O'Connor, R. V., Elger, P., Clarke, P., Continuous Software Engineering – A Microservices Architecture Perspective, Journal of Software: Evolution and Process, Vol.29, No. 11, 2017. http://dx.doi.org/0.1002/smr.1866

Continuous Software Engineering – A Microservices Architecture Perspective

Rory V. O'Connor [1 2], Peter Elger [3] and Paul M. Clarke [2, 4]

[1]*Dublin City University, Ireland, Rory.OConnor@dcu.ie*
[2]*Lero - Irish Software Research Centre, Ireland*
[3]*nearForm Ltd., USA, Peter.Elger@nearform.com*
[4]*Dublin City University, Ireland, Paul.M.Clarke@dcu.ie*

ABSTRACT

From its earliest days, software development has been beset with challenges in relation to timely delivery, appropriateness of features and quality of deliverables. Many advances in software development processes have helped to address these concerns. For example, agile software development has helped to deliver working software more frequently and capability maturity frameworks have brought about improved consistency in quality levels. However, the age-old challenge of *better*, *cheaper*, *faster* software has continued to beguile developers. In this paper, we discuss an emerging approach to software development, *continuous software engineering (CSE),* wherein software of operational quality may be delivered on a very frequent basis, up to many times in a given day. This approach employs a complex set of independent tools that together reduce the lead time in delivering commercial-grade software. Having examined the situational context of one industrial organisation applying CSE, we conclude that the approach may not presently be appropriate to all manners of software development. Nonetheless, the authors are of the view that CSE represents a significant progression in software engineering and that software development settings of all types stand to gain from aspects of the CSE approach.

Keywords

Software Development Process; Continuous Software Engineering; Agile; Microservices; Situational Factors.

1. INTRODUCTION

It is generally accepted that no single software development process is perfectly suited to all software development undertakings [1] and that all software development settings are continually witnessing change [2], [3]. The result is that some amount of process adaption and situational tailoring [4] is required in order to render a process suitable to a given situational context, evidence for which can be seen in the reported amount of process improvement in practice [5], [6]. As noted in the literature, a software process itself is therefore a continuous rather than a static concern [7], and so we should seek to identify techniques that can improve our understanding of interactions between software processes and their situational contexts [8]. We should furthermore take account of the significant complexity associated with the production of commercial grade software [9]. This paper identifies the aggressive adoption and integration of toolsets which replace aspects of development work which were at one time human-intensive and detailed in nature, and for which automated tooling can offer an opportunity to manage the complexity and associated error-proneness.

Continuous software development and deployment has started to receive greater attention in recent years [10] and in the context of our work, we define *Continuous Software Engineering* (CSE) as the application of automation via tools to increase the deployment frequency of new releases of commercial-grade software. CSE permits software feature delivery at rates which just a few short years ago may have been considered unachievable. The authors take the view that one can consider CSE to be the next major wave in software engineering to follow the agile software development [11] and lean software development [12] movements. It is different to these earlier approaches in that CSE emphasises the role of tooling in software development [13], which at least to some extent runs contrary to the agile manifesto which values *individuals and interactions over processes and tools*. With CSE, the importance and investment in tool chains is large and its centrality to the delivery lifecycle is likely to raise its value to at least be on a par with the value associated with individuals and interactions. Furthermore, CSE may also impose greater emphasis on processes. However, with CSE the nature of the process is quite different to earlier software development process initiatives. Accordingly, whereas the ISO 9001 [14] perspective on process may involve very detailed work process definitions and adherence, a CSE implementation involves the realisation of well-thought through development, build and deployment procedures which are realised through knitting together various independent tool chains and automating aspects of the lifecycle. Therefore, if by the term *process* we mean the sequence of steps required to develop or maintain software [15], then clearly CSE places considerable emphasis on process, and in particular, on process automation.

Core to the concept of CSE is *Continuous Integration*, a set of software engineering practices that speed up the delivery of software by decreasing integration times. It emerged in the Extreme Programming (XP) community [16] and is described as "a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day, where each integration is verified by an automated build (including test) to detect integration errors as quickly as possible" [17].

Continuous integration promises advantages in large-scale software development by enabling software development organizations to deliver new functions faster. According to studies of commercial projects using daily builds and thus some form of continuous integration, advantages of continuous integration include [18] reduced integration risks (errors are found early) and motivation (you see a working system). Continuous integration is also an alternative to 'big bang' integration, where all modules are combined in one go, and which usually results in large numbers of errors, hard to isolate and correct owing to the vast expanse of the program. Disadvantages of continuous integration may include degeneration of architecture due to lack of focus on overall design and time spent on too frequent releases of too poor quality [18]. However, implementing continuous integration in large software development organizations is challenging because of organizational, social and technical reasons. One of the technical challenges is the ability to rapidly prioritize the test cases which can be executed quickly and trigger the most failures as early as possible [19].

The CSE concept is perhaps only possible at this point in time as a result of the innovation that has taken place in the tooling domain, specifically in the tools that support software manufacturers. These tools range from configuration management tools, such as *Git*[1], to build tools such as *Jenkins*[2], and deployment tools, such as *Docker*[3]. Central to the CSE approach that we present in this work is the microservices architecture [20], which seeks to deliver small, self-contained and rigidly enforced atoms of functionality. The resulting tool set and microservices architecture promotes the frequent delivery of new commercial-grade software features that can seamlessly interact with preexisting operational systems.

To examine the impact of CSE, we undertake a case study with an industrial partner who is presently fully committed to a CSE approach. We investigate the industrial partner in detail, seeking to understand why CSE has proven to be very effective in their instance. To enable the examination, we employ the situational factors reference model [8] which was previously developed by the authors. This model ensures that we examine in great detail the specific aspects of the industrial partner's business that have influenced their adoption of CSE. Indeed, these same factors – as we shall present in the paper – have contributed to the success of our industrial partner's CSE agenda.

This paper is organized as follows: Section 2 outlines the situational factors framework which we adopted to examine the suitability of CSE for our industrial partner; Section 3 presents an overview of the company studied, including details of its CSE process; Section 4 evaluates why CSE has proven effective for our industrial partner; and finally, Section 5 presents a conclusion.

---

[1] Git is a free and open source distributed version control system. See website for more information: https://git-scm.com

[2] Jenkins is an open source cross-platform, continuous integration and continuous delivery application. See website for more information: https://jenkins.io/

[3] Docker is an open-source project that automates the deployment applications inside software containers. See website for more information: https://www.docker.com/

2. SITUATIONAL FACTORS

The importance of context in software process decisions has been acknowledged for some time [21]. Whilst the literature has noted that "the organization's processes operate in a business context that should be understood" [22] and that a "life cycle model… [should be] appropriate for the project's scope, magnitude, complexity, changing needs and opportunities" [6], contributions to the literature in relation to software process context space are lacking. Software development necessarily occurs in a development context, which includes a large number of concerns and factors [23], [24] and it is this contextualization which provides a better understanding of what works for whom, where, when, and why [25]. In support of the importance of understanding the impact of situational factors, authors such as Dyba [26] point out that it is this dependence on a potentially large number of context variables in any study that is an important reason for why software engineering is so hard.

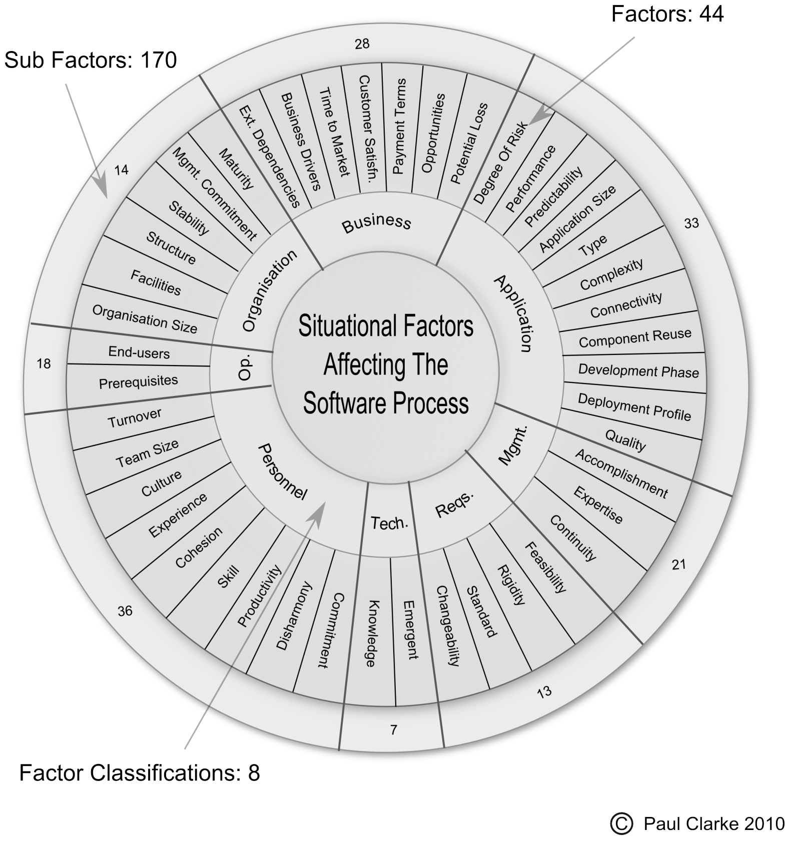

Figure 1. Situational Factors Reference Framework

Despite the frequent references to the importance of situational context in the literature, it was the apparent lack of a comprehensive situational factors framework for software development that led two of the authors to produce and publish an initial reference framework [8], itself an amalgamation of earlier contributions. To raise the scope and coverage of the situational factors reference framework [8], various important contributions throughout software development's history were incorporated, including models and standards [22], [27], risk factors [28], [29], [30], [31], [32], [33], cost estimation [34], [35], [36], environmental factors [37], process tailoring [38], [39], process agility [40] and bodies of knowledge [41]. In each case, the contribution was thoroughly assessed to identify factors that might influence the software development process. Later, these factors were systematically integrated into a unified model using the constant comparison and memoing techniques from grounded theory [42], [43]. The result is a systematically derived, comprehensive reference framework of factors that influence software processes that is well suited to the task of identifying specific situational factors in specific software development settings. This reference framework is in the view of its authors a stepping stone towards greater appreciation of the complexity of software development settings, and the rigorous approach employed in its creation from a rich variety of sources has given rise to a framework that they consider to present a broadly informed reference for the software development community [44]. We therefore employed this framework so as to evaluate the situational context of our industrial partner with a view to understanding why CSE has proven effective in that particular setting.

Table 1. Situational Factors Classification

Classification	Description
Personnel	Constitution and characteristics of the non-managerial personnel involved in the software development efforts.
Requirements	Characteristics of the requirements.
Application	Characteristics of the application(s) under development.
Technology	Profile of the technology being used for the software development effort.
Organization	Profile of the organization.
Operation	Operational considerations and constraints.
Management	Constitution and characteristics of the development management team.
Business	Strategic and tactical business considerations.

The framework incorporates 44 individual factors (ref. Figure 1) which are categorized using 8 classifications (ref. Table 1), and which are based upon 170 underlying sub-factors. A sample listing of the sub-factors in the *Personnel* classification is presented in Table 2.

Table 2. Personnel Factors & Sub-Factors

Factor	Sub-Factors
Turnover	Turnover of personnel
Team size	Size of team
Culture	Team culture/resistance to change
Experience	General team experience; team diversity; team ability to understand the human implications of a new information system/team ability to work with management; application experience; analyst / programmer /tester experience; experience with development methodology platform experience.
Cohesion	General cohesion; team members who have not worked for you; team not having worked together in the past/team ability to successfully complete a task; team ability to work with undefined elements and uncertain objectives; overdependence on team members; distributed or geographically distant team.
Skill	Operational knowledge; team expertise (task); team ability to work with undefined elements and uncertain objectives; training development.
Productivity	Team ability to carry out tasks quickly; general productivity.
Commitment	Commitment to project among team members.
Changeability	Scope creep; continually changing system requirements; ill-defined project goals; gold plating; unclear system requirements.
Disharmony	Interpersonal conflicts.

3. CASE STUDY COMPANY

The case study firm nearForm Ltd., is a software development company with a presence in the US and Europe and which has experienced substantial growth in recent times through the continual delivery of high quality software to some of the largest companies in the world, including blue chip financial institutions. Value is a key focus in the nearForm lifecycle and it is concerned with an acute responsiveness to client needs (be they new features or defect resolutions). The organization works to a regular 5-day iteration for software development, deploying working software to customer site(s) weekly (sometimes daily) through a standard feature bundle. While regular iterations can be predictable from the outset, continual analysis of the value stream ensures that each iteration may be re-planned in real time, delivering the highest possible value from organizational capacity (ref. Figure 2).

Whilst it is acknowledged that tooling can affect the design of a software process [45], the impact of technology on shaping the process in this case is profound and may even run contrary to the Agile Manifesto value of '*Individuals and interactions over processes and tools*'. Within nearForm the continual software evolution and delivery is made possible through the aggressive incorporation of contemporary and predominately open source software tools. While the speedy delivery of innovative features is a vital enabler of competitive advantage, it is only effective if it is accompanied by reliable and high quality deployments.

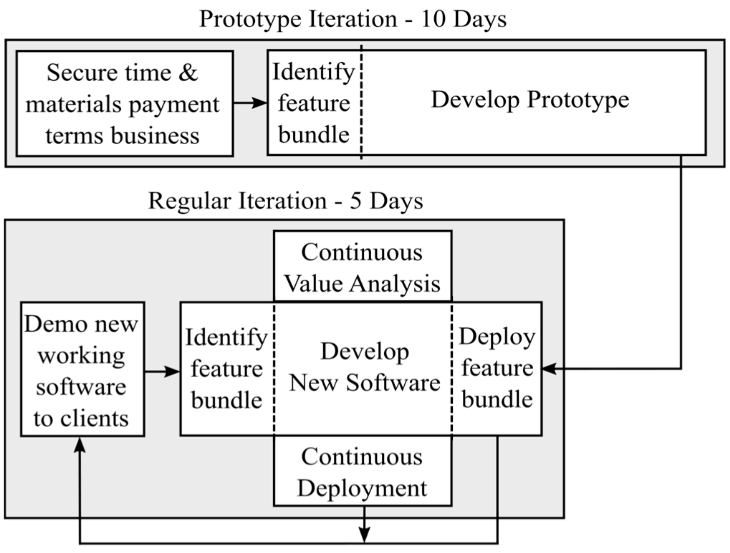
Prototype Iteration - 10 Days
Secure time & materials payment terms business
Identify feature bundle
Develop Prototype
Regular Iteration - 5 Days
Continuous Value Analysis
Demo new working software to clients
Identify feature bundle
Develop New Software
Deploy feature bundle
Continuous Deployment

Figure 2. nearForm Process Lifecycle

There are five key technology enablers driving the process architecture: (1) JavaScript and Node.js which enable extremely rapid code development by utilizing the same programming language across the entirety of the system; (2) Alongside a distributed microservices architecture, under which the system is broken down into a set of discrete co-operating processes, typically each service is of the order of several hundred lines of code only; (3) The adoption of cloud technologies, whereby all infrastructure may be treated as code, supports the rapid creation of testing and production environments; (4) The architectural approach is coupled with a continuous deployment model, layered over the Docker container engine, whereby individual services (or several services at a time) may be deployed without perturbing the system as a whole; (5) Finally the company ensures quality through steps such as code commit hooks via GitHub[4] and the Travis[5] Continuous Integration tool set to construct continuous delivery pipelines. Together, these technologies enable the company to perform well under a time and materials contract basis, whereby clients are initially attracted through the rapid delivery of a prototype in 10 days, and thereafter, regular iterations of new working software are reviewed every 5 days.

3.1 JavaScript and Node.js

JavaScript is an interpreted programming language with object-orientated capabilities. Once considered a 'toy' language by many developers [46]. The growing use of JavaScript has created whole new technical and business models of program construction and deployment [47]. JavaScript now presents as an ideal language for full-stack, enterprise development [48].

[4] GitHub is a web-based Git repository hosting service for revision control and source code management. See website for more information: https://github.com/

[5] Travis CI is a hosted, distributed continuous integration service used to build and test software projects hosted at GitHub. See website for more information: https://travis-ci.org/

Node.js - also called Node - is a cross platform runtime environment originally developed in 2009 by for developing server-side applications. It can be regarded as server-side JavaScript. It was created to address the issues platforms can have with the performance in network communication time dedicating excessive time processing web requests and responses. Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices. Node has become popular as it makes creating high performance, real-time web applications easy. Node allows JavaScript to be used end to end, both on the server and on the client.

Node.js when coupled with its supporting package management system – *npm*[6] – provides a lean and efficient platform that enables developers to be highly productive. This, when combined with an effective front-end framework (such as *angular* or *react*) provides a powerful and rapid development platform enabling the same language to be used in all tiers. The rapid adoption of Node.js is evidenced by Figure 3, which shows the number of open source modules available for the various popular open source platforms (Node.js is the top line). As of January 2016, there are over 225,000 modules available for Node.js with module downloads running in excess of 2.5 billion per month [49], a very strong indicator that this technology stack has some significant momentum behind it.

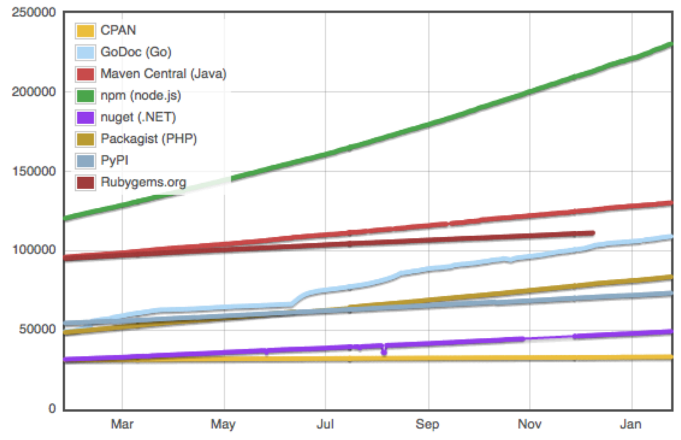

Figure 3 module counts

3.2 Microservice Architecture

Microservices are a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility [13]. The term microservice architecture refers to a style of development under which a system is broken down into a number of small co-operating components [20], where these components typically interact over a direct point-to-point interface (for example, http) or utilizing an event bus technology such as Apache Kafka or RabbitMQ.

The microservice architectural style can be viewed as an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API [50]. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies

By way of comparison a monolithic architectural style is where an application is built as a single unit, typically compromising three main parts: a client-side user interface a database and a server-side application. In this case the server-side application is a 'monolith' - a single logical executable. Any changes to the system involve building and deploying a new version of the server-side application. Monolithic applications can be successful, but increasingly people are feeling frustrations with them [50], in particular as more applications are being deployed to the cloud. Change cycles are tied

[6] *npm* is the default package manager for the JavaScript runtime environment Node.js.. See website for more information: https://www.npmjs.com/

together - a change made to a small part of the application, requires the entire monolith to be rebuilt and deployed. Over time it's often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module. Scaling requires scaling of the entire application rather than parts of it that require greater resource.

These frustrations have led to the microservice architectural style: building applications as suites of services. As well as the fact that services are independently deployable and scalable, each service also provides a firm module boundary, even allowing for different services to be written in different programming languages. They can also be managed by different teams

The key benefits of microservices architectural style includes:

- A highly modular and decoupled system that can be easier to maintain than a traditional class hierarchy;
- The ability to deploy services rapidly to a production system – because services are independently deployable entities, only the service under question need undergo rigorous testing and the rest of the system has not been changed;
- Microservices are highly cohesive units of code that are easier to reason about and manage in isolation, this tends to reduce the burden on developers and if implemented responsibly can lead to simpler code with less defects.

As a corollary to these benefits, microservice systems require a more sophisticated DevOps infrastructure [51], typically requiring the construction of a service deployment pipeline. Use of cloud and container technologies enables the construction of such pipelines and it is this technology enabler that is driving the adoption of these hyper-agile, lean processes. It is the final piece in the jigsaw that makes the technology stack so powerful.

3.3 Cloud Infrastructure and DevOps

The traditional world of IT infrastructure management and systems operations teams is fundamentally and rapidly changing. The world of capital expenditure, hardware lead times and capacity planning is fast becoming outmoded. It has become very clear over recent years that most IT organizations are embracing the notion of commoditized, elastically scaled compute power delivered by infrastructure as a service providers (IAAS) such as Amazon Web Services (AWS). Whilst there will remain certain pockets of specialized co-located services with dedicated operations teams, these will be restricted to specialized application domains. Even in these environments cloud technology is being applied at the periphery in so called hybrid cloud computing models. Under this model, infrastructure management and operations becomes just another programming discipline, often termed DevOps. This rise can be witnessed by the growth of AWS (figure 4), which is expected to exceed over USD10Bn in revenues in 2016 [52].

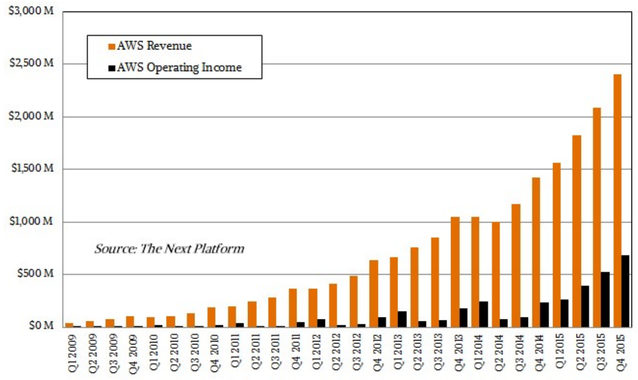

Figure 4 AWS revenue growth

The power of the cloud computing model for rapid software creation and deployment becomes apparent when one observes that environments like AWS are fully programmable through an extensive API set that allows third party vendors and implementers to control all aspects of the platform (figure 5). Use of power tools such as Hashicorp's *Terraform*[7] make this environment simple to configure and use. This allows companies like nearForm to rapidly spin up build, staging and production environments based on the formulaic application of a few simple scripts.

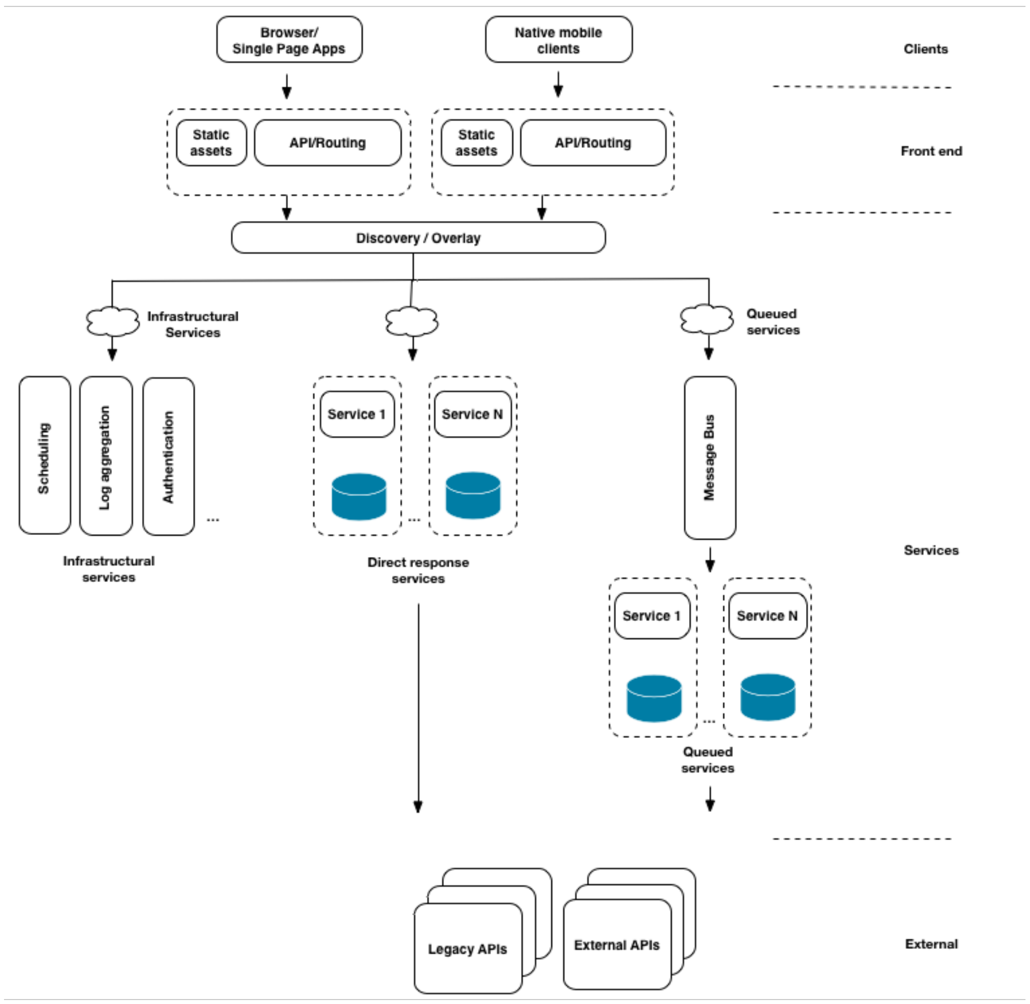

Figure 5 microservice reference architecture

3.4 Software Container Technology

Software containers provide a means of encapsulating functionality within an isolated process space, i.e. a single operating system level process can attend to just a specific, small piece of executable code. The concept of software containers originated in the late seventies with the addition of the *chroot* system call to the BSD Unix operating system. This feature was largely unused until FreeBSD *jails* were introduced in 2000. This was followed by Solaris *zones* in 2004. A more mainstream user-land implementation in the Linux kernel followed in 2008 with the advent of *LXC*[8] (LinuX Containers). However the technology first began to gain wide adoption in 2013 via the Docker project, and it has resulted in the capability of developers to regularly inject new, easily digestible features into live systems with less risk than traditional software development and deployment models.

[7] Terraform is a tool for building, changing, and combining infrastructure safely and efficiently. See website for more information: https://www.terraform.io/

[8] LXC (Linux Containers) is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) on a control host using a single Linux kernel. See website for more information: https://linuxcontainers.org/

Container technology may become the mainstream for certain types of software development, especially with the development of container management and orchestration systems such as *Kubernetes*, *Docker Swarm* and *AWS container services*.

Container technology solves two key problems, thereby enabling the construction of rapid deployment pipelines and advanced software deployment strategies. Firstly the 'runs on my machine' problem; the container holds both the code and the execution environment such as library dependencies, underlying OS and environment settings etc. this means that a container that executes successfully in a test environment can be promoted to production with a high degree of confidence that it will execute successfully, removing concerns of differences in OS version and complex code dependency chains. Secondly containers provide a homogeneous abstraction allowing deployment scripts and tooling to deal with a single entity irrespective of what is executing inside of the container. It is perfectly possible to have a system comprising multiple execution environments (for example, Node.js, Java, Python etc.) all deployed using the same tool chain.

3.5 Continuous Delivery Pipeline

[53] describe Continuous delivery (CD) is the ability to release software whenever we want. This could be weekly or daily deployments to production. The frequency is not our deciding factor, it is the ability to deploy at will that is a key factor. More broadly CD is a software engineering approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time. CD is attracting increasing attention and recognition. CD advocates claim that it lets organizations rapidly, efficiently, and reliably bring service improvements to market and eventually stay a step ahead of the competition [54].

Continuous delivery treats the commonplace notion of a deployment pipeline [55] (also known as deployment production line): a set of validations through which a piece of software must pass on its way to release. Code is compiled if necessary and then packaged by a build server every time a change is committed to a source control repository, then tested by a number of different techniques (possibly including manual testing) before it can be marked as releasable.

Due to the need to support multiple applications for multiple clients, nearForm creates a CD pipeline for each application. Although each application's pipeline will differ somewhat from another application's pipeline, the pipeline architecture is broadly similar and the generic nearForm CD pipeline is depicted in Figure 6.

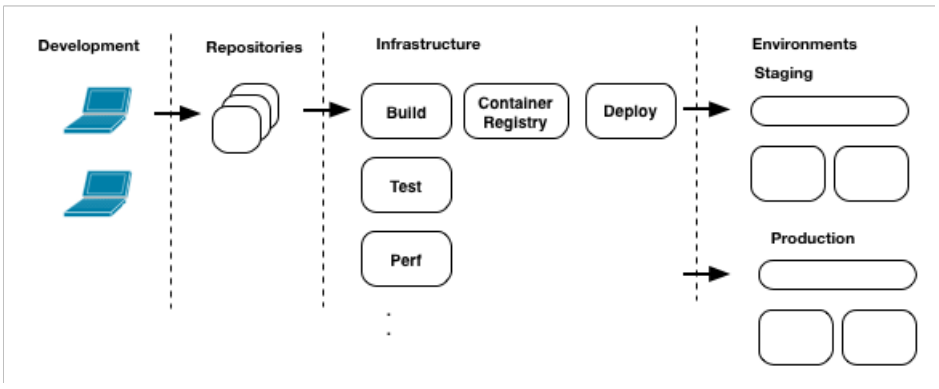

Figure 6 continuous delivery pipeline

A typical pipeline consists of the following stages:

- Code is checked in by developers and immediately pulled onto the build system
- A successful build (i.e. all test passing at the required coverage level) results in a updated set of containers that are pushed to a centralized container registry
- A deployment process is executed to distribute the updated containers into a staging or test environment

- A suite of end to end tests is run against the staging environment and the results made available
- Given that all containers are operating correctly in staging updates may be made immediately to the production environment

Whilst this type of process may be realized with a multitude of different tools, some examples for each phase are provided in Table 3.

Table 3. Example Tools to Enable CSE

Phase	**Tool**
Development	Fuge - http://fuge.io/
	Seneca - http://senecajs.org/
	HAPI - http://hapijs.com/
Repository	Git / github - https://github.com/
	BitBucket - https://bitbucket.org/
Build	Jenkins - https://jenkins.io/
	Drone - https://drone.io/
Test	Tap - https://testanything.org/
	Phantom - http://phantomjs.org/
	Pdiffy - https://github.com/kennychua/pdiffy
Infrastructure	EC2 - https://aws.amazon.com/ec2/
	ELB - https://aws.amazon.com/elasticloadbalancing/
	Consul - https://www.consul.io/
Deployment	Docker registry- https://docs.docker.com/registry/
	Code Deploy – https://aws.amazon.com/codedeploy/
	Swarm - https://docs.docker.com/swarm/
	Kubernetes - http://kubernetes.io/

The above process occurs on every code commit, thus there may be multiple daily releases to the systems production environment. Key benefits of this type of continuous delivery process are:

- Minimization of the delta between development and production. The high frequency of deployment minimizes risk, because each change is a small incremental update allowing rapid diagnosis in the event of production issues.
- Making deployment a common occurrence has the effect of removing fear and uncertainty from the process – contrast this with a quarterly deployment process.
- Everything is automated, human intervention is not required unless a problem is detected.

The case study company applies all of the above technologies, i.e. the node.js platform, cloud infrastructure and containers along with the indicated tooling to build delivery pipelines and software systems aligned with the micro-services architectural paradigm. It is important to note that it is the architectural and systemized approach that is key and that this is readily transferrable across platforms and tool chains. For example the company has applied the same formula to RackSpce (a provider of cloud infrastructure) with similar results. The company has also built systems using a combination of Scala, Java and node.js, demonstrating this transferability.

4. APPLYING THE SITUATIONAL FACTORS REFERENCE FRAMEWORK

Two researchers in association with the Director of Engineering from nearForm undertook a detailed analysis of the company's situational factors, the primary results of which are presented in Table 4. Since the researchers were also the creators of the Situational Factors model adopted in this study, familiarity with the model allowed the researchers to progress the investigation of factors with the benefit of significant amassed competence in this domain. The nature of the engagement between the researchers and the case study organization took the form of a series of detailed discussions during which the nearForm development process was examined in detail, at each step taking note of the key motivators that informed the process design. This was achieved by the researchers consulting the Situational Factors references model, and querying the motivation for the process through the lens of the model. A number of meetings were held by telephone, following which various understandings were documented and shared with the nearForm to seek to established their validity and accuracy.

From the researchers' perspective, one of the researchers would conduct the discussion with the nearForm process specialist, thereafter documenting the results and discussing them with the second researcher. This allowed the first researcher to elicit objective feedback on the proposed relationships and to provide a checkpoint for concerns that may have been overlooked in the initial analysis. This could be considered to be a form of informal internal review. Gradually, this iterative approach to building up the understanding of the nearForm process motivations yielded a more concrete and complete understand, with each iteration providing feedback opportunities from both the perspective of the case study organisation and the research team. It is important to stress that this process took the form of telephone calls, email communications and informal feedback, sometimes delivered verbally. From a strictly academic perspective this approach does raise some issues relating to reproducibility and traceability. However, with each step, the understanding was refined and shared in some type of documented form, with the final iteration agreed upon by both researchers and nearForm.

Table 4. Situational Factors Influencing Software Process in Case Study

	Factors Identified in Case Study
Personnel	**Cohesion**: The company has a geographically distributed team which whose effectiveness is made possible through the adoption of tools, especially with respect to geographically diverse programming as supported by GitHub; **Culture**: The team culture has a low resistance to change. Change is in fact promoted as a highly desirable characteristic and it is enabled at a technical level through the various tools and technologies identified in this paper; **Experience, Skill & Productivity**: The experience, skill and productivity of personnel are all at the upper end of the scale – what are sometimes referred to as premium people. The staff cohort in the company tend to be of high to very high core technical competency, with the result that individuals may operate fluidly and efficiently without the need for extensive training or up-skilling; **Turnover**: Personnel turnover is low (especially with key technical staff) with the result that continuity of technical excellence and know-how is high. There is therefore a reduced need for documented artefacts in relation to product architecture and process descriptions.
Reqs.	**Changeability:** Requirements are subject to frequent, sudden and significant change, a reality of operating in a fast moving and highly innovative market. As a result, a lean/agile approach to software development (such as was outlined in Section 3) is preferable for this setting.
Application	**Quality:** Operational product quality requirement is high and the technology adopted, including Continuous Integration systems, assists greatly in achieving product quality targets; **Type:** The applications under development and evolution (though requiring a high level of quality) do not need to be at the level of safety-critical software, nor are they directly affected by market regulation. As a result, a lean process, enabled via the technology and development stack, is suitable for the needs of this organization.
Technology	**Emergent:** The technology is emergent and innovative thus there is a high level of adoption of new technologies and tools to enable process initiatives. Embracing the rapid supporting technology offerings means that the process itself is subject to change as a result of technology strengths and limitations. This too is a feature of the context that has reduced the desirability of precise and extensive process descriptions which would continually need to be revisited as a result of the rapid pace of change.
Org.	**Size:** Organizational size is small – with the result that information exchange and communications can occur efficiently through video conferences or calls or face-to-face meetings thus enabling more agile/lean software approaches. Organization is experiencing substantial growth in staff headcount.

Operation	**End-Users:** Operational end-users of the software are open to changing requirements and rapidly evolving software systems. In fact, end-users are in this case demanding such capability from their software supplier in pursuit of competitive advantages in a fast moving market. This fact is key in shaping much of the process design – which is capable of working to a time and materials payments model and accommodates rapidly changing requirements.
Mgmt	**Expertise & Accomplishment:** Management expertise and accomplishment is high in key markets and product technology stacks, meaning that the business can pivot in harmony with the emerging technology without the risk of the business and technical strategic directions becoming discommoded.
Business	**Time to Market:** The company are in a fast moving market where the need for rapid delivery is paramount (smooth, regular and rapid delivery is enabled through the adoption of a microservices architecture along with deployment infrastructure such as Docker); **Business Drivers:** The company's business drivers are leveraged upon vanguard activities in key open source emerging technologies - technical excellence and high levels of innovation are key to differentiation and business development; **Payment Arrangements:** Payment terms tend to be time and materials based which supports the type of near-real feature elaboration with clients that is made possible by the microservices architecture.

5. DISCUSSION

Continuous software engineering (CSE) refers to the application of tools, which in many cases are open-source projects, to increase the deployment frequency of new releases of commercial-grade software. It is an emerging software development concept which has already been the subject of dedicated publication [10] but which we examine in the specific context of a microservices architecture. Through background research and industrial collaboration, we have established one working template for this concept: the continuous delivery pipeline enabled by microservice architectures together with container technology and innovative and full lifecycle integration of a suite of software development tools. While many of the tools have existed for some time, for example *Git* and *Jenkins*, it is the integrated use of sets of tools that delivers increased production and deployment speed. The result that the company has experienced that new features can be rapidly deployed to operational systems but also with confidence in relation to quality. This confidence is supported by the automation of many tasks that were once human intensive and which perhaps suffered from volatility as a result.

In the case of nearForm the adoption of tooling allows for regressively test existing code bases at a unit, feature and system level, while also validating new features or modifications through adaptation of earlier test routines. This can significantly reduce the risk of introducing new defects when delivering new features (or when resolving existing defects). Tooling can also address other aspects of software delivery that were once-upon-a-time dependent on direct human activity, such as configuration management, system building, and deployment. Collectively, software development, build and deployment tools can be chained together to bring about an *almost turnkey* software development and deployment framework.

Work in this area holds great promise for future eras of software development as it supports two vital software development concerns: Quality and Speed. A third vital consideration – Cost – presents a somewhat different proposition since there is a necessary investment in tooling infrastructure and knowhow, which itself impacts on initial setup speeds. The findings from our case study are limited as a result of not quantifying the investment required to establish the CSE infrastructure in the participant organisation. Although the tooling utilised is in the main available via open source projects, the cost of adapting and integrating these individual tools into a functioning tool chain, and the hardware and time required to do so, should be researched so that companies wishing to adopt a CSE approach can be better informed of the cost implications.

A further limitation in this work stems from the approach taken when adopting the situational factors reference model. Since the researchers involved in this work were also the creators of the Situational Factors reference model, it was possible to quickly progress the situational investigation in an iterative and fluid form in discussions with nearForm. This might not be the case for researchers not so familiar with the Situational Factors reference model and therefore similar related future research conducted by alternative researchers might first need to review and develop an understand of the reference model prior to conducting the empirical work, thereby introducing increased costs and timelines. In the fullness of time, the authors of this paper intend to provide greater assistance to future researchers seeking to apply the Situational Factors reference model but at this time, there is a shortfall in the supporting resources, e.g. published survey instruments, to assist third parties.

On the point of limitations, the Situational Factors reference framework itself may also have introduced some limitations to the study. It was chosen as it is the most comprehensive framework of its type presently published. However, with the relationship between software processes and situational contexts being so richly varied and complex [CITE 7], it will necessarily prove very challenging to attempt to model such phenomena in bounded frameworks. Consequently, factors not presently addressed in the Situational Factors reference framework may have been overlooked in this study. For the author's however, this reality does not diminish the importance of attempting to improve our understanding of the relationship between a process and its context, and we advocate that to be effective, process optimisation must be sensitive to the peculiarities of individual settings. This viewpoint may not meet with opposition from those familiar with software development, however, we are presently grappling with the complexity of the interactions between a process and its context. It is not our objective to fully quality all manners of interactions, but rather to begin to better understand the relationships. Each successive research undertaken with respect to this complex problem helps to gradually improve our understanding of the domain.

Our on-going work in the case study company is focused on a process formalization roadmap for our industrial partner who is presently witnessing business expansion – this roadmap will act to sustain the speed and quality characteristics of the present process while enabling significant growth in personnel. Certain challenges present when scaling this approach for larger numbers of development personnel. Firstly, since greater numbers of personnel will be introducing changes to build and deployment environments, it will be necessary to introduce mechanisms to avoid build and deployment conflicts and ensuing crashes. In some cases, this may be resolved through the introduction of more hardware, though hardware alone will not be sufficient to completely resolve this challenge. It is expected that further investment by the organization in build and deployment routines will be required, in effect this can be considered build and deployment architecture (or in our case, re-architecture).

A further challenge at the present time is the complexity involved from a tooling perspective – up to 20 independent tools may need to be employed to form a continuous software engineering and deployment chain (refer to Table 3 for examples of potential tools). In time however, this complexity can be reduced and more integrated toolsets will, we expect, appear on the software engineering landscape before long. It is the view of the authors that toolset integration and consolidation is not just beneficial, it is inevitable. We suggest that within 5-10 years from now, integrated continuous software engineering tooling (or tool management) will emerge, perhaps through extensions to existing integrated development environments and source code control systems.

Despite the clear benefits of CSE using microservice architectures, the authors do not suggest that it is a suitable approach for all software development, not least because it is a well-established position that no single software development process is perfectly suited to all software development undertakings [1]. For this reason, we examined the various factors of the situational context [5] that presented in our partner industrial organisation. We found that in this respect, the web-based non-safety-critical characteristic of the applications and the predilection of customers for rapid feature delivery imposed significant weight on the suitability (and success) of CSE in the case of our specific situation. For the time-being, other environments may not stand to benefit from the CSE approach, including the various safety-critical domains (e.g. medical, nuclear and automotive). It is also the case that there remain some challenges in terms of the reliability of the associated operationalised microservices architectures, since they present a particular set of volatility considerations when up-scaling – though these, the authors suggest, can be managed and reduced through mechanisms such as load balancing. In any case, where a client-vendor relationship is characterised by formal contracts, fixed-price arrangements and multiple integrators, the need for CSE may be significantly reduced. This however does not imply that all software development environments should not seek to exploit the advantages of automation through tooling throughout the software development lifecycle.

Targeted automation can help reduce traditionally repetitious human-led activities, for example in testing, building and deploying, and in so doing can offer favourable outcomes for all aspects of software development, including safety considerations.

Of interest is the interaction of the CSE approach that we have outlined with agile software development as underpinned by the agile manifesto. In this respect, the authors suggest that it is emerging developments in technology and tooling that are perhaps the primary reason that the CSE approach is even possible; an observation that may not be entirely congruent with the Agile Manifesto value of '*Individuals and interactions over processes and tools*'. It may therefore be the case that there is a need for a new manifesto to charter the path for CSE progression.

A final point we wish to acknowledge relates to the terminology we have chosen to describe the approach we have presented in this paper: *Continuous Software Engineering* (CSE). As Georg Hegel, the German Philosopher who offered valuable insights on language, observed "*truth is found neither in the thesis nor the antithesis, but in an emergent synthesis which reconciles the two*." Indeed, in other research, the authors have argued that inconsistency in relation to software process terminology application may be a somewhat underappreciated concern at this time [56], [57]. Thus, there are contemporary implications for our application of the word *continuous*. The authors wish to clarify that at the present time, the CSE approach that we have presented permits for commercial-grade software to perhaps be deployed to operational environments on an hourly basis and it is in this sense that we consider the approach to be *continuous*. Clearly, however, this is not akin to the real-time continuity we might associate with the passing of time. Nevertheless, the CSE approach that we have identified herein represents a considerable reduction in the granularity of new feature delivery to operational platforms when compared with many earlier software delivery timelines. Perhaps in the fullness of time, a concentrated focus on test, build and deployment automation may push the delivery granularity ever closer to real time continuous deployment, though clearly it can never quite get there. The term *asymptote* is perhaps appropriate as a description of the likely progression but such concerns are not likely to bear major influence on software engineers, certainly not at the level that they impact on our colleagues in mathematics.

REFERENCES

[1] P. Clarke, R. O'Connor, B. Leavy and M. Yilmaz. "Exploring the Relationship between Software Process Adaptive Capability and Organisational Performance," *IEEE Transactions on Software Engineering,* vol. 41, no. 12, pp. 1169-1183, 2015.

[2] R. V. O'Connor and P. Clarke. "Software process reflexivity and business performance: Initial results from an empirical study," *Proceedings of the 2015 International Conference on Software and System Process,* pp. 142-146, 2015.

[3] P. Clarke and R. V. O'Connor. "An empirical examination of the extent of software process improvement in software SMEs," *Journal of Software: Evolution and Process,* vol. 25, no. 9, pp. 981-998, 2013.

[4] G. Coleman and R. O'Connor. "Investigating software process in practice: A grounded theory perspective," *Journal of Systems and Software,* vol. 81, no. 5, pp. 772-784, 2008.

[5] P. Clarke and R. V. O'Connor, "The influence of SPI on business success in software SMEs: An empirical study," *Journal of Systems and Software,* vol. 85, no. 10, pp. 2356-2367, 2012.

[6] P. Clarke, R. V. O'Connor and M. Yilmaz, "A hierarchy of SPI activities for software SMEs: Results from ISO/IEC 12207-based SPI assessments," *Proceedings of the 12th International Conference on Software Process Improvement and Capability dEtermination (SPICE 2012),* pp. 62-74, 2012.

[7] B. Curtis. "Three problems overcome with behavioral models of the software development process," *11th International Conference on Software Engineering,* pp. 398-399, 1989.

[8] P. Clarke and R. V. O'Connor. "The situational factors that affect the software development process: Towards a comprehensive reference framework," *Journal of Information and Software Technology,* vol. 54, no. 5, pp. 433-447, 2012.

[9] Clarke, P., O'Connor, R. V., Leavy, B., "A complexity theory viewpoint on the software development process and situational context," *Proceedings of the 2016 International Conference on Software and System Process (ICSSP 2016),* 2016.

[10] J. Bosch, *Continuous Software Engineering: An Introduction.* Berlin, Germany: Springer International Publishing, 2014.

[11] M. Fowler and J. Highsmith, "The Agile Manifesto," *Software Development,* pp. 28-32, 2001.

[12] C. Ebert, P. Abrahamsson and N. Oza, "Lean Software Development," *IEEE Software,* vol. 29, no. 5, pp. 22-25, 2012.

[13] P. Clarke, P. Elger and R. V. O'Connor. "Technology enabled continuous software development," *Proceedings of the International Workshop on Continuous Software Evolution and Delivery,* pp. 48-48, 2016.

[14] ISO, *ISO 9001:2000 - Quality Management Systems - Requirements.* Geneva, Switzerland: ISO, 2000.

[15] W. S. Humphrey, *A Discipline for Software Engineering.* Reading, Massachusetts, USA: Addison-Wesley, 1995.

[16] S. Stolberg. "Enabling agile testing through continuous integration," *Proceedings of the 2009 Agile Conference,* pp. 369-374, 2009.

[17] M. Fowler and M. Foemmel, "Continuous integration," http://www. martinfowler. com/articles.

[18] J. Holck and N. Jørgensen, "Continuous integration and quality assurance: A case study of two open source projects," *Australasian Journal of Information Systems,* vol. 11, no. 1, pp. 40-53, 2003.

[19] E. Knauss, M. Staron, W. Meding, O. Söder, A. Nilsson and M. Castell, "Supporting continuous integration by code-churn based test selection," *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering,* pp. 19-25, 2015.

[20] J. Thönes, "Microservices," *IEEE Software,* vol. 32, no. 1, pp. 116-118, 2015.

[21] P. Feiler and W. Humphrey, *Software Process Development and Enactment: Concepts and Definitions.* CMU/SEI-92-TR-004. Pittsburgh, Pennsylvania, USA: Software Engineering Institute, Carnegie Mellon University, 1992.

[22] SEI, *CMMI for Development, Version 1.3.* CMU/SEI-2006-TR-008. Pittsburgh, PA, USA: Software Engineering Institute, 2010.

[23] L. McLeod and S. MacDonell. "Factors that affect software systems development project outcomes: A survey of research," *ACM Comput.Surv.,* vol. 43, no. 4, pp. 24:1-24:56, 2011.

[24] W. J. Orlikowski and J. J. Baroudi. "Studying Information Technology in Organizations: Research Approaches and Assumptions." *Information Systems Research,* vol. 2, no. 1, pp. 1-28, 1991.

[25] T. Dyba. "Contextualizing Empirical Evidence," *IEEE Software,* vol. 30, no. 1, pp. 81-83, 2013.

[26] T. Dyba, D. I. K. Sjoberg and D. S. Cruzes. "What works for whom, where, when, and why?: On the role of context in empirical software engineering," *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement,* pp. 19-28, 2012.

[27] ISO/IEC, *ISO/IEC 12207-2008 - Systems and Software Engineering – Software Life Cycle Processes.* Geneva, Switzerland: ISO, 2008.

[28] J. Ropponen and K. Lyytinen. "Components of software development risk: how to address them? A project manager survey," *IEEE Transactions on Software Engineering,* vol. 26, no. 2, pp. 98-112, 2000.

[29] M. Keil, L. Wallace, D. Turk, G. Dixon-Randall and U. Nulden. "An investigation of risk perception and risk propensity on the decision to continue a software development project," *The Journal of Systems & Software,* vol. 53, no. 2, pp. 145-157, 2000.

[30] M. Benaroch and A. Appari. "Financial Pricing of Software Development Risk Factors," *IEEE Software,* vol. 27, no. 5, pp. 65-73, 2010.

[31] L. Wallace, M. Keil and A. Rai. "How Software Project Risk Affects Project Performance: An Investigation of the Dimensions of Risk and an Exploratory Model." *Decision Sciences,* vol. 35, no. 2, pp. 289-321, 2004.

[32] L. Wallace and M. Keil. "Software project risks and their effect on outcomes," *Commun ACM,* vol. 47, no. 4, pp. 68-73, 2004.

[33] B. Boehm. "Software Risk Management: Principles and Practices," *IEEE Software,* vol. 8, no. 1, pp. 32-41, 1991.

[34] L. Putnam. "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," *IEEE Transactions on Software Engineering,* vol. 4, no. 4, pp. 345-361, 1978.

[35] A. J. Albrecht and J. E. Gaffney. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Transactions on Software Engineering,* vol. 9, no. 6, pp. 639-648, 1983.

[36] B. Boehm, B. Clark, E. Horowitz, A. Brown, D. Reifer, S. Chulani, R. Madachy and B. Steece, *Software Cost Estimation with Cocomo II.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[37] P. Xu and B. Ramesh. "Software Process Tailoring: An Empirical Investigation," *J. Manage. Inf. Syst.,* vol. 24, no. 2, pp. 293-328, 2007.

[38] J. Cameron. "Configurable development processes," *Commun ACM,* vol. 45, no. 3, pp. 72-77, 2002.

[39] T. Ferratt and B. Mai. "Tailoring software development," *SIGMIS-CPR '10: Proceedings of the 2010 Special Interest Group on Management Information System's 48th Annual Conference on Computer Personnel Research on Computer Personnel Research,* pp. 165-170, 2010.

[40] B. Boehm and R. Turner, *Balancing Agility and Discipline - A Guide for the Perplexed.* Boston, Massachusetts, USA: Pearson Education Limited, 2003.

[41] IEEE, *Guide to the Software Engineering Book of Knowledge (SWEBOK).* Los Alamitos, CA, USA: IEEE Computer Society, 2004.

[42] J. Corbin and A. Strauss, *Basics of Qualitative Research.* Thousand Oaks, CA, USA: Sage Publications Limited, 2008.

[43] B. Glaser and A. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Hawthorne, NY, USA: Aldine de Gruyter, 1976.

[44] P. Clarke and R. V. O'Connor, "Changing situational contexts present a constant challenge to software developers," *Proceedings of the 22nd European and Asian Conference on Systems, Software and Services Process Improvement (EuroSPI 2015), CCIS (Vol. 543),* pp. 100-111, 2015.

[45] M. Mora, O. Gelman, R. V. O'Connor and F. Alvarez "A conceptual descriptive-comparative study of models and standards of processes in SE, SwE, and IT disciplines using the theory of systems," *Emerging Systems Approaches in Information Technologies: Concepts, Theories, and Applications,* pp. 156-181, 2010.

[46] H. M. Kienle, "It's about time to take JavaScript (more) seriously," *IEEE Software,* vol. 27, no. 3, pp. 60-62, 2010.

[47] G. Arjun, C. Saftoiu and S. Krishnamurthi, "The essence of JavaScript," *European Conference on Object-Oriented Programming,* pp. 126-150, 2010.

[48] A. Nitze. "Evaluation of JavaScript quality issues and solutions for enterprise application development," *Proceedings of the 7th International Conference on Software and Systems Quality in Distributed and Mobile Environments,* pp. 108-119, 2015.

[49] E. DeBill, "Comparison of how many packages are available across different programming languages," www.modulecounts.com.

[50] M. Fowler and J. Lewis, "Microservices a definition of this new architectural term," http://martinfowler.com/articles/microservices.

[51] L. Bass, I. Weber and L. Zhu, *DevOps: A Software Architect's Perspective.* Boston, USA: Addison-Wesley Professional, 2015.

[52] Polar Capital Technology Trust plc., *Into the Eye of the Storm: Cloud Computing Makes Landfall, Annual Report.* UK: Polar Capital Technology Trust plc., 2016.

[53] S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," *Proceedings of the 2013 Agile Conference (AGILE 2013),* pp. 121-128, 2013.

[54] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software,* vol. 32, no. 2, pp. 50-54, 2015.

[55] J. Humble, C. Read and D. North, "The deployment production line," *Proceedings of the 2006 AGILE Conference (AGILE 2006),* pp. 113-118, 2006.

[56] P. Clarke, A. L. Mesquida Calafat, D. Ekert, J. Ekstrom, T. Gornostaja, M. Jovanovic, J. Johansen, A. Mas, R. Messnarz, B. Nájera Villar, A. O'Connor, R. V. O'Connor, M. Reiner, G. Sauberer, K. D. Schmitz and M. Yilmaz, "An investigation of software development process terminology," *Proceedings of the 16th International SPICE Conference,* pp. 351-361, 2016.

[57] P. Clarke, A. L. Mesquida Calafat, D. Ekert, J. J. Ekstrom, T. Gornostaja, M. Jovanovic, J. Johansen, A. Mas, R. Messnarz, B. Najera Villar, A. O'Connor, R. V. O'Connor, M. Reiner, G. Sauberer, K. D. Schmitz and M. Yilmaz, "Refactoring software development process terminology through the use of ontology," *Proceedings of the 23rd European and Asian Conference on Systems, Software and Services Process Improvement (EuroSPI 2016),* pp. 47-57, 2016.