Advances in Streamlining Software Delivery on the Web and its Relations to Embedded Systems

Kasper Hirvikoski

Master's thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, April 1, 2015

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Kasper Hirvikoski

Työn nimi — Arbetets titel — Title

Advances in Streamlining Software Delivery on the Web and its Relations to Embedded Systems

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	April 1, 2015	44

Tiivistelmä — Referat — Abstract

Abstract.

ACM Computing Classification System (CCS):

Avainsanat — Nyckelord — Keywords
keyword

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Software delivery on the web has evolved over the years into a rather established process. A software is developed iteratively through multiple phases, which ensure the user's requirements and the quality of the product or service. These phases form what is called the deployment pipeline [Fow06, HF11, Fow13a, Fow13b].

A deployment pipeline nowadays usually consists of at least three stages: development, staging and production. Organisations alter these depending on their size and needs. Using modern iterative and incremental processes, a software is developed feature-by-feature by iterating through these steps. Development starts in the development stage where developers build the feature requested by the customer or user. The feature is then tested in the staging phase, which represents the production setting. When the feature has been validated, it is then deployed to production. If necessary, each stage can be repeated until the feature is accepted. Each step is short and features are deployed frequently — in some cases even multiple times a day [O'R11, Sny13, Rub14].

Software engineering consists of various different processes and practises for ensuring the quality of a product or service — nowadays more or less based on Agile and Lean ideologies and practises [Ōno88, BBvB+01a, Fow05, Mon12]. At the low level, developers use source code management to keep track of changes to the software and to collaborate with other team members. To reinforce that features work as intended, developers write automated test cases. Teams can also use more social methods — such as reviewing each other's code — to validate the implementations. These practises form the basis for Continuous Integration and Continuous Deployment [Fow06, HF11, Fow13a, Fow13b]. Software changes are frequently integrated, tested and deployed — automatically in each stage. The first two form Continuous Integration and the latter Continuous Deployment. If any stage fails, the process starts from the beginning.

The web enables the use of the deployment pipeline and its practises in an unprecedented way [KLSH09]. Due to the distributed nature of the Internet, software can be deployed as needed and the user always sees the newest version without the need of any interaction. This eases the use of many cutting-edge methods [KLSH09, FGMM14]. Deploying software as needed has allowed developers to experiment with different implementations of a feature. These changes can target anything from a more optimised algorithm to something more user-faced, such as improvements to the user experience of a product [KLSH09]. These practises have started to formalise as Continuous Experimentation [FGMM14].

Not all software can be developed easily this way. Many embedded systems, which have a dedicated function within a larger mechanical or electrical system, require hardware to accompany the software. Many of the features

are not user-focused and are limited by hardware. This presents a variety of challenges to overcome. Hardware can require thorough planning and iterating can take time. Contexts such as cross-platform support, robotics, aerospace and other embedded systems pose interesting cases. Many of these contexts can at a glance seem regarded as models for more traditional sequential software engineering processes with heavy planning, documentation and long development phases. Partly, this is still the case. However, even NASA's earlier missions have iterated on the successes and failures of previous ones [LB03]. Even though it can be more difficult, software related to hardware can be build and tested iteratively [LB03]. New approaches from prototyping electronics to 3D-printing have provided novel ways for building hardware iteratively.

This raises an interesting research topic — *presenting the advances in streamlining software delivery on the web and relating its practises and their advantages and challenges to embedded systems.* Using case studies to identify which Agile and Lean practises are used, how they could be improved and how new practises could be incorporated to embedded settings. Moreover, the aim is to identify if modern Continuous Integration, Delivery and Experimentation practises are used. Not just in a strict sense, but trying to discover what practises are possible in these settings. Can we determine how they compare to the way the web is utilised as a platform?

My hypothesis is that there should be no reason why many of these practises could not be successfully used and cleverly adapted to hardware settings. (In the context of this thesis, I also refer to embedded settings as hardware related.) Progress is an organisational issue above all. My research method for this thesis was reviewing the current practises in literature and industry. I also conducted several semi-structured interviews with the academia and industry working on leading embedded systems to get a view on if and how the deployment pipeline has changed the development of hardware related products.

This thesis is structured into seven chapters. Following the introduction, Chapter 2 outlines how software delivery has progressed from a structureless process following a code-and-fix mentality, to what is now considered the leading edge of iterative development. This sets the scene for understanding the rationality behind being adaptive to change and how the user is an essential part of the process. Chapter 3 describes how software delivery has embraced primarily a three staged pipeline for deploying new features to the user and how the web has provided an effective platform for the deployment pipeline to exist by streamlining and automating many of the practises used by modern development. Chapter 4 delves into the challenges related to delivering software that is firmly linked to hardware. It deliberates about how the deployment pipeline could be integrated into these embedded systems. Chapter 5 presents the results from the interviews collected from the field. The idea is to incite discussion — through the view of people working on

embedded systems — about what is the current state of software delivery in these settings and how it could be improved. Finally Chapter 6 concludes this work by making conclusions about the gathered knowledge.

2 Software Delivery

Software development has changed notably in the past few decades, nonetheless it still being a young field. Most software development can be seen as disordered chaos with a mentality of coding first and fixing later [Boe88, Fow05]. A software is built without much of an underlying plan and the design of the system is a result of many short term decisions. This can work well if the system is small, but as it grows, adding new features becomes easily too much to handle.

Going back, it was not until 1968, when the term software engineering was introduced by the NATO Science Committee [NR69]. By that time, it was considered that software development had drifted into a crisis, where a wider gap was forming between the objectives and end results of software projects. Additionally, it was getting increasingly difficult to plan the length and cost of development. A typical consequence was a long and manual test phase after a system was considered "feature complete" [Fow05]. As a consequence, projects did not meet their deadlines and budgets. A collective effort was put in place to establish a more formalised method for software development — similar to traditional engineering such as building bridges. It was considered necessary that the foundation for delivering software should be more theoretical with laid principles and practises [NR69]. Software development had to become more predictable and efficient. By 1969, the term software engineering had become well-established in the field [BR70].

Software development processes began to form. One of the primary functions of software processes was to determine the flow and order of how software is developed in stages [Boe88]. Notably in 1970, Winston W. Royce published a paper that described a formal approach for sequentially developing a software based on previously used practises [Roy70]. It was only later named as the Waterfall model [Boe88, LB03]. See figure 1. The process consists of multiple stages that should be carried after the previous has been reviewed and verified. It begins by mapping the requirements for the entire software, then proceeding to designing the architecture, followed by implementing the plan, verifying the result is according to the set requirements, and finally maintaining the product [Roy70]. Generally, all this is considered as a linear timeline with a start and end. Each stage is planned and documented thoroughly. The concept thus being that as each step progresses, the specification of the software becomes further detailed.

However, contrary to what has been referred, Royce presented the model as a somewhat flawed, non-working model [Roy70]. If any of the stages fail,

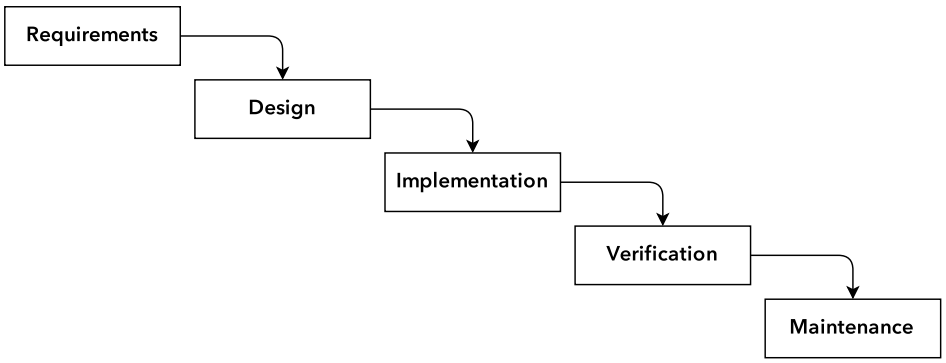

Figure 1: Waterfall Model

serious reconsideration of the plan or implementation might be necessary. Therefor sequentially following the stages would not produce what was intended and inevitably previous stages would need to be revisited [Roy70]. Royce however found the approach fundamentally sound and proposed the method should be carried out twice [Roy70, Boe88]. It should begin by creating a prototype and only then proceed in executing it. Nevertheless, this was overlooked and the single pass Waterfall model became the dominant software development process for software standards in government and industry [Boe88, LB03]. It is still used widely in some fields. It is noteworthy to mention that Royce has later been stated as a supporter for iterative approaches [LB03].

Engineering methodologies, also called as plan-driven methods, are considered heavy. They also have not been noted for being terribly successful [Fow05]. The Waterfall model has been criticised as too linear, controlled, managed and documentation-oriented [Boe88, LB03, Fow05]. Waterfall pushes high-risk and difficult elements towards the end of the project [VB09]. Royce considered a software completed only when in addition to its implementation the documentation of it was acceptable — sometimes hundreds or even thousands of pages [Roy70]. It was declared that developers should prioritise keeping the documentation up to date over everything.

More lightweight iterative processes were proposed as opponents for sequential software development in the later part of the nineteen hundreds [LB03]. In fact, early applications of iterative and incremental development dates as far back as the mid-1950s — with many names such as incremental, evolutionary, spiral and staged development [Boe88, LB03, Fow05]. All of these sought in developing a useful compromise between no process

and too much process [Fow05]. They also focused to be less documentation-oriented and in many ways more code-oriented. It was considered that the documentation for a project should be the code itself, not some external specification.

Fast-forward to 2001, when a group of software developers met to discuss new lightweight development principles. As the result of these discussions, a manifesto for Agile software development was published [BBvB+01a]. Four principles were proposed for Agile software development: *focusing on individuals and interactions* over processes and tools, *focusing on working software* over comprehensive documentation, *focusing on customer collaboration* over contract negotiation and *responding to change* over following a plan. The manifesto does not dismiss the value of the latter, but considers the former even more valuable [BBvB+01a]. From thereon, iterative processes have started to gain mainstream traction in the field [LB03, Fow05].

Software development is now considered as an ongoing process, where a product should be build in small increments, iteratively going through the development stages. Repeating this process as long as required. Software delivery moved from a linear approach to a more recurrent cycle. See figure 2. The notion is not to resist change. Most of the ideas were not new and had been successfully used already in the industry for a long time before the manifesto [Fow05]. At that time, an urge revived to treat the ideas more seriously. Instead of planning, designing and implementing a whole software once, a software should be build iteratively by repeating all of these steps in shorter more controllable parts. Hence, any issue or miss-communication could be discovered early on and fixed accordingly.

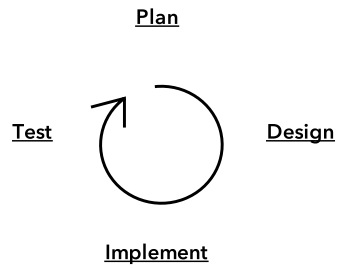

Figure 2: Iterative Development

2.1 Adapting to Change

The demands for software products are continuously shifting. It is not always obvious what the users want. In some cases, users do not know what they are looking for, until you show them what they need. It is hard to know

what the value of a feature is before you see it in reality [Fow05]. Reality allows the user to learn how a feature works. An average client has little knowledge on how software products work or how they are built. Therefor, it is exceedingly difficult for a client to map specifically what they require from a software product. Software development should be more people-oriented than process-oriented [Fow05]. This requires a different kind of relationship with a customer. In most cases a user can be considered as the customer. What is notable, even Royce emphasised, although loosely, the value of customer commitment during development [Roy70].

In most cases, rigorously planning a software beforehand will not work [LB03]. It is not uncommon that an idea will change quite a bit during its lifetime. The key problem that plan-driven methods face is the separation of design [LB03, Fow05]. The concept is similar to traditional engineering: engineers will build a precise plan which will then be followed by a different set of people. In such, architects and engineers would first design a bridge and then a construction company would build it. A classic example is how Henry Ford standardised car parts and assembly techniques so that even low-skilled workers with specialised machines could manufacture low-priced cars to the masses [Pop02]. This lead to an explosion of indirect labour from production planning, engineering to management. All of this required a lot of overhead [Pop02]. Designing, which involves creative and more talented individuals, is far more difficult and less predictable than construction [Fow05]. Commonly expensive as well. Construction on the other hand, although more labour intensive, is considered more predictable and straightforward after a plan has been completed. The premise is that by following this methodology in software engineering, we could reasonably predict the time and cost of software "construction".

When Royce first defined the Waterfall model, he stated that the documentation of a software is both its specification and design [Roy70]. Without documentation, there would be no design nor communication. Still to this day, no one has found a solid way of designing software in a manner that the plans can be thoroughly verified before construction [Fow05]. A design can look good on paper, but be seriously flawed when you actually program it. When building a bridge, the cost of the design is fractional to the cost of construction [Fow05]. It was thought beneficial that "low-skilled" programmers would produce the code, while a few "talented" architects and designers did the critical thinking [Pop02]. Naturally, this lead to a Waterfall-like process with different people involved in different stages. In software, the time spent implementing is fractional to the time spent designing. Essentially, coding is designing. Coding requires creative and talented people. People are considered one of the most important factors in software development. Developers should be in control of all the technical decisions. There are serious flaws in separating different tasks to different "specialists", but this is how software engineering was regarded as [Roy70]. It is still quite common

that a developer writing the code and a tester writing the tests, are not the same person. Even though studies have indicated that developers familiar with their own implementation tend to write more thorough tests [MNDT09]. The metaphor for traditional engineering is in practise flawed [Fow05]. Many projects simply fail in what they are trying to achieve and as a consequence the results will never be used [LB03]. Some reports have indicated that one of the top reasons for project failures is related to Waterfall-practises [LB03].

Andy Whitlock, a product strategist, drew a fitting mental picture about changes [Whi14]. You see the road ahead as a clear and straight path to an objective you have set. What you do not always realise, is that the path will have its twists and turns along the way. What you can really only do, is to plan to a certain point ahead. The rest of your path will be a gloomy fog in the distance. You need to be ready to make difficult choices along the way. Agile development tries to create a framework, where processes and practises can take these requirements into consideration. Even to the point of changing the process itself [Fow05].

2.2 Being Agile

Prominently, being "agile" means effectively responding and adapting to change and not resisting it. After all software is supposed to be "soft" [Fow05]. These course corrections are rapid and adaptive. The highest priority is to satisfy the customer trough continuously delivering valuable software from early on [BBvB+01b]. Software should be delivered frequently in short increments. These increments, also referred as iterations in Agile development, should take no more than a couple of weeks to a couple of months — the shorter the better [Fow05]. After each iteration a working software is delivered with a subset of the required features. These features should be as carefully tested as a final delivery. Throughout the project, one of the ways for a team to respond to change is by having effective communication among all stakeholders for the product daily. The best means for conveying information is face-to-face conversation — not documentation [BBvB+01b]. At every iteration the customer has control over the process by getting a look on the progress and then altering the direction as needed. This continuous feedback has been attributed as a key factor for success in agile projects [DD08].

Commonly a stakeholder represents the views for the users or clients. By taking the stakeholders as part of the team, developers can react when something is not working as intended. The importance of customer reviews and acceptance was already noted in the Spiral model, which dates as early as the 1980s [Boe88]. Studies also show that developers see the ongoing presence of stakeholders helpful for development [DD08]. An Agile process is driven by the customer's descriptions of what is required [BBvB+01b]. These requirements may be short-lived and that must be kept in focus. Changes are unavoidable [Fow05]. Users' desires evolve and this must be

harnessed to the customer's competitive advantage [BBvB+01b, Fow05]. Even if deciding a stable set of requirements would be possible, outside forces are changing the value of features too fast [Fow05]. It is not uncommon for requirements to change even late in development. If you cannot get a fixed set of requirements, you cannot get a predictable plan. This is what makes plan-driven development inefficient. Royce stated that required design changes can be so disruptive that the software requirements upon which the design is based on and which provide the rationale for everything can be breached [Roy70]. Even so, predictability is highly desirable [Fow05]. It is an essential force in what makes a model work. Adaptivity is about making unpredictability predictable. This creates a framework for risk-control in the project.

One key premiss for Agile development is to reduce the burden of the process. Working software is the primary measure of progress [BBvB+01b]. A process should not hinder the work of a team — on the contrary it should permit the team to function to its full extent. By organising the team to be in control of the process, the framework facilitates rapid and incremental delivery of software. Still, no process will make up for the skill of the individuals working on the project [Boe88, Fow05]. Projects should be based on motivated individuals [BBvB+01b]. Motivation is maintained by creating a constructive environment and giving the necessary support when needed. Trusting the team is of utmost importance [BBvB+01b]. Morale has direct effects on the productivity of people [Fow05].

One of the weaknesses of adaptability is that in its essence it implies that the usual notion of fixed-priced software development does not work [Pop02, TFR02, Fow05, HOAB12]. Instead completely new approaches have to be used. Contracts should allow incremental deliveries which are not pre-defined in the contract, yet still ensuring the customer receives business value [Pop02]. You cannot fix scope, time and price in the same way as plan-driven methods have tried. The usual agile approach is to fix time and price and allow the scope to vary in a predetermined manner. Value is not only created by building software on-time and on-cost, but by building software that is valuable to the customer. Yet, value unquestionably still is a philosophical problem.

2.3 Ensuring Quality

Assuring quality is not an easy task. Applying measurements to software development is demanding. Something as simple as productivity is exceedingly hard to quantify. Let alone defining the value of something — from monetary significance to anything like user interpretations. ISO 9000 -standard defines quality as the extent of how well the characteristics of a product or service fulfil all of the requirements, the needs and expectations, set by the stakeholders [ISO05]. IEEE defines software quality as the degree to which a

system, component or process meets the specified requirements as well as the customer's and user's needs or expectations [IEE06]. Both definitions focus strongly on fulfilling the user's needs. In this sense, quality and value have similar interpretations. It is also relatively hard to distinguish what is success, most of times this is based on the impressions of the people involved though sometimes some kind of measurement can be used as an indicator. Most of times these are focused on time and monetary value.

Software development is challenging. Users perceive quality as working software, but most of all emphasising good technical design and implementation makes the development process easier. People, time and money are limiting factors for ensuring quality. Strict deadlines and scarce resources have direct effects. Furthermore, human factors play a considerable role [DD08]. Several empirical studies reinforce the significance of Agile development processes and practises as improving quality in software [DD08, SS10, DNBM12]. Evidently being "agile" should in the long term make development more predictable and eventually lead to shorter development times and minimised costs [DD08]. This provides an environment for being adaptive.

In addition to focusing on satisfying the customers needs, Agile development promotes continuous attention on technical excellence and good design practises [BBvB$^+$01b]. Even so, this should not be accomplished by hindering simplicity. Simplicity maximises the amount of work that can be accomplished. The Agile Manifesto states that the best architectures, requirements and designs emerge from self-organising teams [BBvB$^+$01b]. After regular intervals, the team members reflect on how they have performed and how they can become more effective. This is how the team can then tune and adjust its behaviour appropriately. The problem with traditional engineering is the separation of responsibility [Pop02]. Employees are not expected to take responsibility for the quality of a product. By giving responsibility back, you add accountability to the process. Developers will take quality more seriously.

Achieving quality is above all an ambition. No process or practise will account for quality if the developers are not willing to pursue it. A team must set mutual working principles which define how development will aim to deliver quality. These include anything from coding conventions to reviewing each other's work. Quality should be a concerted effort. Above all, in addition of partly being a quantitative metric, value is qualitative.

The Agile practises also have their critics. Firstly, there is a lot of preconceptions about being Agile mostly driven by seeing the process as supporting no design nor documentation [HMP$^+$10]. Secondly, one of the biggest criticism is that there is a shortage of scientific support for many of the claims made by the Agile community [DD08, DNBM12]. However, empirical studies have shown favourable results and studies have increased significantly [DD08, SS10, DNBM12]. Agile development has been critiqued for a lack of focus on the architecture and design behind software. Practises

are also rarely applicable by the book and therefor they are rarely used as such. Additionally, Agile development has a strong focus on small teams and as such many have struggled in seeing them used in larger distributed environments [TFR02]. One of the key concern has been also how to handle subcontracting which tends to lean heavily on documentation. Regards to embedded systems, there is issues in adopting Agile principles and practises to safety-critical environments [TFR02]. It is no surprise that it takes time and effort to introduce the methods properly [DD08]. In most cases, once you get past the first obstacles many of these hurdles are not an issue.

2.4 Processes and Practises

Processes and practises assist the development process. They create the framework and guidelines within a team can develop a suitable environment to deliver software [Kni07]. Martin Fowler discusses about a process as a part of the design [Fow05]. Processes and practices also help to maintain quality. Agile development has become well known and organisations are showing interest in adopting these methods [DD08].

At the low level, developers use source code management to keep track of changes to the software and to collaborate with other team members. Source code management enables multiple developers to work on a single project, while also creating a history for the entire project. When a problem arises, developers can go back in time to look at the source code at any given point in time. To ensure features work as intended, developers use automated test cases to verify expected behaviour. There is a clear correlation between higher test coverage resulting in fewer errors in software [MNDT09]. By and large, writing tests for code has a better chance of signalling errors than untested code. Teams can also use more social methods — such as reviewing each other's code — to validate the implementations. Pair programming, coding dojos and hackathons provide tools for improving skills and solving complex problems together [DD08, HHLV13, RKDB$^+$13].

Most iterative development processes vary by the iteration length and how iterations are time-boxed — from a couple of weeks to a couple of months [LB03]. Agile development only provides a framework for software delivery. It does not specify concretely how development should be organised. Instead, development methods are incorporated to give focus on how software should be develop. Most notably, Scrum and Extreme Programming have created a structure for Agile development [LB03, Fow05, SS10]. Scrum provides a framework for managing development. It focuses on how development should be planned, managed and scheduled. It does not provide any strict practises, instead it gives guidelines for how customer requirements should be discovered, prioritised, and how the development of these features is split into iterations.

Scrum has been strengthened with ideas and practises which focus on

simple design, small releases and coding standards. These also include test-driven development, refactoring, pair programming, collective ownership of the code, utilising on-site customers and continuous integration. These are defined in Extreme Programming [Bec00]. Continuous Integration aims at creating a process where developers integrate new features in small chunks and as often as possible into the software. In test-driven development, features are developed by writing the expectations for a feature as tests before actually implementing the code. When possible, code should also always be refactored to improve existing implementations. In pair programming, developers develop features in pairs.

Extreme Programming practises have been easier to be studied than management processes such as Scrum [DD08, DNBM12, KRM+13]. Most of the practises have been regarded as improving the quality of software and most developers tend to support them [DD08, SS10]. What is more, these practises make software development progress visually and aurally available. This increases the confidence that you are building what users want. Teams also improve the quality of their work: communication and understanding is improved, knowledge is transferred among the team and developers are more confident about their work. This increases morale and productivity [SS10]. A productive team is a right mixture of talented people. A team will not work if its members cannot work together. Regardless, it is still clear that many of the practises need more empirical studies to validate their claims [DD08].

2.5 From Agile to Lean

As time has passed, developers have simplified software delivery even more. Agile has turned into Lean. Popularised, being "lean" means reducing the amount of "waste" around software development. Craig Larman and Bas Vodde have criticised this simplification [LV09]. Above all lean thinking is defined by respect for people and continuous improvement (kaizen). You need to challenge everything and embrace change. One way to achieve this is to remove anything from the process that does not have benefit. Some principles for Lean development are: *eliminating waste, amplifying learning, deciding as late as possible, delivering as fast as possible, empowering the team, building integrity in and seeing the whole* [PP03]. Lean refers to an approach in manufacturing that was originally developed by Toyota in the 1950s [Fow08]. It became well known for the rest of the world in the 1990s when westerners started to explore why Japanese where leading in so many industries. Principles of lean thinking are universal and have been applied successfully in many disciplines [Pop02]. Many of the ideas presented by Lean Manufacturing have influenced the roots of Agile in software development. Both place notable attention on adaptive planning and people-focused approaches. In recent history, the software community has started to embrace Lean principles with more clarity [Fow08]. Agile and

Lean are deeply entwined — you are not only agile or lean, you are both agile and lean.

Lean emphasis doing work just-in-time, not too early and not too late. Instead of dealing with a lot of up front design, just-in-time delivers a better paradigm [Pop02]. The principle is to structure processes so that they do nothing but add value and as fast as possible. This is accomplished by removing unnecessary waste and moving decision-making to the developers. "Mass-production" requires immensive amounts of work to create a process that does not directly add any value. This takes time. Time that is of the essence. Being "lean" means reducing this framework to the minimum and providing customers value with significantly fewer resources. As a notable example, Pierre Omidyar created eBay by responding to daily requests for improvements to the service [Pop02]. Many of these improvements where integrated overnight.

Iterations have in some cases even turned into building single features at a time. The idea of time-boxed iterations has become less important, you build a single feature and once done continue to the next one. Instead of building a frame for a ship, a development process should essentially start with building a boat first. To evaluate an idea, developers should begin by developing a minimum viable product (MVP) to validate the implementation has value [Rie11]. The notion is that sometimes ideas can be evaluated quicker by implementing them rather than spending time with a committee to decide the requirements [Pop02]. Even Royce hinted on prototyping in the Waterfall model and later the Spiral model integrated this as a principal concept [Roy70, Boe88]. Only a minimal effort should be put into place to specify the overall nature of a product. Being "adaptive" has transformed into quantitatively assessing what effects changes have. This so called build-measure-learn cycle or continuous innovation has transformed how features are developed and validated [Rie11]. See figure 3. Either you change you heading by pivoting or you persevere with the choice you have made.

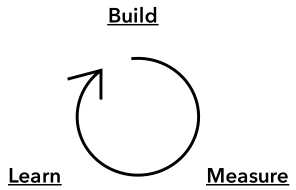

Figure 3: Build-Measure-Learn Cycle

This mentality of continually innovating has become popular among

software startups [Rie11]. An entrepreneur with a big vision and stubborn determination can charge through obstacles and make whatever their ambition is. The passion, energy and vision that people can bring to new ventures are resources that should not be disregarded. However, it is difficult to choose when to take a new direction. These decisions can be backed by anything from intuition to external indicators such as user feedback. In any case, making changes requires courage and determination. The build-measure-learn cycle makes it possible to test reactions, learn and iterate. Making decisions purely based on intuition can be risky. Learning, adapting and making changes are guided by data [Rie11]. It has even been suggested that experiments with negative user effects should be conducted [KLSH09, KDF+12, Bos12]. Still, personally I would argue that these experimentations need to be carefully planned. Not all users tend to agree with the use of somewhat unethical experimentation practises [RM13]. If a minimum viable product does not focus at all on the user experience, there is a high chance that users will seek for alternative options.

2.6 Focusing on the Essential

You eliminate waste by using activities and resources that are only absolutely necessary. Everything else is waste. The idea of doing things right has been widely misused as a justification for doing plan-driven development with heavy planning [Pop02]. Instead, software should be developed with short incremental cycles to ensure feedback and learning. This way, developers learn when something can be adjusted and most of all the customer can have direct effect. You concentrate on building features that will bring value by moving other decisions to as late as possible. Commitment should be delayed until there is certain demand that indicates what the users really want.

By delivering as fast as possible you ensure you can concretely see whether the feature has value or not. Development should centre on the people that have most effectiveness. Responsibility should not be transferred away from these people. Developers should have control on every part of the process. If something does not work, they have the chance to make a difference. Developers should challenge their skills instead of separating different tasks to different people. Maintaining responsibility and keeping a keen awareness and interest on the process builds integrity. All the skills required to build the product should reside in the team: from understanding the customers' needs, to architecture, design, development, testing and management. When these principles are applied to software development you ensure you see the product as a whole. Fundamentally, Lean development tries to not hide the unknown.

Such as Agile development, Lean development is more or less a mindset. It emphasis certain aspects that guide development. Developers still have a lot of flexibility in how they utilise these guidelines in development. Lean

development has also brought some popular practises such as Kanban, which is a visual way of organising work into tasks and limiting the amount of work currently in progress [Mon12]. These tasks are for example written down on sticky notes and their progress is made evident by moving them through different production stages: to do, doing and done (for example on a wall of an office).

3 Deployment Pipeline

Someone thinks of a good idea, but how do we deliver a feature as quickly as possible? In many software projects, releasing new features is a manually intensive process. Typically delivery of software to the users occurs at the very end of the project [HOAB12]. Optimally, this should not be the case. Releasing software has a tendency to fail. Fixing major production issues after development can be hard to accomplish. How can we for example determine a software will work in its intended environment and not just on the developer's machine?

A deployment pipeline is the foundation for many modern software development practises. Anything that can be treated as construction should be automated [Fow05]. One of the obstacles of build and test environments is that you want to be able to build fast so that you can get fast feedback [Fow13b]. Deploying software manually is a fragile and time-consuming process. Ideally a software should be able to be deployed by anyone with the simplicity of a push of a button. No struggle in finding out the steps to do so and automated ways in discovering if something has gone wrong along with rollbacking when this happens. To ensure quality, you have a comprehensive set of test cases for your code. Running these tests manually can take a long time. A deployment pipeline handles this by breaking up your build — with automated scripts and tasks — into multiple stages. Each stage increases your trust that everything is working as expected.

Jez Humble and David Farley describe three common anti-patterns for software delivery: deploying software manually, deploying to a production-like environment only after development is complete and manually managing production environments [HF11]. Most applications are rather complex to deploy and the process involves many moving parts. Eventually this leaves the process prone to human error. The purpose of a deployment pipeline is to provide automated and frequent releases of features. Any change in the software should trigger a feedback process. Features should be deployed so that developers receive feedback promptly and can act upon it. Features should be considered complete only when they are deployed to production [HF11]. Reflecting the ideas behind Agile and Lean development. The end result of a deployment pipeline is a production ready software or a cloud deployment. Without a deployment pipeline, development undoubtedly

slows down. One is not truly incited to develop features incrementally.

Figure 4: Deployment Pipeline

Typically a deployment pipeline consists of at least three stages: development, staging and production [HF11]. See figure 4. Stages can be automated or require human interaction. A deployment pipeline begins by a developer implementing a feature and building the software. The pipeline runs automated test cases and other build related checks, but can also include manual checks that cannot be automated. These tests are also run in a production-like "staging" environment to be sure the feature will work once it will be deployed. The final stage is deploying the software to production. The purpose of a deployment pipeline is to detect any changes that will lead to issues in production [Fow13b]. In addition, it gives you visibility about changes in your development process. This visibility is easier to follow.

3.1 From Development to Production

Developing a software feature starts with the developer. Developers carry out ideas and turn them into code that implement a feature. Everything that is required to build an application should reside in a shared code repository [HF11]. This source code management keeps track of changes and makes it possible for several people to work on the same project. It also creates an invaluable history, where developers can go back in time and look through how the project has changed over time. This makes troubleshooting easier. A developer should be able to pull a local copy of the shared repository and with minimal effort — such as installing the required programming frameworks — get the application building and running. This should be an automated and straightforward process.

A developer should be encouraged to implement features in small chunks. Continuously integrating code is first of all a practise, not a tool [HF11]. Development practises require a degree of commitment and discipline from developers. Developers write code, related automated test cases and manually test that the feature is working as desired. This includes interacting with the actual software. After finishing, a developer runs all the test cases for the project. Thus making sure that any local changes have not broken something else in the application. Finally, the work is integrated back into the shared repository. In small chunks, even multiple times a day. Integrating code is

about conveying information amongst the team. Other developers can see what has changed, test new features and make sure nothing conflicts with the work they are doing. This prevents any major integration troubles in the future. It is common that developers also have social practises for validating implementations. New features might not be integrated to the main branch before they are reviewed by another team member.

A local development machine is only a local setting. An application can seemingly work as expected on a local environment, but this must be verified against the production environment. Once the feature has been integrated into the repository, it is then immediately tested in a production-like setting, usually called as "staging". A server runs the scripts and task related to building and testing the application. These include tests cases and any other checks that make sure the code is satisfactory. Tasks can include anything from analysing code style which include spotting human errors to verifying the test coverage. If anything fails, the developers should notice the issues relatively soon and fix them accordingly. The idea of a staging environment is to simulate the production environment. Tests should be run under this controlled environment to make sure the software works as intended in the desired final environment.

Finally, the last step includes deploying the application to production. It is not always feasible or desired to deploy software straight to production. Staging software adds a secondary barrier to verify the application. Customers can also see if the feature works as intended and any changes can still be made before deploying the feature to users. In web-application, features can even be deployed gradually starting from a subset of users [Bos12]. If this goes well, gradually more servers will be deployed with the new feature. At any point in time, the deployment can be rollbacked to a previous version if any issues are raised.

In addition, managing the staging and production environments should be made as easy as possible. The application stack should be easy to maintain and all related configurations should be in a repository [HF11]. Any developer should be able to create a production environment precisely, preferably in an automated fashion. Virtualisation and service-oriented platforms can help to achieve this.

3.2 Continuous Integration

Continuous Integration (CI) is a development practise where members of a team integrate their work frequently, usually multiple times a day [Fow06]. This leads to multiple integrations of the software every day. As described previously, each integration is verified by an automated build and test process to detect any errors as soon as possible. Less time is spent in trying to find bugs, because they are discovered quickly. Only if the source builds and tests without any error, can the overall build be considered good [Fow06]. If and

when a developer breaks the build, it is their responsibility to directly fix and repeat until the shared state is functional.

The essence for continuous integrating is maintaining a controlled source code repository [Fow06]. Software projects involve a lot of files and manually keeping track of these is hard. Source code management allows developers to keep track of changes to the source code and to collaborate with other team members. Any individual developer works only a few hours at time from this shared project state. After the work is done, the developer integrates their changes back into the repository.

Integration is a way of communicating with the team. Frequent integrations let team members know about changes to the software. This eases any changes necessary in their work. Developers can also see if their work conflicts with any other team member. It also encourages developers the keep their work in as small chunks as possible. This significantly reduces the amount of integration problems by shortening the integration cycle and removing any unpredictability. Conflicts that stay undetected for weeks are hard to resolve [Fow06].

The integration process is run locally, but in addition the process should be run on a separate automated integration machine, a CI -server [Fow06]. A build can be started manually, but most of the time this process is automated as soon as the developer integrates their work back to the shared repository. This prevents any flaws that might not be discovered on a local environment. On a CI -server, the build should never stay failed for long.

Continuous Integration assumes a comprehensive test suite for the software. The tests are a integral part of the integration and build processes which in affect results in a stable platform for future development. It is easy to add new features since it is easy to integrate and test them against previous functionality. An integrated system and well-tested software is key for bringing a sense of reality and progress into a project [Fow05]. Documentation can hide flaws that have not yet been discovered. Untested code can hide even more flaws. Practises such as test-driven development enhance integration by introducing programmers into writing simultaneously tests as they write production code. In addition, writing tests before the implementation is a design practise. Of course, you cannot count tests to find every single bug, but imperfect tests are better than no test at all [Fow06]. It has been stated that projects that use CI, tend to have dramatically less bugs [Fow06].

3.3 Continuous Deployment

Continuous Deployment is a development practise where you build software throughout its lifecycle so that it can be deployed automatically at any given time [Fow13a]. Continuous Deployement requires that your pipeline enables you to do Continuous Delivery. The difference between Continuous Delivery

and Deployment is that the first enables you to deliver new versions of your software easily with a push of a button whenever you so desire, instead the latter automates this process by doing deployments automatically to production, resulting in many production deployments each day [O'R11, Sny13, Rub14]. The ability to delivery software functionality frequently to users subsequently enables to continuously learn from real-time usage [HOAB12]. Usage data can be utilised throughout development, delivery and deployment. As a result the feedback-cycle becomes shorter.

You achieve Continuous Deployment by continuously integrating the features completed by the development team. Teams prioritise keeping software in a deployable state. Features are integrated, built and automatically tested to detect any issues. If no issues are raised, the software can be deployed automatically to production. Furthermore, you use environments that closely resemble the production environment to first see how the software performs before finally deploying it to users. By making small changes, there is a lower risk of something going wrong. When this happens, it is likely that these issues will be easier to fix.

The value of doing continuous deployments is that the current version of the software can be deployed at a moments notice without panic. Resources are not wasted in doing manual tasks. Deploying software frequently gives a sense of believable progress, not just developers declaring features done [Fow13a]. In addition of requiring extensive automation throughout the deployment pipeline, it also involves a close and collaborative working relationship between everyone from developers to system specialists involved in the software delivery process [HOAB12, Fow13a]. Lately this has been referred to as a "DevOps culture" [Fow13a]. In practise, developers should have control on how software is hosted and this should not be primarily outsourced [HF11]. Developers can make appropriate choices based on these decisions.

Continuous Deployment also provides a way of making the latest version of the software being always accessible. Other developers and customers can then effortlessly demonstrate, explore and see what has changed since the previous version. It enables stakeholders to test the system and give feedback. A substantial risk in the effort of building something is whether or not it is useful to the user. The earlier you have the change of evaluating the value of a feature (MVP), the quicker you get feedback on it. Using the web has enabled the possibility to deploy and explore features on a subset of users [Fow06, Fow13a]. This can then be used as factor in making decisions about how to proceed.

3.4 Continuous Experimentation

Innovation is a moving force for organisations, but notoriously hard to get right [BE12]. The world is never static, being able to figure out what works

and what does not can mean the difference between being on the top or becoming invisible [KLSH09]. Innovation is maintained by balancing between the number of ideas presented and those being practical. The web has for example provided a platform for easily establishing a causal relationship between changes and their influence on user-observed behaviour [KLSH09].

In the simplest form of these controlled experimentations, users are randomly assigned to two different variants of a feature: a) the Control and b) the Treatment. The Control represent the existing version of the feature and the Treatment a new version being evaluated. At large this is called as A/B testing. Data is collected with predetermined metrics from these experiments — metrics such as how the user behaves with the feature. From these results we can determine by statistical analysis which implementation is better, although surprisingly not always why. Different implementations can have very unexpected results [KLSH09, KDF+12, McK12]. It is intriguing how poor we are at assessing the values of our ideas — many assumptions are simply wrong [BE12, KDF+12]. Regardless of these assumptions having significant effects. Features are built because developers believe they are useful. Even worse, these opinions can come from managers not familiar with the area in question [KLSH09, BE12, Bos12]. Of course, the significance of intuition and luck should not purely be belittled.

Controlled experiments provide a methodology to reliably evaluate the value of ideas [KR04, KLSH09, McK12, Rho14, Wan14]. Passive feedback can provide much more valuable information than actively trying to ask feedback from users. Users can be blinded by how they act with features. By building a system for experimentation, the cost of testing and failure becomes small. This encourages innovation by enabling experimentation. Failing fast and knowing when an idea is not great is essential in making course corrections and developing better ideas. When we fail fast, we can also make improvements more faster. Due to the distributed nature of the web, these experimentations can be done in the background. New versions of features can be deployed frequently without the user even noticing the changes. This provides a thriving environment for experimentation. Experimentation can be used to understand what these user truly want [Wan14].

Continuous Experimentation is a development practise where you build an environment where you can continuously deploy new features and enhancements to the user [FGMM14]. As a result, developers can continuously get direct feedback from the user by observing usage behaviour. This requires an environment where you automatically deploy new features, collect metrics from usage, analyse them and furthermore integrate the results into the development pipeline. Instead of heavy up front testing, alerts and post-deployment fixing should be tried [FGMM14]. When an issue is discovered, the feature can be rollbacked promptly, sometimes even automatically. The adoption of cloud computing has clearly shown a different approach to adding frequent and rigorous experimentation to the development process [Bos12].

Continuous Experimentation makes substantial use of minimum viable products as the basis for an hypothesis and experiment. Choices are made by analysing the data gathered from this minimal implementation. A hypothesis is either supported by the data or not. It is necessary to base decisions on sound evidence rather than guesswork [FGMM14]. Controlling every aspect of development will not work, instead you need to sustain a culture where teams can move and innovate with the experimentation system [Rie11].

Indeed, the leading edge of Continuous Experimentation is even starting to favour experiments over predefined test cases [New15]. Instead of rigorously testing features beforehand, automatic analytics are run in production. Heuristics are used to immediately discovers issues and alert about their consequences. This is also referred to as Canary testing [HF11, Sat14]. (As cruel as it can sound, canaries where used to test whether toxic gases where present in coal mines [Sat14].) Changes are rolled out slowly to a small subset of users before rolling out new features to the entire infrastructure. Used by companies like Google and Netflix [Whi11, Sch13]. Testing in production will be as production-like as it can be. Another variation of production-testing is Blue–Green deployment, where you maintain two practically identical production environments as a backup to enable hot-switching between two alternatives and even feeding some transactions to the blue variant and some to the green one [Fow10, HF11]. Lately, Continuous Experimentation has become popular among companies building web-products such as Etsy, Facebook and Twitter [McK12, Boh13, New13, Rho14, Wan14].

3.5 Using Web as a Platform

People have barely touched the surface of what the web can provide. The acceleration of digital products and services means the web will become more and more irreplaceable for software-intensive products and services. Cloud computing has emerged as a new model for hosting and delivering services over the Internet [ZCB10]. Infrastructure has become more cheaper, more powerful and more available than ever before. This made many of the current practises impossible back in the day [Roy70]. The cost of infrastructure is becoming negligible [ZCB10, Bos12]. Cloud computing has made it possible for general utilities such as computing power and storage to be leased and released over the network when necessary. This is highly scalable and adaptive, mirroring many of the Agile and Lean ideologies. Organisation can start small and increase resources only when there is rise in demand. One of the key reasons is the simplicity associated with not having to deal with hardware constraints [BE12].

Cloud computing uses a service-driven model. Typically, cloud computing provides three categories of services: infrastructure such as computing and storage (Infrastructure as a Service), platforms such as operating systems and software development frameworks (Platform as a Service) and on-demand

software applications (Software as a Service) [ZCB10]. It is no surprise that many of these services have become platforms for the deployment pipeline. Amongst all, this movement has generated service-oriented platforms that provide many of the common functionalities involved in software delivery. There is clear trend for continuously testing and experimenting with new innovative functionalities and deploying these regularly to users [BE12]. Especially web-applications and services can be developed and deployed with ease. Collecting data is a well-established strategy [HOB14].

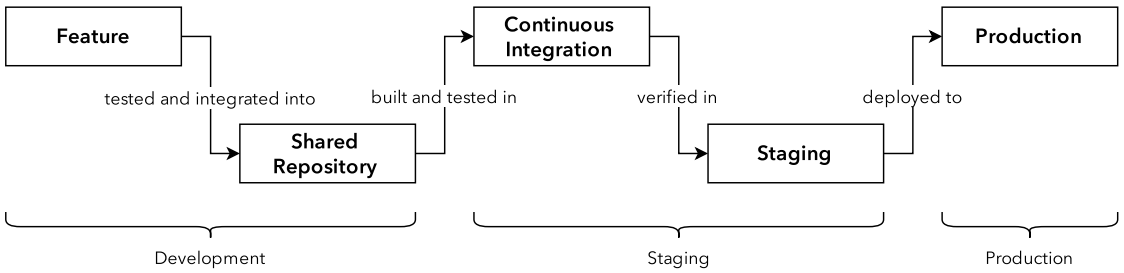

Figure 5: Deployment Pipeline Flow

Using cloud-based services has transformed software delivery: there is a fundamental shift in how products and services are developed and deployed. A popular paradigm nowadays is for instance to use GitHub for shared code repositories and project management, Travis CI for continuous integration and Heroku for web-application deployment [Git, Tra, Her]. Many of these services provide a high-level of interaction. A developer can push local changes to GitHub, GitHub can then start the continuous integration build on Travis CI automatically, and if successful the software can be deployed to Heroku. The use of cloud-based testing is accelerating: tests and analytics can be run post-deployment [RK14]. See figure 5 for an example of a deployment pipeline flow.

4 Towards Embedded Systems

Software is considered "soft", hardware "hard". Initially, software was only considered a convenient way to configure mechanisms for electronic systems [BE12]. It is not always obvious how products or features that combine software with hardware can be developed step-by-step. This combination, usually referred to as an embedded system, provides challenges in being agile and adaptive. Many of the Agile practises such as an identifiable customer, co-

located development and minimal architectural design are outright opposite of what hardware-related embedded system development currently is [RA03]. There is a wide range of applications using embedded systems and the complexity and required functionality of these is increasing [KRM+13, EHOS14]. Intricate systems are getting increasingly difficult to verify and validate.

The industry has started to recognise that setting requirements for products is the most difficult and deciding part of the software development process. All of this requires new thinking on how hardware products are being developed. Eminently, this is why Agile and Lean philosophies are starting to get attention on the embedded field, not only for small organisation but for large ones as well. Prominently, while teams have succeeded in adopting Agile software practises, the organisation level is still governed by plan-driven approaches [EB12, EHOS14]. Notably, a previous background in Agile and Lean practises seems to influence many of the development practises in embedded systems [KRM+13].

Nevertheless, there is still uncertainty whether the same agile ideologies and practises can improve the product development of embedded systems as much as they have reshaped the way user-focused software is being developed. Agile methods were not targeted for developing embedded systems where usually the object is not a person but instead a hardware. This manifests itself as limited customer–developer interaction. A development process tends to be focused on the integration of the whole product rather than being feature-driven. There are restrictions that are inescapable. For instance, it is not practical to develop a new working hardware prototype for each iteration. This does not alleviate the fact that experimentation is still required, because hardware constraints tend to have direct effects on later stages of development. Still, studies show that the use of Agile and Lean methodologies do have a positive effect also on the development of embedded systems by reducing development times and improving the overall development process and quality of products [CWR10, KRM+13]. Agile methods can be used with success if the underlying restrictions are addressed accordingly [RA03]. Even Boehm observed in the 1980s that iterative development suited equally both software and hardware development [Boe88]. Lehtonen et al. researched agility in embedded systems particularly from the perspective of well-being at work [LTR+14]. Using case studies, they conclude that increasing communication and being able to estimate workload evidently improve the meaningfulness, satisfaction and motivation of individuals.

4.1 Embracing Agile Development

It has become clear that methods and practises need to be adapted to suit each specific field [VB09, CWR10, HMP+10, JLP12, KRM+13]. There are also alterations of Agile that may be more suitable for plan-driven and

large-scale environments, such as Scaled Agile [Sca]. There is no silver bullet. The wide diversity of products and their domain-specific problems are very distinct. No single method will work, but rather a combination of best practises need to be utilised. The research field on embedded systems is still very young and most of the input is coming from the industry, which in turn does not tend to share internal practises to the public [KRM+13]. Some doubt has been casted on whether current Agile and Lean practises alone are sufficient in embedded settings — especially when related to safety-critical environments [TFR02, EB12]. Hardware-software systems are playing an increasing role in our everyday life making safety considerations a paramount concern [CWR10]. It is obvious though that agile and formal software development are not incompatible and features from plan-driven development can be adopted in iterative development. Agile practises such as test-driven development, early and exploratory releases and pair reviews support many of the requirements for formal development [TFR02, VB09, CWR10, JLP12]. Importantly, this provides a feedback-cycle for improving development.

The most explored Agile methods in embedded systems are unsurprisingly Extreme Programming and Scrum [KRM+13]. However, many of these practises have different focuses than in "traditional" software development. A different focus makes sense, since the practises are considered mostly as a baseline for practical usage instead of rigid guidelines. For example in the context of embedded systems, refactoring may focus on making improvements to the speed, memory or power consumption instead of improving the quality of code. Sometimes resulting in even hurting the simplicity and clarity of the code. Performance and software reliability are key factors [RA03, EHOS14]. Systems have to perform tasks within a defined time slot. Refactoring can even be risky, since hardware is very sensitive to changes in timing. Small changes can have somewhat large effects. Many of these effects are impossible to tell without hands-on experience with the hardware [RA03].

Embedded systems have a dedicated function within a larger mechanical or electrical system. This requires hardware to accompany the software. An embedded system is a specialised computer-based system designed for a dedicated task or purpose. There is a clear distinction between embedded software and embedded systems [KRM+13]. The former is constrained usually by restrictions set by the hardware, but the latter is not only constrained by the hardware itself, but also restrictions set by the development process of hardware. To simplify, an embedded software is part of an embedded system. An embedded software targets an existing system, whilst an embedded system involves building a hardware product from the ground up. I have set my perspective to emphasise not only the hardware aspect, but also the hardware development process as a key part of software development in embedded systems. Embedded systems can vary from mobile phones, cameras, robots to aeroplanes. Many of these systems require thorough planning and this is why plan-driven methodologies have been used widely. Even so, requirement

changes are still very prominent, not all requirements can be mapped before starting product development.

Increasing requirements and unpredictability for embedded systems has led to the adoption of agile methods in hardware environments. Customer collaboration has become invaluable. Nonetheless, this is still a relatively new shift. Organisations have started to become more aware of the advantages that agility has brought to software projects, but the use of Agile and Lean practises is not widespread in the embedded field [CWR10, EB12, KRM+13]. One of the biggest barriers on adopting new practises is organisational — mostly caused by conservative views [Pop02, HOAB12]. Another big barrier is technical challenges related to the context [KRM+13].

A cultural change is essential [HOAB12, KRM+13]. Acceptance and knowledge of Agile methods are still rather limited in industrial settings related to embedded systems [HMP+10]. The support for continuing with current plan-driven methods is conflicted with the desire to get the benefits from Agile practises. It is important to have a comprehensive view of the whole organisation [KRM+13]. In a large organisation, it is not only sufficient to look at how single teams work, but how the organisation works as a whole. In current practises, teams are often well ahead of the organisation as a whole [HOAB12]. An organisation should understand what they are trying to achieve with Agile and Lean methodologies. Not all companies should adopt practises the same way [KRM+13]. As a solution, piloting and coaching new practises has been suggested to overcome resistance and convince on the advantages of iterative practises [CWR10, EB12, HMP+10, KRM+13]. Piloting can indicate what adjustments are called for adopting Agile practises within the organisation. People familiar with the practises tend to see the benefits more easily. Mental acceptance is crucial for Agile and Lean practises to work [HMP+10]. Otherwise there is substantial risk that Agile practises will be adopted to suit plan-driven approaches and not the other way around.

4.2 Integrating Hardware and Software Development

A challenge in developing embedded systems is maintaining hardware and software development and integrating these together [EB12, EHOS14]. Developing embedded systems faces the same challenges which were posed by seeing software development as an engineering practise. Software and hardware development are still rather separated. Larger organisations are struggling with aligning hardware and software development cycles and practises [EHOS14]. Face-to-face conversation might not be sufficient and other means of communication are necessary [RA03]. Usually the hardware is not a major part of the software development until very late in the project [RA03]. Hardware development is more expensive and has longer lead times, i.e. how long it takes from initiation to the completion of production process [EHOS14]. Different teams handle different aspects of the process:

engineers design the hardware and developers the software. Furthermore, different people have different domain knowledge, which hinders the principle of shared responsibility [KRM+13, EHOS14]. Software development in embedded systems is mostly driven by the hardware [BE12]. An organisation is typically more experienced either in software or hardware development, balancing between these is a skill.

The Agile notion of moving all control to development teams is hard to accomplish. Hardware development should be deeply intertwined with software development. This has been referred as a hardware–software co-design. Developers and engineers should work on small teams more closely on systems to get the most out of Agile development. Clearly this requires cross-functional teams to transition to an agile research and development (R&D) approach [HOAB12, EHOS14]. An engineering process should be transparent to all stakeholders [KRM+13]. There is an apparent need for developing software and hardware simultaneously [RA03]. It is true that some of the architecture emerges through experience gained during development, but initial design cannot be avoided. Some organisations developing mass-produced embedded systems have been successful in allowing agility for individual teams to define their own ways on working to facilitate speed, shorter iterations and improving quality [EHOS14]. A key issue is how companies scale these practises beyond single teams.

One the most common approach to develop embedded systems is to use an integration-centric approach [EB12, EHOS14]. Commonly, hardware and software development are separated into two parallel streams and the work is later in development process integrated into a product. Early in the development, requirements are allocated to hardware and software components by a central engineering team. This is followed by multiple development teams implementing the requirements allocated to each component. The work of each team, hardware and software, is synchronised to a common project model. After the components are finalised, they are integrated together to form complete systems. Once this is completed, a system level testing can be conducted. At worst, this is were most of the integration problems are discovered. This cycle is repeated up to five times according to the project's stages. Typically one integration cycle lasts at least six months, meaning that complete systems require lead times of multiple years. Software and hardware cycles have a very weak link between them. The culture is to focus on predictability by foreseeing activities months ahead. A prime purpose of this stage gate model is to ensure the feasibility of releasing large investments for the following stages after each stage completes. Essentially this results in a linear approach where teams are dependant on others thus forming bottlenecks. Typically the idea of introducing Agile methodologies is to try to increase the rate of new features being developed within this cycle — mostly in a module development phase. Above all, software development is driven by the hardware [EB12]. See figure 6 for a typical approach for

applying Agile development in an integration-centric V-model approach.

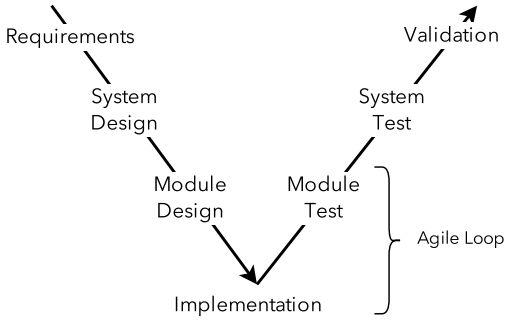

Figure 6: V-model [EHOS14]

Another key issue is that hardware and software development usually relies on many suppliers [EB12, HOAB12, EHOS14]. A situation that distinctly makes the development process complex. It is uncommon that an organisation can do everything related to hardware development in-house. Different suppliers provide components and factories combine them into hardware products. This also includes subcontracting different software components to different companies. Communication can be slow between all these moving parts. Not to mention difficult contractual matters. To get the benefits from Agile development and embrace shorter loops, all of these suppliers must follow similar principles and essentially abandon plan-driven processes.

Predictability may be desired in some circumstances. Organisations such as NASA are prime examples where software development must be predictable. NASA's space operations consist of plenty of procedure, time, large teams and stable requirements [Fow05]. Having said that, NASA is also a prime example of an organisation where iterative development has been used with good results [LB03]. The challenge is to combine predictability with the dynamic capabilities of modern iterative development [EHOS14].

It is true that defining fully elaborated requirements work in certain applications where for example real-time systems and safety are important criteria [Boe88, KRM$^+$13]. At least some sort of top-level documentation might be required to support iterative development, most of times purely required by legislation, but also for managing and conveying information among the different levels of stakeholders involved in the project [KRM$^+$13, EHOS14]. Regulatory standards can be rather rigorous, although they mostly do not impose any particular software development processes or practises [CWR10]. Traditionally safety regulations have been most regulated

in medical, nuclear and avionics sectors [JLP12]. Lately, this has extended to automotive and railway sectors as the use of software has increased in these fields. Within the aerospace industry for example, nearly all Agile practises can be mapped to regulatory standards — albeit the industry has been slow in adopting them [VB09, CWR10]. This is also true for the railway industry [JLP12]. Though difficult, a transition is possible by incorporating and adapting Agile methods into existing processes [VB09].

In embedded systems, the role of architecture and up front design cannot be avoided. In a regulated environment, there is a burden of proof which must demonstrate the compliance of the process [CWR10]. This makes it very rigid. It is imperative that there is full traceability throughout the development lifecycle. This obviously is a challenge for Agile and Lean practises. In any case, it has been suggested that Extreme Programming's focus on test-driven development and the use of source control management can act as documentation and assist traceability. Also, Agile development in no way dismisses the importance of documentation. What it does emphasise however, is creating methods that make documentation easier to handle instead of making it a barrier for iterative and adaptive development. Organisations need to organise support for light-weight documentation methods such as issue trackers, wikis, whiteboards and even cameras [HMP+10]. Some advocate using auto-generated documentation as much as possible [VB09, JLP12].

4.3 Historical Perspective

History has many successful examples of the usage for iterative development in software development in embedded systems [LB03]. The X-15 hypersonic jet applied iterative and incremental development already back in the 1950s. In fact, the X-15 was only a hardware project. In the 1960s this knowledge was carried through to NASA, where iterative development was used in Project Mercury's software. The project used surprisingly short iterations that only lasted a half-day. Interestingly, they also applied Extreme Programming practises such as test-driven development. Essentially, the platform for Project Mercury allowed the development team to build the system incrementally.

Later in the 1970s, the US Department of Defence used iterative development on large, life-critical space and avionics systems [LB03]. As an other example, the command and control system for the first US Trident submarine also used iterative development. Although, the project still used very long iterations taking as long as six months each. Other applications of iterative development included TRW/Army Site Defence's missile defence systems and the US Navy's Light Airborne Multipurpose System (LAMPS) part of a weapons system. The missile defence software project progressed by the team refining each iteration in response to the preceding iteration's feedback, an early use of reflection and learning by doing. LAMPS was one of the earliest

projects that used short iterations that only took one month per iteration. The project succeeded, deliveries were on time and under budget [LB03]. Notably, since many Waterfall project had fell short on this.

Another remarkable story is the primary avionics software for NASA's space shuttle program in the late 1970s [LB03]. The motivation for using iterative development came from need to be able to handle changing requirements for the shuttle program during its software development. NASA used eight week iterations and these made feedback-driven refinements to specifications. In the early 1990s, a new-generation Canadian Automated Air Traffic Control System was developed using risk-driven iterative development. The project was also a success, despite its near-failure predecessor that applied the famed Waterfall model [LB03]. Still, it used rather long iterations of six months by modern standards.

Notably, all these examples are early examples of being agile, but it has to be said that they only applied a fraction of ideas presented by current ideologies. Recently Agile and Lean practises have been used with good results in the development of instruments, cameras, telecommunications software, health products, the automotive industry, automated vehicles and even aeroplanes and satellites [RA03, VB09, BE12, KRM+13, HOB14].

4.4 Using Hardware as a Platform

Hardware is something concrete (sometimes even bare metal). Not only being something physical, but having many dependencies between components and the way they interface together. It is usual that hardware sets tight requirements for the software. The lifecycle of devices is measured in years, sometimes decades — resulting in many legacy platforms [BE12]. Obviously systems like traffic lights, railway signalling and ticketing systems for the Underground are upgraded rarely in contrast to for instance mobile devices. Maintaining these platforms requires legacy software and physical spare parts. Embedded systems need to evolve to stay attractive for users especially in sectors that are moving fast [BE12]. Product use evolves over time and features should be adjusted duly.

Most of the challenges regarding iterative development can be overcome by designing a platform that can be easily extended and modified for different needs [KRM+13]. A principal obstacle is caused by the lack of a base product which can be improved continually [HOAB12]. This also includes modularising the software into smaller more manageable components. Investing in hardware development is many times more expensive than investing in software — requiring immense amounts of effort [BE12]. Let alone the undertaking caused by deploying these systems to actual use. A single product should be customisable for different customers instead of developing completely separate products. Subsequently, hardware becomes a platform for delivering value with software solutions. Increasingly software has become

the core for almost all hardware systems [BE12]. Software is the enabler for new innovation. Keeping the process "lean" and reducing the unnecessary can provide a competitive edge [CWR10].

The benefits of the web and cloud computing extend well beyond web-applications [BE12]. Remote deployments are possible on embedded systems. Microprocessor-based architectures have made it possible to update systems with new software releases [RA03]. These architectures are combined with application-specific integrated circuits that are customised for a particular use. A current trend is connecting these systems, from cameras to cars, to the Internet, sometimes also referred as the Internet of Things [BE12, HOB14]. This enables Continuous Delivery on embedded systems.

Helena Holmström Olsson and Jan Bosch state three issues: 1) post-deployment data is being used for new products rather than for improving existing systems, 2) post-deployment data is used for troubleshooting and support rather than for innovating new features and 3) post-deployment data is being used to understand operation and performance rather than for providing insight in individual feature usage [HOB14].

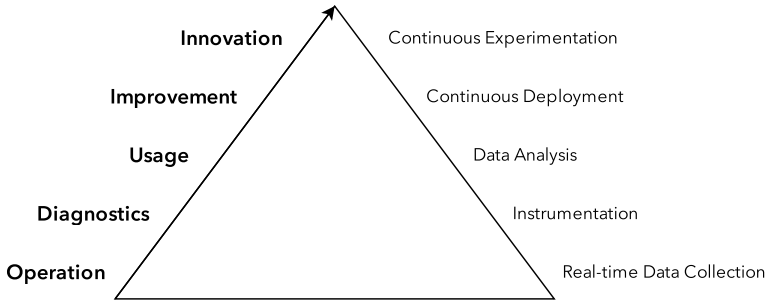

Figure 7: Levels of Data Usage, adapted from [HOB14]

Although cloud computing currently is applied in delivering software to the web, these techniques can be applied basically to any product that is able to collect and provide data about its usage — including even software-intensive embedded systems [BE12, Bos12]. See figure 7 for how data can be put to use. Data can be used in the research and development process of embedded products. This enables Continuous Experimentation on embedded systems. The organisation can spend time to developing the right things instead of fixing mistakes in needless functionality [HOAB12]. By and large, organisation have significant quantities of data, but little is used for research and development [HOB14]. For example cars can collect real-time fuel consumption data and telecom devices real-time bandwidth

data [Bos12]. Previously these points of interest where only collected for management purposes, now data is used for development purposes as well. Still, post-deployment data is mostly used as input for development of successor products, but surprisingly not for improving features on current products [HOB14].

Service-oriented models have started to find their way to more embedded settings [BE12, Bos12]. Organisations are moving towards software-houses to be able to concentrate on bringing value through software. While high-quality hardware systems is still important, it is no longer the only differentiator between organisation and what makes a product competitive [EHOS14]. In some cases, hardware products are being leased instead of them being expensive investments. Mobile phones and even cars are starting to take advantage of frequent, post-deployment updates to their software. Organisation are also interested in collecting usage and other performance related metrics. Instead of freezing requirements before starting product development, the requirements evolve and have effect on how users use the products. Users are becoming increasingly accustomed to frequent updates that add value. Consequently this also increases the expectations for users. Traditional static and unconnected hardware are fast becoming unsustainable. Upgrading is expensive and integrating new solutions involve risk and complexities that can be hard to predict.

In embedded systems, software is deployed as part of the overall system, including its hardware. Instead of servers, the software is deployed to the destination point. The process obviously requires changes to the architecture of the platforms to facilitate remote deployments and experimentation. Notably creating structures to collect data from performance to usage, analysing these, creating experiments and enabling remote deployments [BE12]. The ability to evolve and conduct experiences must be supported in a safe and controlled manner [BE12]. Jan Bosch refers to this as building an innovation experiment system, well suited for connected embedded systems [BE12, Bos12]. A research and development system responds to actual customer feedback collected from experimenting and testing features that the users actually need [HOAB12]. Many embedded systems are deployed in customer locations. For that reason, there is demand for customers who can see the benefits for Continuous Experimentation and are willing to explore the concept in production [HOAB12]. Furthermore, connecting devices to the Internet obviously poses security and privacy issues that must be taken account [BE12].

New approaches such as electronic testing platforms like Arduino, 3D-printing and laser-cutting are bringing hardware development to the general public [Ard]. Lately, it has become even possible to print circuit boards [Vol15]. Previously all these activities where limited to specific industries. Experimenting with hardware products is getting even more accessible. Microprocessor-based architectures can be rapidly expanded and used in a variety of different applications [KRM+13]. Now, building a

prototype for experimentation purposes is within reach.

4.5 Adapting for Deployment Pipeline

Embedded development is exploration-driven by nature: the development process includes extensive research and development resources [EHOS14]. Still, testing is the cornerstone of embedded systems [RA03, HOAB12, KRM+13, EHOS14, Ngy15]. Bugs tend to be hard to detect. Trying to detect whether they are caused by the software or hardware is demanding on its own. A bug might be caused by something completely not related to software [Ngy15]. Most of the software implementations need to be tested against the hardware since most of the code is dependant on it. Hardware development has slow development cycles, even lasting several years. Developer need to verify proper co-operation between the software and hardware. The more complex the interactions are, the more important experimentation is. That is why hardware simulation is an essential practise. This is problematic since test environments in the embedded domain have very different performance and memory constraints. Real-time requirements are hard to satisfy [RA03, HMP+10]. Testing can take several days to achieve on actual hardware — if developers even have certified access to hardware environments [VB09]. The overhead of processes can make the lead time of simple bug fixes to several weeks. It is also common that stringent contexts such as aerospace need separate review teams [VB09]. In these contexts, developers have very strict access to only specific sections of code. Implementation and testing should be carried out by separate people to satisfy regulations [VB09, JLP12].

The concept of running tests relentlessly on a target platform is hard to accomplish. Performance and memory constraints often prevent installing and running all tests on the hardware at the same time. It also takes significantly longer than say in modern computers. Anyway, basic usage of Continuous Integration is not hard to achieve in embedded settings — especially when related to unit testing [RA03, KRM+13]. Integration and acceptance testing is harder to achieve. As with "traditional" software development, automatic testing will help to identify failures in an early stage. It will also increase awareness on what effect each build has on the overall system. People should not do the job of computers, Continuous Integration handles many error prone tasks [Ngy15].

Organisations developing embedded systems are starting to seek for opportunities presented by Continuous Deployment [HOAB12]. However, there is still a rather long leap for frequently delivering new features to users. Sometimes this is not even possible within internal processes. The need for both Continuous Integration and Deployment are well understood, but their implementation is unfortunately rather hard to accomplish without more deep changes. A key focus is for an organisation to develop a fully automated testing infrastructure to continuously verify development [HOAB12]. Without

a doubt, the distributed nature of web is a valuable platform for enabling remote deployments as well as collecting, analysing and experimenting with data.

With the high cost associated with testing new ideas, experimentation should be a well thought and simple process. Organisations are inclined to use observation techniques and expert reviews in pre-deployment research and development, but there is less evidence on organisations using methods focused on continuously collecting customer feedback [HOB14]. Turning most promising ideas from concepts to prototypes should receive significant attention [BE12]. Iterating on features in embedded systems should be possible in short cycles pursuing Agile practises. Many current processes, such as the V-model, are focused in creating a process focused in rigid milestones to try to minimise losses instead of experimenting in small steps. An obvious cause for mundane innovation [BE12]. Instead organisations should focus on maximising the amount of iterations by reducing the related costs for each iteration. Post-deployment data can provide understanding of the operation and performance of embedded systems, but in addition how the user uses different features [BE12, HOB14]. See figure 8 for the steps related for moving towards Continuous Experimentation.

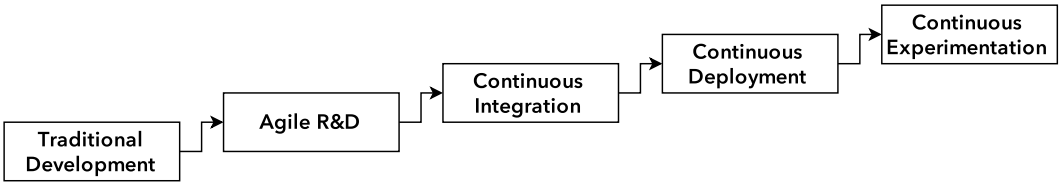

Figure 8: Towards Continuous Experimentation, adapted from [HOAB12]

Specific solutions are required to mitigate the restriction in scaling the deployment pipeline in embedded systems. A special test environment may be needed in environments where the implications of feature changes are broad and the customer may have reluctance towards experimenting with new features [FGMM14]. To avoid safety concern, Jan Bosch and Ulrik Eklund have suggested separating experimentation systems from the development of critical components [BE12]. Setting up an experimentation cycle can be rather challenging for developing software that requires hardware. Longer release cycles with hardware and potential synchronisation problems between the development schedules is an issue [FGMM14]. An experimentation system can be used to guide research and development efforts [HOAB12].

In certain life-critical environments, experimentation can just be too ex-

pensive or undesirable to achieve [BE12]. Heavyweight sequential processes are outside of Agile research and development [EHOS14]. Heavy verification and validation is required in contexts such as aerospace. This is why simulation becomes important [KRM+13]. Simulation enables developers to work on software simultaneously while hardware is being evolved. Developers should be able to experiment by simulating in a production-like setting. The problem is that simulation is never really "production-like". At least in the sense we are able to achieve in web-development. Software most likely will behave differently on the actual hardware.

At the leading edge, some organisation are even pushing web design philosophies to native applications to tackle the rigid environment of embedded systems [Boh13, GZ14]. In another example, the UC Berkeley Solar Vehicle Team is using Travis CI as a platform to test their embedded software that powers their solar vehicle [Ngy15].

5 Views from Embedded Settings

I conducted five semi-structured interviews with six people working in the academia and industry on leading embedded systems to get a view on if and how Agile and Lean methodologies and the deployment pipeline has changed the development of hardware related products. The interview consisted of the following topics:

Process

1. Do you consider that your organisation follows the principles and practises of Agile and Lean development? Which of these principles have most significance?

2. If so, has this recently changed the way you develop products or features into production?

3. Do you approach development from the point-of-view of the whole product or by single features?

4. Please describe the process behind developing an idea into a single feature. How long does it take?

5. Do you recognise distinct development, staging and production environments in your process?

6. If so, are these automated?

Adapting to Change

7. How easy or hard is it to adapt changes in hardware related products?
8. Can you deploy software changes automatically or even remotely?
9. How do you keep software and hardware development in sync?
10. How short iterations do you use to adjust for feedback from your stakeholders?

Experimentation

11. Do you have an automated process for deploying or experimenting a feature?
12. How do you experiment with software related to hardware?
13. How do you value an idea (prototypes, minimum-viable products, A/B testing)?
14. Related to hardware, has new approaches such as electronic testing platforms, 3D-printing or laser-cutting changed your process?

Many of the mentioned topics and terms are prone to many interpretations. Their meaning is somewhat ambiguous. To address this, I started the interviews by introducing the topics and relaying my understanding on the concepts. This way we shared understanding on what I was trying to achieve. I conducted the interviews both face-to-face and via email, which ever suited best for the situation. Each interview included additional discussion about interesting topics which arose during the conversations.

Most of the interviews are from European organisations or subsidiaries of multinational corporations in Finland. The participants include: Lauri Koivulehto from Airbus Defence and Space Oy, Stefan Baggström and Marko Taipale from GE Healthcare Finland Oy, Niklas Holsti from Tidorum Ltd and Harri Holopainen from ZenRobotics Ltd [Koi15, BT15, Hol15b, Hol15a]. My interviews included also one participant from academia in Switzerland: Maximilian Kriegleder from ETH Zürich, Institute for Dynamic Systems and Control [Kri15].

From the industry, Airbus Defence and Space Oy works in the telecom-industry, GE Healthcare Finland Oy produces health-care equipment, Niklas Holsti has an extensive background in the aerospace industry and finally ZenRobotics Ltd works on waste sorting robots. As for the academia, ETH Zürich, Institute for Dynamic Systems and Control experiments among other things with intelligent quadcopters famous from TED [D'A13].

5.1 About the Process

About the process.

5.2 About Adapting to Change

About adapting to change.

5.3 About Experimentation

About experimentation.

5.4 About Building Hardware for Software

About building hardware for software.

6 Conclusions

A collective effort is essential. Regardless, it is likely that small, agile and adaptive companies will lead the way, existing rigid larger companies will struggle.

7 Acknowledgements

I would like to express my gratitude to everyone who helped me to get in contact with the embedded industry for the interviews. Most of all, I would like to thank all the interviewees: Stefan Baggström, Harri Holopainen, Niklas Holsti, Lauri Koivulehto, Maximilian Kriegleder and Marko Taipale [BT15, Hol15a, Hol15b, Koi15, Kri15]. Additionally, thanks to Matti Luukkainen for supervising this thesis!

References

[Ard] *Arduino*. `http://arduino.cc`.

[BBvB+01a] Beck, Kent, Beedle, Mike, Bennekum, Arie van, Cockburn, Alistair, Cunningham, Ward, Fowler, Martin, Grenning, James, Highsmith, Jim, Hunt, Andrew, Jeffries, Ron, Kern, Jon, Marick, Brian, Martin, Robert C., Mellor, Steve, Schwaber, Ken, Sutherland, Jeff, and Thomas, Dave: *Manifesto for Agile software development*, 2001. `http://agilemanifesto.org`, (accessed 13 January 2015).

[BBvB+01b] Beck, Kent, Beedle, Mike, Bennekum, Arie van, Cockburn, Alistair, Cunningham, Ward, Fowler, Martin, Grenning, James, Highsmith, Jim, Hunt, Andrew, Jeffries, Ron, Kern, Jon, Marick, Brian, Martin, Robert C., Mellor, Steve, Schwaber, Ken, Sutherland, Jeff, and Thomas, Dave: *Principles behind the Agile Manifesto*, 2001. http://agilemanifesto.org/principles.html, (accessed 10 February 2015).

[BE12] Bosch, Jan and Eklund, Ulrik: *Eternal embedded software: Towards innovation experiment systems.* In Margaria, Tiziana and Steffen, Bernhard (editors): *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, page 19–31. Springer Berlin Heidelberg, 2012, ISBN 9783642340253. http://dx.doi.org/10.1007/978-3-642-34026-0_3.

[Bec00] Beck, Kent: *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000, ISBN 9780201616415.

[Boe88] Boehm, Barry W.: *A spiral model of software development and enhancement.* Computer, 21(5):61–72, May 1988, ISSN 00189162. http://dx.doi.org/10.1109/2.59.

[Boh13] Bohn, Dieter: *How Facebook secretly redesigned its iPhone app with your help.* The Verge, 2013. http://theverge.com/2013/9/18/4744904/how-facebook-secretly-redesigned-its-iphone-app-with-your-help/, (accessed 18 March 2015).

[Bos12] Bosch, Jan: *Building products as innovation experiment systems.* In Cusumano, Michael A., Iyer, Bala, and Venkatraman, N. (editors): *Software Business*, volume 114 of *Lecture Notes in Business Information Processing*, page 27–39. Springer Berlin Heidelberg, 2012, ISBN 9783642307454. http://dx.doi.org/10.1007/978-3-642-30746-1_3.

[BR70] Buxton, John N. and Randell, Brian (editors): *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27–31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970.

[BT15] Baggström, Stefan and Taipale, Marko. Personal correspondence, 2015. GE Healthcare Finland Oy.

[CWR10] Cawley, Oisín, Wang, Xiaofeng, and Richardson, Ita: *Lean/agile software development methodologies in regulated environments*

– *state of the art*. In Abrahamsson, Pekka and Oza, Nilay (editors): *Lean Enterprise Software and Systems*, volume 65 of *Lecture Notes in Business Information Processing*, page 31–36. Springer Berlin Heidelberg, 2010, ISBN 9783642164156. http://dx.doi.org/10.1007/978-3-642-16416-3_4.

[D'A13] D'Andrea, Raffaello: *The astounding athletic power of quadcopters*. TED, 2013. http://ted.com/talks/raffaello_d_andrea_the_astounding_athletic_power_of_quadcopters/, (accessed 1 April 2015).

[DD08] Dybå, Tore and Dingsør, Torgeir: *Empirical studies of agile software development: A systematic review*. Information and Software Technology, 50(9–10):833–859, 2008, ISSN 09505849. http://dx.doi.org/10.1016/j.infsof.2008.01.006.

[DNBM12] Dingsøyr, Torgeir, Nerur, Sridhar, Balijepally, VenuGopal, and Moe, Nils Brede: *A decade of agile methodologies: Towards explaining agile software development*. Journal of Systems and Software, 85(6):1213–1221, June 2012. http://dx.doi.org/10.1016/j.jss.2012.02.033.

[EB12] Eklund, Ulrik and Bosch, Jan: *Applying agile development in mass-produced embedded systems*. In Wohlin, Claes (editor): *Agile Processes in Software Engineering and Extreme Programming*, volume 111 of *Lecture Notes in Business Information Processing*, page 31–46. Springer Berlin Heidelberg, 2012, ISBN 9783642303494. http://dx.doi.org/10.1007/978-3-642-30350-0_3.

[EHOS14] Eklund, Ulrik, Holmström Olsson, Helena, and Strøm, Niels Jørgen: *Industrial challenges of scaling agile in mass-produced embedded systems*. In Dingsøyr, Torgeir, Moe, Nils Brede, Tonelli, Roberto, Counsell, Steve, Gencel, Cigdem, and Petersen, Kai (editors): *Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation*, volume 199 of *Lecture Notes in Business Information Processing*, page 30–42. Springer International Publishing, 2014, ISBN 9783319143576. http://dx.doi.org/10.1007/978-3-319-14358-3_4.

[FGMM14] Fagerholm, Fabian, Guinea, Alejandro Sanchez, Mäenpää, Hanna, and Münch, Jürgen: *Building blocks for continuous experimentation*. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, page 26–35, New York, NY, USA, 2014. ACM, ISBN 9781450328562. http://doi.acm.org/10.1145/2593812.2593816.

[Fow05] Fowler, Martin: *The new methodology*, 2005. http://martinfowler.com/articles/newMethodology.html, (accessed 13 January 2015).

[Fow06] Fowler, Martin: *Continuous integration*, 2006. http://martinfowler.com/articles/continuousIntegration.html, (accessed 13 January 2015).

[Fow08] Fowler, Martin: *AgileVersusLean*, 2008. http://martinfowler.com/bliki/AgileVersusLean.html, (accessed 4 March 2015).

[Fow10] Fowler, Martin: *BlueGreenDeployment*, 2010. http://martinfowler.com/bliki/BlueGreenDeployment.html, (accessed 31 March 2015).

[Fow13a] Fowler, Martin: *ContinuousDelivery*, 2013. http://martinfowler.com/bliki/ContinuousDelivery.html, (accessed 13 January 2015).

[Fow13b] Fowler, Martin: *DeploymentPipeline*, 2013. http://martinfowler.com/bliki/DeploymentPipeline.html, (accessed 13 January 2015).

[Git] *GitHub*. https://github.com.

[GZ14] Grant, Ari and Zhang, Kang: *Airlock — Facebook's mobile A/B testing framework*, 2014. https://code.facebook.com/posts/520580318041111/airlock-facebook-s-mobile-a-b-testing-framework/, (accessed 18 March 2015).

[Her] *Heroku*. https://heroku.com.

[HF11] Humble, Jez and Farley, David: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. The Addison-Wesley Signature Series (Fowler). Addison-Wesley, 2011, ISBN 9780321601919.

[HHLV13] Heinonen, Kenny, Hirvikoski, Kasper, Luukkainen, Matti, and Vihavainen, Arto: *Learning agile software engineering practices using coding dojo*. In *Proceedings of the 14th Annual ACM SIGITE Conference on Information Technology Education*, SIGITE '13, page 97–102, New York, NY, USA, 2013. ACM, ISBN 9781450322393. http://doi.acm.org/10.1145/2512276.2512306.

[HMP+10] Heidenberg, Jeanette, Matinlassi, Mari, Pikkarainen, Minna, Hirkman, Piia, and Partanen, Jari: *Systematic piloting of agile methods in the large: Two cases in embedded systems development.* In Ali Babar, M., Vierimaa, Matias, and Oivo, Markku (editors): *Product-Focused Software Process Improvement*, volume 6156 of *Lecture Notes in Computer Science*, page 47–61. Springer Berlin Heidelberg, 2010, ISBN 9783642137914. http://dx.doi.org/10.1007/978-3-642-13792-1_6.

[HOAB12] Holmström Olsson, Helena, Alahyari, Hiva, and Bosch, Jan: *Climbing the "stairway to heaven" — a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software.* In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, page 392–399. IEEE, September 2012. http://dx.doi.org/10.1109/SEAA.2012.54.

[HOB14] Holmström Olsson, Helena and Bosch, Jan: *Post-deployment data collection in software-intensive embedded products.* In Bosch, Jan (editor): *Continuous Software Engineering*, page 143–154. Springer International Publishing, 2014, ISBN 9783319112824. http://dx.doi.org/10.1007/978-3-319-11283-1_12.

[Hol15a] Holopainen, Harri. Personal correspondence, 2015. ZenRobotics Ltd.

[Hol15b] Holsti, Niklas. Personal correspondence, 2015. Tidorum Ltd.

[IEE06] IEEE: *IEEE standard for developing a software project life cycle process.* IEEE Std 1074-2006 (Revision of IEEE Std 1074-1997), 2006. http://dx.doi.org/10.1109/IEEESTD.2006.219190.

[ISO05] ISO: *Quality management systems — fundamentals and vocabulary.* 2005. http://iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=42180.

[JLP12] Jonsson, Henrik, Larsson, Stig, and Punnekkat, Sasikumar: *Agile practices in regulated railway software development.* In *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, page 355–360. IEEE, November 2012. http://dx.doi.org/10.1109/ISSREW.2012.80.

[KDF+12] Kohavi, Ron, Deng, Alex, Frasca, Brian, Longbotham, Roger, Walker, Toby, and Xu, Ya: *Trustworthy online controlled experiments: Five puzzling outcomes explained.* In *Proceedings of*

the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, page 786–794, New York, NY, USA, 2012. ACM, ISBN 9781450314626. http://doi.acm.org/10.1145/2339530.2339653.

[KLSH09] Kohavi, Ron, Longbotham, Roger, Sommerfield, Dan, and Henne, Randal M.: *Controlled experiments on the web: survey and practical guide*. Data Mining and Knowledge Discovery, 18(1):140–181, 2009, ISSN 13845810. http://dx.doi.org/10.1007/s10618-008-0114-1.

[Kni07] Kniberg, Henrik: *Scrum and XP from the Trenches*. InfoQ, 2007, ISBN 9781430322641.

[Koi15] Koivulehto, Lauri. Personal correspondence, 2015. Airbus Defence and Space Oy.

[KR04] Kohavi, Ronny and Round, Matt: *Front line internet analytics at Amazon.com*, 2004. http://ai.stanford.edu/~ronnyk/emetricsAmazon.pdf, (accessed 18 March 2015).

[Kri15] Kriegleder, Maximilian. Personal correspondence, 2015. ETH Zürich, Institute for Dynamic Systems and Control.

[KRM+13] Kaisti, Matti, Rantala, Ville, Mujunen, Tapio, Hyrynsalmi, Sami, Könnölä, Kaisa, Mäkilä, Tuomas, and Lehtonen, Teijo: *Agile methods for embedded systems development - a literature review and a mapping study*. EURASIP Journal on Embedded Systems, 2013(1), 2013. http://dx.doi.org/10.1186/1687-3963-2013-15.

[LB03] Larman, Craig and Basili, Victor R.: *Iterative and incremental developments: a brief history*. Computer, 36(6):47–56, June 2003, ISSN 00189162. http://dx.doi.org/10.1109/MC.2003.1204375.

[LTR+14] Lehtonen, Teijo, Tuomivaara, Seppo, Rantala, Ville, Känsälä, Marja, Mäkilä, Tuomas, Jokela, Tero, Könnölä, Kaisa, Kaisti, Matti, Suomi, Samuli, Ylitolva, Marko, and Isomäki, Minna: *Sulautettujen järjestelmien ketterä käsikirja*. University of Turku, 2014, ISBN 9789512958382. http://embedded.utu.fi/kasikirja/, (only in Finnish).

[LV09] Larman, Craig and Vodde, Bas: *Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*. Addison-Wesley, 2009, ISBN 9780321480965.

[McK12] McKinley, Dan: *Design for continuous experimentation*, 2012. http://slideshare.net/danmckinley/design-for-continuous-experimentation/, (accessed 18 March 2015).

[MNDT09] Mockus, Audris, Nagappan, Nachiappan, and Dinh-Trong, Trung T.: *Test coverage and post-verification defects: A multiple case study.* In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, page 291–301, October 2009. http://dx.doi.org/10.1109/ESEM.2009.5315981.

[Mon12] Monden, Yasuhiro: *Toyota Production System: An Integrated Approach to Just-In-Time, 4th edn.* CRC Press, 2012, ISBN 9781439820971.

[New13] Newton, Casey: *Building a better nest: Inside Twitter's continuous redesign.* The Verge, 2013. http://theverge.com/2013/12/23/5238802/building-a-better-nest-inside-twitters-continuous-redesign/, (accessed 18 March 2015).

[New15] Newman, Sam: *Building Microservices.* O'Reilly Media, 2015, ISBN 9781491950333.

[Ngy15] Ngyuen, Christopher: *UC Berkeley CalSol team runs on Travis CI*, 2015. http://blog.travis-ci.com/2015-03-19-uc-berkeley-calsol-team-runs-on-travis-ci/, (accessed 21 March 2015).

[NR69] Naur, Peter and Randell, Brian (editors): *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.* 1969.

[Ōno88] Ōno, Taiichi: *Toyota Production System: Beyond Large-Scale Production.* Productivity Press, 1988, ISBN 9780915299140.

[O'R11] O'Reilly: *Velocity 2011: Jon Jenkins, "velocity culture".* YouTube, 2011. https://youtube.com/watch?v=dxk8b9rSKOo, (accessed 13 January 2015).

[Pop02] Poppendieck, Mary: *Principles of lean thinking*, 2002. http://leanessays.com/2002/11/principles-of-lean-thinking.html, (accessed 4 March 2015).

[PP03] Poppendieck, Mary and Poppendieck, Tom: *Lean Software Development: An Agile Toolkit.* The Agile Software Development Series. Addison-Wesley, 2003, ISBN 9780321150783.

[RA03] Ronkainen, Jussi and Abrahamsson, Pekka: *Software development under stringent hardware constraints: Do agile methods have a chance?* In Marchesi, Michele and Succi, Giancarlo (editors): *Extreme Programming and Agile Processes in Software Engineering*, volume 2675 of *Lecture Notes in Computer Science*, page 73–79. Springer Berlin Heidelberg, 2003, ISBN 9783540402152. http://dx.doi.org/10.1007/3-540-44870-5_10.

[Rho14] Rhoades, Lacy: *Mobile app feature configuration and A/B experiments*, 2014. http://slideshare.net/lacyrhoades/mobile-feature-configuration-and-ab-experiments/, (accessed 18 March 2015).

[Rie11] Ries, Eric: *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business, 2011, ISBN 9780307887894.

[RK14] Riungu-Kalliosaari, Leah: *Empirical study on the adoption, use and effects of cloud-based testing*. PhD thesis, Lappeenranta University of Technology, Computer Science, 2014.

[RKDB+13] Raatikainen, Mikko, Komssi, Marko, Dal Bianco, Vittorio, Kindstöm, Klas, and Järvinen, Janne: *Industrial experiences of organizing a hackathon to assess a device-centric cloud ecosystem*. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, page 790–799. IEEE, July 2013. http://dx.doi.org/10.1109/COMPSAC.2013.130.

[RM13] Rein, Alexander Derek and Münch, Jürgen: *Feature prioritization based on mock-purchase: A mobile case study*. In Fitzgerald, Brian, Conboy, Kieran, Power, Ken, Valerdi, Ricardo, Morgan, Lorraine, and Stol, Klaas Jan (editors): *Lean Enterprise Software and Systems*, volume 167 of *Lecture Notes in Business Information Processing*, page 165–179. Springer Berlin Heidelberg, 2013, ISBN 9783642449291. http://dx.doi.org/10.1007/978-3-642-44930-7_11.

[Roy70] Royce, Winston W.: *Managing the development of large software systems*. In *Proceedings of IEEE WESCON*, 1970.

[Rub14] RubyKaigi: *Continuous delivery at GitHub - RubyKaigi 2014*. YouTube, 2014. https://youtube.com/watch?v=Rhvri5cozTc, (accessed 13 January 2015).

[Sat14] Sato, Danilo: *CanaryRelease*. Martin Fowler, 2014. http://martinfowler.com/bliki/CanaryRelease.html, (accessed 31 March 2015).

[Sca] *Scaled Agile Framework*. http://scaledagileframework.com.

[Sch13] Schmaus, Ben: *Deploying the Netflix API*. The Netflix Tech Blog, 2013. http://techblog.netflix.com/2013/08/deploying-netflix-api.html, (accessed 31 March 2015).

[Sny13] Snyder, Ross: *Continuous deployment at Etsy: A tale of two approaches*, 2013. http://slideshare.net/beamrider9/continuous-deployment-at-etsy-a-tale-of-two-approaches/, (accessed 13 January 2015).

[SS10] Sfetsos, Panagiotis and Stamelos, Ioannis: *Empirical studies on quality in agile practices: A systematic literature review*. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, page 44–53, September 2010. http://dx.doi.org/10.1109/QUATIC.2010.17.

[TFR02] Turk, Dan, France, Robert, and Rumpe, Bernhard: *Limitations of agile software processes*. Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, page 43–46, May 2002. http://arxiv.org/abs/1409.6600.

[Tra] *Travis CI*. https://travis-ci.org.

[VB09] VanderLeest, Steven H. and Buter, Andrew: *Escape the waterfall: Agile for aerospace*. In *Digital Avionics Systems Conference, 2009. DASC '09. IEEE/AIAA 28th*, pages 6.D.3–1–6.D.3–16. IEEE, October 2009. http://dx.doi.org/10.1109/DASC.2009.5347438.

[Vol15] Voltera: *Voltera: Your circuit board prototyping machine*, 2015. https://kickstarter.com/projects/voltera/voltera-your-circuit-board-prototyping-machine/, (accessed 6 March 2015).

[Wan14] Wang, Lynn: *Etsy's culture of continuous experimentation and A/B testing spurs mobile innovation*. Apptimize, 2014. http://apptimize.com/blog/2014/01/etsy-continuous-innovation-ab-testing/, (accessed 18 March 2015).

[Whi11] Whittaker, James: *How Google tests software — part four*. Google Testing Blog, 2011. http://googletesting.blogspot.fi/2011/03/how-google-tests-software-part-four.html, (accessed 31 March 2015).

[Whi14] Whitlock, Andy. Twitter, 2014. https://twitter.com/andywhitlock/status/524545897737494528/, (accessed 22 February 2015).

[ZCB10] Zhang, Qi, Cheng, Lu, and Boutaba, Raouf: *Cloud computing: state-of-the-art and research challenges*. Journal of Internet Services and Applications, 1(1):7–18, 2010, ISSN 18674828. http://dx.doi.org/10.1007/s13174-010-0007-6.