

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, March 12, 2015

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	March 12, 2015	27	
Tiivistelmä — Referat — Abstract			
<p>Abstract.</p> <p>ACM Computing Classification System (CCS):</p>			
Avainsanat — Nyckelord — Keywords			
keyword			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — Övriga uppgifter — Additional information			

Contents

1	Introduction	1
2	Software Delivery	3
2.1	Adapting to Requirements	5
2.2	Being Agile	7
2.3	Ensuring Quality	8
2.4	Processes and Practises	9
2.5	From Agile to Lean	11
2.6	Focus on the Essential	12
3	Deployment Pipeline	13
3.1	Three Stages	15
3.2	Using Web as a Platform	16
3.3	Continuous Integration	17
3.4	Continuous Deployment	18
3.5	Continuous Experimentation	19
4	Towards Embedded Systems	20
4.1	Using Hardware as a Platform	21
4.2	Adapting for Deployment Pipeline	22
5	Cases from Embedded Settings	22
6	Conclusions	24
	References	24

1 Introduction

Software delivery on the web has over the years evolved 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 even 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 these settings. Moreover, the aim is to identify if modern Continuous Integration, Delivery and Experimentation practises are used. Not just in a literal way, but trying to discover how ideas turn into features. 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. 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]. 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].

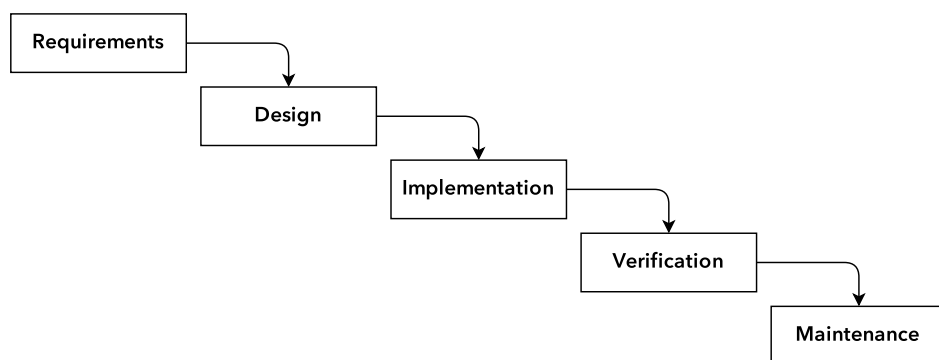


Figure 1: Waterfall Model

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 a straight timeline with a start and end. Each stage is planned and documented thoroughly. The concept thus being that as each step progresses, the design 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, 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-like plan and only then proceed in executing it. Nevertheless, this was overlooked and the Waterfall model became the dominant software development process for software standards in government and industry for the time-being [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]. Royce considered a software completed only when 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 methods sought in developing a useful compromise between no process and too much process [Fow05]. They also focused to be less document-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.

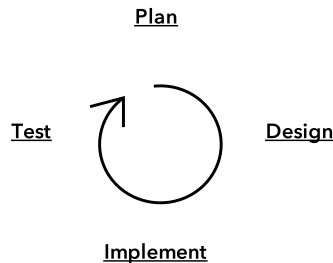


Figure 2: Iterative Development

2.1 Adapting to Requirements

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 life-time. 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 led 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, 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 led to a Waterfall-like process with different people involved in different stages. In software, the time spent coding is fractional to the time spent designing. Essentially, coding is designing. Coding requires creative and talented people. People are considered the most important factor 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 writing the functionality tend to write better tests, because they are more familiar with the implementation [MNMT09].

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 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 through 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, teams respond to change 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, Fow05]. 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.

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. 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. Quality and value have similar interpretations.

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]. 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 others work. Quality should be a concerted effort. Above all, in addition of being a quantitative metric, value is qualitative.

The Agile practises also have their critics. 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]. However, empirical studies have shown favourable results. 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 in larger environments. 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]. The process is 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, tested code has a better chance of detecting 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].

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 [DD08]. These are defined in Extreme Programming. 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, existing code should also always be refactored to be better. In pair programming, developers develop features in pairs.

Extreme Programming practises have been more widely studied than Scrum [DD08]. 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. Being “lean” means reducing the amount of “waste” around software development. The 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* [DD08]. 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 90s when westerners started to explore why Japanese were 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.

Iterations have turned into building single features at a time. Instead of building a frame for a car, a development process should essentially start with building a bicycle first. To evaluate an idea, developers should begin by developing a minimum viable product to validate the implementation has value [Rie11]. To retort, the notion is that a feature can be completed in the time it takes a committee to decide its goals [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 your heading by pivoting or you persevere with the choice you have made.

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 ambi-

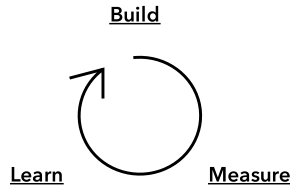


Figure 3: Build-Measure-Learn Cycle

tion 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 even negative effect should be investigated [KLSH09, KDF⁺12, Bos12]. Still, personally I would argue that these experimentations need to be carefully planned. 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.

Lean emphasis doing work just-in-time, not too early and not too late. Instead of dealing with a lot of upfront 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 immense 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 were integrated overnight.

2.6 Focus on the Essential

You eliminate waste by using activities and resources that are only absolutely necessary. Everything else is waste. You create an environment where you amplify learning, by doing it right the first time. 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 emphasises 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 small tasks [Mon12]. These tasks are for example written down on small paper 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 it as quickly as possible? In many software projects, releasing new features is a manually intensive process. This should not be the case. Releasing software has a high risk of failure. Will it for example work in its intended environment and not just on the developer machine? A deployment pipeline is the basis for many modern development practises. Anything that you can treat as construction should be automated [Fow05]. This was expensive back in the days, but is not an issue anymore [Roy70]. One of the obstacles of an automated build and test environment is that you want to be able to build fast so that you can get fast feedback [Fow13b]. To ensure quality, you have a comprehensive set of tests for your code. Running these tests can take a long time. A deployment pipeline handles this by breaking up your build into multiple stages. Each

stage increases trust that everything is working as expected.

Jez Humble and David Farley describe three common anti-patterns: 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 they involve many moving parts. This leaves the process prone to human error. Deploying software manually is a fragile and time-consuming process. Ideally a software should be able to be deployed by anyone with a push of a button. No hassle in finding out the steps to do so and automated ways in discovering if something has gone wrong. This is accomplished by creating scripts and tasks that build a production ready installer or deploy the software to the cloud.

Something can still go wrong, this is why you first deploy your software to a production-like environment to make sure it works as expected. This increases your confidence that once the software is finally deployed to production, the likelihood for issues is smaller. If this “staging” is part of your development process, you have time to discover these issues when they are still easy to fix. In addition, managing the production environments should be made as easy as possible. The application stack should be easy to maintain and all configurations should be in a repository. You should be able to create your production environment precisely, preferably in an automated fashion. Virtualisation and service-oriented platforms can help to achieve this.

The notion 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 quickly so that the developers receive feedback promptly and can act upon it. Features should be considered done only when they are released [HF11].



Figure 4: Deployment Pipeline

Usually a deployment pipeline consists of three stages: development, staging and production. See figure 4. The stages can be automated or require human interaction. In fact, they can also be executed in parallel to each other when possible. A deployment pipeline usually begins by building the software. The pipeline runs the automated tests, but can also include manual checks that cannot be automated. Usually deploying software to

production is one of the final stages of a deployment pipeline. 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.

3.1 Three Stages

It all starts with a 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 repository. This source code management keeps track of changes and makes it possible for multiple 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 code has changed over time. This makes troubleshooting easier. A new developer should be able to pull a local copy of the repository and with minimal effort get the application building and running. It should only be required that the developer has to install the basic frameworks such as the programming language and related libraries to get going. Everything else should be in the repository. Building the application should be an automated process.

A developer is encouraged to implement features in small chunks. Continuously integrating code is practise, not a tool [HF11]. Development practises require a degree of commitment and discipline from the team. Developers write the code, any related automated test cases for the new code and manually test that the feature is working as desired. Usually the developer also interacts with the actual software. After this the developer runs all the test cases for the project. Thus making sure that local changes have not broken something else in the application. Finally, the work is integrated into the shared repository. In small chunks, multiple times a day. Integrating code most of all is about conveying information amongst the team. Other developers can swiftly see what has changed, test new features and make sure nothing conflicts with the work they are doing. This prevents any major integration troubles.

A local development machine is only a local setting and developers are humans who make mistakes. An application can seemingly work correctly on a local environment, but this must be verified. Once the feature has been integrated into the repository, it is then immediately tested in a production-like setting. A server runs the scripts and task related to building and testing the application. Automated tests are run and any other checks are made to make sure the code is satisfactory. These can include anything from statically analysing the code to verifying the test coverage. If anything fails, the developers should notice the issues relatively soon and fix them.

Once a feature has been integrated and tested, it first deployed to a production-like staging environment. 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 environment. By testing on multiple platforms, you ensure the likelihood of the software working also on an other environment. Of course, it is still not practical to test you code on every single platform.

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 final product to users.

Finally, the last step includes deploying the application to production. 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 rolledback to a previous version if any issues are raised. According to Agile and Lean principles, only once a feature is deployed it is considered to be finally done.

3.2 Using Web as a Platform

The world is not static. The acceleration of digital products and services means the web will become more and more irreplaceable for software-intensive products and services. We have barely touched the surface of what the web will provide. 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. The cost of this 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 when necessary over the network. 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. This has also transformed software delivery: there is a fundamental shift in how products and services are developed and deployed.

Cloud computing uses a service-driven model. Typically, cloud computing provides three services: infrastructure such as computing and storage (Infrastructure as a Service, IaaS), platforms such as operating systems and software development frameworks (Platform as a Service, PaaS) and on-demand software applications (Software as a Service, SaaS) [ZCB10]. It is no surprise that many of these services have become platforms for the deployment pipeline. This move has generated service-oriented platforms that provide many of the common functionalities involved in software delivery. Web-applications and services can be developed and deployed with ease.

Popular service-oriented platforms for the deployment pipeline include: GitHub for shared code repositories and project management, Travis CI for continuous integration and Heroku for web-application deployment [Git, Tra, Her]. See figure 5 for the deployment pipeline flow.

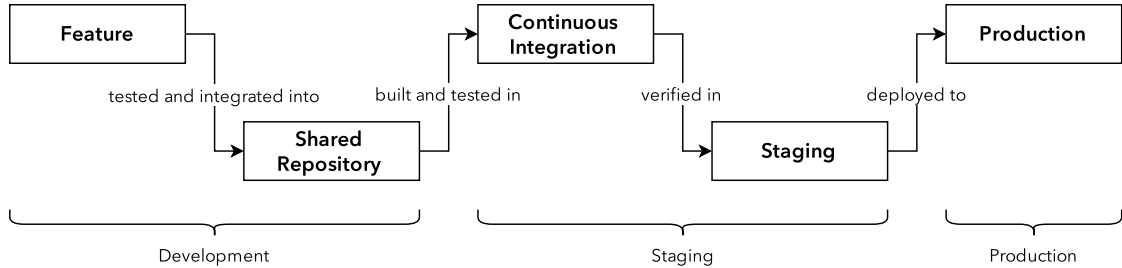


Figure 5: Deployment Pipeline Flow

3.3 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. 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 fix and repeat until the shared state is functional. A feature is only considered done when the CI-process succeeds.

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. Everything you need to build the software should be in the repository. 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 an 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 ran on a separate automated integration machine [Fow06]. This is generally accomplished with a CI -server. The 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 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 comprehensively automated test suite for the software. The tests are integrated into an integration and build process which in affect results in a stable platform for future development. 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]. Projects that use CI, tend to have dramatically less bugs [Fow06].

Continuous Integration also provides a way of making the latest version of the software being always accessible. Other developers and customers can then demonstrate, explore and see what has changed since the previous version with ease.

3.4 Continuous Deployment

Continuous Deployment (CD) is a development practise where you build software throughout its lifecycle so that it can be deployed automatically at any given time [Fow13a]. CD 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, the latter automates this process by doing deployments automatically to production. This results in many production deployments each day [O'R11, Sny13, Rub14].

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 then 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. Deploying software frequently gives a sense of believable progress, not just developers declaring features done [Fow13a]. This requires extensive automa-

tion throughout the deployment pipeline, but also a close and collaborative working relationship between everyone from developers to system specialists involved in the software delivery [Fow13a]. Lately this has been referred to as the “DevOps culture” [Fow13a]. In practise, developers should have also control on how software is hosted and this should not be primarily outsourced [HF11]. Stakeholders can then test the system and the feedback cycle is short. 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 a feature, the quicker you get feedback on it. The web has enabled the possibility to deploy and explore web applications to a subset of users [Fow06, Fow13a]. This can then be used as factor in making decisions about where to proceed.

3.5 Continuous Experimentation

The world is never static, being able to figure out what works and what does not can mean the difference between being in on the top and becoming invisible [KLSH09]. The web has 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. This is also called A/B testing. Data is then 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 not always why. Different implementations can have very unexpected results [KLSH09, KDF⁺12]. It is intriguing how poor we are at assessing the values of our ideas — many assumptions are simply wrong [KDF⁺12]. Most of times these assumptions have significant effects. Features were built because developers believed they were useful. Even worse, these opinions could come from managers with higher positions [KLSH09, Bos12]. Of course, the significance of intuition and luck should not be belittled.

Controlled experiments provide a methodology to reliably evaluate the value of ideas [KLSH09]. Passive feedback can provide much more valuable information than actively asking feedback from users. By building a system for experimentation, the cost of testing and failure becomes small. This encourages innovation by doing experimentation. Failing fast and knowing when an idea is not great is essential in making course corrections and developing more successful 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 an thriving environment for experimentation. Experimentation can be used to understand what the user wants. When an issues 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 pipeline [Bos12].

Continuous Experimentation (CE) 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]. Continuous Experimentation makes use of minimum viable products as the basis for an hypothesis and experiment. Choices are made by analysing the data gathered from this minimum implementation. The hypothesis is either supported by the data or not. It is necessary to base decisions on sound evidence rather than guesswork [FGMM14]. Micromanagement 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 suites. Instead of rigorously testing features beforehand, automatic analytics are run in production. Heuristics are used to immediately discovers issues and alert about their consequences.

4 Towards Embedded Systems

The biggest barrier to adopting new practises is organisational [Pop02]. 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. The agile notion of moving all control to developers is hard to accomplish. Hardware development should be intertwined with software development. Developers and engineers should work more closely on these systems to get the most out of Agile development. Defining fully elaborated requirements work in certain applications where for example security is a important criteria [Boe88]. It does not however work particularly well with interactive applications that are targeted to end-users. Be that as it may be, even Boehm observed that iterative development suited equally both software and hardware development [Boe88].

Predictability may be desired. Organisations such as NASA are prime examples where software development must be predictable. NASA's 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].

History has many successful examples of the usage of iterative development in software development in embedded systems. The X-15 hypersonic jet applied iterative and incremental development already back in the 1950s [LB03]. 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 the 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. 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 weapon 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. LAMPS was one of the earliest project that used short iterations that only took one month per iteration. The project succeeded, deliveries were on time and under budget [LB03].

Another noticeable story is the primary avionics software for NASA’s space shuttle program in the late 1970s. 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. Still, it used rather long iterations of six months by modern standards. All these examples are early examples of being agile, but they only applied a fraction of ideas presented by current ideologies.

4.1 Using Hardware as a Platform

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 [Bos12]. This includes even software-intensive embedded systems. The current trend is connecting these systems to the Internet, sometimes also referred as the Internet of Things. Data can be used in the research and development process of these products. For

example cars can collect real-time fuel consumption data and telecom devices real-time bandwidth data [Bos12]. Previously these points of interest were collected for management purposes, now they can be used for development purposes. In embedded systems, software is deployed as part of the overall system, including its hardware.

Service-oriented models have started to find their way to more embedded settings. 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. Organisations are also interested in collecting usage and other performance related metrics. Instead of freezing requirements before starting product development, the requirements evolve and have an 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 unconnected static hardware are fast becoming unsustainable. Upgrading is expensive and integrating new solutions involve risk and complexities that can be hard to predict.

Remote deployments are possible also on embedded systems [Bos12]. Instead of servers, the software is deployed to the destination point. This obviously requires changes to the architecture of the platforms to facilitate remote deployments and experimentation. Jan Bosch talks about building an innovation experiment system [Bos12].

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]. Experimenting with hardware products is getting more accessible and cheaper.

4.2 Adapting for Deployment Pipeline

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]. Setting up an experimentation cycle can be rather challenging developing software that requires hardware. Longer release cycles with hardware and potential synchronisation problems between the development schedules is an issue [FGMM14]. Certain life-critical environments, experimentation can just be too expensive or undesirable for the stakeholders.

5 Cases from Embedded Settings

Software is considered “soft”, hardware “hard”. 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. I 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? 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?

6 Conclusions

Conclusions.

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).
- [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>.
- [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*, pages 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.
- [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>.

- [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, pages 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).
- [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>.
- [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, pages 97–102, New York, NY, USA, 2013. ACM, ISBN 9781450322393. <http://doi.acm.org/10.1145/2512276.2512306>.
- [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.
- [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, pages 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.
- [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>.
- [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*, pages 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.
- [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=dxk8b9rSK0o>, (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).
- [Rie11] Ries, Eric: *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business, 2011, ISBN 9780307887894.
- [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).
- [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*, pages 44–53, September 2010. <http://dx.doi.org/10.1109/QUATIC.2010.17>.
- [Tra] *Travis CI*. <https://travis-ci.org>.
- [Vol15] Voltera: *Voltera: Your circuit board prototyping machine*, 2015. <https://kickstarter.com/projects/voltera/voltera-your-circuit-board-prototyping-machine/>, (accessed 6 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>.