Abstract

Lean software development has become more and more popular. It is especially suited for development of Software as a Service-applications (SaaS). One reason for this is the way the SaaS-applications are released; they are usually released in fast cadence of iterations, in some cases even multiple times a day. This requires a new approach to the tools and methods used in software development. The objective of this thesis is to find out what requirements there are for development tools in context of small software development company.

The lean software development has its roots in lean manufacturing. Many of the methods used in lean software development are somewhat similar to the methods of lean manufacturing. So it is relevant to know out what kind of methods and processes are used in lean manufacturing, and learn about their experiences, as it has been around from 1940s.

Two high level requirements for development tools were derived from the methods and processes found in lean manufacturing and lean software development:

Every change must be reviewed and tested

Installation and update process must be automated

Two different tools were then selected for implementation. These tools were then used as a part of for a new development process, which was focused for short lead time of new features using lean software development methodologies. This process was found out to be useful in internal acceptance, but there were problems in acceptance testing by the customer as the pace of changes were too fast.

Keywords

Lean, Continuous deployment
Foreword

I would like to thank my family, friends and colleagues at Sysart for help and support during this thesis process. I would also like to thank my thesis supervisor Professor Ilkka Tervonen for advices, tips and especially for patience during this long process.

Jyrki Puttonen
Oulu, April 19, 2015
1. Introduction

The lean software development has become one of the main software development processes used in modern software development organizations. The lean software development process has a new set of requirements for supporting tooling, as the release process changes from scheduled releases to continuous release.

1.1. Background and incentives for research

Software development organizations are transforming into service providing organizations. Software is no longer delivered to the customer as an installation package on physical media and as a set of written instructions on how to operate the software. The responsibility of operating the software is moving to the development organization. This is especially true when this responsibility is fulfilled by developing an web-based application.

When developing a web-based application the software development process can, and must, be changed from traditional processes. The reason for this is that the release of a new version becomes easier, as there is no longer need for delivering the end product to the customer on a physical media. This makes it possible to release new versions of the application as needed, for example when a problem is detected or a new feature has been implemented.

Humble and Farley (2010), Chen (2015) and Leppänen et al. (2015) have all listed multiple benefits for delivering applications continuously. Examples of these benefits are shorter time to market, better customer satisfaction and faster feedback loops.

The continuous delivery of applications seems to offer benefits, but it also has new challenges. For example, when application is deployed frequently, there cannot be a long testing period before the release, and the deployment itself must be simple as it is not possible to long periods of scheduled downtime on regular basis.

1.2. Research questions and research methods

The research question of this thesis is: "How can the lean development process be supported by tooling". This problem is approached with a constructive research method. The constructive research method is appropriate as there is a need to find solutions for real world problems (March & Smith, 1995; Lukka, 2003). The problem is solved with innovative artefacts, but the artefacts, or the process of building them, is not the result of the research. The resulting artefacts are used for evaluating the new ideas.

In this study, the artefacts are evaluated as a case study. The artefacts were initially created and used for a specific case, which was one project at Sysart. The evaluation of the
artefacts, namely deployment pipeline and its parts, was done in the context of the Sysart case. The case used to evaluate the artefacts is described in detail in the next section.

A core concept of the constructive research is a construct. The construct is an artefact, such as documentation, model, diagram or product. A notable thing in the constructive research method, according to Lukka (2003), is the that resulting construct is not found, it is invented and built. Then the construct must be tested in a real world situation. This creates a need for close collaboration with the researcher and the real world users.

Constructive research process has typically seven phases (Lukka, 2003).

1. Find a practically relevant problem
2. Examine the potential for research co-operation
3. Obtain a deep understanding of the topic area
4. Innovate a solution idea and develop a problem solving construction
5. Implement and test the solution
6. Consider the applicability of the solution
7. Identify and analyze the theoretical contribution

In this thesis, the problem relevancy was defined by Sysart. Understanding of the topic area was gathered by literature analysis, which was used for defining potential solution. This solution was then implemented and tested in a real world project. The applicability of the solution was considered by observing the usage of the developed artefacts.

1.3. Limitations and Threats to Validity

This thesis is limited to process and tools used from feature development to the deployment of software. So, for example, the requirements gathering and operating the application is left out.

Biggest threat to validity of this thesis is that there is no written data available. The analysis is based on informal discussions and mostly undocumented project retrospectives.

1.4. Sysart Case

During 2014, Sysart was selected as the provider for "Espoon Hyvinvointipalveluiden Verkkoneuvontajärjestelmä" (Sysart, 2014b). The system consists of knowledge base and question/answer section, and it is based on Requeste platform.

The Requeste platform is a highly configurable, extensible web application, which is designed for this kind of projects. The platform is a separate from Requeste -product, which is older application based on client-server -design.

The project consists of four pilot areas, which are done one after another. Pilot areas had already been selected, but there wasn't any clear specifications available. The final design and specification was left to Sysart. This meant that there would be rapid pace of changes; initial version of the pilot was first done, then displayed to the customer representative who
then commented and requested changes. When writing this thesis, only the first of these pilots has been completed.

The application is offered as Software as a Service (SaaS). When describing SaaS-applications, Choudhary (2007) noted that "whereas traditional software publishers typically release new product features as part of new versions of software once in a few years, publishers using SaaS have an incentive to release new features as soon as they are completed". Sysart has a history of being a "traditional software publisher" with Requeste product (Sysart, 2014a).

These two reasons, unclear specifications and offering the system as SaaS, were main drivers for researching a better way for developing and deploying the application. There was a need for a better process which would utilize the benefits of producing SaaS, and for the actual implementation of this process and supporting tooling.

The initial decision was made that the process model best suited for this project was lean software development. The process and supporting tools were developed in the beginning of the project, and used to develop and deploy the first pilot area.

Petersen and Wohlin (2009) provide a checklist for context variables, which can be used to define the context of the study. The context variables for this study are outlined in table 1.

Table 1: Context variables

<table>
<thead>
<tr>
<th>Product</th>
<th>Customized web application build on Requeste platform. The size of customization is few thousand lines of code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>The process used was kanban. A small part of the system was designed, developed and tested before starting working on next.</td>
</tr>
<tr>
<td>Practices, Tools, Techniques</td>
<td>All of the code was reviewed and integrated constantly. The product was deployed into testing environment daily, and to deployed to production as needed.</td>
</tr>
<tr>
<td>People</td>
<td>The project had a project manager, a technical project manager and 1-2 developers. Most of the people were working also in other projects.</td>
</tr>
<tr>
<td>Organization</td>
<td>Sysart has two different business units, one on product development and other in provides software development services to other companies</td>
</tr>
<tr>
<td>Market</td>
<td>The product is targeted to a strict audience, namely citizens and employees of City of Espoo.</td>
</tr>
</tbody>
</table>

In other words, the context of this study a small project, done by a small company. One noteworthy issue in the context is that there wasn't any assigned system administrators or testers. These roles were done by participants as needed.

1.5. Contributions and goals

The major goal of this thesis is to define requirements and implementation for tooling meant to support lean software development in small development organization. Most of the research of tooling and development pipelines for lean development has been targeted to big development organizations, like Facebook and Netflix. This thesis describes how a small organization can benefit from lean software process by using proper tooling.
1.6. Structure

First chapter of this thesis introduces Lean Production. This was used as inspiration on how the process should work. Second chapter is a literature review of methods and practices in Lean software development. Third chapter describes the build pipeline and process which was the result of this research. Finally, the in the fourth chapter, the applicability of the solution is considered and theoretical contributions are identified and analyzed.
2. Lean Production

The term "Lean Production" (or "Lean Manufacturing) was introduced in "The Machine That Changed the World" (Womack, Jones, & Roos, 1990). The book was a result of the International Motor Vehicle Program (IMVP), a five-year collaborative investigation into the performance of the global motor industry. Womack, Jones and Roos credit the actual coining of the term to IMVP researcher John Krafcik. Lean production is "lean" because it uses less of everything compared to with mass production (Womack et al., 1990). Lean production has also been defined as a multi-dimensional approach that encompasses a wide variety of management practices, including just-in-time, quality systems, work teams, cellular manufacturing and supplier management in an integrated system (Shah & Ward, 2003).

One of the fundamental concepts of lean production is "muda", or waste. Waste is everything that doesn't add value to the product, or in other words, "any activity that takes time and money but does not add value from the customer's perspective is waste". and "waste or muda is any activity in a process that consumes resources without adding value for the customer" (Morgan & Liker, 2006). Elimination of waste is the goal of lean production (Ohno, 1988; Middleton, 2001).

The lean methodologies have been used in manufacturing since Second World War (Womack et al., 1990). One of the implementations of Lean methodologies in manufacturing is the Toyota Production System (Ohno, 1988). According Womack et al. (1990), the Toyota Production System was the source of Lean methodologies. The lean methodologies have also been implemented by Toyota for developing new products. This version of Lean is called Toyota Product Development System (Morgan & Liker, 2006).

The lean methodologies have recently become more well known in software development (Anderson, 2003; Ikonen, 2012; M. Poppendieck & T. Poppendieck, 2007), and there are multiple different implementations of lean methodologies in software development. The "Kanban" model introduced by Anderson (2003) is one the implementations. The kanban process model is based on the pull -method, which is one of the key operation management tools in lean manufacturing. The pull system in Kanban process model contains value stream mapping (Mujtaba, Feldt, & Petersen, 2010), inventory management (Anderson, 2003), including queuing theory and the theory of constraints (Goldratt, Cox, & Whitford, 2012).

Ohno (Ohno, 1988) says that the beginnings of Toyota Production System can be traced to Toyoda Sakichi (1867-1930), the founder of the Toyota Motor Company, and his son Toyoda Kiichirō (1894-1952), the first president of Toyota Motor Company and the father of the first Japanese passenger car. Toyota entered the motor-vehicle industry in the late 1930s having previously been in textile machine industry (Womack et al., 1990). After World War II, Eiji Toyoda and Taichii Ohno from the Toyota Motor Company pioneered the concept of lean production. According to Womack, Ohno visited the big auto manufacturers in Detroit, but quickly realized that the methods and tools used there were not suitable in Japanese environment as the Japanese market was a lot smaller which required
flexibility in the manufacturing; they could not manufacture huge lots of one kind of products at a time, as the market required a mixed product variety.

There's two, essentially similar views of what Toyota Production System is, one from "The Toyota Way" (Liker, 2004) and one from "Toyota Production System" (Ohno, 1988). Ohno gives autonimation and just-in-time as the two pillars of Toyota Production System (Ohno, 1988). In the book "The Toyota Way", Liker says that the two pillars of the Toyota Production System are just-in-time and jidoka (Liker, 2004). On the other hand, he says that "Jidoka is also referred to as autonimation" (Liker, 2004), so he is essentially using same terms as Ohno.

Autonimation is defined by Liker as "equipment endowed with human intelligence to stop itself when it has a problem" (Liker, 2004) and by Morgan as "the practice of recognizing an abnormal condition and responding quickly" (Morgan & Liker, 2006). Ohno himself describes autonimation to be also know as "automation with human touch" (Ohno, 1988). The concept of autonimation originates from auto-activated looms made by Toyoda Sakichi. These looms were equipped with devices that stopped if vertical or lateral threads broke or ran out (Ohno, 1988). If one of those threads was missing, the resulting product would be defective. Main purpose of autonimation is to prevent production of defective products.

Just-In-Time is defined by Ohno as follows: "in a flow process, the right parts needed in assembly reach the assembly line at the time they are needed and only in amount needed" (Ohno, 1988). Pull system has been recently the commonly applied term (Womack et al., 1990). Ohno used the term "pull" to describe the way the customers pull the goods when they need them in the amount they need them and at the time they need them. A Just-In-Time -system eliminates the need of extra inventories (Ohno, 1988). Inventories are considered harmful as they will age and can contain defective products. As products produced by earlier process are used in latter process almost immediately, possible defective products are noticed earlier (Ohno, 1988).
3. Lean in Software Development

There has been multiple attempts to use and define lean manufacturing methods in software development. One of the most used process model that tries to bring lean manufacturing into software development is called "Kanban". Other, earlier definitions of lean software development were given in books "Lean Software Development - An agile toolkit" (M. Poppendieck & T. Poppendieck, 2003) and "Lean-agile software development" by Shalloway, Beaver, and Trott (2009).

Following sections present these three views of what lean software development is. First a general overview of Kanban in Software development is given, and then the views of Poppendiecks and Shalloway are presented. These two views are some of the best available in literature.

3.1. Kanban in Software Development

One suggestion for implementing the lean Manufacturing in software development is the Kanban process model (Gross & McInnis, 2003). The process model is based on "pull" -method, and is one of the key operation management tools. The "pull" system in Kanban process model contains value stream mapping (Mujtaba et al., 2010), inventory management (Anderson, 2003) including queuing theory and the theory of constraints (Goldratt et al., 2012). As the kanban process model uses the pull -method, it has benefits in improved responsiveness, prevention of overproduction and reduced inventory (Gross & McInnis, 2003).

The value-stream map is typically visualized with index cards moving on board from one phase to another. These index cards act as an flow-control tickets between the different workstations and processes (Kniberg & Skarin, 2009). This visualization can then be realized as a table, such as a wall-paper with sticky notes (Ikonen, 2012).

Shalloway et al. (2009) argue that Kanban has following benefits:

- Lessens the fear of committing to estimates per story
- Focuses on the team's performance over individuals
- Focuses on improving the workflow
- Allows reflection of improvement measures
- Due to transparency, allows the management to involve in process improvement

Kanban in software engineering is based on the following beliefs: (1) software development is about managing and creating knowledge, (2) software development processes can be managed and described in terms of queues and control loops accordingly, and (3) some representation of information that flows through the system is required (Shalloway et al., 2009).
The Kanban management model can be applied into software development by using tasks as Work-In-Progress items (Hiranabe, 2008). One important thing to remember is that the use of Kanban is not actually a method, it is a concept that is used to justify the triggering of activities putting the idea of "pull" practice. In software development, this means that the Kanban is used to organize the activities within a team (Hopp & Spearman, 2004). The basic principles of Kanban can be applied to software development when the concept of physical material is reinterpreted by replacing material flows with information flows.

In his foreword to "Kanban and Scrum - making the most of both" Kniberg and Skarin (2009) explain that "What we've come to realize about Kanban is that it is an approach to change management. It isn't a software development or project management lifecycle or process. Kanban is an approach to introducing change to an existing software development lifecycle or project management methodology". Larman (Larman & Vodde, 2008) has criticized that Kanban merely optimizes something that needed to be changed.

Kniberg and Skarin define three distinct ideas behind the Kanban process model they use:

**Visualize the workflow**

- Split the work into pieces, write each item on a card and put on the wall.
- Use named columns to illustrate where each item is in the workflow.

**Limit Work In Progress (WIP)**

- Assign explicit limits to how many items may be in progress at each workflow state.

**Measure the lead time**

- Optimize the process to make lead time as small and predictable as possible.

Ikonen (2012) says that "In spite of the potential in manufacturing, there are only a few studies regarding how Kanban fits in software development projects.". However, there is studies of Kanban resulting in higher maturity (Middleton & Joyce, 2012; Vega, 2010).

### 3.2. Poppendiecks’ Approach on Lean Software Development

In the book "Lean Software Development - An agile toolkit" (M. Poppendieck & T. Poppendieck, 2003), p. xxv - xxvii Poppendieck gives 22 different tools of lean software development which are divided into seven different categories:

**Eliminate Waste**

1. Seeing waste
2. Value Stream Mapping

**Amplify Learning**

3. Feedback
4. Iterations
5. Synchronization
6. Set-Based Development
Decide as late as Possible
7. Options thinking
8. The Last Responsible Moment
9. Making Decisions

Deliver as Fast as Possible
10. Pull systems
11. Queuing Theory
12. Cost of Delay

Empower the Team
13. Self-Determination
14. Motivation
15. Leadership
16. Expertise

Build Integrity In
17. Perceived integrity
18. Conceptual integrity
19. Refactoring
20. Testing

See the Whole
21. Measurement
22. Contracts

(M. Poppendieck & T. Poppendieck, 2003).

Many of these tools sound familiar with the tools of lean manufacturing in Toyota Production System and the tools of lean product development system.

In 2007 Poppendiecks revised their wording. "Building integrity in" became "building quality in", "amplify learning" turned into "create knowledge", "decide as late as possible" is now "defer commitment", "deliver as fast as possible" got shortened to "deliver fast", "Empower the Team" is "respect the people" and "see the whole" is now action, "optimize the whole" (M. Poppendieck & T. Poppendieck, 2007)

3.2.1. Eliminate waste

Poppendiecks defines waste as "Waste is anything that does not add value to a product. If a development cycle has collected requirements in a book gathering dust, that is waste. If developers code more features than are immediately needed, that is waste. Whatever gets in the way of rapidly satisfying a customer need is waste." (M. Poppendieck & T. Poppendieck, 2003).
There are two tools in this category, seeing waste and value stream mapping. For the first tool, seeing waste, to work, it is important to understand what waste is in the context of software development. As Liker, Ohno and Morgan, Poppendiecks describes seven different wastes specific to software development.

- **Partially done work**
  Partially done development ties up resources in investments that have yet to yield results.

- **Extra processes**
  Many software development processes require paperwork for customer to sign-off, or to provide trace-ability, or to get a approval for a change.

- **Extra features**
  Every feature adds code into the system, and that code must be tracked, compiled, integrated, and tested every time the code is touched, and then it has to be maintained for the life of the system.

- **Task Switching**
  Assigning people to multiple projects is a source of waste. Every time software developer switch between tasks, a significant switching time is incurred as they get their thoughts gathered and they get into the flow of the new task.

- **Waiting**
  One of the biggest wastes in software development is usually waiting for things to happen. Delays in starting a project, delays in staffing, delays due to excessive requirements documentation, delays in reviews and approvals, delays in testing and delays in deployment are waste. Delays keeps the customer from realizing value as quickly as possible.

- **Motion**
  Moving artifacts from one group to another is a huge source of waste in software development.

- **Defects**
  The amount of waste caused by defect is the product of the defect impact and the time it goes undetected.

(M. Poppendieck & T. Poppendieck, 2003), p. 6-8.

In Implementing Lean Software Development Poppendiecks have replaces "Extra processes", "Motion" and "Waiting" with "Relearning", "Hand-offs" and "Delays" (M. Poppendieck & T. Poppendieck, 2007).

- **Relearning**
  Wastes resources and adds no value for the customer. Rediscovering a known but forgotten thing is rework. Ignoring knowledge that people bring to the workplace, destroys utilizing their potential.

- **Handoffs**
  Handoffs leave the major part of knowledge behind in the mind of originators. Tacit knowledge is difficult to transport to other people through documentation. In brief, documents cannot contain all of the information that the other people in line need to know.

- **Delays**
Delays slow down realizing value for the customer. Waiting for people to be available who are being busy in other areas causes waste.

Poppendieck's also provide a translation from manufacturing wastes to software development waste.

- [In-process] inventory -> Partially done work
- Over-production --> Extra features
- Extra processing --> Relearning
- Transportation --> Hand-offs
- Motion --> Task switching
- Waiting --> Delays
- Defects --> Defects

There is also a lot of waste in the management activities, as they don't directly add value to the product. For example, project tracking and control systems do not add value, and may indeed be an indication of too much work in the system. In Just-In-Time environment the work flows through the system so fast, that there's no need for tracking as the customer demands are fulfilled in one-piece flow. (M. Poppendieck & T. Poppendieck, 2003).

The second tool for eliminating waste is value stream mapping. In value stream mapping, the process of delivering the product is followed and written down step-by-step, including possible wait times. This process will lead to deeper insights of the processes and how they add value, or waste, to customer demands. The actual process of value stream mapping is best done with pencil and paper, while walking through the path of customer requirement through organization (M. Poppendieck & T. Poppendieck, 2003).

3.2.2. Amplify Learning

The software development is an exercise in discovery while production is exercise in reducing variation, and for this reason, a lean approach to development results in practices that are quite different than lean production practices (M. Poppendieck & T. Poppendieck, 2003). Development is a learning process involving trial and error (M. Poppendieck & T. Poppendieck, 2003), but in production the purpose is to produce a product with minimal errors.

Poppendiecks claim that "somehow, the idea that variation is bad has found its way into software development, where people have tried to develop standardized processes to reduce variation and achieve repeatable results every time". But development is not intended to produce repeatable results; development produces appropriate solutions to unique customer problems. (M. Poppendieck & T. Poppendieck, 2003). Software development is still compliant with reducing of variability in lean product development process. The lean product development process differs from lean production system as in the former, the reducing of variability means reducing variability in the process steps, not in the products. Also, the lean product development process tries to reduce variability in the end product also by using shared architectures and components where needed. This sharing of architectures and components is compliant with software development.
Today it is widely accepted that design is a problem-solving process that involves discovering solutions through short, repeated cycles of investigation, experimentation, and checking the results (Guindon, 1990). This can be described as knowledge-generation loop. Quite often, the problem to be solved is understood best by the people in the business with the problem, so it is usually necessary to have business people - or representatives of business people, such as focus groups - in the knowledge-generation loop. Most users relate better to seeing working screens than to seeing a requirements document, so working software tends to generate better knowledge faster. (M. Poppendieck & T. Poppendieck, 2003). So the goal of the software development process is to have many, short learning cycles.

First tool of amplifying learning is feedback. Feedback is related to the knowledge generation loops as the knowledge generation loops should provide feedback to the development process. Feedback gathering can be on multiple levels, starting from compile time errors to customer feedback surveys. Poppendiecks give few specific examples of feedback in different situations:

- Instead of letting defects accumulate, run tests as soon as the code is written
- Instead of adding more documentation or detailed planning, try checking out ideas by writing code
- Instead of gathering more requirements from users, show them an assortment of potential user screens and get their input
- Instead of studying more carefully which tool to use, bring the top three candidates in-house and test them
- Instead of trying out to figure out how to convert an entire system in a single massive effort, create a Web front end to the legacy system and try the new idea out

(M. Poppendieck & T. Poppendieck, 2003).

One way to increase the frequency of feedback loops is the usage of the second tool, iterations. Poppendiecks describe iterations as points of synchronization across individual and multiple teams and with the customer. Iterations are the points when feature set are completed and the it is brought as close as possible to a releasable or shippable state - even if it will not actually be released to the customers. (M. Poppendieck & T. Poppendieck, 2003).

There are three fundamental principles in iterations.

1. Small batches moving rapidly through a system enforces quality and worker level commitment and they provide better resource utilization and shorter feedback loops
2. Short iterations are options-based approach to software development as they allow when system to respond to facts rather than forecasts
3. Iteration are points of synchronization across individual and multiple teams and with the customer

(M. Poppendieck & T. Poppendieck, 2003).

Synchronization has been mentioned when talking about the two previous tools, and it is the third tool to amplify learning. Whenever several individuals are working on the same thing, a need for synchronization occurs. (M. Poppendieck & T. Poppendieck, 2003). Traditionally the integrity of a module was ensured by having only one developer working
on it, but in lean development the idea of one-piece-flow forces multiple people to work on same features at the same time. So the need of synchronization is inherent in lean software development.

In software development environment with collective code ownership, the idea is to build the system every day, after a very small batch of work has been done by each of the developers. At the end of the day, a build takes place, followed by a set of automated test. If the build works and the tests pass, the developers have been synchronized. This technique is often called the daily build and smoke test. There are many variations of this theme: A build might occur every few days, or it might run every time new code is checked in. More frequent builds are better; they provide much more rapid feedback (M. Poppendieck & T. Poppendieck, 2003). Poppendiecks continue that the build and test should be automated, as if they aren't, the build process itself will introduce errors, and the amount of manual work will prohibit sufficiently frequent builds.

Another way to synchronize the work of several teams is to start by having a small advance team to develop a simple spanning application through the system. Now the overall architecture is sketched and then the separate teams will develop separate components and subsystems (M. Poppendieck & T. Poppendieck, 2003). This makes it possible to test different components and gather real understanding about the software.

Fourth tool in amplifying learning is set-based development. This is similar to the set-based approach in the lean product development system. In the domain of software development set-based development is applied by developing multiple options, by communicating constraints and lettings the solutions emerge.

3.2.3. Decide as late as possible

Development practices that provide for late decision making are effective in domains that involve uncertainty, because they provide an options-based approach. Delaying decisions is valuable because better decisions can be made when they are based on facts, not speculations. A key strategy for delaying commitments when developing a complex system is to build a capacity for change into the system. (M. Poppendieck & T. Poppendieck, 2003).

The problem with sequential development is that it forces designers to take a depth-first rather than breadth first approach to design. Depth-first forces making low-level dependent decisions before experiencing the consequences of the high-level decisions. (M. Poppendieck & T. Poppendieck, 2003). And depth-first approach also prevents the usage of set-based development. Poppendiecks call breadth-first approach "concurrent software development". For concurrent software development to be effective, it requires developers with enough expertise in the domain to anticipate where the emerging design is likely to lead. Concurrent software development also requires close collaboration with the customers and analysts who are designing how the system will solve the business problem at hand (M. Poppendieck & T. Poppendieck, 2003).

The first tool in deciding as late as possible is options thinking. Option means that there's an option to do something, but it is not required. One of the hot debates in software development has been the trade off between predictive and adaptive processes. In predictive processes the development should be specified in detail prior to implementation, as changes cost more to make later. The adaptive process uses options to delay some of the
decisions until the customer needs are clearly understood. It is important to notice that options are not free. (M. Poppendieck & T. Poppendieck, 2003).

Second tool is the last responsible moment. As concurrent software development makes it possible to start developing before all requirements are known, it is possible to delay commitment until the last responsible moment. That moment is when failing to make a decision eliminates an important alternative. (M. Poppendieck & T. Poppendieck, 2003). Poppendiecks give following tactics for making decisions as late as possible:

- Share partially complete design information
- Release unfinished designs to get feedback early
- Organize for direct, worker-to-worker collaboration
- Releasing incomplete information requires that participants of development process communicate directly.
- Develop a sense of how to absorb changes
- Use common object-oriented and general design principals to constraint the changes.
- Develop a sense of what is critically important in the domain
- Forgetting some critical feature of the system until too late is the fear that drives sequential development, but in practice, early commitments are more likely to overlook such critical elements as early commitments rapidly narrow the field of view.
- Develop a sense of when decisions must be made
- Late commitments must not degenerate into no commitment, so there must be knowledge about the timing and a mechanism to cause decisions to be made when their time has come.
- Develop a quick response capability
- If response times are long, the decisions must be made earlier.

The making of decisions is the third tool when deciding as late as possible. There are two strategies for problem solving: breadth-first and depth-first. Breadth-first problem solving might be thought of as funnel, while depth-first problem solving is more like a tunnel. Breadth-first involves delaying commitments, while depth-first involves making early commitments. (M. Poppendieck & T. Poppendieck, 2003). There is a risk in depth-first problem solving that the field under consideration is narrowed too soon and if there's need for change of course, work will be wasted. It is important to notice that domain expertise is needed in both strategies, as when doing breadth-first problem solving, it is crucial to have some knowledge about the expected direction of the business domain.

One way of making decision at the lower level of organization is to use simple rules. Simple rules allow everyone in the organization to act quickly, synchronously, in coordinated manner and without instructions from above (M. Poppendieck & T. Poppendieck, 2003). Poppendiecks give following simple rules for software development, one per their principle:

1. Eliminate waste: Spend time only on what adds real customer value.
2. Amplify learning: When you have tough problems, increase feedback.
3. Decide as late as possible: Keep your options open as long as practical, but no longer.
4. Deliver as fast as possible: Deliver value to customers as soon as they ask for it.
5. Empower the team: Let the people who add value use their full potential.

6. Build integrity in: Don't try to tack in integrity after the fact - build it in.

7. Optimize the whole: Beware of the temptation to optimize parts at the expense of the whole.

These rules allow the lowest level of organization make decisions about how to proceed. They should be based on the culture and values of the organization. In a way, these simple rules could be considered as standardized work from Toyota Production System.

3.2.4. Deliver as fast as possible

Until recently, the rapid software development hasn't been valued, even if it has many advantages. Rapid development makes it possible to delay decisions, enables getting feedback from customers earlier and assures the customers that they get what they need now, not what they needed yesterday (M. Poppendieck & T. Poppendieck, 2003). Delivering software rapidly diminishes the risks of having inventory, such as partly done work. It also supplement the principle of deciding as late as possible (M. Poppendieck & T. Poppendieck, 2003).

First tool of rapid delivery is usage of pull systems. These pull systems are similar to the ones in Toyota Production System. But as there usually isn't physical items to work on in software development, it is not actual items that usually get pulled from the previous process. The previous process will produce requirements for features, which are then pulled through the development flow. (M. Poppendieck & T. Poppendieck, 2003).

Second tool if queuing theory. Important concept in lean manufacturing is takt time, which describes the rhythm of the manufacturing line. Rhythm of the manufacturing line can also be measured by checking how long every step has time to get completed. Poppendiecks use the term cycle time in somewhat different meaning: the cycle time means the "average time something to get from one end of a process to the other" (M. Poppendieck & T. Poppendieck, 2003). There's two ways to reduce the cycle, controlling the arrival of the work into the process and refining the process itself.

When controlling the arrival of the work into the process the goal is to move form batch processing to one-piece flow, as defined in Toyota Production System. This thinking should be executed in all levels of organizations, from project initiation to individual sprint planning (M. Poppendieck & T. Poppendieck, 2003).

Last tool in delivering as fast as possible is cost of delay. This tool explains how the delay in delivery might affect into to the profits and losses for the whole lifetime of the product. The basic idea is that when delivered late, the delay might prevent the early high pricing and might result in smaller market share as the competitors bring their products into market first (M. Poppendieck & T. Poppendieck, 2003).

3.2.5. Empower the Team

The first tool of empowering team is called "Self determination". This means that those who do the work should define the processes and tools used. This is somewhat similar with the standard work in Toyota Production System, where those who do the work will
write the actual standard. Poppendiecks say that "transferring practices from one environment to another is often a mistake." (M. Poppendieck & T. Poppendieck, 2003). In Toyota Production System, standard work is used to spread practices around the whole organization, so there seems to some disagreement here. But the answer might be that standard work is just few steps to do something, like attaching a front bumper into car, as practices are bigger things, like usage of version control.

Motivation is second tool for this principle. There's much written about motivation (for example, Pink (2011), Kahneman (2013)). One of the important factors in motivation is purpose, and Poppendiecks list six different examples on what to do make the work more purposefully:

Start with clear and compelling purpose
   The vision of the product must be compelling for the members of the team.

Be sure the purpose is achievable
   The team must have within itself capability of accomplish the purpose.

Give the team access to customers
   Talking to customer gives the team clear understanding about the purpose and it helps them to see how their work fits into overall picture.

Let the team make its own commitments
   The team should commit to a set of feature.

Management's role is to run interference
   Highly motivated team does not need to be told what to do, but it may need protection, resources or advises.

Keep skeptics away from the team
   Nothing kills purpose faster than someone who knows it can't be done and has plenty of good reasons why. The team does not need to hear it.

(M. Poppendieck & T. Poppendieck, 2003).

Purpose alone isn't enough. Motivation needs requires also feeling of belonging, a feeling of safety, a sense of competence and a sense of progress.

Belonging
   Everyone on the team should know what the goal is and they should be committed to its success. Team members should respect each other and be honest.

Safety
   Team must be allowed to make mistakes, there should not be zero mistakes -mentality.

Competence
   Feeling of competence comes from knowledge, skill, positive feedback, high standards and meeting difficult challenges.

Progress
   Projects should have meaningful measurements that show progress toward the goal posted in a place for everyone to see.

(M. Poppendieck & T. Poppendieck, 2003).

Third tool in this category is leadership. Poppendiecks present the term master developer (M. Poppendieck & T. Poppendieck, 2003), which is more or less the same as chief
engineer in lean product development system. The chief engineer in lean product development system holds the overall responsibility for the product, starting from requirements gathering to production (Liker & Hoseus, 2008).

Expertise is the final tool in empowering the team. There's two different kind of expertise involved in software development, technical expertise and domain expertise. The technical expertise is usually divided by functions, such as database experts, user interface experts or embedded software experts. The domain expertise can be, for example, on health care or on security. All of these should be arranged into communities of expertise, where experts from same area can help each other (M. Poppendieck & T. Poppendieck, 2003). Just like in the lean product development system, Poppendiecks talk about matrix organization where people from communities of expertise work in different product development programs.

Poppendiecks claim that one important purpose for communities of expertise is development of standards such as naming standards, language standards, code checkout and check-in standards, build standards and so on. This idea has similarities with the standard work from Toyota Production System.

3.2.6. Build integrity in

There are two different kinds of integrity, perceived integrity and conceptual integrity. Poppendiecks define them in following way: Perceived integrity means that the totality of the product achieves a balance of function, usability, reliability, and economy that delights customers. Conceptual integrity means that the system's central concepts work together as a smooth, cohesive whole. (M. Poppendieck & T. Poppendieck, 2003).

There's four tools in this principle, perceived integrity, conceptual integrity, refactoring and testing.

Perceived integrity is present in lean product development. The chief engineer system in lean product development is the key to the perceived integrity. (M. Poppendieck & T. Poppendieck, 2003). The chief engineer can establish the flow of information from customers to engineers by understanding the customer needs and by having the technical knowledge to communicate with engineers. There are other techniques for establishing this information flow between customer and developers, such as customer tests, domain specific languages and models and customer feedback.

Second tool is conceptual integrity. The automotive industry uses few key practices for achieving conceptual integrity, they use existing parts where appropriate and they use integrated problem solving (M. Poppendieck & T. Poppendieck, 2003). The latter means the problem solving and understanding should happen at the same time, preliminary information about the problem is released early, information is transmitted in small batches and the information flows in two directions. The preferred media for transmitting information is face-to-face communication. (M. Poppendieck & T. Poppendieck, 2003).

Refactoring is the third tool in this principle, and it is closely related to conceptual integrity. Refactoring is a tool for keeping architecture healthy. The need for refactoring comes from the addition of new features. These new features might require some architectural capacities, and without refactoring the existing architecture, the conceptual integrity of architecture will suffer. (M. Poppendieck & T. Poppendieck, 2003). Refactoring is comparable to the stopping of the line in Toyota Production System, where production
workers can stop the production line when problems occur so the root cause of problem can be found and solved.

Final tool in building integrity in is testing. Poppendiecks divide tests into to different parts, developer tests and customer tests (M. Poppendieck & T. Poppendieck, 2003). Developer tests are tests which assures that code does what the developer intended it to do, and customer tests assure that the system will do what the customer expects it to do. According to Poppendiecks, tests have five different roles in software development, as follows:

1. Communication
   Tests will communicate unambiguously how things are supposed to work
2. Feedback
   Tests give feedback if the system actually works like it's supposed to work.
3. Scaffolding
   Tests will support tools like last responsible moment, set-based development and refactoring.
4. As-Built documentation
   After development, tests will provide representation how the system was actually built.
5. Maintenance
   Tests make changes to production system more safer.

3.2.7. See the whole

Last principle from Poppendiecks is called "See the Whole", which has two tools: measurements and contracts. The roots of this principle are in systems thinking, which is a research area that look at organizations as systems (M. Poppendieck & T. Poppendieck, 2003).

The first tool, measurements, is about how local measurements can lead to local optimizations. Local optimizations can lead to growing inventory, which is the worst of the seven wastes. The second tool of contracts defines different types of contracts and specifies that the trust between two companies is the most import thing.

3.3. Lean Software Development according to Shalloway

Shalloway et al. (2009) presented their view on lean software development in "Lean-agile software development". This view was expressed in following categories and items:

Minimizing Complexity and Rework

1. Eliminating Waste and Deferring Commitment
2. Using iterative development to minimize complexity and rework
3. Create Knowledge
4. Deliver early and often
5. Build quality in
6. Optimize the whole

Fast-Flexible-Flow
1. Focus on Time

Value Stream Mapping

Shalloway et al. (2009) compares the hidden costs and risks between Manufacturing Mass Production and Waterfall Model as defined by Royce (1970). This comparison is presented in table 2. There are a lot of similarities between Manufacturing Mass Production and Waterfall Model, so the methods of lean manufacturing should be applicable to software development.

Table 2: Comparison of Manufacturing Mass Production and Waterfall

<table>
<thead>
<tr>
<th></th>
<th>Manufacturing Mass Production</th>
<th>Waterfall Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hidden Costs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transportation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Managing inventory</td>
<td>Lots of open items; can lead to</td>
<td></td>
</tr>
<tr>
<td>and storage</td>
<td>overwhelming the workforce</td>
<td></td>
</tr>
<tr>
<td>Capital costs of</td>
<td>Costs of training people to</td>
<td></td>
</tr>
<tr>
<td>inventory</td>
<td>build software</td>
<td></td>
</tr>
<tr>
<td><strong>Risks</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Building things you</td>
<td>Building things you don't need</td>
<td></td>
</tr>
<tr>
<td>don't need because</td>
<td>because requirements aren't</td>
<td></td>
</tr>
<tr>
<td>production goes on</td>
<td>clear or customers change</td>
<td></td>
</tr>
<tr>
<td>after need go away</td>
<td>their minds</td>
<td></td>
</tr>
<tr>
<td>Inventory becoming</td>
<td>Knowledge degrading quickly</td>
<td></td>
</tr>
<tr>
<td>obsolete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Huge latency if an</td>
<td>Errors in requirements</td>
<td></td>
</tr>
<tr>
<td>error occurs</td>
<td>discovered late in the process</td>
<td></td>
</tr>
<tr>
<td></td>
<td>and errors in completed code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>discovered late in testing</td>
<td></td>
</tr>
</tbody>
</table>

3.3.1. Minimizing Complexity and Rework

The first category that Shalloway et al. present is "Minimizing complexity and Rework". They emphasize that complexity and rework cannot be removed, but that the Lean Principles can reduce them. (Shalloway et al., 2009).

The first item combines two items from Poppendiecks. (Shalloway et al., 2009) says that the waste comes from the system, and Lean practitioner should look at the system to see how the it can be fixed. Deferring commitment, according to Shalloway et al., means making the decisions at the right time, "the last responsible moment".

Shalloway et al. present "Emergent design" as a approach to unclear design issues(Shalloway et al., 2009). Emergent design consists of three disciplines:

1. Using the thought process of design patterns to create application architectures that are resilient and flexible
2. Limiting the implementation of design patterns to only those features that are current
3. Writing automated acceptance- and unit-tests before writing code both to improve the thought process and to create a test harness.

All of these disciplines are aimed to make the code easy to change.

The second item is usage of iterative development to minimize complexity and rework. Shalloway et al. claim that biggest causes fro complexity are 1. writing code that isn't
needed and 2. writing code that is tightly coupled together (Shalloway et al., 2009). Iterative development avoids writing of unnecessary code, and emergent design helps with the latter.

Creation of knowledge is the third item. Shalloway et al. believe that the software development process is more a discovery process than a building process, and they divide product development process into three steps:

1. Discover what the customer needs
2. Figure out how to build that
3. Build it

They continue to claim that in software development the third step is what we spend most of our time talking about, but the first two steps take the most time. (Shalloway et al., 2009).

Iterative development also makes it possible to deliver early and often. Shalloway et al. makes it clear that the principle should be "remove delays" instead of "delivering fast", as delays represent waste. The ultimate goal is adding value to the customer without delay, but this has to be done in sustainable manner. (Shalloway et al., 2009).

For previous items to be sustainable, quality must be built into both the code and the process. One way to build quality in to the process is acceptance tests which are defined with the customer, the developer and the tester all together, as they improve communications. Acceptance tests also prevent errors from creeping in. (Shalloway et al., 2009).

The final principle in minimizing complexity and rework is optimizing the whole. This concept is similar with the one explained previously. Shalloway et al. emphasize that local optimizations creates large inventories between steps. Inventories in software world are partially done work, like requirements written but not designed, coded and tested. (Shalloway et al., 2009).

3.3.2. Fast-Flexible-Flow

Fast-Flexible-Flow is a term used to describe the development pipeline which takes ideas in and out to the customer as fast as possible. Fast means two different things here, first is that the idea must go into the pipeline fast, which gives flexibility to the process, and the other meaning is that when the idea is in the pipeline, it must be fast to go out also.

Shalloway says that the elimination of delays is something that is wanted in the lean world. Then he gives the following examples of delays in software development:

The time from when a requirements is stated until it is verified as correct
The time from when code is written until it is tested
The time from when a developer asks a question of a customer or analyst until she gets an answer.

(Shalloway et al., 2009).

Shalloway suggests creation of self-directed teams that have all of the resources they need and which pull the work from the backlog. (Shalloway et al., 2009).
3.3.3. Value Stream Mapping

Value stream mapping is a tool which is used to analyze value streams in lean manufacturing (Shalloway et al., 2009). It is a picture of the current process stream which shows every step of the process. Value stream map can be used in finding of root causes, because it shows the whole picture of process. It doesn't focus on one particular process step (Shalloway et al., 2009).

Liker describes value stream map to be a diagram, which captures processes, material flows, and information flows of a given product family and helps to identify waste in the system (Liker, 2004).

Value stream mapping highlights the critical error in many Agile approaches. They focus on the team, when the problem might not be there (Shalloway et al., 2009).
4. Continuous Integration and Deployment

Continuous integration and continuous delivery are crucial methods in both previous sections. These two have been defined in "Continuous integration" by Martin Fowler and in "Continuous Delivery" by Humble and Farley (2010). These two tools are closely related, and important to understand. Following sections present an overview of both Continuous Integration and Continuous Delivery.

4.1. Continuous Integration

Continuous Integration is term coined by Martin Fowler in 2000 and clarified in 2006 (Fowler, 2000, 2006). Currently he gives the following principles for Continuous Integration:

- Maintain a Single Source Repository
- Automate the Build
- Make Your Build Self-Testing
- Everyone Commits To the Mainline Every Day
- Every Commit Should Build the Mainline on an Integration Machine
- Keep the Build Fast
- Test in a Clone of the Production Environment
- Make it Easy for Anyone to Get the Latest Executable
- Everyone can see what's happening
- Automate Deployment

In his article, Fowler says that "Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible." (Fowler, 2006).

Duvall, Matyas, and Glover (2007) present four components required for Continuous Integration 1. A connection to version control repository, 2. a build script, 3. some sort of build mechanism (such as email) and 4. a process for integrating the source code changes (manual or CI server)(Duvall et al., 2007). They also specify following practices for Continuous Integration in their book (Duvall et al., 2007):

- Commit code frequently
  - Commit changes into repository at least daily
Don't commit broken code
 Compile and run tests before commit

Fix broken builds immediately
 The developer who committed the code that broke the build should be involved in fixing it.

Write automated developer test
 Verify software with automated tests

All tests and inspections must pass
 Every test must pass before committing code into repository

Run private builds
 Get latest changes from repository and and run full integration build locally

Avoid getting broken code
 Don't get broken code from repository.

Automate builds
 Create build scripts that can be executed without the Integrated Development Environment (IDE).

Perform single command builds
 Build should be possible to execute with one command

Separate build scripts from your IDE
 Build should be possible to be executed without IDE

Centralize software assets
 Use centralized repository for dependencies

Create a consistent directory structure
 Create logical directory structure that makes building the software easy

Fail builds fast
 For fast feedback, execute tests that are likely to fail fast first (smoke test).

Build for any environment
 Run same automated build on all necessary environments

Use dedicated integration build machine
 Use one, clean machine for integration builds.

Use a CI server
 In addition or as alternative to a dedicated integration machine, use CI-server which polls version control for changes and runs builds as necessary.

Run manual integration builds
 Run sequential integration build manually using an automated build to reduce integration errors.

Run fast builds
 Try to keep integration builds under 10 minutes.

Stage builds
 Separate long integration build process into different stages for faster feedback.
Automate database integration
   Rebuild database and insert test data automatically.

Use a local database sandbox
   Every developer should have own database which is generated via SQL scripts.

Use a version control repository to share database assets
   Commit database scripts into source control system.

Give developers the capability to modify the database
   Avoid bottleneck which realizes when one database admin does all change into the database.

Make the Database Administrator part of the development team
   Make sure that the database administrator can run the same automated build to ensure consistency.

Automate unit tests
   All levels of tests should be automated using appropriate tools.

Automate component tests
   All levels of tests should be automated using appropriate tools.

Automate system tests
   All levels of tests should be automated using appropriate tools.

Automate functional tests
   All levels of tests should be automated using appropriate tools.

Categorize developer tests
   Categorize different tests into "buckets" so they can be executed separately.

Run faster tests first
   For faster feedback, run faster tests first.

Write tests for defects
   Make sure that same defect doesn't happen again.

Make component tests repeatable
   Tests results should be same every time, so the state of the application at the start of testing has to be defined.

Limit test cases to one assert
   If there's only one assert per test, tracking down the failure reason is easier.

Reduce code complexity
   Measure cyclomatic complexity in your code base with automated inspectors and identify areas that need attention.

Perform design reviews continuously
   Use tools to study dependencies between packages to detect if some packages are highly dependent on others and thus may lead to brittle architecture.

Maintain organizational standards with code audits
   Use tools to detect standards violations in code.

Reduce duplicate code
   Use tools to detect duplicate code.
Assess code coverage
Find out what areas need more testing by tracking code coverage.

Release working software any time, any place
With fully automated build, which includes compilation, testing, inspections, packaging and deployments, release of working software should be possible at any time and into any known environment.

Label a repository's assets
Label files in version control, typically at the end of a milestone.

Produce a clean environment
Remove everything related to your software from your integration build machine and ensure you can rebuild back to a state where your integration build is successful.

Label each build
Label the binary of a build distribution in your version control library.

Run all tests
Run all tests that ensure that the software is ready to be delivered

Create build feedback reports
List the changes that were made in the most recent build.

Possess capability to roll back release
Make it possible to roll back into previous release.

Use continuous feedback mechanisms
Get the right information to the right people and in the right way.

4.2. Continuous Delivery

Continuous Delivery, or continuous deployment, can be considered as a next step from continuous integration. Continuous delivery takes continuous delivery a step forward by adding automatic deployment of applications to production (Humble & Farley, 2010). Humble and Farley (2010) say that the title comes from first principle of agile manifesto, "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." (Humble & Farley, 2010).

Chen (2015) lists multiple benefits from continuous deployment. The time to market is faster which have helped to stay a step ahead of the competition. Frequent releases lets the development team to get feedback from customers so they (the development team) knows that they are building the right product. The productivity and efficiency improved significantly as the time required to set up different environments was reduced. The releases became more reliable as the deployment scripts are executed for multiple environments and the number of changes per release decreases. Product quality improved significantly as bugs and issues were discovered earlier and fixes could be deployed in few days. Lastly the customer satisfaction improved.

Leppänen et al. (2015) interviewed multiple information and communications technology companies and found out following perceived benefits from continuous delivery: faster feedback, more frequent releases, improved quality and productivity, improved customer satisfaction, effort savings and closer connection between development and operations.
These are similar to what Chen presented, only the closer connection between development and operations is different. By that, one of the companies interviewed by Leppänen et al. (2015) meant that the continuous deployment prevents silos between development and operations as the frequent releases must be communicated and coordinated in much tighter schedule.

Humble and Farley (2010) also presents multiple benefits from continuous delivery. First of all, it empowers the teams. The deployment pipeline allows personnel to self-service the version of the application they want into the environment of their choice. Continuous delivery reduces errors, mostly the errors which come from manual configuration of system. Stress is lowered as the deployment itself is done by push of button and rolled back if needed. This leads to smaller releases and eventually the release becomes a non-event, something that is just done regularly. Then there is the flexibility of deployments, as the deployment can be easily done to different environments as needed. The last benefit mentioned is that practice makes perfect. The same deployment mechanisms are used for multiple different environments, so the final production deployments are rehearsed again and again.

The main principle in continuous delivery is the deployment pipeline (Humble & Farley, 2010; Chen, 2015) or deployment chain (Leppänen et al., 2015). According to Humble and Farley (2010) deployment pipeline has three benefits. First, it makes every step of the process visible to everyone, secondly it improves feedback and thirdly it enables teams to deploy and release any version of their software to any environment at will through a fully automated process.

Chen (2015) divides the deployment pipeline into six stages: code commit, build, acceptance test, performance test, manual test and production. This stages are described in table 3.

Table 3: The stages of deployment pipeline according to Chen (2015)

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code commit</td>
<td>Compiles the source code and executes unit tests</td>
</tr>
<tr>
<td>Build</td>
<td>Executes the unit tests again to generate a code coverage report, runs integration tests and various static code analyses, builds the artifacts for release and publishes artifacts for later use</td>
</tr>
<tr>
<td>Acceptance test</td>
<td>Sets up acceptance test environment and executes acceptance test suite</td>
</tr>
<tr>
<td>Performance test</td>
<td>Sets up performance test environment and executes performance test suite</td>
</tr>
<tr>
<td>Manual test</td>
<td>Sets up environment for manual testing</td>
</tr>
<tr>
<td>Production</td>
<td>Deploy application to production</td>
</tr>
</tbody>
</table>

Humble and Farley (2010) describes how the deployment pipeline works in following way: "Every change made to an application's configuration, source code, environment, or data, triggers the creation of a new instance of the the pipeline. One of the first steps in the pipeline is to create binaries and installers. The rest of the pipeline runs a series of tests on the binaries to prove that they can be released. Each test that that the release candidate passes gives us more confidence that this particular combination of binary code, configuration information, environment and data will work. If the release candidate passes all the tests, it can be released.".
In addition to description, Humble and Farley (2010) presents following principles of software delivery:

- Create a Repeatable, Reliable Process for Releasing Software
- Automate almost everything
- Keep everything in Version Control
- If it hurts, do it more frequently, and bring the pain forward
- Build Quality In
- Done Means Released
- Everybody Is Responsible for the Delivery Process
- Continuous Improvement

First of these principles, creation of a repeatable, reliable process for releasing software, means that releasing of software should be easy. Repeatability and reliability derives from two principles: automate almost everything and keep everything in version control. Reliability comes from using automated, repeatable process and repeat-ability is achieved with version control.

As mentioned previously, automation brings reliability to deployment process. Although there are things that are impossible to automate, like exploratory testing, the list of things that cannot be automated is surprisingly short. The build and deployment process should be automated up to the point where specific human decision is needed. Automation is a prerequisite for deployment pipeline, because there is no other way to make it possible to deploy the application with push of a button. (Humble & Farley, 2010).

Version control system is usually used for versioning source code, but that is not enough. The version control should include everything needed to build, deploy, test and release your application. This includes requirement documents, test scripts, automated test cases, network configuration script, deployment scripts, database creation, upgrade, downgrade, and initialization scripts, application stack configuration scripts, libraries, tool-chains, technical documentation and so on. The purpose of this is to enable releasing of any version of your application into any environment defined at any time (Humble & Farley, 2010).

The fourth principle is a general one, "If it hurts, do it more frequently, and bring the pain forward". This is what Extreme Programming does to the software development process. The goal is to make to continually get better in what is hurting you. (Humble & Farley, 2010).

"Build quality in" principle is achieved by integrating testing into software development process. Testing is not a distinct phase which begins after software is done, and it is not the domain of testers. Everybody in the team is responsible for building quality in. Techniques like continuous integration, comprehensive automated testing, and automated deployment, are designed to catch defects as early as possible. These techniques enables testing in early phases of the project, and there will be time to fix the defects (Humble & Farley, 2010).

Many agile practices emphasize the meaning of "done". For some agile teams, done means that product is released in to production. This isn't always possible, so another option
would be to define done as demoing product in production like environment to all stakeholders. This meaning of done has interesting side-effect: if done means released into production, multiple people are usually needed to make this happen, so it forces that everybody works together, from developer to the system administrator. (Humble & Farley, 2010). As getting task done needs everyone to work together, then everybody is responsible for the delivery process (Humble & Farley, 2010). This requires continuous communication between the participants.

Final principle for continuous delivery is continuous improvement. This continuous improvement should happen throughout the organization, because otherwise it will lead to local optimization. (Humble & Farley, 2010).

Deployment pipeline and related principles creates a release process that is repeatable, reliable, and predictable, which in turn generates large reductions in cycle time, and hence gets features and bug fixes to users faster. in addition to those benefits, the release process empowers teams, reduces errors, lowers stress, makes deployments flexible and forces to practice deployment. Those techniques also gives the ability to verify changes, to make the process reproducible across a range of environments, and to largely eliminate the opportunity for errors to creep into production. Automated system also encourages the implementation of other good practices. (Humble & Farley, 2010).

4.3. Comparing Lean software development with Continuous integration and Continuous delivery

Lean software development, continuous integration and continuous deployment have a lot of common principles. These principles are mapped together in tables 4 and 5. This mapping can be then used to define requirements for lean development pipeline.

Automation is important concept in both of continuous integration and continuous delivery. Automation seems to have impact to multiple principles of lean software development. There's a multiple different types of automation, from automated building of software to automated, repeatable delivery.

Reliable software delivery is a key component in creation of automated, repeatable delivery. This is achieved by building quality in. Building quality in is somewhat unclear term in software development. It can be approached by thinking the opposite of building quality in, which is inspecting the final product. In manufacturing industry, there has traditionally been an inspection unit that checked the final product (Morgan & Liker, 2006). This inspection unit is similar to traditional software testing. In lean manufacturing industry, the quality is built in by using tools, methods and parts that prevent mistakes, for example a bolt that can be used only in one place, or a tool that cuts the part in right place (Liker, 2004). The mistake preventing tools and parts can be related to having automated tests in software development.

Lean manufacturing industry also has other ways of building quality in. one of these is "Andon" (Liker, 2004). Andon is a visible and possible signal that tells that there's a problem and help is needed to resolve the problem. Andon signals that the production line must be stopped to resolve problems. Andon is also a mechanism for any employee to stop production if they see a problem. In software development, review can be considered as Andon. When reviewer sees a problem, review system can be used to signal problems existence.
Table 4: Mapping of practices in continuous integration and delivery to M. Poppendieck and T. Poppendieck (2007)

<table>
<thead>
<tr>
<th>Eliminate Waste</th>
<th>Continuous integration</th>
<th>Continuous delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Eliminate Waste</td>
<td>Create a Repeatable, Reliable Process for Releasing Software</td>
</tr>
<tr>
<td></td>
<td>Waste</td>
<td>automate almost everything</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Keep everything in Version Control</td>
</tr>
<tr>
<td></td>
<td>Maintain a Single Source Repository</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Make Your Build Self-Testing</td>
<td></td>
</tr>
<tr>
<td>Amplify Learning</td>
<td>Make it Easy for Anyone to Get the Latest Executable</td>
<td>Continuous Improvement</td>
</tr>
<tr>
<td></td>
<td>Everyone Commits To the Main-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>line Every Day</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Test in a Clone of the Production Environment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Everyone can see what's happening</td>
<td></td>
</tr>
<tr>
<td>Decide as late as possible</td>
<td>Automate the Build</td>
<td>Create a Repeatable, Reliable Process for Releasing Software</td>
</tr>
<tr>
<td></td>
<td>Keep the Build Fast</td>
<td>automate almost everything</td>
</tr>
<tr>
<td></td>
<td>Automate Deployment</td>
<td>Keep everything in Version Control</td>
</tr>
<tr>
<td>Deliver as fast as possible</td>
<td>Automate the Build</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Keep the Build Fast</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Automate Deployment</td>
<td></td>
</tr>
<tr>
<td>Empower the Team</td>
<td>Make it Easy for Anyone to Get the Latest Executable</td>
<td></td>
</tr>
<tr>
<td>Build integrity in</td>
<td>Make Your Build Self-Testing</td>
<td></td>
</tr>
<tr>
<td>See the whole</td>
<td>Make it Easy for Anyone to Get the Latest Executable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Everyone Commits To the Main-line Every Day</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Every Commit Should Build the Mainline on an Integration Machine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Test in a Clone of the Production Environment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Everyone can see what's happening</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Everybody Is Responsible for the Delivery Process</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Continuous integration</td>
<td>Continuous delivery</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Eliminating Waste and</td>
<td>Maintain a Single Source Repository</td>
<td>Create a Repeatable, Reliable Process for Releasing Software</td>
</tr>
<tr>
<td>Deferring Commitment</td>
<td>Automate the Build</td>
<td>Automate almost everything</td>
</tr>
<tr>
<td></td>
<td>Make Your Build Self-Testing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Automate Deployment</td>
<td></td>
</tr>
<tr>
<td>Using iterative</td>
<td>Make it Easy for Anyone to Get the Latest Executable</td>
<td>Create a Repeatable, Reliable Process for Releasing Software</td>
</tr>
<tr>
<td>development to</td>
<td>Everyone Commits To the Mainline Every Day</td>
<td>Automate almost everything</td>
</tr>
<tr>
<td>minimize complexity</td>
<td>Everyone can see what's happening</td>
<td>If it hurts, do it more frequently, and bring the pain forward</td>
</tr>
<tr>
<td>and rework</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create Knowledge</td>
<td>Make it Easy for Anyone to Get the Latest Executable</td>
<td>Continuous Improvement</td>
</tr>
<tr>
<td></td>
<td>Everyone Commits To the Mainline Every Day</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Everyone can see what's happening</td>
<td></td>
</tr>
<tr>
<td>Deliver early and</td>
<td>Automate the Build</td>
<td>Create a Repeatable, Reliable Process for Releasing Software</td>
</tr>
<tr>
<td>often</td>
<td>Automate Deployment</td>
<td>Automate almost everything</td>
</tr>
<tr>
<td></td>
<td>Make Your Build Self-Testing</td>
<td>Done Means Released</td>
</tr>
<tr>
<td>Build quality in</td>
<td>Make Your Build Self-Testing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Build Quality In</td>
<td></td>
</tr>
<tr>
<td>Optimize the whole</td>
<td>Every Commit Should Build the Mainline on an Integration Machine</td>
<td>Create a Repeatable, Reliable Process for Releasing Software</td>
</tr>
<tr>
<td></td>
<td>Keep the Build Fast</td>
<td>Automate almost everything</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If it hurts, do it more frequently, and bring the pain forward</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Everybody Is Responsible for the Delivery Process</td>
</tr>
<tr>
<td>Focus on Time</td>
<td>Automate Deployment</td>
<td>Create a Repeatable, Reliable Process for Releasing Software</td>
</tr>
<tr>
<td></td>
<td>Keep the Build Fast</td>
<td>Automate almost everything</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Done Means Released</td>
</tr>
</tbody>
</table>
5. Implementing Lean development pipeline

The project started with four pilot areas, which were done one after another. Pilots had already been selected, but there wasn't any strict specifications available. This meant that there would be rapid pace of changes; initial version of the pilot was first done, displayed to the internal customer who then commented and requested changes. All of these changes needed to be tested and verified before they could be released into public system.

In addition to the production environment, the customer also wanted to have their own testing and training environments. Sysart also needed to have own internal testing environment. So there was four different environments which must work similarly.

In tables 4 and 5, the methods and practices of continuous integration and continuous delivery are mapped into principles of lean software production. This mapping was then used to decide requirements for Lean development pipeline.

After reviewing methods and practices from previous chapters and the needs and the context of the project, requirements for development pipeline were decided. On high level, the pipeline needed to fulfill following requirements:

1. Every change must be reviewed and automated tests must be executed
2. Installation and update process must be automated

These requirements were considered to be small enough to be implemented with the resources available, and they covered multiple principals of lean software development. The following sections will describe how each of these high level requirements were filled.

5.1. Every change must be reviewed and automated tests must be executed

Automated tests are executed by continuous integration server. Jenkins (Jenkins Project, 2014b) was the tool of choice for continuous integration server. One of the requirements for CI-server was extensibility and integration to other systems. Jenkins is extensible with plugins, and it has been integrated into wide range of different tools and systems with community provided plugins. Currently there's over one thousand (1000) plugins available (Jenkins Project, 2014a). Other continuous integration servers weren't considered, as the Jenkins has been the traditional tool of choice in projects at Sysart.

Software inspection is widely considered beneficial (Doolan, 1992; Fagan, 1976). However, the inspection tends to be highly formal (Russell, 1991). The traditional software inspection meeting is considered to be time-consuming (Hedberg & Lappalainen, 2005). So an online tool was needed to enable asynchronous review.

Three different tools were considered as a option for code review; Gerrit (Gerrit Project, 2014), Crucible (Atlassian, Inc, 2015) and Review Board (Beanbag, Inc, 2015). Two main
requirements for code review tool was compatibility with Git and Jenkins, which were existing tools in the project. There were also other requirements that were considered, and the comparison is presented in table 6.

Table 6: Comparison of review tools

<table>
<thead>
<tr>
<th></th>
<th>Gerrit</th>
<th>Crucible</th>
<th>Review Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jenkins Support</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Git Support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>License</td>
<td>Apache License 2.0</td>
<td>Proprietary</td>
<td>MIT</td>
</tr>
</tbody>
</table>

The tool of choice was Gerrit (Gerrit Project, 2014). Main reasons for selection was compatibility with Git, existing support with Jenkins and familiarity with the tool. As Git was selected as the version control tool, the compatibility with it was strict requirement. In fact, Gerrit does not support any other version control systems, but that was not considered as harmful.

The integration of Gerrit to Jenkins was done with Jenkins Gerrit Plugin. This plugin makes it possible to build and test changes before they are submitted to the main development branch. The Jenkins instance detects that there's a new change added to the review tool, executes tests and then marks the result into Gerrit. This prevents broken code from entering the main development branch.

One of the developers had been managing and using a Gerrit installation before. This made it easy to start using Gerrit, because there was at least one person who was familiar with the tool.

After the change has been reviewed and tested successfully, it can be then merged into the main development branch. If either one of review or testing fails, a new change must be submitted. This workflow is described in figure 1.

![Review and testing workflow](image_url)

Figure 1: Review and testing workflow

In addition of building and testing changes in review phase, the main development branch was tested every time that there was a change added to it. This made it sure that there were no regressions in the main development branch.
By using Gerrit and Jenkins, every change is reviewed and tested before they were made available to other developers. This means that the main development branch has almost always a version which does not have any compilation errors and which passes all tests..

5.2. Installation and update process must be repeatable

As there is four different environments which must work similarly, there was a need for repeatable installation process. Configuration management tools make the installation and deployment to different environment easier (Wettinger et al., 2013; Günther, Haupt, & Splieth, 2010). The installation and deployment should lead to predictable outcome constantly (Burgess, 2011). Configuration errors are considered to be the biggest cause of service failures (between 40% and 51%) (Delaet & Joosen, 2007). And many of these errors are caused by human (Brown, 2004).

As the installation and update process is repeatable, the process can be tested. By using virtual machines, such as VirtualBox (Virtual Box Project, 2014), the process can be tested locally by developers. So the normal workflow of first testing This meant that the changes done into production environment could be completely tested before they are applied.

The installing and updating of the systems consists of two parts, configuring the infrastructure and deploying applications. In this case, the infrastructure was not complicated, as it consisted of one host. In this host, there were three major applications installed; Nginx as a web server, Request application and PostgreSQL database. This deployment is described in figure 2.

![Deployment of applications](image)

There's multiple different tools available for configuration management. For selecting the tool to be used, few different requirements were decided. The requirements are introduced in table 7 in order of importance.

The tool of choice for configuration management was Ansible (Ansible, Inc, 2014a). First reason for selecting Ansible as a tool for configuration management was its simple install. Ansible is a agentless configuration management tool, and it does not require any custom security infrastructure (Ansible, Inc, 2014b). The connection to the hosts is done with SSH, and the target host does not require any additional servers, databases or daemons.
Table 7: Comparison of configuration management tools

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple installation</td>
<td>The tool should be easy to install on the management host and target hosts</td>
</tr>
<tr>
<td>Simple configuration</td>
<td>The language the tool use should be easy to understand</td>
</tr>
<tr>
<td>language</td>
<td></td>
</tr>
<tr>
<td>Easy to use</td>
<td>Deploying configurations should be easy</td>
</tr>
</tbody>
</table>

The configuration language used in Ansible YAML. YAML is a simple, text based markup language. Below is an example of YAML used in Ansible:

tasks:

- name: make sure apache is running
  service: name=httpd state=running

Deploying configurations with Ansible is simple, as it is a standalone command line tool which can be used as long as the user has SSH connection to the target host.

Other configuration management tools that were considered were Puppet (Puppetlabs, 2015), Chef (Chef Software, Inc, 2015), Saltstack (SaltStack Inc, 2015) and CFEngine (CFEngine AS, 2015). All of these require installation of applications to management and target hosts. This was against the most important requirement, so there was no need for additional analysis.

Ansible has a concept called "inventory" (The Ansible Project, 2014). Inventory describes the hosts available and possible groups. The inventory file can also be used for setting variables for hosts. There can be multiple inventory files, and this was used for separating different environments from each other. So the same set of scripts can be kept in version control system and then used to configure different environments, as described in figure 3.

There was a separate inventory file for every environment, and these inventory files contained environment specific variables. These variables contained information about IP -addresses, database settings, integration points and host names. The variables are then used as parameters during installation and updates. This way the installation process can be repeated multiple times for different hosts. For example, in the figure 3 the contents of inventory file for development environment was similar to what is seen below.

host1.localdomain ansible_ssh_host=192.168.1.50 database_ip=192.168.1.50

First, there's a fully qualified domain name (FQDN) for host. Usually, this FQDN is used when the ansible makes connection to the host, but when doing local development is usual that the FQDN does not resolve to an ip. So the "ansible_ssh_host" tells what ip to use when making connection. The "database_ip" tells the IP -address of database, which in development use was same same as the host.

The contents inventory file of test environment were similar to what is seen below

host1.example.com database_ip=192.168.20.21

In this file, one host is defined, host1.example.com. As this is more stable environment, it's FQDN resolves into address. The database is now deployed in another host, so ip must be different.
5.3. Experiences

Experiences from the usage of the tools and process were gathered by observations, informal discussions and from different project review and retrospective meetings. Unfortunately, these were mostly undocumented.

During the project one benefit of the developed system rose above others, as the system enabled short internal feedback loops. Changes to the system were deployed into testing environment for acceptance, and the internal customer gave feedback to developers multiple times during one work day. With automated deployment, the deployment of new version into internal testing environment took only minutes. This meant that even small changes could be published for testing, as there were no overhead on deployment.

Even while the internal customer was giving feedback daily to developers, it was difficult to get the representatives of external customer to give feedback. One reason for this was that there were multiple representatives, and scheduling a testing session with all of them was difficult. Without explicit testing session, the representatives usually did not test the system.

Acceptance testing of every short iteration was considered to be too time consuming. Waiting for approval or change request from external customer after each small iteration would have slowed the development down as it would either caused a lot context switching or downtime as the developer would have been forced either to start developing a new feature or wait for feedback.

Even as the environment was simple, the testing of infrastructure changes was considered beneficial. The infrastructure changes were mostly installation of new applications, like anti-virus software, and configuration of web server proxies. As there were a testing en-
environment, the changes could be tested and validated before executing them in production environment. And with the usage of configuration management tool, the changes were executed identically in every environment.

In tables 8 and 9 these observed results of implemented requirements are mapped to principles of lean software development. In these tables the method from literature is in the leftmost column, and the effects of the goals are in the two rightmost columns. For example, fulfillment of the first of the goals, "Every change must be reviewed and automated tests must be executed", had effect to goal "Create Knowledge" from Shalloway as "Reviewing spreads knowledge". The fulfillment of the second goal, "Installation and update process must be automated", had effect to the same goal as "Assumptions can be validated faster".

Table 8: How fullfilling of the selected requirements affected to Poppendiecks principles and methods

<table>
<thead>
<tr>
<th></th>
<th>Effect of goal &quot;Every change must be reviewed and automated tests must be executed&quot;</th>
<th>Effect of goal &quot;Installation and update process must be automated&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eliminate Waste</td>
<td>Reduces defects</td>
<td>Reduces processes in deployment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduces waiting when deploying</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduces motion as artefacts are not moved around manually</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduces hand-offs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduces delays</td>
</tr>
<tr>
<td>Amplify Learning</td>
<td>Faster feedback</td>
<td>Faster feedback</td>
</tr>
<tr>
<td>Decide as late as possible</td>
<td>n/a</td>
<td>Shortens the time needed for deployment</td>
</tr>
<tr>
<td>Deliver as fast as possible</td>
<td>n/a</td>
<td>Shortens the time needed for deployment</td>
</tr>
<tr>
<td>Empower the Team</td>
<td>n/a</td>
<td>The team could make changes to environment</td>
</tr>
<tr>
<td>Build integrity in</td>
<td>&quot;Testing is done all the time&quot;</td>
<td>n/a</td>
</tr>
<tr>
<td>See the whole</td>
<td>n/a</td>
<td>Whole application is always running</td>
</tr>
</tbody>
</table>

39
Table 9: How fulfilling of the selected requirements affected to Shalloways principles and methods

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Every change must be reviewed and automated tests must be executed</th>
<th>Installation and update process must be automated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eliminating Waste and Deferring Commitment</td>
<td>n/a</td>
<td>Fast deployments reduce overhead</td>
</tr>
<tr>
<td>Using iterative development to minimize complexity and rework</td>
<td>n/a</td>
<td>Enables iterations</td>
</tr>
<tr>
<td>Create Knowledge</td>
<td>Reviewing spreads knowledge</td>
<td>Assumptions can be validated faster</td>
</tr>
<tr>
<td>Deliver early and often</td>
<td>n/a</td>
<td>Reduces effort needed when delivering</td>
</tr>
<tr>
<td>Build quality in</td>
<td>Enforces quality</td>
<td>n/a</td>
</tr>
<tr>
<td>Optimize the whole</td>
<td>n/a</td>
<td>Enforces whole production pipeline</td>
</tr>
<tr>
<td>Focus on Time</td>
<td>n/a</td>
<td>Reduces time needed for deployment</td>
</tr>
</tbody>
</table>
6. Conclusions

The research question of this thesis was: "How can lean development process be supported by tooling". The scope of this thesis was restricted to process and tools used from feature development to the deployment of software. The question was answered by doing a literature analysis of lean development process. This literature analysis covered the lean manufacturing and the lean software development.

After the literature analysis, two requirements for tooling were defined:

1. Every change must be reviewed and automated tests must be executed
2. Installation and update process must be automated

These requirements were implemented as described in chapter 3.

Using of proper tooling enables lean development, as it makes it possible to have short lead times while keeping quality at proper level. There is a usually multiple steps when software is deployed into testing or production environment. By automating these steps, the overhead of deployment can be minimized. This overhead can be considered similar to "setup time" in lean manufacturing.

Proper tooling is also critical for doing review and testing. The tooling must be able to force review and testing of every change. Review can be considered as a form of "Andon", a mechanism which can be used to stop production by any employee. Different forms of tests have similarities with "Baka-Yoke" -devices. These devices are mechanism which will prevent mistakes, like a part which only fits in one way.

The tooling enabled a lean software development process. From the developers' point of view, the process is straightforward. In the version control system, there's a branch which is used for current development. The developer implements required feature and commits it to the review system. This commit forms first "Patch Set" in a "Change". Then the CI -tool is notified about the new patch set and it starts running different test suites, like unit and integration tests. If tests are passed and reviewer has accepted the change, the patch set can be submitted. If either tests fail, or reviewer rejects the changes, the committer makes changes and uploads a new patch set to same change. This stage of development is called the review stage.

When change is submitted into the development branch, it becomes part of next release candidate. The current development branch is deployed into different environments using CI- and configuration management tools. These environments are used for acceptance and exploratory testing. The submission to main repository also starts the acceptance stage, where the patch set goes through different acceptance tests. These tests can include some automated tests, for example performance and browser compatibility -tests, and some manual acceptance tests in integration testing environment. If those tests pass, the patch set can be deployed into production environment. There are few differences between tests in review stage and tests in acceptance stage is that in the review stage the tests needs to run faster as they are meant to give rapid feedback to the developers. This
means that, for example, when running full user interface tests, only one browser is used and the main purpose of these tests is to ensure that the logic works in the client. In the acceptance stage tests are allowed to take more time.

Changes to the environment were done in similar fashion. Developer made necessary changes to Ansible playbooks and tested them locally. Then the changes were submitted into code review system, and after review, changes were applied to testing environment. Biggest difference from application development was that there were no tests. Testing of infrastructure changes was considered to be too expensive as the environment was simple enough to be tested manually.

The context variables of this study were presented in table 1. Within these variables, the results are applicable to different cases. The most notable of these variables were the size of team, kanban process and lack of dedicated system administrators and testers.

The most interesting aspect for future research would be the engagement of end user in the process. As lean development tries to optimize small batch sizes, the end-user faces a new set of challenges. The rate of changes is become faster, but the changes are smaller. This causes challenges in acceptance testing, as there aren't any clear releases.
Bibliography


