the documentation of previous installation scripts and manual steps related to them was in poor condition, a complete rewrite of self-documenting configuration management software will give the National Land Survey better control over the product.

# Bibliography


[1] Markus Juopperi. Deployment automation with chatops and ansible, 2017. https://www.theseus.fi/bitstream/handle/10024/127446/Deployment%20automation%20with%20ChatOps%20and%20Ansible.pdf?sequence=1.

[2] Beck, K., Cockburn, A., Jeffries, R., and Highsmith, J. Agile Manifesto, 2002.

[3] Board, T. G. I. S. M. Instructions on Implementing the Decree on Information Security in Central Government, 2b/2010, 2010.

[4] Documentation, A. Playbooks - Best Practices , 2017. https://docs.ansible.com/ansible/latest/playbooks_best_practices.html. Accessed 17.8.2017.

[5] Documentation, A. Use case: Configuration management , 2017. https://www.ansible.com/configuration-management. Accessed 8.11.2017.

[6] Documentation, A. Playbooks - Intro to Playbooks, 2017. https://docs.ansible.com/ansible/latest/playbooks_intro.html. Accessed 17.8.2017.

[7] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Montesi, M. M. F., Mustafin, R., and Safina, L. Microservices: yesterday, today, and tomorrow - Chapter 1, 2017. https://arxiv.org/pdf/1606.04036.pdf. Accessed 8.11.2017.

[8] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Montesi, M. M. F., Mustafin, R., and Safina, L. Microservices: yesterday, today, and tomorrow - Chapter 3, 2017. https://arxiv.org/pdf/1606.04036.pdf. Accessed 8.11.2017.

[9] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Montesi, M. M. F., Mustafin, R., and Safina, L. Microservices: yesterday, today, and tomorrow, page 9, 2017. `https://arxiv.org/pdf/1606.04036.pdf`. Accessed 8.11.2017.

[10] Ebert, C., Gallardo, G., Hernantes, J., and Serrano, N. DevOps, IEEE Software May/June 2016, 2016.

[11] Eficode Oy. Property transaction service deployment history, 2017. `https://confluence.nls.fi/display/MMLSKV/KVP+asennushistoria`. Accessed 14.10.2017. National Land Survey internal material.

[12] Fowler, M. Continuous Integration, 2006. `https://martinfowler.com/articles/continuousIntegration.html`. Accessed 11.10.2017.

[13] Fowler, M. Mocks Aren't Stubs , 2012. `https://martinfowler.com/articles/mocksArentStubs.html`. Accessed 8.11.2017.

[14] Fowler, M. Service Stub , 2012. `https://martinfowler.com/eaaCatalog/serviceStub.html`. Accessed 7.7.2017.

[15] Fowler, M. Phoenix Server, 2012. `https://martinfowler.com/bliki/PhoenixServer.html`. Accessed 2.6.2017.

[16] Fowler, M. Snowflake Server, 2012. `https://martinfowler.com/bliki/SnowflakeServer.html`. Accessed 2.6.2017.

[17] Jez Humble, D. F. *Continuous delivery : Reliable software releases through build, test, and deployment automation, Chapter 1: The Problem of Delivering Software.* 2010.

[18] Jez Humble, D. F. *Continuous delivery : Reliable software releases through build, test, and deployment automation, Page 109* . 2010.

[19] Jez Humble, D. F. *Continuous delivery : Reliable software releases through build, test, and deployment automation, Page 5.* 2010.

[20] Ken Schwaber. SCRUM Development Process, Chapter 2: Overview, 1997.

[21] Ken Schwaber, Jeff Sutherland. The Definitive Guide to Scrum: The Rules of the Game, Scrum Theory, 2014. `https://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-us.pdf`, Accessed 29.10.2017.

[22] Maanmittauslaitos. Property Transaction Service , 2017. `https://www.kiinteistoasiat.fi/english_info`. Accessed 11.5.2017.

[23] Mary Poppendieck, T. P.

[24] Ministry of Justice, Finland. Code of real estate, chapter 9 a, 2017. `http://www.finlex.fi/fi/laki/ajantasa/1995/19950540#L9a`. Accessed 2.6.2017.

[25] Mohan, V. Microservices, Supergiant Architecture for Stability and Scale, 2017. `https://supergiant.io/blog/microservices-supergiant-architecture-stability-scale`. Accessed 21.8.2017.

[26] Pekka Abrahamson, Outi Salo, Jussi Ronkainen, Juhani Warsta. *Agile Software Development Methods: Review and Analysis Chapter 2.1 What does it mean to be Agile.* 2002.

[27] Pekka Abrahamson, Outi Salo, Jussi Ronkainen, Juhani Warsta. *Agile Software Development Methods: Review and Analysis, Chapter 2.2 Selection of Agile methods* . 2002.

[28] Pressman, R. S. Software Engineering: A Practitioner's Approach (7th International ed.), 2009.

[29] Red Hat Inc. The Benefits of Agentless Architecture , 2016.

[30] Smart, J. F. *Jenkins: The Definitive Guide, Chapter 1: introducing Jenkins.* 2011.

[31] Stahl, Daniel and Bosch, Jan. Experienced benefits of continuous integration in industry software product development: A case study.

[32] Vanish Talwar, Qinyi Wu, Calton Pu, Wenchang Yan, Gueyoung Jung, Dejan Milojicic. Comparison of approaches to service deployment. Proceedings of the 25th IEEE International Conference on Distributed Computing Systems.