The Journal of Systems and Software 000 (2015) 1–14

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Continuous software engineering: A roadmap and agenda

Brian Fitzgerald, Klaas-Jan Stol*

Lero—the Irish Software Research Centre, University of Limerick, Ireland

ARTICLE INFO

Article history:
Received 11 November 2014
Revised 24 April 2015
Accepted 25 June 2015
Available online xxx

Keywords:
Continuous software engineering
Lean software development
DevOps

ABSTRACT

Throughout its short history, software development has been characterized by harmful disconnects between important activities such as planning, development and implementation. The problem is further exacerbated by the episodic and infrequent performance of activities such as planning, testing, integration and releases. Several emerging phenomena reflect attempts to address these problems. For example, Continuous Integration is a practice which has emerged to eliminate discontinuities between development and deployment. In a similar vein, the recent emphasis on DevOps recognizes that the integration between software development and its operational deployment needs to be a continuous one. We argue a similar continuity is required between business strategy and development, *BizDev* being the term we coin for this. These disconnects are even more problematic given the need for reliability and resilience in the complex and data-intensive systems being developed today. We identify a number of continuous activities which together we label as 'Continuous *' (i.e. *Continuous Star*) which we present as part of an overall roadmap for Continuous Software engineering. We argue for a continuous (but not necessarily rapid) software engineering delivery pipeline. We conclude the paper with a research agenda.

1. Introduction

Software development has been characterized by harmful disconnects between important activities, such as planning, analysis, design and programming. This is clearly reflected in the traditional waterfall process for software development described (and criticized) by Royce (1987). In the last two decades, there has been a widespread recognition that increasing the frequency of certain critical activities helps to overcome many challenges. Practices such as 'release early, release often' are well established in open source software development (Feller et al., 2005), which offer several benefits in terms of quality and consistency (Michlmayr et al., 2015). The pervasive adoption of agile methods (Kurapati et al., 2012; Papatheocharous and Andreou, 2014) provides ample evidence of the need for flexibility and rapid adaptation in the current software development environment. Very complex and business- and safety-critical software is being developed, often by distributed teams. A tighter connection between development and execution is required to ensure errors are detected and fixed as soon as possible. The quality and resilience of the software is improved as a result. This is manifest in the increasing adoption of continuous integration practices. The popularity of continuous integration is facilitated by the explicit recommendation of the practice in the Extreme Programming agile method (Beck, 2000), and indeed the practice is highly compatible with the frequent iterations of software produced by agile approaches. Also, many open source toolsets such as Jenkins CI[1] are freely available to automate the continuous integration process which makes this practice readily available to potential adopters.

However, a number of recent trends illustrate that a more holistic approach is necessary rather than one which is merely focused on continuous integration of software. For example, the *Enterprise Agile* (Overby et al., 2005) and *Beyond Budgeting* (Bogsnes, 2008) concepts have emerged as a recognition that the benefits of agile software development will be sub-optimal if not complemented by an agile approach in related organizational functions such as finance and HR (Leffingwell, 2007; Overby et al., 2005). In a similar vein, the recent emphasis on *DevOps* recognizes that the integration between software development and its operational deployment needs to be a continuous one (Debois, 2009). Complementing this, we argue that the link between business strategy and software development ought to be continuously assessed and improved, "BizDev" being the term we coin for this process. Several researchers across the management, information systems and software engineering disciplines have provided arguments which reinforce the BizDev concept and we elaborate on this in Section 4 below.

Furthermore, the holistic approach mentioned above should not assume that the process is complete once customers have initially

* Corresponding author. Tel.: +353 61 233737.
E-mail addresses: bf@lero.ie (B. Fitzgerald), klaas-jan.stol@lero.ie (K.J. Stol).

[1] https://jenkins-ci.org/

http://dx.doi.org/10.1016/j.jss.2015.06.063
0164-1212/© 2015 Elsevier Inc. All rights reserved.

Table 1
Seven additional practices to scale agile to the enterprise level according to Leffingwell (2007).

Practice	Description
Intentional Architecture	For large systems consisting of many subsystems, agile components team fit their components into an intentional architecture, which is component-based and is aligned with the team's core competencies, physical location and distribution.
Lean requirements at scale	Non-functional/cross-cutting system requirements such as performance and reliability must be considered and understood by all teams contributing to the system. Define a *vision* stating the overall objectives; a *roadmap* that defines an implementation strategy, and defer specific requirements as long as possible, implement them *just-in-time*.
Systems of systems and the agile release train	Create a release schedule with fixed dates that is imposed upon all teams, and leave decisions regarding the delivered features/functionality up to those teams.
Managing highly distributed development	Large-scale software development is inherently distributed, as developers are necessarily located in different physical locations, ranging from different rooms, floors, and buildings to different countries and time zones. Additional coordination practices and enterprise-level tool support are necessary.
Impact on customers and operations	More frequent delivery has an impact to a variety of stakeholders, including sales and marketing, operations, support, distribution organizations, and ultimately customers. The interaction with each of these stakeholders must be considered and adapted to fit enterprise-level agile adoption.
Changing the organization	Transforming the enterprise to an agile one requires a combination of top-down leadership, bottom-up adoption and expansion and empowered managers in the middle, all with a common vision.
Measuring business performance	Whereas the key measure of small-scale agile is the presence of working software, enterprise agile requires a number of additional measures to monitor efficiency, value delivery, quality, and agility.

adopted a software product. Digital natives, the term for those who have been born in the technology era (Vodanovich et al., 2010) have high expectations of software and are not put off by the high switching costs associated with moving to alternatives. Frequently, third-party opinions and word-of-mouth can cause customers to switch, and software providers must be more proactive in such a market-place. Also, privacy and trust issues loom much larger in the data-intensive systems being used today. Run-time adaptation is increasingly a factor as software is expected to exhibit some degree of autonomy to respond to evolving run-time conditions.

We believe that rather than focusing on agile methods *per se*, a useful concept from the lean approach, namely that of 'flow' (Reinertsen, 2009) is useful in considering continuous software engineering. Rather than a sequence of discrete activities, performed by clearly distinct teams or departments, the argument for continuous software engineering is to establish a continuous movement, which, we argue, closely resembles the concept of flow found in lean manufacturing and product development (Morgan and Liker, 2006), a school of thought that is called 'Lean Thinking' (Womack and Jones, 2003). In recent years, there has been much interest in lean software development (Conboy and Fitzgerald, 2004; Fitzgerald et al., 2014; Maglyas et al., 2012; Petersen, 2011; Wang et al., 2012), however, there has been a frequent tendency to adopt a somewhat narrow view on the topic by closely linking it to agile practices only.

In this paper we review a number of initiatives that are termed 'continuous.' This paper extends our preliminary view on this topic (Fitzgerald and Stol, 2014). The goal of this paper is to sketch a holistic view of these initiatives and position them within the continuous software engineering context, and illustrate how Lean Thinking is a useful and relevant lens to view continuous software engineering. Furthermore, we look beyond software development, and consider issues such as continuous use, continuous trust, etc. and coin the term 'Continuous *' (pronounced as "Continuous Star") to refer to this holistic view.

The various developments are by and large at different levels of maturity—continuous integration is a concept and practice that has gained widespread currency, probably in large part due to it being a formal practice in XP. However, recent research shows that different organizations implement this practice in different ways (Ståhl and Bosch, 2013). In contrast, continuous delivery is an idea that has not widely been established in the software industry. Thus, we consider our holistic overview as a conceptual research agenda, and we envisage future research to operationalize each of the concepts identified, much as Ståhl and Bosch have initiated for continuous integration.

This paper proceeds as follows. Section 2 describes a number of developments that have attempted to scale up the agile paradigm, such as Enterprise Agile, Beyond Budgeting and DevOps. Section 3 reviews a number of key concepts from the school of Lean Thinking, of which the concept of 'flow' is the most important as it can be used as a suitable foundation for the holistic concept of 'Continuous *' that we propose in this paper. Section 4 presents the activities which comprise Continuous * in more detail, and we observe how the various concepts of lean thinking can be identified within these activities. Section 5 discusses a number of directions for future research and implications for practice.

2. Trends in the software engineering landscape

A number of recent trends have focused on the need to transform organizations to deliver software more rapidly. We discuss these in turn.

2.1. Enterprise agile

Agile methods were initially considered to be only suitable for small teams, and research on agile methods has long focused quite narrowly on the software development function only. In recent years, several authors have identified the need to scale the agile concept to the enterprise level (Kettunen and Laanti, 2008; Reifer et al., 2003). Leffingwell (2007), for example, documented a set of seven practices to complement the practices that are common to agile methods such as Scrum, XP and DSDM (see Table 1). These seven principles address several dimensions in which agile approaches should scale, such as the link to other functions of the organization (e.g., marketing, operations), the product (e.g., architecture, requirements), and the development process (e.g., distributed development). Leffingwell also initiated the Scaled Agile Framework (SAFe).[2] The SAFe is based on experiences of organizations that have adopted agile at enterprise scale and describes practices and activities, roles, and artifacts.

Overby et al. (2005) defined 'enterprise agility' as *"the ability of firms to sense environmental change and respond appropriately,"* hence identifying the two main components of 'sensing' and 'responding' appropriately. As organizations may possess different capabilities to sense and respond, these can be seen respectively as two dimensions along which organizations may be positioned. For instance, an organization may have well-developed capabilities to *sense* new market

[2] http://www.scaledagileframework.com/

opportunities, but less-developed capabilities to *respond* appropriately (e.g., deliver newly requested features rapidly). Clearly, software organizations should aim at developing their sensing capabilities, for example through usage analytics of their software delivered to customers, as well as develop their responding capabilities, for example through quickly adapting and delivering new features.

2.2. DevOps

The DevOps concept (Debois, 2009) emerged from the increasing disconnect between the development and operations functions that arise within large software companies. As organizations scale, so do development and operations teams, and while they may initially be co-located and have close communication links, increased team size and more strict separation of responsibilities can weaken such links (Swartout, 2012). The DevOps term represents the need to align the development of software and the deployment of that software into production (Debois, 2011). Another trend in this context is that software is increasingly delivered through the Internet, either server-side (so-called Software-as-a-Service) or as a channel to deliver directly to a customer's machine or device. Furthermore, the platforms on which this software runs become increasingly pervasive, such as smartphones and tablets, and an emerging trend is that of wearable technology. Examples of software for mobile devices delivered through the internet include Apple's iOS and Google's Android mobile operating systems.

While no common definition exists for DevOps, Humble and Molesky (2011) defined four principles:

- Culture—DevOps requires a cultural change of accepting joint responsibility for delivery of high quality software to the end-user. This means that code no longer can be "thrown over the wall" to operations.
- Automation—DevOps relies on full automation of the build, deployment and testing in order to achieve short lead times, and consequently rapid delivery and feedback from end-users.
- Measurement—Gaining an understanding of the current delivery capability and setting goals for improving it can only be done through measuring. This varies from monitoring business metrics (e.g., revenue) to test coverage and the time to deploy a new version of the software.
- Sharing—Sharing happens at different levels, from sharing knowledge (e.g. about new functionality in a release), sharing tools and infrastructure, as well as sharing in celebrating successful releases to bring development and operations teams closer together.

These principles again reflect the trends of changing perspectives on responsibility and measurement which are also found in Enterprise Agile. Furthermore, automation emphasizes a greater reliance on tools and a sound development and delivery infrastructure.

2.3. Beyond Budgeting

Another related trend is that of Beyond Budgeting, an innovation with its roots in management accounting (Bogsnes, 2008; Lohan, 2013). The Beyond Budgeting trend started when the "Beyond Budgeting Round Table" (BBRT) of the Consortium for Advanced Manufacturing International (CAM-I) was founded in 1998 by Hope and Fraser, and was formalized in their book (Hope and Fraser, 2003). The Beyond Budgeting school of thinking seeks improvements over traditional budgeting approaches, which tend to be annual time-consuming activities, resulting in rigid budget plans that reinforce departmental barriers (Lohan, 2013). Budgets serve different uses in organizations, including performance management and evaluation, strategy implementation and formation (Lohan, 2013).

Traditional budgeting approaches suffer from a number of drawbacks: they tend to be infrequent (mostly annual), time-consuming, and as a result do not allow agile responses to changing markets (see Enterprise Agile, Sec. 2.1). To address these issues, the Beyond Budgeting management model defines a set of 12 principles which resonate strongly with lean principles such as defined by Liker (2004) and Womack and Jones (2003), organized into two sets, namely *Leadership* principles and *Process* principles (see Fig. 2). Similar to the lean philosophy, the Beyond Budgeting model advocates a customer-centric focus, systems thinking and self-organization and empowerment of people within an organization.

2.4. Lean startups

The principles of lean manufacturing have also influenced the Lean Startup concept (Bosch et al., 2013; Maurya, 2012; Ries, 2011, p.6). The Lean Startup method was proposed by Ries in 2011 and emphasizes continuous learning. Most startups fail due to the fact that they go out of business before sufficient revenue comes in; in other words, the time between 'order' and 'cash' (see Fig. 1) is too long. The solution to this is, of course, to shorten the time between the key idea that a startup tries to market, and the delivery to its customers. One common challenge in startups is that well-defined requirements are lacking due to a high level of uncertainty about the startup's product idea. In other words, an understanding of what constitutes 'value'

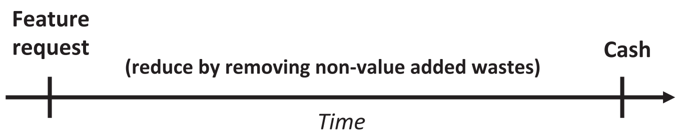

Fig. 1. Lean Thinking aims to reduce the time between order and cash (adapted from Ohno (1988)).

Leadership Principles

- Customers. Focus everyone on improving customer outcomes, not on hierarchical relationships.
- Organization. Organize as a network of lean, accountable teams, not around centralized functions.
- Responsibility. Enable everyone to act and think like a leader, not merely follow the plan.
- Autonomy. Give teams the freedom and capability to act; do not micromanage them.
- Values. Govern through a few clear values, goals, and boundaries, not detailed rules and budgets.
- Transparency. Promote open information for self-management; do not restrict it hierarchically.

Process Principles

- Goals. Set relative goals for continuous improvement; do not negotiate fixed performance contracts.
- Rewards. Reward shared success based on relative performance, not on meeting fixed targets.
- Planning. Make planning a continuous and inclusive process, not a top-down annual event.
- Controls. Base controls on relative indicators and trends, not on variances against plan.
- Resources. Make resources available as needed, not through annual budget allocations.
- Coordination. Coordinate interactions dynamically, not through annual planning cycles.

Fig. 2. 12 Principles of Beyond Budgeting (Bogsnes, 2009).

to the customer is unclear. Informed by the principles of lean manufacturing, lean startups attempt to solve this problem by continuous learning through a process of experimentation (as discussed in more detail below). Ries (2011) defined five principles of the Lean Startups approach:

1. *Entrepreneurs are everywhere.* Ries defines a ‘startup’ as:

   > *“a human institution designed to create new products and services under conditions of extreme uncertainty. That means entrepreneurs are everywhere and the Lean Startup approach can work in any size company, even a very large enterprise, in any sector or industry.”*

   In other words, the Lean Startup approach is not restricted to startups, but can also be applied within large organizations.
2. *Entrepreneurship is management.* Ries argues that startups require careful management that is appropriate to deal with the extreme uncertainty inherent in developing new ideas into products.
3. *Validated learning.* A core aspect of a startup is *“to learn how to build a sustainable business.”* This principle highlights the need to take the ‘scientific’ approach, namely by running frequent experiments that allow entrepreneurs to evaluate their hypotheses.
4. *Build-Measure-Learn.* This principle summarizes the key activity in startups, namely that of establishing a continuous cycle of getting feedback. The Build-Measure-Learn cycle implies that ideas should be implemented as products as quickly as possible, evaluate customer response (e.g. its usage) and draw lessons based on that feedback.
5. *Innovation accounting.* Ries argues that innovators must be held accountable, and to facilitate that, startups need measurements, milestones and prioritization of tasks.

As suggested by the first principle, the Lean Startup methodology is not exclusive to startups and can also be used within large organizations. The Lean Startups movement focuses specifically on those contexts characterized by a high degree of uncertainty. In order to reduce this uncertainty and get feedback on a product as soon as possible, it is important that this testing of assumptions and hypotheses is done in short cycles.

For instance, a startup company may have an idea for a new feature that it believes will offer valuable functionality to users; however, implementing the feature would take three months. Should the company implement this feature? Some techniques adopted in the software industry include the presentation of mock-interfaces in web-based systems, whereby features are ‘advertised’ in the interface but not yet implemented. By monitoring the requests for that feature (i.e., counting clicks) the interest for that feature can be assessed. If nobody clicks the button, the hypothesis that the feature offers value is rejected, and the company can refrain from implementing the feature, thus saving three months of development time. For startup companies with only a few developers, this will have a significant impact on potentially wasted development time.

Another form of experimentation is A/B testing, whereby two (or more) different versions of a system are provided to different groups of users. For instance, a company that wishes to test the effectiveness of a particular interface might present two different versions to two different groups of users. This way, the company can monitor and compare user behavior based on the two different ‘treatments’ (the different interfaces). This form of experimentation was used by Google, for instance, to test ‘click’ behavior based on different shades of blue used to color page links (Feitelson et al., 2013).

The dramatic growth in the size of software systems has long been evidenced (Müller et al., 1994; Sommerville, 2007). Also, the vast amount of legacy systems that exist in organizations, allied to the amount of open source software that is freely available, means that much software development does not begin from scratch with new source code, but builds on existing software. This in turn leads to a greater need for continuous software engineering.

Table 2
Seven types of waste according to Ohno and examples of how they can manifest in software development.

Waste	Example in software development
Overproduction	Unwanted features
Waiting	Waiting for build process (compilation) or test suites to finish
Transportation / hand-overs	Handing over software from an agile development team to a traditional operations team
Too much machining (over-processing)	Recompiling unchanged software; running test suites unnecessarily
Inventories	Unfinished features
Moving (motion)	Task switching e.g. developers working on different projects, losing ‘state of mind’ whenever they refocus on a different project
Making defective parts and products	Software defects

3. Lean Thinking

The term ‘lean’ was coined by Krafcik (1988) to describe the mode of operation in the Toyota Production System (TPS) (Liker, 2004). Numerous books have been published on TPS and ‘lean manufacturing,’ such as ‘Lean Thinking’ (Womack and Jones, 2003). While a full outline of the lean philosophy is outside the scope of this paper, we introduce a number of key lean concepts that can be observed in many distinct software engineering practices (see Table 3 for a summary of terms).

3.1. Value and waste

A fundamental focus in lean thinking is to shorten the time between a customer order and the delivery of that order; any activities that do not add ‘value’ are considered ‘waste’ and should be removed (Ohno, 1988; Womack and Jones, 2003) (see Fig. 1). One of the founding fathers of the TPS, Taiichi Ohno, identified seven types of waste (Ohno, 1988) (see Table 2), and others identified additional types such as unused skill and creativity. One such waste is overproduction: producing something that is unwanted, such as unused product

Table 3
Lean thinking terminology and examples in software development.

Lean term	Example in software engineering
Andon	Traffic light connected to continuous integration server – this is a tool to achieve *jidoka*
Chaku Chaku	Automated delivery pipeline
Genchi Genbutsu	Daily stand-up meeting
Hanedashi	Automatic ejection, for instance automatic testing of newly checked in code
Heijunka	Workload leveling of features with a kanban
Jidoka	Tools and techniques to detect faults so that they can be corrected as soon as possible, e.g., continuous integration
Kaikaku	Reimplementation, architectural overhaul, transition from waterfall to agile
Kanban	Kanban board to let developers ‘pull’ stories to implement once they have capacity to do so
Kaizen	Sprint retrospective meetings, refactoring
Obeya	Scrum board
Poka Yoke	Fool proofing mechanisms to prevent mistakes and defects early, e.g., syntax highlighting, unit tests, static source code checkers such as `splint`
Single-minute Exchange of Dies (SMED)	Automatic deployment of a new software version with a single press of a button

features. This type of waste is common in traditional plan-driven software development methods, most notably the waterfall model, whereby requirements are identified, converted into a design and implemented in a product (Petersen and Wohlin, 2010). While in theory this process should work, in practice, requirements virtually always change (Larman and Basili, 2003), resulting in the wrong product being built. Other authors, including Royce who documented the waterfall approach, also argued that the waterfall approach does not work for any but the most trivial systems (Larman and Basili, 2003; Petersen and Wohlin, 2010; Royce, 1987). However, without any customer feedback as to whether a feature is needed, there may be significant waste resulting in the development of a bloated software product. Techniques such as those described above (counting clicks in mock-interfaces) can help to gauge the level of interest in features.

One of the first steps for organizations that wish to reduce waste in their processes is to conduct a *value stream mapping* exercise, whereby the current processes are identified and visualized so as to be able to gauge where improvements can be made. Mujtaba et al. (2010) and Khurum et al. (2014) have demonstrated the use of value stream mapping in a software development context.

3.2. Flow and batch size

As already mentioned, flow is a central concept within Lean Thinking (Womack and Jones, 2003). Flow can be contrasted with 'batch-and-queue' thinking, in which actions are done on batches of products, after which they are queued for the next processing step. By contrast, flow refers to a connected set of value-creating actions—once a product feature is identified, it is immediately designed, implemented, integrated, tested, and deployed. Establishing a continuous flow does not refer just to a software development function in isolation, but should be leveraged as an end-to-end concept that considers other functions within an organization such as planning, deployment, maintenance and operation.

Many traditional software development environments are still operating according to the principles of batch-and-queue. While the software development function might 'flow' to some degree, the planning and deployment of features is still done in batches, and not in a continuous flowing movement. Increasingly, the software industry is adopting *kanban* as an alternative approach for scheduling work, or as a source to augment the longer established Scrum method (Fitzgerald et al., 2014). Kanban can help to level the daily workload, a concept known in lean thinking as *heijunka* (Liker, 2004).

Whereas agile software development is mostly focused on the software development function, Lean Thinking has a very explicit focus on the end-to-end process: from customer to delivery. Based on a comparison of principles and practices of agile and lean software development, Petersen (2011) concluded this end-to-end focus is the key difference between agile and lean.

3.3. Autonomation and building-in quality

A third key concept found within TPS and lean thinking is autonomation, or "*automation with a human touch*" (Ohno, 1988, p.6). Another term used for this is *jidoka*, or *built-in quality*. This refers to tools and visual aids in closely monitoring quality during the production process. For instance, an *andon* is a line stop indication board, which indicates the location and nature of troublesome situations at a glance (Ohno, 1988, p.21). This can also be observed in software development; for instance, many organizations that practice continuous integration have an indicator, sometimes an actual traffic light, that turns to red as soon as the build fails. Continuous integration servers, such as Tinderbox[3] provide a visual interface that can link specific code changes to build failures.

Another related term is *Poka Yoke* which has been defined as consisting of checklists, test plans, quality matrices, standard architecture, shared components, and standardized manufacturing processes (Morgan and Liker, 2006, p.95). Poka Yoke, or *Baka-Yoke* are fool proofing mechanisms to help eliminate mistakes, and assist an engineer in identifying problems as soon as possible. Software developers can choose from a range of tools to assist them in early fault detection. Besides normal compilers which highlight errors and warnings, specialized tools such as `splint`[4] can statically check software and identify potential errors. Many software development organizations also use coding standards which can help to maintain a consistent style of source code, which in turn can help to prevent the introduction of programming errors. Another form of standardization is the use of reference architectures, which can help software architects and developers to prevent the mis-use of software components. Modern integrated development environments (IDEs) offer built-in calculation of a variety of source code metrics; the use of object-oriented metrics, for instance, could help to identify potential code "smells" – this too, is a form of poka yoke.

The Japanese term *Chaku Chaku* means 'load load,' and refers to the idea to efficiently load parts (work-in-progress) from machine to machine, which are conveniently placed closely together. Together with *hanedashi*—automatic ejection—this can contribute to improve the flow. This concept is also gaining increasing attention in software engineering, with practices such as continuous integration and continuous delivery. Once new features, functionality or additions are implemented, they are automatically tested, and upon successful completion can be deployed automatically.

3.4. Kaizen and continuous improvement

Adopting 'lean thinking' is a continuous cycle of process improvement. Process improvement can take place through radical steps (*kaikaku*) in the beginning of a transformation initiative, followed by incremental improvements (*kaizen*). For instance, the decision to adopt an agile method such as Scrum to replace traditional, so-called plan-driven methods (waterfall, V-model) is an instance of 'kaikaku,' whereas review and retrospective meetings at the end of a sprint are forms of 'kaizen.' The goal of kaizen events is to make incremental improvements to the process.

Daily Scrum meetings are similar to daily build wrap-up meetings found within the Toyota Product Development System (TPDS), and are an integral mechanism of *genchi genbutsu*, which refers to the idea of *"going to see the actual situation first hand to understand deeply the current reality"* (Morgan and Liker, 2006, p.173). Typically, daily Scrum meetings take place at a Scrum Board, which can be considered an *obeya*, the Japanese term for "large room" or "war room."

Research on improving agile methods has tended to focus on the software development function within organizations. However, very little attention has been paid to the interaction with—and *improvement of*—functions such as planning, deployment, operations and maintenance. This could partially explain the barriers that organizations encounter in further improving their software development activities, as they encounter tension points with those parts of the organization that are not considered in a holistic manner.

One critical development in the TPS was that of the Single-Minute Exchange of Dies (SMED) (Shingo, 1989, p.43). Initially, changing dies in the TPS (to produce differently-shaped parts) could take a significant amount of time, up to three hours. Through a number of adjustments made after observations of how exchange of dies took place (*genchi genbutsu*), this time was reduced to only a few minutes (Ohno, 1988, p.39), a striking reduction in time wasted. A similar development can be observed in the Continuous Delivery approach: whereas

[3] https://developer.mozilla.org/en-US/docs/Tinderbox

[4] http://www.splint.org

software releases could previously take hours, days, or even weeks, Continuous Delivery allows immediate delivery of a new version of the software with a single press of a button (Swartout, 2012).

4. Continuous*: continuous software engineering and beyond

We view Continuous * as a holistic endeavor. The '*' implies that other 'continuous' activities may emerge over time which could also be positioned within this holistic view. Continuous * considers the entire software life-cycle, within which we identify three main sub-phases: Business Strategy & Planning, Development, and Operations. Within these sub-phases, we position the various categories of continuous activities (see Fig. 3 and Table 4).

4.1. *Business strategy & planning*

Historically, a gulf has emerged between business strategy and IT development, with IT departments being viewed as a necessary evil rather than a strategic partner (CA Technologies, 2012). We argue that a closer and continuous linkage between business and software development functions is important, and term this *BizDev*, a phenomenon which complements the DevOps one of integrating more closely the software development and operations functions (Debois, 2009). Continuous planning would certainly facilitate BizDev as it requires tighter connection between planning and execution.

We have argued earlier that several recently emerging phenomena are symptomatic of a need for closer integration between business needs and development. This is clearly evidenced in the short cycles of agile which seek frequent feedback from customers, and also the explicit identification of a proxy or surrogate customer in the Scrum role of Product Owner. Likewise, the Enterprise Agile and Beyond Budgeting phenomena have a similar expanded rationale. Lean thinking also advocates a more holistic end-to-end approach, much more than is the case with agile, for example. In a software engineering context, this requires a closer connection and flow between business and development and even business and operations, as meeting business needs is of paramount importance (Reifer, 2002).

The disconnect between software developers and business managers is an old and often cited problem. Software practitioners have been criticized for being excessively tool-oriented, looking for simplistic technocratic solutions without ever coming to terms with the complex reality of the business process (Earl and Hopwood, 1980). The eventual upshot has been that the software development function has receded in importance. In the past, technical staff had information power as business managers were unsure as to what was technically feasible and conceded to technical staff knowledge. While technical staff could gain business knowledge, business managers could also gain more awareness of technical issues, and the latter has largely been the case. As a consequence, the software development function is often placed in the organizational hierarchy as reporting to the financial function. This represents a demotion over time, and so it has often been characterized as a failure of technical staff to understand business needs. Many software developers understand this and express a desire for getting involved in strategic business decision-making rather than being consulted after decisions have been taken. The need to connect business management and the software development function has been identified (e.g., Rautiainen et al. (2003)). Lehtola et al. (2009) differentiate between upstream activities (e.g., business planning) and the downstream software development processes, and argue that the typical feature level roadmapping that takes place in software development is inadequate and that a more business-oriented approach is needed. They suggest greater involvement of extra stakeholders (R&D, sales and marketing, and management) earlier in the development cycle, and collectively setting the high-level targets in a co-operative fashion so that sales and marketing function, for example, can prepare their activities at the same time as the software development activities. Furthermore, a different approach needs to be followed depending on whether the software is for internal consumption (i.e., the customer is internal) or for sale to external customers. From a business perspective these two cases are different, and should be managed differently.

4.1.1. *Continuous planning*

While the topic of continuous planning has only been introduced within software engineering research relatively recently (Lehtola et al., 2009), it has long been a topic of study in the strategic management literature (e.g., Mintzberg (1994); Ozden (1987)). In the context of software development, planning tends to be episodic and performed according to a traditional cycle usually triggered by annual

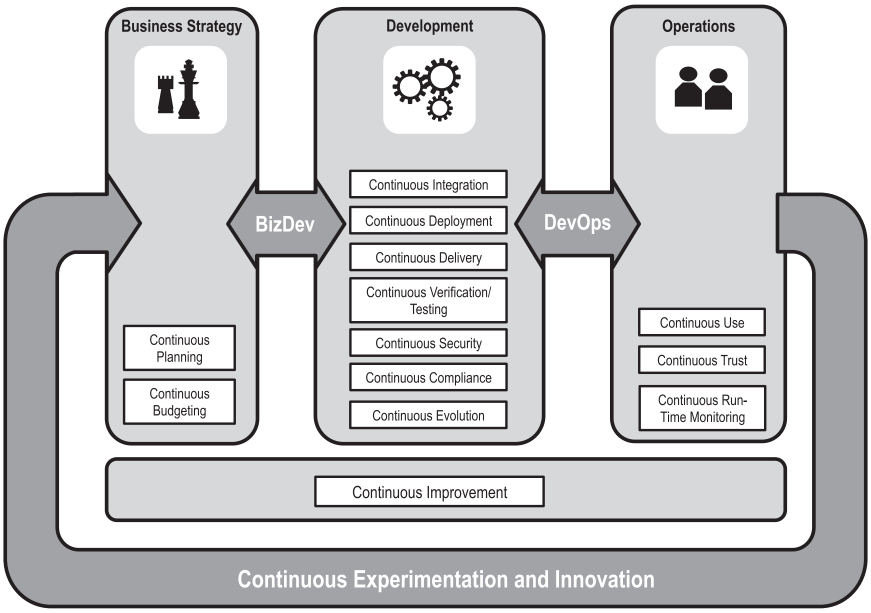

Fig. 3. Continuous*: a holistic view on activities from business, development, operations and innovation.

Table 4
Continuous* activities and definitions.

Activity	Description & references
Business strategy and planning	
Continuous planning	Holistic endeavor involving multiple stakeholders from business and software functions whereby plans are dynamic open-ended artifacts that evolve in response to changes in the business environment, and thus involve a tighter integration between planning and execution. (See Knight et al. (2001), Myers (1999); Lehtola et al. (2009)).
Continuous budgeting	Budgeting is traditionally an annual event, during which an organization's investments, revenue and expense outlook are prepared for the year to come. The Beyond Budgeting model suggests that budgeting becomes a continuous activity also, to better facilitate changes during the year. (See Bogsnes (2008), Frow et al. (2010), Lohan (2013)).
Development	
Continuous integration	A typically automatically triggered process comprising inter-connected steps such as compiling code, running unit and acceptance tests, validating code coverage, checking coding standard compliance and building deployment packages. While some form of automation is typical, the frequency is also important in that it should be regular enough to ensure quick feedback to developers. Finally, any continuous integration failure is also an important event which may have a number of ceremonies and highly visible artifacts to help ensure that problems leading to integration failures are solved as quickly as possible by those responsible. (see Kim et al. (2008); Lacoste (2009); Rogers (2004); Ståhl and Bosch (2013); Stolberg (2009))
Continuous delivery	Continuous delivery is the practice of continuously deploying good software builds automatically to some environment, but not necessarily to actual users (See Neely and Stolt (2013); Humble and Farley (2010)).
Continuous deployment	Continuous deployment implies continuous delivery and is the practice of ensuring that the software is continuously ready for release and deployed to actual customers (see Claps et al. (2015); Fitz (2009); Holmström Olsson et al. (2012)).
Continuous verification	Adoption of verification activities including formal methods and inspections throughout the development process rather than relying on a testing phase towards the end of development (see Chang et al. (1997); Cordeiro et al. (2010)).
Continuous testing	A process typically involving some automation of the testing process, or prioritization of test cases, to help reduce the time between the introduction of errors and their detection, with the aim of eliminating root causes more effectively. (See Bernhart et al. (2012); Marijan et al. (2013); Muslu et al. (2013); Saff and Ernst (2003)).
Continuous compliance	Software development seeks to satisfy regulatory compliance standards on a continuous basis, rather than operating a 'big-bang' approach to ensuring compliance just prior to release of the overall product. (See Fitzgerald et al. (2013); McHugh et al. (2013)).
Continuous security	Transforming security from being treated as just another non-functional requirement to a key concern throughout all phases of the development lifecycle and even post deployment, supported by a smart and lightweight approach to identifying security vulnerabilities. (See Merkow and Raghavan (2011)).
Continuous evolution	Most software systems evolve during their lifetime. However, a system's architecture is based a set of initial design decisions that were made during the system's creation. Some of the assumptions underpinning these decisions may no longer hold, and the architecture may not facilitate certain changes. In the last years, there has been increased focus on this topic. When an architecture is unsuitable to facilitate new requirements but shortcuts are made nevertheless, technical debt is incurred. (See Del Rosso (2009); Riaz et al. (2009)).
Operations	
Continuous use	Recognizes that the initial adoption versus continuous use of software decisions are based on different parameters, and that customer retention can be a more effective strategy than trying to attract new customers. (See Bhattacherjee (2001); Gebauer et al. (2013); Ortiz de Guinea and Markus (2009)).
Continuous trust	Trust developed over time as a result of interactions based on the belief that a vendor will act cooperatively to fulfill customer expectations without exploiting their vulnerabilities. (See Gefen et al. (2003); Hoehle et al. (2012); Zhou (2013)).
Continuous run-time monitoring	As the historical boundary between design-time and run-time research in software engineering is blurring (Baresi and Ghezzi, 2010), in the context of continuously running cloud services, run-time behaviors of all kinds must be monitored to enable early detection of quality-of-service problems, such as performance degradation, and also the fulfillment of service level agreements (SLAs). (See van Hoorn et al. (2009)).
Improvement and Innovation	
Continuous improvement	Based on lean principles of data-driven decision-making and elimination of waste, which lead to small incremental quality improvements that can have dramatic benefits and are hard for competitors to emulate. (See Chen et al. (2007), Fowler (1999), Järvinen et al. (1999), Krasner (1992)).
Continuous innovation	A sustainable process that is responsive to evolving market conditions and based on appropriate metrics across the entire lifecycle of planning, development and run-time operations. (See Cole (2001); Holmström Olsson et al. (2012), Ries (2011)).
Continuous experimentation	A software development approach based on experiments with stakeholders consisting of repeated Build-Measure-Learn cycles. (See Adams et al. (2013); Bosch (2012); Fagerholm et al. (2014); Steiber and Alänge (2013)).

financial year-end considerations, for example. This traditional planning model is effectively a batch formulation of the problem (Knight et al., 2001). When addressing an ongoing planning problem, time is divided into a number of planning horizons, each lasting a significant period of time. The only form of continuous planning is that which emerges from agile development approaches and is related to sprint iterations or at best, software releases, and is not widespread throughout the organization. However, just as agile seeks to enable software development to cope with frequent changes in the business environment, the nature of the business environment also requires that planning activities be done more frequently to ensure alignment between the needs of the business context and software development (Lehtola et al., 2009), and also requires a tight integration between planning and execution (Knight et al., 2001). Given the current interest in autonomous systems, it is also interesting that Knight et al. (2001) identify continuous planning as a key prerequisite for delivering autonomous systems.

In the traditional planning model, a failure in the plan may require another cycle of planning activity before it is resolved, but the typical cadence of annual once-per-year planning is certainly not adequate. Continuous planning may be defined as a holistic endeavor involving multiple stakeholders from business and software functions whereby plans are dynamic open-ended artifacts that evolve in response to changes in the business environment, and thus involve a tighter integration between planning and execution. In addition to iteration and release planning, product and portfolio planning activities would also be conducted (Ruhe, 2010). The relationship between planning at the individual project and the portfolio level is also an important issue here, with a need for flow between the project and portfolio level. For example, it is possible to you achieve success at the individual project level achieving close cooperation with the customer (with appropriate continuous feedback and learning). However, while the project may be a success, at the project portfolio level, there can still be overall failure. A portfolio approach to project management is consistent with the holistic approach adopted in this paper. A portfolio approach is also more likely to lead to success at the overall organizational level (Young and Conboy, 2013), but it is still the case that more attention is paid to individual project management rather than a portfolio approach (Gale, 2007). Further discussion of portfolio management is beyond the scope of this paper but it is an important research area.

4.1.2. *Continuous budgeting*

A budget is *"the formal expression of plans, goals, and objectives of management that covers all aspects of operations for a designated time period"* (Shim and Siegel, 2008, p.1). A budget helps management of an organization to evaluate plans and actions before implementing them. Traditionally, budgeting is an annual event or even a multi-year event; Shim and Siegel (2008) describe an annual event as 'short term,' and a long-term to be three years or more. However, such a long budgeting interval makes an organization rigid in terms of its decision making and planning capabilities. To address this, the Beyond Budgeting (Bogsnes, 2008) model has been proposed, which suggests that budgeting becomes a continuous activity also, to better facilitate changes during the year (see Section 2.3). Frow et al. (2010) introduced the concept of "continuous budgeting," and reported that a more flexible approach to budgeting encourages managers to make operational decisions at their own discretion as they encounter unexpected events which could not have been foreseen in a master budget plan. Bogsnes (2008) introduced the concept of 'ambition to action' which makes the point that activities should not be simply be mandated in a one-way fashion by management. Rather, teams must choose their own actions, but these must be aligned with business strategy. Thus, it links well with the BizDev concept outlined above. Bogsnes suggested that these actions should be open and transparent, and teams should operate with a degree of self-regulation in this regard, and not be constrained to an annual cycle.

4.2. *Development*

The Development phase, in our conception, comprises the main software development activities of analysis, design, coding and verification/testing. The following Continuous * activities are considered in this phase: continuous integration (incorporating continuous deployment/release, continuous delivery, continuous verification/testing). In recognition of the increasing focus on security and regulatory compliance, we also consider continuous compliance and continuous security activities in this phase.

4.2.1. *Continuous integration and related constituent activities*

Continuous integration is the best known of the Continuous * family. This is clearly helped by the fact that continuous integration is an explicit practice identified in the very popular Extreme Programming (XP) method. One consequence of this popularity however, is that there is considerable variability in how the topic is defined and in what activities are considered to be part of continuous integration (Ståhl and Bosch, 2013). However, at heart, continuous integration may be defined as a process which is typically automatically triggered and comprises inter-connected steps such as compiling code, running unit and acceptance tests, validating code coverage, checking compliance with coding standards, and building deployment packages. While some form of automation is typical, the frequency of integration is also important in that it should be regular enough to ensure quick feedback to developers. Finally, continuous integration failures are important events which may have a number of ceremonies and highly visible artifacts to help ensure that problems leading to these failures are prioritized for solution as quickly as possible by whoever is deemed responsible.

Continuous integration has increased in importance due to the benefits that have been associated with it (Ståhl and Bosch, 2013). These benefits include improved release frequency and predictability, increased developer productivity, and improved communication.

Continuous integration requires a link between development and operations and is thus very relevant to the DevOps phenomenon (Debois, 2009). Within continuous integration, a number of further modes of continuous activities can be identified, namely continuous deployment and continuous delivery (Humble, 2010; Lacoste, 2009). These concepts are related in that continuous delivery is a prerequisite for continuous deployment, but the reverse is not necessarily the case. That is, continuous deployment refers to releasing valid software builds to users automatically, whereas continuous delivery refers to the *ability* to deploy the software to some environment, but not automatically deploying to customers.

Neely and Stolt (2013) describe the experience of an organization that adopted continuous delivery. The organization implemented a number of lean principles, such as implementing a Kanban system (migrating from Scrum), documenting their development process (value stream mapping), and automation whenever possible. The transformation of continuous delivery cannot be limited to the software development team, but should also consider other functions, in particular Sales and Marketing. This suggests that an end-to-end consideration of the software development lifecycle is important, which is also a characteristic of Lean Thinking. Neely and Stolt also reported that continuous monitoring (through tests, gates and checks) is important (Poka Yoke, see Section 3.3) and the organization used a number of tools to monitor the state of the system.

By moving from time-based releases (e.g., Sprint-based) to continuous delivery of software, the number of reported defects (e.g., by customers) is likely to level out (*heijunka*, see Section 3.2).

4.2.2. *Continuous verification and continuous testing*

Given the extent to which various forms of testing are a key component of continuous integration, the topics of continuous verification and continuous testing are also included here. The traditional waterfall approach leads to a tendency to consider verification and quality as separate activities, to be considered only after requirements, design and coding are completed (Chang et al., 1997). Agile and iterative approaches have introduced prototyping which is used for earlier verification of requirements. Continuous verification seeks to employ verification activities including formal methods and inspection throughout the development process rather than relying on a testing phase towards the end of development. Chang et al. (1997) presented a case study of the development of a mission-critical weapons system that followed a waterfall lifecycle augmented with the concept of 'continuous verification.' Each phase (requirements analysis, high level design, detailed design, code, unit testing, integration testing) was augmented by adding an inspection or verification phase that could include prototyping. Chang et al. reported that only 3.8% of the total development time was needed for testing phases, which they attributed to the additional time spent on inspection and verification activities, which represented 21% of the total time. While verification activities took a significant amount of time, they were found to be very effective in achieving quality.

Inspections based on pre-defined checklists were found to be more effective than those performed without a checklist. This is clearly a form of task standardization (*Poka Yoke*, see Section 3.3).

Continuous testing seeks to integrate testing activities as closely as possible with coding. Similar to continuous integration, there are potential benefits to this (Muslu et al., 2013). Firstly, errors can be fixed quickly while the context is fresh in the developers' minds and before these errors lead to knock-on problems. This is a clear instance of one of the types of waste, namely that of motion (switching tasks; see Table 2). Also, the underlying root causes that led to the problems may be identified and eliminated. Furthermore, there is usually some level of automation of the testing process and a prioritization of test cases (Marijan et al., 2013). Saff and Ernst (2003) introduced the concept of 'continuous testing,' and argued that continuous testing can reduce wasted development time. Their experiment showed that continuous testing can help to reduce overall development time by as much as 15%, suggesting that continuous testing can be an effective tool to reduce one of the types of wastes, namely that of waiting time.

4.2.3. Continuous compliance and continuous security

Agile methods were initially seen as suited to small projects with co-located developers in non-safety critical contexts (Ambler, 2001; Boehm, 2002). However, over the past decade or so agile methods have been successfully applied on large projects with distributed developers (Fitzgerald et al., 2006), and in recent times, the final frontier, that of applying agile methods on safety critical systems is being addressed. Fitzgerald et al. (2013) discuss the tailoring of the Scrum method for a regulated environment, *R-Scrum* (i.e., Regulated Scrum) as it is termed. In keeping with the move from a waterfall approach to an agile approach comprising three-week sprints (a radical transformation, or *kaikaku*), a mode of continuous compliance was achieved. That is, rather than compliance being ensured on an annual basis in a single frenetic activity, new ceremonies, roles and artifacts were added to R-Scrum to allow compliance to be assessed at the end of each sprint. Non-conformance issues were fed back to sprint planning after each sprint, and this led to very efficient organizational learning whereby non-conformance issues tended not to reappear. As a result, the compliance assurance at final release time was more of a formality given that the issues had been ironed out during the various sprints.

Continuous security seeks to prioritize security as a key concern throughout all phases of the development lifecycle and even post deployment (Merkow and Raghavan, 2011). As a non-functional requirement, security is often relegated to a lower priority even unintentionally. Continuous security also seeks to implement a smart and lightweight approach to identifying vulnerabilities.

4.2.4. Continuous evolution

Software evolution has a long history (e.g., Lehman (1980); Swanson (1976)), and as already mentioned, given the vast amount of legacy software already developed, and the ready availability of open source software, much software development in practice involves evolving this software rather than developing from scratch. This has challenges in terms of the architecture not supporting the new functionality which may be required. This is exacerbated by the fact that the original developers are no longer available, but the software may be mission-critical and it is vital that it be evolved to support the organization. A system's maintainability depends on its architecture, which is formed through a set of initial design decisions that were made during its creation (Jansen and Bosch, 2005). Some of the assumptions underpinning these decisions may no longer hold due to changes in the context or environment in which the system operates, or the architecture may not facilitate certain changes. In recent times, increased attention has been paid to this topic using terms such as architectural decay and technical debt. When an architecture is unsuited to facilitating new requirements but 'shortcuts' are taken nevertheless, technical debt is incurred.

Del Rosso (2009) discussed how architecture evaluation methods (such as ATAM) can be used to gauge the evolvability of a software architecture. As systems evolve, an architecture may start to decay which results in a degenerated architecture (Riaz et al., 2009). Riaz et al. (2009) investigated architectural decay, identifying factors that contribute to this, and potential measures to mitigate such decay.

Software evolution is not only dependent on the software product's inherent changeability (which is greatly affected by architectural decisions as described above), but also on the expertise of developers (Rajlich and Bennett, 2000).

4.3. Operations

4.3.1. Continuous use

While much emphasis has been placed on the initial adoption of software systems, much less attention has been devoted to the continuing use of these systems (Gebauer et al., 2013). That is, continuous use cannot be automatically assumed for any given customer after the initial decision to use the software. However, continuous use is necessary given that the economic payoff from systems comes from continued use rather than initial adoption (Bhattacherjee, 2001). It is estimated that it costs ten times as much to acquire a new customer than retain and existing one (Daly, 2002, p.85).

Gebauer et al. (2013) point out that the models which are used to study initial adoption (e.g., the Technology Acceptance Model (TAM) (Davis et al., 1989) and Unified Theory of Acceptance and Use of Technology (UTAUT (Venkatesh et al., 2003)) are not necessarily suited to study continuous use as they do not consider variables such as automatic and unconscious use, or habitual characteristics. These have been found to be important in actual continuous use (Ortiz de Guinea and Markus, 2009). Also, the theoretical concepts underpinning these models were derived in an era where the stakeholders behaved differently. In the past, consumers of technology were *digital immigrants* rather than *digital natives* who have known technology all their lives (Vodanovich et al., 2010). The latter are motivated very differently and have different attitudes to software use. Digital immigrants are less comfortable with technology, need to be encouraged and trained in its use, and are less likely to experiment with new technology. Digital natives, on the other hand, are proactive in seeking out new technology and are far more likely to switch if they perceive a benefit.

A final shortcoming is that many studies consider *intention* to continue using a system rather than the *actual* continuous use. The latter requires longitudinal studies and ideally objective measures rather than just the self-reporting measures which are typical when personal intention is being assessed. The trend towards rapid experimentation and split A/B testing (see Section 2.4) with users to assess acceptance of various feature sets is also relevant to the continuous use category.

4.3.2. Continuous trust

Drawing on Hoehle et al. (2012) and Pavlou and Fygenson (2006), we define continuous trust as trust developed over time as a result of interactions based on the belief that a vendor will act cooperatively to fulfill customer expectations without exploiting their vulnerabilities. Continuous use is strongly dependent on continuous trust. Also, just as the initial adoption scenario is quite different to continued use, the relationship between initial trust and continuous trust is a complex one. Hoehle et al. (2012) suggest that initial trust is more important in circumstances that occur in a single transaction, such as buying a car. However, in contexts where activities are transacted over an extended period of time with remote providers, such as cloud services, continuous trust is critical. Continuous trust evolves over time and even if initially high, it is constantly being recalculated by users, and can be eroded due to user experience, for instance, with security or privacy concerns. Interestingly, even if nothing changes in a software product or service, trust can be eroded solely by changes in the external environment, for instance, by media reports on security or privacy vulnerabilities. Given the extent to which continuous use is dependent on continuous trust, ensuring the latter is clearly critical.

4.3.3. Continuous run-time monitoring

The historical boundary between design-time and run-time research in software engineering is blurring due to increased dynamic adaptation at run-time (Baresi and Ghezzi, 2010). This is especially significant in the context of cloud services which involve continuously running software services. Runtime behaviors of all kinds, including adaptations, must be predictable and bounded to ensure safety properties are satisfied and end-user expectations are met, hence linking to continuous security. Van Hoorn et al. (2009) suggest continuous monitoring may enable early detection of quality-of-service problems, such as performance degradation, and also the fulfillment of service level agreements (SLAs).

4.4. Continuous improvement, continuous innovation and continuous experimentation

Software process improvement has a long history in the software field and is expressed in several standards such as Capability Maturity Model Integration (CMMI), ISO 9000, and Software Process Improvement and Capability dEtermination (SPICE). Also, the Scrum notion of Sprint Retrospectives is intended to facilitate continuous improvement. The topic continues to evolve with emphasis on human factors (Korsaa et al., 2013), distributed teams (O'Connor and Basri, 2012), and small companies (Valtierra et al., 2013).

Continuous improvement, or *kaizen*, is a key tenet of Lean Thinking (see Section 3.4). In software development terms, continuous product improvement manifests itself in the refactoring concept, a key practice in Extreme Programming (Fowler, 1999). Continuous process improvement has also been a prominent theme in the software arena (Chen et al., 2007; Järvinen et al., 1999; Krasner, 1992). These initiatives are important contributors to software quality and are very much based on the lean principles of using data to drive decision-making and eliminate waste. While continuous improvement initiatives are typically incremental and may appear small, Tushman et al. (1997) argued that continuous improvement leverages organizational tacit knowledge and is thus difficult for other organizations to easily emulate. However, continuous improvement activities are essentially *reactive* initiatives and eventually are limited in the extent to which they can add customer value. Hence, there has been a move to place greater emphasis on innovation as a more proactive strategy.

Innovation in a business context refers to the process whereby new ideas are transformed to create business value for customers, i.e. invention plus exploitation. Innovation has been one of the most widely used buzzwords in recent times, especially in the context of open innovation (Chesbrough, 2003). Also, the theme of continuous innovation has emerged, most notably in the software domain through the concept of Lean Startups (see Section 2.4). An early activity in the continuous innovation space was that of beta testing, which became a widespread practice in the software industry, where it was used to elicit early customer feedback prior to formal release of software products (Cole, 2001). The concept has matured considerably over the years, and now techniques such as A/B testing are widely used where features such as text, layouts, images and colors are manipulated systematically and customer reaction is monitored (Holmström Olsson et al., 2012). This can be an effective way to identify value-adding features.

Interestingly, planning has been identified as a prerequisite for continuous innovation, in that inadequate planning and strategic alignment at the front-end of the development process is a major cause of failure for consumer products companies. Continuous innovation seeks to establish a sustainable process that is responsive to evolving market conditions and based on appropriate metrics across the entire lifecycle of planning, development and run-time operations.

While some have seen continuous improvement and innovation as incompatible, it has been argued that continuous improvement can be a useful base upon which to achieve continuous innovation (Cole, 2001). As a consequence, we position continuous innovation and continuous improvement as the foundation upon which the other Continuous * activities can be grounded (See Fig. 3).

5. Discussion and conclusion

The current identification of Continuous Software Engineering as an important topic is presaged by a number of emergent phenomena which at their core reflect the necessity of a focus on continuous activities. Here, we identify the sub-topics that underpin the emergence of Continuous *.

*5.1. The emergence of Continuous **

The numerous 'continuous' phenomena discussed in the previous section, which we have categorized under the Continuous * umbrella term, clearly indicate a common trend, namely the increasing need to establish an end-to-end flow between customer demand and the fast delivery of a product or service. This observation is, of course, not entirely new; within the software development context this has already been established in the Agile Manifesto. However, the 'big picture' by which this might be achieved goes beyond the Agile Manifesto and surfaces a more holistic set of Continuous * activities as discussed here, such as Enterprise Agile and Beyond Budgeting. It also requires tight integration and flow between the various continuous activities. The age-old disconnect between the business strategy and technical development components is recognized in the BizDev concept which seeks to tighten this integration. The Scrum role of Product Owner, who acts as a surrogate for the business customer, is a recognition of the need for this connection by the agile methods community. However, we contend that the Product Owner concept does not go far enough. The central problem is captured in the apocryphal, but very apt quote from the business manager, who described the key problem with his company's flagship software product, as being that he knew that half of the features being offered were not used by any customers, but the trouble was that he did not know *which* half. What is needed is a *feature analytics* capability whereby a business manager can systematically identify a feature or set of features and quickly experiment with delivery of those features, the cost of their delivery, the usage by customers, and the actual return on investment from these features, similar to what the Lean Startup approach aims to achieve (see Section 2.4). This is necessary if the "Stairway to Heaven" as suggested by Holmström Olsson et al. (2012) is to be realized.

5.2. The shift left strategy

Another emergent concept that is gaining in popularity is the "Shift Left" strategy. In the agile world, this has been characterized as the need to address the *technical debt* that accrues from not addressing potentially problematic issues at the time issues are first encountered, but rather postponing these to be dealt with at a subsequent stage (and moving them to the 'right' into the software development life cycle). While this may lead to a perception of greater speed initially, in the long run it is very detrimental. However, economic tradeoffs may prohibit the investments needed to remove technical debt. Further research that can help to calculate break-even points in the paying off of technical debt is needed to assist industry in this matter.

It is worth noting that the need to shift left has a long history in the software field. The initial concept that was proposed in what became known as the Structured Approach (Fitzgerald, 1996) was Structured Programming, on the basis that bad programming practices led to difficulties. While this was undoubtedly true, resolving bad programming practices through applying Structured Programming concepts had limited benefit as it mainly revealed the problems earlier in the design process. This in turn led to the identification of Structured Design concepts. Again, this has limited benefit as efficient design of the wrong solution was not worthwhile when it became apparent that the core problem was the incorrect analysis of the actual business requirements in the first place. This series of shift left stages then culminated in the Structured Analysis concept.

The shift left strategy of identifying problems as early as possible resonates very well with lean concepts identified in Section 3, such as root cause analysis, jidoka, poka yoke and the andon concept.

5.3. Challenges for continuous software engineering

Overall, the implementation of Continuous * surfaces a number of high level challenges for practice. We summarize these high level challenges here.

5.3.1. Win the war, not the battles

When asked the question *"What is the Toyota Production System [...] most people (80 percent) will echo the view of the average customer and say: 'It's a kanban system.' "* (Shingo, 1989, p. 67). Lean experts would argue that kanban is a *technique*, a system for eliminating waste. In a similar fashion, we argue that true continuous software engineering is more than adopting continuous delivery and continuous deployment. These are merely techniques, but the ultimate goal is to take a holistic view of a software production entity, whether this be a single software organization or an ecosystem where different organizations together deliver a final product. Rather than focusing on winning these battles (i.e. successfully implementing such techniques), the holistic view that we advocate is that of winning the war; in this case, to focus on pursuing the Continuous * agenda and establish a holistic view from customer to delivery.

5.3.2. The importance of culture and context

The success of continuous integration and deployment is often viewed in terms of the multiple product release per day, as achieved by Google or Facebook, for example (Feitelson et al., 2013). However, the functionality being released in these examples does not often require major architectural or design changes to enable deployment; the best example of this being Google's A/B experiments with 41 different shades of blue for displaying hyperlinks, so as to determine with evidence which shade led to the greatest number of click-throughs. Similarly, Amazon reportedly deploys its software into production every 11.6 seconds (Jenkins, 2011).

By contrast, it is not obvious how to establish such a continuous process in a real business environment through the 'bleeding' in of real and significant new functionality to production systems. A lot of work needs to be done to understand the specifics of different development contexts. The importance of context becomes immediately clear if we take avionics software as an example, in that few people would be willing to fly in an airplane in which a new version of the software was being deployed every 11.6 seconds.

There are numerous dimensions in which contexts vary, for instance the business domain in which organizations operate. The telecommunication sector is an example that depends on legacy systems that may be much less amenable to a continuous software engineering approach. In such systems, rapid delivery of new functionality is a major challenge, as there may be dozens of different interacting systems that together deliver hundreds of different services to both internal and external customers. Documentation may be missing or outdated, thus relying on the tacit knowledge of hundreds of different engineers working in different departments. In such domains, technology may also prove to be less suitable for a continuous software engineering approach. The mere size of legacy systems consisting of hundreds of thousands of lines of COBOL code, for instance, represents a significant barrier to 'quick delivery' of new features and services. Likewise, rapid updates of embedded systems may also be highly challenging as these systems require cross-compilation and any failures may require significant effort to recover.

In these domains that have relied on automation for decades, there may also be a cultural tendency to assume that the status quo is the only possible way. Similar to what could be observed in some lean transformations, a disbelief that "this could work here" may result in considerable resistance to change within organizations. Womack and Jones have argued that a change agent is needed: *"a leader—someone who will take personal responsibility for change—is essential"* (Womack and Jones, 2003, p. 313). This cultural change may very well be the most significant barrier to change.

Another dimension is that of software sourcing; the use of outsourcing of components or the use of commercial off-the-shelf (COTS) components are very common approaches in numerous domains. Such dependency on software components produced elsewhere may introduce additional challenges when aiming for delivering new software releases frequently. Novel approaches such as opensourcing and crowdsourcing are emerging within the software engineering domain, which have consequences for aspects such as innovation and time-to-market (Ågerfalk et al., 2015). Again, software development is not unique in this respect; Toyota recognized the need to consider its parts suppliers, and convinced them to adopt lean manufacturing principles as well (Liker, 2004).

5.3.3. Misplaced focus on speed rather than continuity

There is a general trend towards a 'need for speed' (Tabib, 2013) that is, a need to deliver new features and defect fixes to the end-user as fast as possible, as suggested by project names such as 'Need 4 Speed'[5] and the International Workshop on Rapid and Continuous Software Engineering (RCoSE) (Tichy et al., 2014). However, in this paper we have argued the need for *continuity*. Ohno (1988) (p. 62), one of the founding fathers of the Toyota Production System, described the story of the tortoise and the hare:

> *"A slower but more consistent tortoise causes less waste and is much more desirable than the speedy hare who races ahead and then stops occasionally to doze."*

Ohno argued that the TPS can only be realized if all workers become tortoises, and referred here to the term 'high-performance.' Ohno's point was that *"speed is meaningless without continuity."* Likewise for software engineering, we argue that achieving flow and continuity is much more important in first instance than speed.

5.4. The need for discontinuous software engineering

Russell Ackoff, the veteran organizational theorist, concluded that *"continuous improvement isn't nearly as important as discontinuous improvement"* (Ackoff, 1994). Hamel captured the issue more vividly using the example of human species evolution: *"if six-sigma ruled, we would still be slime"* (Hamel, 2012). The point being made is that creativity and innovation require discontinuous thinking—in some cases an abrupt, discontinuous change is needed rather than a gradual change; in lean terms: kaikaku, not kaizen. Undoubtedly, additional value is often delivered through a series of incremental improvements, a path that can be compared to 'kaizen.' However, there are cases where this path is no longer sustainable and more significant, abrupt changes are needed, very much comparable to the 'kaikaku' phenomenon of radical change. Thus we see that while Continuous * is a necessary evolution in the software development field, it is not the only way that progress can be achieved. New development models, such as that offered by the open source software phenomenon, for example, would not have emerged from a continuous paradigm.

5.5. Achieving the Continuous * agenda

In order to achieve the envisaged Continuous * agenda, we believe that the concepts presented in the holistic overview (see Fig. 3) need to be further operationalized and studied. Table 5 presents some of the highlights of the research agenda that we envisage. Research activities focus on the three types of stakeholders that we mentioned

[5] http://www.digile.fi/N4S

Table 5
Research agenda for continuous software engineering.

Business strategy	Development	Operations
• Feature analytics: What types of information does executive management need for useful planning and evaluation of features?	• How can the continuous evolution and maintenance of software systems be facilitated?	• Usage: How can a continued use of product features be predicted?
• Continuous planning: How can management define a project portfolio in alignment with strategy and implement continuous planning while not changing course too frequently which would hamper product development?	• What architectural solutions offer the highest degree of flexibility to facilitate continuous evolution?	• Trust: What factors help to sustain customers' continuous trust in a product?
• Bizdev: How can the mismatch in expectations between sales/marketing on the one hand and development on the other hand be addressed?	• What factors influence the decision to take a radical (discontinuous) approach to re-engineering a product? • How can hardware and software co-development follow a continuous software engineering approach?	• DevOps: What are the key barriers between development and operations and how can these be removed?

before: executive management (business strategy), development, and operations; for each type of stakeholder, we have identified a number of issues that need further research.

Some concepts have been studied in greater detail than others; for instance, Ståhl and Bosch (2013) studied continuous integration and identified variation across different implementations. In a similar way we envisage that other concepts, including continuous delivery, but also continuous trust and continuous use, will manifest in different ways as well. Documenting these variations is one important avenue for future research.

The BizDev concept that we advocate in this paper has thus far been a missing link, but we believe it is essential to connect the business and development functions within organizations. There are fundamental mismatches between expectations and planning strategies of business and sales managers on the one hand, and the capability to deliver functionality in a timely fashion on the other hand. For instance, a very common scenario is that companies base their products and services on open source components; this introduces a dependency on such open source communities, which complicates the ability to make a precise planning for product release dates. Part of realizing the Continuous * agenda thus involves the development of appropriate metrics that can help in predicting how fast interactions with external communities and other players in an ecosystem can be achieved. Furthermore, metrics will also be important in realizing the continuous experimentation concept, including the suggested 'feature analytics,' which will facilitate measuring the value-add of specific features. Development of appropriate tools and infrastructure to support this will also be an important activity in achieving the Continuous * agenda.

5.6. Conclusion

Delivering the Continuous * agenda highlights a number of significant challenges which need to be overcome if the concept is to be successful. This work attempts to provide a roadmap of the overall territory, an important step in its own right, since there is much confusion as terms are used interchangeably and synonymously without rigorous definition, similar to early research on agile methods (Conboy and Fitzgerald, 2004). The need for the Continuous * concept is evident when one considers the emergence of phenomena such as Enterprise Agile, Beyond Budgeting, DevOps, Lean Startups and many other concepts from Lean Thinking in general. These are all symptomatic of the need for a holistic and integrated approach across all the activities that comprise software development.

Acknowledgments

We thank the anonymous reviewers for constructive feedback to an earlier version of this paper. Thanks to Garry Lohan for his insights on Beyond Budgeting. This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 to Lero—the Irish Software Research Centre (www.lero.ie), Enterprise Ireland grant IR/2013/0021 to the ITEA2-SCALARE project, the Irish Research Council under the New Foundations programme, and the Royal Irish Academy (www.ria.ie).

References

Ackoff, R., 1994. The learning and legacy of Dr. W. Edwards Deming. Available at: http://www.youtube.com/watch?v=OqEeIG8aPPk (accessed on July 8, 2015).
Adams, R.J., Evans, B., Brandt, J., 2013. Creating small products at a big company: Adobe's "pipeline" innovation process. In: Proceedings of CHI'13 Extended Abstracts on Human Factors in Computing Systems. ACM, pp. 2331–2332.
Ågerfalk, P.J., Fitzgerald, B., Stol, K., 2015. Not so shore anymore: the new imperatives when sourcing in the age of open. In: Proceedings of the 23rd European Conference on Information Systems.
Ambler, S., 2001. When does(n't) agile modeling make sense. http://www.agilemodeling.com/essays/whenDoesAMWork.htm (accessed on July 8, 2015).
Baresi, L., Ghezzi, C., 2010. The disappearing boundary between development-time and run-time. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER'10). ACM, pp. 17–22.
Beck, K., 2000. Extreme Programming Explained: Embrace Change. Addison-Wesley.
Bernhart, M., Strobl, S., Mauczka, A., Grechenig, T., 2012. Applying continuous code reviews in airport operations software. In: Proceedings of the 12th International Conference on Quality Software, pp. 214–219.
Bhattacherjee, A., 2001. Understanding information systems continuance: an expectation–confirmation model. MIS Q. 25 (3), 351–370.
Boehm, B., 2002. Get ready for agile methods, with care. IEEE Comput. 35 (1), 64–69.
Bogsnes, B., 2008. Implementing Beyond Budgeting: Unlocking the Performance Potential. Wiley.
Bogsnes, B., 2009. Keynote: beyond budgeting in a lean and agile world. In: Abrahamsson, P., Marchesi, M., Maurer, F. (Eds.), Proceedings of XP 2009, LNBIP 31. Springer, pp. 5–7.
Bosch, J., 2012. Building products as innovation experiment systems. Proceedings of the Third International Conference on Software Business (ICSOB), LNBIP. Springer, pp. 27–39.
Bosch, J., Holmström Olsson, H., Björk, J., Ljungblad, J., 2013. The early stage software startup development model: a framework for operationalizing lean principles in software startups. In: Fitzgerald, B., Conboy, K., Power, K., Valerdi, R., Morgan, L., Stol, K. (Eds.), Proceedings of the 4th International Conference on Lean Enterprise Software and Systems, pp. 1–15.
CA Technologies, 2012. The innovation imperative: Why it needs to lead now. http://www.ca.com/us/~/media/Files/Presentations/the-innovation-imperative-external-presentation-final.pdf (accessed on July 8, 2015).
Chang, T.-F., Danylyzsn, A., Norimatsu, S., Rivera, J., Shepard, D., Lattanze, A., Tomayko, J., 1997. "Continuous verification" in mission critical software development. In: Proceedings of the 30th Hawaii International Conference on System Sciences, pp. 273–284.
Chen, X., Sorenson, P., Willson, J., 2007. Continuous SPA: continuous assessing and monitoring software process. In: Proceedings of the IEEE Congress on Services (SERVICES), pp. 153–158.
Chesbrough, H., 2003. Open Innovation: The New Imperative for creating and Profiting from Technology. Harvard Business School Press.
Claps, G.G., Svensson, R.B., Aurum, A., 2015. On the journey to continuous deployment: technical and social challenges along the way. Inf. Softw. Technol. 57 (1), 21–31.
Cole, R., 2001. From continuous improvement to continuous innovation. Qual. Manage. J. 8 (4), 7–21.
Conboy, K., Fitzgerald, B., 2004. Towards a conceptual framework of agile methods. Extreme Programming and Agile Methods-XP/Agile Universe 2004, LNCS, vol. 3134. Springer, pp. 105–116.
Cordeiro, L., Fischer, B., Marques-Silva, J., 2010. Continuous verification of large embedded software using smt-based bounded model checking. In: Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems, pp. 160–169.
Daly, J.L., 2002. Pricing for Profitability: Activity-Based Pricing for Competitive Advantage. Wiley Press.

Davis, F., Bagozzi, R., Warshaw, P., 1989. User acceptance of computer technology: a comparison of two theoretical models. Management Science 35 (8), 982–1003.
Debois, P., 2009. DevOps Days Ghent. http://www.devopsdays.org/events/2009-ghent/ (accessed on July 8, 2015).
Debois, P., 2011. Devops: a software revolution in the making? Cutter IT J. 24 (8).
Del Rosso, C., 2009. Continuous evolution through software architecture evaluation: a case study. J. Softw. Maint. Evol.: Res. Pract. 18 (5), 351–383.
Earl, M., Hopwood, A., 1980. From management information to information management. Trends in Information Systems. North-Holland Publishing Co. Amsterdam, The Netherlands, pp. 315–325.
Fagerholm, F., Guinea, A.S., Mäenpää, H., Münch, J., 2014. Building blocks for continuous experimentation. In: Tichy, M., Bosch, J., Goedicke, M., Larsson, M. (Eds.), Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering (RCoSE). ACM, pp. 26–35.
Feitelson, D., Frachtenberg, E., Beck, K., 2013. Development and deployment at facebook. IEEE Internet Comput. 17 (4), 8–17.
Feller, J., Fitzgerald, B., Hissam, S., Lakhani, K., 2005. Perspectives on Free and Open Source Software. MIT Press.
Fitz, T., 2009. Continuous deployment at IMVU: doing the impossible fifty times a day. http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/ (accessed on July 8, 2015).
Fitzgerald, B., 1996. Formalized systems development methodologies: a critical perspective. Inf. Syst. J. 6 (1), 3–23.
Fitzgerald, B., Hartnett, G., Conboy, K., 2006. Customising agile methods to software practices at Intel Shannon. Eur. J. Inf. Syst. 15 (2), 197–210.
Fitzgerald, B., Musiał, M., Stol, K., 2014. Evidence-based decision making in lean software project management. In: Proceedings of the 36th International Conference on Software Engineering (ICSE-SEIP). ACM, pp. 93–102.
Fitzgerald, B., Stol, K., 2014. Continuous software engineering and beyond: Trends and challenges. In: Proceedings of the First International Workshop on Rapid and Continuous Software Engineering (RCoSE). ACM, pp. 1–9.
Fitzgerald, B., Stol, K., O'Sullivan, R., O'Brien, D., 2013. Scaling agile methods to regulated environments: an industry case study. In: Proceedings of the 35th International Conference on Software Engineering (ICSE-SEIP), pp. 863–872.
Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley.
Frow, N., Marginson, D., Ogden, S., 2010. "Continuous" budgeting: reconciling budget flexibility with budgetary control. Account., Organ. Soc. 35 (4), 444–461.
Gale, A., 2007, The Wiley Guide to Project Organisation and Project Management Competencies. In: Morris, P., Pinto, J. (Eds.). Wiley & Sons, pp. 143–167.
Gebauer, L., Sollner, M., Leimeister, J., 2013. Towards understanding the formation of continuous IT use. In: Proceedings of the 34th International Conference on Information Systems.
Gefen, D., Karahanna, E., Straub, D., 2003. Trust and tam in online shopping: an integrated model. MIS Q. 27 (1), 51–90.
Hamel, G., 2012. What Matters Now. Jossey Bass.
Hoehle, H., Huff, S., Goode, S., 2012. The role of continuous trust in information systems continuance. J. Comput. Inf. Syst. 52 (4), 1–9.
Holmström Olsson, H., Alahyari, H., Bosch, J., 2012. Climbing the "stairway to heaven": a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: Proceedings of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 392–399.
Hope, J., Fraser, R., 2003. Beyond budgeting: how managers can break free from the annual performance trap. In: Proceedings of the Harvard Business Review.
Humble, J., 2010. Continuous delivery vs continuous deployment. http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/ (accessed on July 8, 2015).
Humble, J., Farley, D., 2010. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley.
Humble, J., Molesky, J., 2011. Why enterprises must adopt devops to enable continuous delivery. Cutter IT J. 24 (8), 6–12.
Jansen, A., Bosch, J., 2005. Software architecture as a set of architectural design decisions. In: Proceedings of teh 5th Working IEEE/IFIP Conference on Software Architecture, pp. 109–120.
Järvinen, J., Hamann, D., van Solingen, R., 1999. On integrating assessment and measurement: towards continuous assessment of software engineering processes. In: 6th International Software Metrics Symposium.
Jenkins, J., 2011. Velocity culture. In: Talk at Velocity 2011 Conference.
Kettunen, P., Laanti, M., 2008. Combining agile software projects and large-scale organizational agility. Softw. Process: Improv. Pract. 13 (2), 183–193.
Khurum, M., Petersen, K., Gorschek, T., 2014. Extending value stream mapping through waste definition beyond customer perspective. J. Softw.: Evol. Process 26 (12), 1074–1105.
Kim, S., Park, S., Yun, J., Lee, Y., 2008. Automated continuous integration of component-based software: an industrial experience. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 423–426.
Knight, R., Rabideau, G., Chien, S., Engelhardt, B., Sherwood, R., 2001. Casper: space exploration through continuous planning. IEEE Intell. Syst. 16 (5), 70–75.
Korsaa, M., Johansen, J., Schweigert, T., Vohwinkel, D., Messnarz, R., Nevalainen, R., Biro, M., 2013. The people aspects in modern process improvement management approaches. J. Softw.: Evol. Process 25 (4), 381–391.
Krafcik, J., 1988. Triumph of the lean production system. MIT Sloan Manage. Rev. 30 (1), 41–52.
Krasner, H., 1992. The ASPIRE approach to continuous software process improvement. In: Proceedings of the 2nd International Conference on Systems Integration.
Kurapati, N., Manyam, V.S.C., Petersen, K., 2012. Agile software development practice adoption survey. In: Proceedings of the XP 2012, LNBIP 111, pp. 16–30.
Lacoste, F., 2009. Killing the gatekeeper: introducing a continuous integration system. In: Proceedings of the Agile Conference, pp. 387–392.
Larman, C., Basili, V.R., 2003. Iterative and incremental developments: a brief history. IEEE Comput. 36 (6), 47–56.
Leffingwell, D., 2007. Scaling Software Agility: Best Practices for Large Enterprises. Addison–Wesley.
Lehman, M., 1980. On understanding laws, evolution, and conservation in the large-program life cycle. J. Syst. Softw. 1, 213–221.
Lehtola, L., Kauppinen, M., Vähäniitty, J., Komssi, M., 2009. Linking business and requirements engineering: is solution planning a missing activity in software product companies? Requir. Eng. 14 (2), 113–128.
Liker, J.K., 2004. The Toyota Way. McGraw Hill.
Lohan, G., 2013. A brief history of budgeting: reflections on beyond budgeting, its link to performance management and its appropriateness for software development. In: Fitzgerald, B., Conboy, K., Power, K., Valerdi, R., Morgan, L., Stol, K. (Eds.), Proceedings of the 4th International Conference on Lean Enterprise Software and Systems, LNBIP, 167. Springer, pp. 81–105.
Maglyas, A., Nikula, U., Smolander, K., 2012. Lean solutions to software product management problems. IEEE Softw. 29 (5), 40–46.
Marijan, D., Gotlieb, A., Sen, S., 2013. Test case prioritization for continuous regression testing: an industrial case study. In: Proceedings of the 29th IEEE International Conference on Software Maintenance, pp. 540–543.
Maurya, A., 2012. Running Lean, 2nd edition O'Reilly Media Inc.
McHugh, M., Mc Caffery, F., Fitzgerald, B., Stol, K., Casey, V., Coady, G., 2013. Balancing agility and discipline in a medical device software organization. Proceedings of the 13th International SPICE Conference, Software Process Improvement and Capability Determination, Communications in Computer and Information Science, 349. Springer, pp. 199–210.
Merkow, M., Raghavan, L., 2011. An ecosystem for continuously secure application software. CrossTalk March/April.
Michlmayr, M., Fitzgerald, B., Stol, K., 2015. Why and how should open source projects adopt time-based releases? IEEE Softw. 32 (2), 55–63.
Mintzberg, H., 1994. The Rise and Fall of Strategy Planning. Prentice Hall.
Morgan, J., Liker, J., 2006. The Toyota Product Development System. Productivity Press.
Mujtaba, S., Feldt, R., Petersen, K., 2010. Waste and lead time reduction in a software product customization process with value stream maps. In: Proceedings of the 21st Australian Software Engineering Conference.
Müller, H.A., Wong, K., Tilley, S.R., 1994. Understanding software systems using reverse engineering technology. In: The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS), pp. 41–48.
Muslu, K., Brun, Y., Meliou, A., 2013. Data debugging with continuous testing. In: Proceedings of the 2013 9th Joint ESEC/FSE Meeting on Foundations of Software Engineering, pp. 631–634.
Myers, K., 1999. CPEF: a continuous planning and execution framework. AI Magazine 20 (4), 63–69.
Neely, S., Stolt, S., 2013. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In: Proceedings of the Agile Conference, pp. 121–128.
O'Connor, R., Basri, S., 2012. The effect of team dynamics on software development process improvement. Int. J. Human Cap. Inf. Technol. Prof. 3, 13–26.
Ohno, T., 1988. Toyota Production System: Beyond Large-Scale Production. CRC Press.
Ortiz de Guinea, A., Markus, M., 2009. Why break the habit of a lifetime? rethinking the roles of intention, habit, and emotion in continuing information technology use. MIS Q. 33 (3), 433–444.
Overby, E., Bharadwaj, A., Sambamurthy, V., 2005. A framework for enterprise agility and the enabling role of digital options, business agility and information technology diffusion. Business Agility and Information Technology Diffusion, IFIP International Federation for Information Processing, vol. 180. Springer, pp. 295–312.
Ozden, M., 1987. A dynamic planning technique for continuous activities under multiple resource constraints. Manage. Sci. 33 (10), 1333–1347.
Papatheocharous, E., Andreou, A.S., 2014. Empirical evidence and state of practice of software agile teams. J. Softw.: Evol. Process 26 (9), 855–866.
Pavlou, P., Fygenson, M., 2006. Understanding and predicting electronic commerce adoption: an extension of the theory of planned behavior. MIS Q. 30 (1), 115–143.
Petersen, K., 2011. Is lean agile and agile lean? a comparison between two software development paradigms. Modern Software Engineering Concepts and Practices: Advanced Approaches. IGI Global.
Petersen, K., Wohlin, C., 2010. The effect of moving from a plan-driven to an incremental software development approach with agile practices: An industrial case study. Empir. Softw. Eng. 15 (6), 654–693.
Rajlich, V.T., Bennett, K.H., 2000. A staged model for the software life cycle. Computer 33 (7), 66–71.
Rautiainen, K., Vuornos, L., Lassenius, C., 2003. An experience in combining flexibility and control in a small company?s software product development process. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering (ISESE), pp. 28–37.
Reifer, D., 2002. Making the Software Business Case. Addison Wesley.
Reifer, D.J., Maurer, F., Erdogmus, H., 2003. Scaling agile methods. IEEE Softw. 20 (4), 12–14.
Reinertsen, D.G., 2009. The Principles of Product Development Flow. Celeritas Publishing.
Riaz, M., Sulayman, M., Naqvi, H., 2009. Architectural decay during continuous software evolution and impact of 'design for change' on software architecture. In: Proceedings of the International Conference on Advanced Software Engineering and Its Applications, pp. 119–126.

Ries, E., 2011. The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses. Crown Business.

Rogers, R., 2004. Scaling continuous integration. Extreme Programming and Agile Processes in Software Engineering, LNCS, vol. 3092. Springer, pp. 68–76.

Royce, W.W., 1987. Managing the development of large software systems. In: Proceedings of the 9th international conference on Software Engineering, pp. 328–338.

Ruhe, G., 2010. Product Release Planning: Methods, Tools and Applications. CRC Press.

Saff, D., Ernst, M.D., 2003. Reducing wasted development time via continuous testing. In: Proceedings of the 14th International Symposium on Software Reliability Engineering, pp. 281–292.

Shim, J.K., Siegel, J.G., 2008. Budgeting Basics and Beyond, 3rd edition John Wiley & Sons, Inc.

Shingo, S., 1989. A Study of the Toyota Production System From an Industrial Engineering Viewpoint, revised Edition. Productivity Press.

Sommerville, I., 2007. Software Engineering. Pearson Educated Ltd.

Steiber, A., Alänge, S., 2013. A corporate system for continuous innovation: the case of google inc. Eur. J. Innov. Manage. 16 (2), 243–264.

Stolberg, S., 2009. Enabling agile testing through continuous integration. In: Proceedings of the Agile Conference.

Ståhl, D., Bosch, J., 2013. Modeling continuous integration practice differences in industry software development. J. Syst. Softw. 87 (1), 48–59.

Swanson, E.B., 1976. The dimensions of maintenance. In: Proceedings of the 2nd international conference on Software engineering, pp. 492–497.

Swartout, P., 2012. Continuous Delivery and DevOps: a QuickStart Guide. Packt Publishing.

Tabib, R., 2013. Need 4 speed: leverage new metrics to boost your velocity without compromising on quality. In: Proceedings AGILE 2013.

Tichy, M., Bosch, J., Goedicke, M., Larsson, M. (Eds.), 2014. Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering (RCoSE 2014) ACM.

Tushman, M., Anderson, P., O'Reilly, C., 1997. Technology cycles, innovation streams, and ambidextrous organizations: organizational renewal through innovation streams and strategic change. Managing Strategic Innovation and Change. Oxford University Press.

Valtierra, C., Muñoz, M., Mejia, J., 2013. Characterization of software processes improvement needs in SMEs. In: Proceedings of the International Conference on Mechatronics, Electronics and Automotive Engineering (ICMEAE), pp. 223–228.

van Hoorn, A., Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Frey, S., Kieselhorst, D., 2009. Continuous monitoring of software services: Design and application of the Kieker framework. Berich Nr. 0921, November 2009. Christian-Albrechts Universität zu Kiel.

Venkatesh, V., Morris, M., Davis, G., Davis, F., 2003. User acceptance of information technology: toward a unified view. MIS Q. 27 (3), 425–478.

Vodanovich, S., Sundaram, D., Myers, M., 2010. Digital natives and ubiquitous information systems. Inf. Syst. Res. 21 (4), 711–723.

Wang, X., Conboy, K., Cawley, O., 2012. "leagile" software development: an experience report analysis of the application of lean approaches in agile software development. J. Syst. Softw. 85 (6), 1287–1299.

Womack, J., Jones, D.T., 2003. Lean Thinking: Banish Waste and Create Wealth in Your Corporation. Productivity Press.

Young, M., Conboy, K., 2013. Contemporary project portfolio management. Int. J. Proj. Manag. 31 (8), 1089–1100.

Zhou, T., 2013. An empirical examination of continuance intention of mobile payment services. Decis. Support Syst. 54 (2), 1085–1091.

Prof. Brian Fitzgerald is Chief Scientist at Lero—the Irish Software Research Centre. He holds an endowed chair, the Frederick Krehbiel Chair in Innovation in Business and Technology at the University of Limerick. His research interests include open source software, inner source, crowdsourcing, and lean and agile methods. Contact him at bf@lero.ie.

Dr. Klaas-Jan Stol is a research fellow at Lero—the Irish Software Research Centre at the University of Limerick. His research interests include open source software, inner source, and agile and lean methods. Previously, he was a contributor to an open source project. Contact him at klaas-jan.stol@lero.ie.