# Continuous Software Engineering and Beyond: Trends and Challenges

Brian Fitzgerald
Lero—The Irish Software Engineering Research Centre
University of Limerick, Ireland
bf@ul.ie

Klaas-Jan Stol
Lero—The Irish Software Engineering Research Centre
University of Limerick, Ireland
klaas-jan.stol@lero.ie

## ABSTRACT

Throughout its short history, software development has been characterized by harmful disconnects between important activities e.g., 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, the *Enterprise Agile* concept has 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 function such as finance and HR. 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. Drawing on the lean concept of flow, we identify a number of continuous activities which are important for software development in today's context. These activities include continuous planning, continuous integration, continuous deployment, continuous delivery, continuous verification, continuous testing, continuous compliance,continuous security, continuous use, continuous trust, continuous run-time monitoring, continuous improvement (both process and product), all underpinned by continuous innovation. We use the umbrella term, "Continuous *" (continuous star) to identify this family of continuous activities.

## Categories and Subject Descriptors

D.2 [**Software Engineering**]: Management; K.6.3 [**Software Management**]: Software development, Software maintenance, Software process



*RCoSE'14*, June 3, 2014, Hyderabad, India
ACM 978-1-4503-2856-2/14/06
http://dx.doi.org/10.1145/2593812.2593813

## General Terms

Management, Theory

## Keywords

DevOps, BizDev, Continuous Star, continuous software engineering

## 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 [53]. 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 [18]. The pervasive adoption of agile methods [61] 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, typically 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 [3], and indeed the practice is highly compatible with the frequent iterations of software produced by agile approaches. Also, many open source toolsets are freely available to automate the continuous integration process.

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* and *Beyond Budgeting* [7] 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 function such as finance and HR [36, 48]. 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 [16]. Complementing this, we argue that the link between business strategy and software development ought to be continuously assessed and improved, *BizDev* being the term which we have coined for this process.

Similarly, we cannot assume that the process is complete once customers have initially adopted a software product. Digital natives, the term for those who have been born in the technology era [62] have high expectations of software and are not put off by high switching costs if 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 market conditions.

We believe that rather than focusing on agile methods *per se*, a useful concept for assessing continuous concepts in software development comes for the lean approach, namely that of 'flow' [50].

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, a school of thought that is called 'Lean Thinking' [63]. In recent years, there has been much interest in lean software development [13, 17, 21], however, so far there has been a somewhat narrow view on this topic by closely linking it to agile practices only.

In this paper we review a number of initiatives that are termed 'continuous.' We take a holistic approach and use the umbrella term 'Continuous *' ("Continuous star") to refer to these related practices and concepts. The various developments are by and large at different levels of maturity—continuous integration is a concept and practice that has been around for some time, for instance, but continuous delivery is an idea that has not widely been established. We argue that these developments have strong links to Lean Thinking.

This paper proceeds as follows. Section 2 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 3 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 4 concludes this paper by outlining a number of directions for future research.

# 2. LEAN THINKING

The term 'lean' was coined by Krafcik [33] to describe the mode of operation in the Toyota Production System (TPS) [38]. Numerous books have been published on TPS and 'lean manufacturing,' such as 'Lean Thinking' [63]. 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.

## 2.1 Value and Waste

A fundamental focus in lean thinking is that of 'value,' and 'reducing waste' [46, 63]. Any product feature or development step that does not add value is considered to be waste. One of the founding fathers of the TPS, Ohno, identified seven types of waste [46], and others identified a few additional types. One such waste is overproduction, producing something that is unwanted, such as unused product 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. However, without any customer feedback as to whether a feature is needed, there may be significant waste in product development. 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. 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.

## 2.2 Flow and Batch Size

Flow is another central concept within Lean Thinking [63]. Flow can be contrasted with 'batch-and-queue' thinking, whereby actions are done on batches of products, after which they are queued for the next processing step. Instead, 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 thus not only refers 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 be flowing 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 methodology [21]. Kanban can help to level the daily workload, a concept known in lean thinking as *heijunka* [38].

## 2.3 Autonomation and Building-in Quality

A third key concept found within TPS and lean thinking is autonomation, or "*automation with a human touch*" [46, 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 [46, p.21]. This can also be observed in software development; for instance, many organizations that follow an agile approach have an indicator, sometimes an actual traffic light, that springs to red as soon as the build is 'broken.' Tools such as Tinderbox[1] provide a visual interface that can link specific code changes to build failures.

Another related terms is *Poka Yoke* which has been defined as consisting of checklists, test plans, quality matrices, standard architecture, shared components, and standardized manufacturing processes [42, 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.

[1] http://en.wikipedia.org/wiki/Tinderbox_(software)

## 2.4 Kaizen and Continuous Improvement

Adopting 'lean thinking' is a continuous process of improvement. 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 in favor of 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.'

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.

# 3. CONTINUOUS *

As already mentioned, we view Continuous * as a holistic endeavor. We consider 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 Table 1).

## 3.1 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 [8]. 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 [16]. Continuous planning would certainly facilitate BizDev as it requires tighter connection between planning and execution.

### 3.1.1 Continuous Planning

In the context of software development, planning tends be episodic and performed according a traditional cycle usually triggered by annual financial year-end considerations, for example. This traditional planning model is effectively a batch formulation of the problem [32]. 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 [37], and also requires a tight integration between planning and execution [32]. Given the ongoing interest in autonomous systems, it is also interesting that Knight et al. [32] 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 [54].

## 3.2 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.

### 3.2.1 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 [57]. 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 [57]. 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 [16]. Within continuous integration, a number of further modes of continuous activities can be identified, namely continuous deployment and continuous delivery [28, 35]. These concepts are related in that continuous deployment is a prerequisite for continuous delivery, but the reverse is not necessarily the case. That is, continuous delivery refers to releasing valid software builds to users automatically, whereas continuous deployment refers to the practice of deploying the software to some environment, but not automatically delivering to customers.

Continuous delivery has been defined as the ability to

**Table 1: 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. [32], Myers [44]; Lehtola et al. [37]) |
| | **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 artefacts to help ensure that problems leading to integration failures are solved as quickly as possible by those responsible. (see Kim et al. [31]; Rogers [52]; Ståhl & Bosch [57]; Stolberg [56]) |
| Continuous Deployment | Continuous deployment is the practice of continuously deploying good software builds automatically to some environment, but not necessarily to actual users (See Fitz [19], Holmström et al. [27], Humble & Farley [29], Lacoste [35]) |
| Continuous Delivery | Continuous delivery implies continuous deployment and is the practice of ensuring that the software is continuously ready for release and deployed to actual customers (see Neely & Stott [45]) |
| 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. [9]; Cordeiro et al. [14]) |
| Continuous Testing | A process typically involving some automation of the testing process, or prioritisation 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. [4]; Marijan et al. [39]; Muslu et al. [43]; Saff & Ernst [55]). |
| 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. [22]; McHugh et al. [40]). |
| 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 & Raghavan [41]). |
| | **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 [5]; Gebauer et al. [24]; Ortiz & Markus [47]). |
| Continuous Trust | Trust developed over time as a result of interactions based on the belief that a vendor will act cooperatively to fulfil customer expectations without exploiting their vulnerabilities. (see Gefen et al. [25]; Hoehle et al. [26]; Zhou [64]). |
| Continuous Run-Time Monitoring | As the historical boundary between design-time and run-time research in software engineering is blurring [2], in the context of continuously running cloud services, run-time behaviours of all kinds must be monitored to enable early detection of quality-of-service problems, such as performance degradation, and also the fulfilment of service level agreements (SLAs). (see Van Hoorn et al. [59]). |
| | **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. [10], Fowler [23], Jarvinen et al. [30], Krasner [34]). |
| 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 [12]; Holmström-Olsson et al. [27], Ries [51]) |

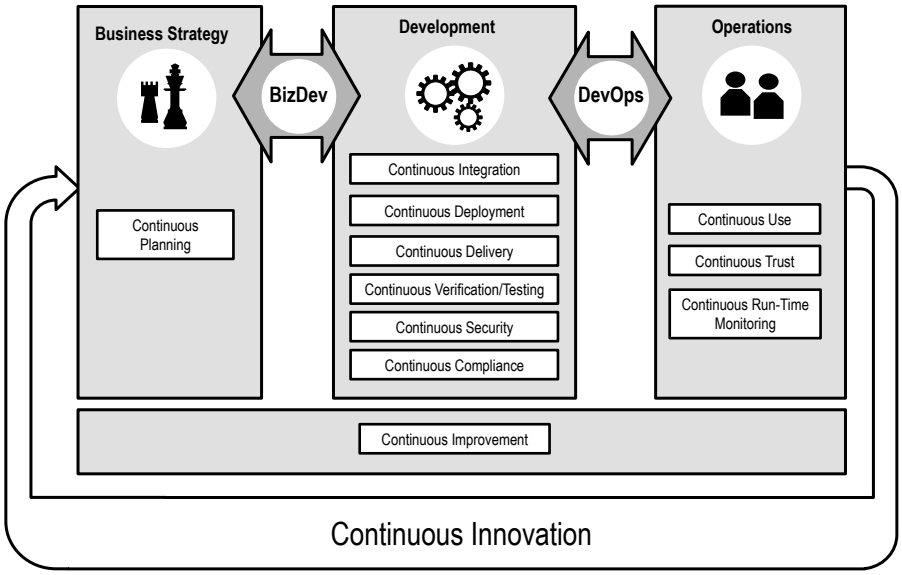


**Figure 1: Continuous *: A holistic view on activities from Business, Development, Operations and Innovation.**

release software whenever an organization wants [45]. In actual practice, this means that new features are deployed into production code as soon as they are finished. This ability and practice to release frequently has long been recognized in open source software communities, where "release early, release often" is a common practice.

Neely and Stolt [45] 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 – in lean vocabulary this is better known as Poka Yoke – and the organization used a number of tools to monitor the state of the system.

By changing 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—the leveling of a workload (i.e., the need to fix defects) is referred to as 'heijunka' in Lean Thinking [63].

### *3.2.2 Continuous Verification/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 [9]. 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. [9] 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 that 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 without a checklist. This is clearly a form of task standardization, which is a key principle in Lean Thinking. Chang et al. found that no significant tool support was required to support continuous verification, which could be an impediment to adopting this activity in some organizations.

Continuous testing seeks to integrate testing activities as closely as possible with coding. Similar to continuous integration, there are potential benefits to this [43]. Firstly, errors can be fixed quickly while the context is fresh in the developers' minds and before these errors lead to knock-on problems. Also, the underlying root causes that led to the problems may be identified an eliminated. Furthermore, there is usually some level of automation of the testing process and a prioritization of test cases [39]. Saff and Ernst introduced the concept of 'continuous testing' [55], and argued that continuous testing can result in wasted development time. Their experiment showed that continuous testing can help to reduce overall development time by as much as 15%. This suggests that continuous testing can be an effective tool to reduce one of the types of wastes, namely that of waiting time.

### 3.2.3 Continuous Compliance and Continuous Security

Agile methods were initially seen as suited to small projects with co-located developers in non-safety critical contexts [1, 6]. However over the past decade or so agile methods have been successfully applied on large projects with distributed developers [20], and in recent times, the final frontier, that of applying agile methods on safety critical systems is being addressed. Fitzgerald et al. [22] discuss the tailoring of the Scrum method for a regulated environment, *R-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 [41]. 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.

## 3.3 Operations

### 3.3.1 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 [24]. However the latter is necessary given that the economic payoff from systems comes from continued use rather than initial adoption [5].

Gebauer et al. [24] point out that the models which are used to study initial adoption (e.g. TAM [15] and UTAUT [60]) are not necessarily suited to study continuous use as they do not consider variables such as automatic and unconscious or habitual characteristics, which have been found to be important in actual continuous use [47]. Also, the theoretical concepts underpinning these models were derived in an era where the consumers of technology were digital immigrants rather than digital natives who have known technology all their lives [62]. The latter are motivated very differently and have different attitudes to software use.

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 self-reporting measures which are inevitable when personal intention is being assessed. The trend towards rapid experimentation and split A/B testing with users to assess acceptance of various feature sets is also relevant to the continuous use category.

### 3.3.2 Continuous Trust

Drawing on Hoehle et al. [26] and Pavlou & Fygenson [49], 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, it just as the initial adoption scenario is quite different to continuous use, the relationship between initial trust and continuous trust is a complex one. Hoehle et al. [26] 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, for example, 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 e.g., 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, e.g., 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.

### 3.3.3 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 [2]. 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 secure. Van Hoorn et al. [59] 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).

## 3.4 Continuous Improvement and Continuous Innovation

Continuous improvement, or Kaizen, is a key tenet of Lean Thinking. In software development terms, continuous product improvement manifests itself in the refactoring concept, a key practice in Extreme Programming [23]. Continuous process improvement has also been a prominent theme in the software arena [10, 30, 34]. 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. [58] argue 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 [11]. Also, the theme of continuous innovation has emerged, most notably in the software domain in the concept of the Lean Start-Up [51]. 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 [12]. 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 [27]. 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 [12]. As a consequence, we position continuous innovation and continuous improvement as the foundation upon which the other Continuous * activities can be grounded (See Figure 1).

# 4. DISCUSSION AND 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 [13]. The need for the Continuous * concept is evident when one considers the emergence of phenomena such as Enterprise Agile, Beyond Budgeting, DevOps, Lean Start-Ups 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.

Among the main contributions of this work are the following:

- Recognition of the need for tighter connection between the various phases of business strategy, development and execution.
- More specifically, the identification of *BizDev* to represent the need for tighter integration between business strategy and software development, a complement to the DevOps concept.
- The need to go *beyond* episodic planning based on a traditional annual financial model to one more in keeping with market needs.
- Recognition that initial adoption of technology is driven by very different factors to those which determine continued use.
- The interconnection between the various continuous activities. For example, to deliver on rapid experimentation and Split A/B testing, required for continuous innovation, there is a need for precise build information to identify exactly which features are included in different builds which facilitate A/B testing in the first place.
- While some tool support is available for certain sub-sets of Continuous *, appropriate tool support is needed for the overall concept. For instance, rapid deployment of different versions of a software product and managing the resulting data that supports decision-making in pursuing certain business opportunities could greatly benefit from dedicated tool support. This area has received very little attention so far.

# 5. ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero—the Irish Software Engineering Research Centre (www.lero.ie).

# 6. REFERENCES

[1] S. Ambler. When does(n't) agile modeling make sense, 2001. www.agilemodeling.com/essays/whenDoesAMWork.htm.

[2] L. Baresi and C. Carlo Ghezzi. The disappearing boundary between development-time and run-time. In *Future of Software Engineering Research*, 2010.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[4] M. Bernhart, S. Strobl, A. Mauczka, and T. Grechenig. Applying continuous code reviews in airport operations software. In *12th International Conference on Quality Software*, pages 214–219, 2012.

[5] A. Bhattacherjee. Understanding information systems continuance: An expectation-confirmation model. *MIS Quarterly*, 25(3):351–370, 2001.

[6] B. Boehm. Get ready for agile methods, with care. *IEEE Computer*, 35:64–69, 2002.

[7] B. Bogsnes. *Implementing Beyond Budgeting: Unlocking the Performance Potential*. Wiley, 2008.

[8] CA Technologies. The innovation imperative: Why it needs to lead now, 2012. www.ca.com/us/ /media/Files/Presentations/the-innovation-imperative-external-presentation-final.pdf.

[9] T.-F. Chang, A. Danylyzsn, S. Norimatsu, J. Rivera, D. Shepard, A. Lattanze, and J. Tomayko. "continuous verification" in mission critical software development. In *30th Hawaii International Conference on System Sciences*, volume 5, pages 273–284, 1997.

[10] X. Chen, P. Sorenson, and J. Willson. Continuous SPA: Continuous assessing and monitoring software process. In *IEEE Congress on Services (SERVICES)*, 2007.

[11] H. Chesbrough. *Open Innovation: The New Imperative for creating and Profiting from Technology*. Harvard Business School Press, 2003.

[12] R. Cole. From continuous improvement to continuous innovation. *Quality Management Journal*, 8(4), 2001.

[13] K. Conboy and B. Fitzgerald. Towards a conceptual framework of agile methods. In *XP and Agile Conference*, 2004.

[14] L. Cordeiro, B. Fischer, and J. Marques-Silva. Continuous verification of large embedded software using smt-based bounded model checking. In *17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, 2010.

[15] F. Davis, R. Bagozzi, and P. Warshaw. User acceptance of computer technology: A comparison of two

theoretical models. *Management Science*, 35(8):982–1003, 1989.

[16] P. Debois. Devops days ghent, 2009. http://www.devopsdays.org/events/2009-ghent/.

[17] C. Ebert, P. Abrahamsson, and N. Oza. Lean software development. *IEEE Software*, 29(5):22–25, 2012.

[18] J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani. *Perspectives on Free and Open Source Software*. MIT Press, 2005.

[19] T. Fitz. Continuous deployment at IMVU: Doing the impossible fifty times a day, 2009. http://timothyfitz.com/2009/02/10/continuous-deployment-atimvu-doing-the-impossible-fifty-times-a-day/.

[20] B. Fitzgerald, G. Hartnett, and K. Conboy. Customising agile methods to software practices at Intel Shannon. *European Journal of Information Systems*, 15(2):197–210, 2006.

[21] B. Fitzgerald, M. Musiał, and K. Stol. Evidence-based decision making in lean software project management. In *36th International Conference on Software Engineering (ICSE-SEIP)*, 2014.

[22] B. Fitzgerald, K. Stol, R. O'Sullivan, and D. O'Brien. Scaling agile methods to regulated environments: An industry case study. In *35th International Conference on Software Engineering (ICSE-SEIP)*, 2013.

[23] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[24] L. Gebauer, M. Sollner, and J. Leimeister. Towards understanding the formation of continuous it use. In *34th Int'l Conf. Information Systems*, 2013.

[25] D. Gefen, E. Karahanna, and D. Straub. Trust and tam in online shopping: An integrated model. *MIS Quarterly*, 27(1):51–90, 2003.

[26] H. Hoehle, S. Huff, and S. Goode. The role of continuous trust in information systems continuance. *Journal of Computer Information Systems*, 52(4), 2012.

[27] H. Holmström Olsson, H. Alahyari, and J. Bosch. Climbing the "stairway to heaven": A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *38th Euromicro Conference on Software Engineering and Advanced Applications*, 2012.

[28] J. Humble. Continuous delivery vs continuous deployment, 2010. http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/.

[29] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[30] J. Järvinen, D. Hamann, and R. van Solingen. On integrating assessment and measurement: Towards continuous assessment of software engineering processes. In *6th International Software Metrics Symposium*, 1999.

[31] S. Kim, S. Park, J. Yun, and Y. Lee. Automated continuous integration of component- based software: an industrial experience. In *ASE*, 2008.

[32] R. Knight, G. Rabideau, S. Chien, B. Engelhardt, and R. Sherwood. Casper: Space exploration through continuous planning. *Intelligent Systems*, 16(5), 2001.

[33] J. Krafcik. Triumph of the lean production system. 1988.

[34] H. Krasner. The ASPIRE approach to continuous software process improvement. In *2nd International Conference on Systems Integration*, 1992.

[35] F. Lacoste. Killing the gatekeeper. In *Agile Conf.*, 2009.

[36] D. Leffingwell. *Scaling software agility: best practices for large enterprises*. Addison-Wesley, 2007.

[37] L. Lehtola, M. Kauppinen, J. Vähäniitty, and M. Komssi. Linking business and requirements engineering: is solution planning a missing activity in software product companies? *Requirements Engineering*, 14(2):113–128, 2009.

[38] J. K. Liker. *The Toyota Way*. McGraw Hill, 2004.

[39] D. Marijan, A. Gotlieb, and S. Sen. Test case prioritization for continuous regression testing: An industrial case study. In *IEEE International Conference on Software Maintenance*, 2013.

[40] M. McHugh, F. Mc Caffery, B. Fitzgerald, K. Stol, V. Casey, and G. Coady. Balancing agility and discipline in a medical device software organization. In *13th International SPICE Conference*, 2013.

[41] M. Merkow and L. Raghavan. An ecosystem for continuously secure application software. *CrossTalk*, March/April, 2011.

[42] J. Morgan and J. Liker. *The Toyota Product Development System*. Productivity Press, 2006.

[43] K. Muslu, Y. Brun, and A. Meliou. Data debugging with continuous testing. In *ESEC/FSE*, 2013.

[44] K. Myers. CPEF: A continuous planning and execution framework. *AI Magazine*, 20(4):63–69, 1999.

[45] S. Neely and S. Stolt. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *Agile Conference*, 2013.

[46] T. Ohno. *Toyota Production System: Beyond Large-Scale Production*. CRC Press, 1988.

[47] A. Ortiz de Guinea and M. Markus. Why break the habit of a lifetime? rethinking the roles of intention, habit, and emotion in continuing information technology use. *MIS Quarterly*, 33(3):433–444, 2009.

[48] E. Overby, A. Bharadwaj, and V. Sambamurthy. A framework for enterprise agility and the enabling role of digital options, business agility and information technology diffusion. In *Business Agility and Information Technology Diffusion*, 2005.

[49] P. Pavlou and M. Fygenson. Understanding and predicting electronic commerce adoption: An extension of the theory of planned behavior. *MIS Quarterly*, 30(1):115–143, 2006.

[50] D. G. Reinertsen. *The Principles of Product Development Flow*. Celeritas Publishing, 2009.

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

[52] R. Rogers. Scaling continuous integration. In *Extreme Programming and Agile Processes in Software Engineering*. Springer, 2004.

[53] W. W. Royce. Managing the development of large software systems. In *9th international conference on Software Engineering*, pages 328–338, 1987.

[54] G. Ruhe. *Product Release Planning: Methods, Tools and Applications*. CRC Press, 2010.

[55] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *14th International Symposium on Software Reliability Engineering*, 2003.

[56] S. Stolberg. Enabling agile testing through continuous integration. In *Agile Conference*, 2009.

[57] D. Ståhl and J. Bosch. Modeling continuous integration practice differences in industry software development. *The Journal of Systems and Software*, 87, 2013.

[58] M. Tushman, P. Anderson, and C. O'Reilly. Technology cycles, innovation streams, and ambidextrous organizations: Organizational renewal through innovation streams and strategic change. In *In Managing strategic innovation and change*. Oxford University Press, 1997.

[59] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: Design and application of the kieker framework, 2009.

[60] V. Venkatesh, M. Morris, G. Davis, and F. Davis. User acceptance of information technology: Toward a unified view. *MIS Quarterly*, 27(3):425–478, 2003.

[61] VersionOne. 7th annual state of agile survey: The state of agile development, 2012.

[62] S. Vodanovich, D. Sundaram, and M. Myers. Digital natives and ubiquitous information systems. *Information Systems Research*, 21(4):711–723, 2010.

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

[64] T. Zhou. An empirical examination of continuance intention of mobile payment services. *Decision Support Systems*, 54(2):1085–1091, 2013.