The Journal of Systems and Software 128 (2017) 72–86

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Continuous Delivery: Overcoming adoption challenges ☆

Lianping Chen

Lianping Chen Limited, Dublin, Ireland

ARTICLE INFO

Article history:
Received 29 August 2016
Revised 23 December 2016
Accepted 16 February 2017
Available online 20 February 2017

Keywords:
Agile Software Development
Continuous Delivery
Continuous Deployment
Continuous Software Engineering
DevOps
Adoption

ABSTRACT

Continuous Delivery (CD) is a relatively new software development approach. Companies that have adopted CD have reported significant benefits. Motivated by these benefits, many companies would like to adopt CD. However, adopting CD can be very challenging for a number of reasons, such as obtaining buy-in from a wide range of stakeholders whose goals may seemingly be different from—or even conflict with—our own; gaining sustained support in a dynamic complex enterprise environment; maintaining an application development team's momentum when their application's migration to CD requires an additional strenuous effort over a long period of time; and so on. To help overcome the adoption challenges, I present six strategies: (1) selling CD as a painkiller; (2) establishing a dedicated team with multi-disciplinary members; (3) continuous delivery of continuous delivery; (4) starting with the easy but important applications; (5) visual CD pipeline skeleton; (6) expert drop. These strategies were derived from four years of experience in implementing CD at a multi-billion-euro company. Additionally, our experience led to the identification of eight further challenges for research. The information contributes toward building a body of knowledge for CD adoption.

1. Introduction

Continuous Delivery (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 (Chen, 2015a).

The CD approach is relatively new. It started gaining wide attention only in 2010, when Humble and Farley published the book titled "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation" (Humble and Farley, 2010). However, the CD approach has become increasingly popular, as shown by Google search trends (Fig. 1).

Companies that practice CD have reported huge benefits, such as significant improvements in time-to-market, customer satisfaction, product quality, release reliability, productivity and efficiency, and the ability to build the right product through rapid experiments (Chen, 2015a; Leppanen et al., 2015).

These benefits have motivated many companies to adopt CD. According to a recent survey of 600 software developers, managers, and executives in the United States and the United Kingdom, only 3% of the respondents said they had no plans to adopt CD (Perforce Software Inc., 2015).

However, implementing CD can be quite challenging (Chen, 2015a; Leppanen et al., 2015; Claps et al., 2015). Although CD as a goal (a target state) is no longer a new idea and has been well documented (Humble and Farley, 2010), the adoption journey for CD is not yet a smooth path. The journey itself is where the challenges lie, where many companies struggle, and where practitioners need help. Some of the fundamental challenges include the following:

- acquiring buy-in from a wide range of stakeholders whose goals may seemingly be different from—or even conflict with—those of the team driving the CD implementation;
- gaining sustained support in a dynamic complex enterprise environment;
- maintaining an application development team's momentum when their application's migration to CD requires an additional strenuous effort over a long period of time.

To help overcome the adoption challenges, I present six strategies we learned along our journey to CD at a multi-billion-euro company called Paddy Power. Paddy Power has been in the process of implementing CD for the past four years and, consequently, we have encountered and overcome many challenges. However, eventually, we achieved huge benefits (Chen, 2015a).

I hope the strategies described here can help fellow practitioners to overcome similar challenges on their way to achieving CD. The information also contributes toward building a body of knowledge for CD adoption.

☆ The work was done when the author was at Paddy Power PLC.

E-mail address: lianping.chen@outlook.com

http://dx.doi.org/10.1016/j.jss.2017.02.013

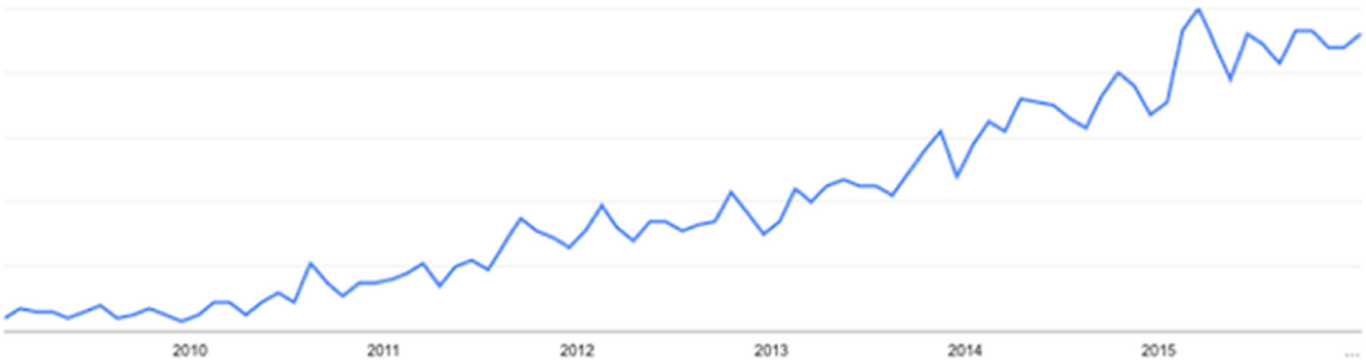

Fig. 1. Five-year Google trend for the search term "Continuous Delivery".

I also outline eight further challenges for research. Although not all of them are new, I provide elaboration on why these challenges are important for the industry, which on its own right can be useful input to researchers.

The remainder of this paper is organized as follows: Section 2 describes the context from which the strategies described here were derived. Section 3 describes the six strategies. I discuss related work in Section 4 and outline further research challenges in Section 5. Section 6 summarizes the paper.

2. The context

2.1. Paddy Power

Paddy Power is a rapidly growing company in the bookmaking industry. It offers its services in regulated markets, through betting shops, phones, and the Internet. It has an annual turnover of approximately €7 billion and employs approximately 5000 people. Paddy Power recently merged with Betfair, making it the world's largest public online betting and gaming company.

The company relies heavily on an increasingly large number of custom software applications that include websites, mobile apps, trading and pricing systems, live-betting-data distribution systems, and software used in betting shops. We developed these applications using a wide range of technology stacks, including Java, Ruby, PHP, and .NET. To run the applications, the company has an IT infrastructure consisting of thousands of servers in different locations. The applications are developed and maintained by the Technology Department, which employs approximately 500 people.

2.2. Continuous Delivery

At Paddy Power, we view Continuous Delivery as 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 (Chen, 2015a). The following sections highlight the characteristics of CD that are important to us.

2.2.1. Valuable software

Developing valuable software is a goal that has long been on the Agile manifesto (Beck et al., 2001). However, it is not an easy goal to achieve. Before adopting CD, some of our teams had been using an Agile method called Kanban (Anderson, 2010); however, due to delivery problems, we still had situations where a team had completed a feature but could not deliver it to production to obtain users' feedback. Consequently, they built additional functionalities on top of that feature, simply assuming it was useful. Unfortunately, when they finally delivered the software to the users, they found out that the feature was not what the users needed. Even worse, by that point, significant effort had been spent on the feature and the additional functionalities. An important objective of our CD implementation was to alleviate this problem. We want teams to build valuable software rather than spend time on features that users do not need.

2.2.2. Short cycles

We use "cycle time" to refer to the time from the conception of a user story to its production. Our cycle time used to be multi-month. Because of this long cycle time, the user stories that were completed earlier in the cycle had to wait for a long time. The value they could otherwise have generated was lost. Moreover, we were unable to obtain early user feedback. Therefore, shortening the cycle time is important.

2.2.3. Releasable at any time

Before implementing CD, applications were only releasable at the end of a long release cycle. This constraint caused problems. For example, business people could not obtain a release to users on demand in the middle of this long cycle, no matter how desperately they needed it; users were simply forced to wait until the next big release to obtain some important features—although those features might have been developed early, at the beginning of the cycle. To eliminate these problems, we want applications to be releasable at any time.

2.2.4. Reliable releases

Our emphasis on reliable releases stem directly from bad release experiences. Each time we were about to release, we had little confidence in the release reliability. Many times, these releases would be followed by P1 (priority 1) incidents (Rob, 2007), meaning that release activity was always full of uncertainty, failures, and stress. Reliable releases are a big point of distinction between our old practices and CD.

A term very similar to CD exists: Continuous Deployment. In academia, people tend to use it interchangeably with Continuous Delivery (Rodríguez et al., 2016).

However, many practitioners tend to clearly distinguish these two terms. The distinction is that under Continuous Deployment we deploy any change to production that passes a series of tests. In contrast, under Continuous Delivery, we ensure that the software can be reliably released at any time, but it is up to a human to decide when to release. In both Continuous Deployment and Continuous Delivery, deployment to production itself is automated. The difference lies in the trigger for making the deployment. One is triggered automatically, while the other is triggered by a human. According to this distinction, Continuous Delivery is compatible with a wide range of scenarios, but Continuous Deployment is suitable only under special conditions (O'Dell and Skelton, 2016). As do most companies, we mainly use Continuous Delivery.

Table 1
Summary of benefits achieved in our CD adoption.

Benefits	Description
Accelerated time to market	The cycle time from a user story's conception to production has decreased from several months to two to five days. Release frequency increased from once every one to six months to once a week.
Building the right product	Frequent releases let the application development teams obtain user feedback more quickly. This lets them work on only the useful features. If they find that a feature is not useful, they spend no further effort on it. This helps them build the right product.
Improved productivity and efficiency	Developers used to spend 20% of their time setting up and fixing their test environments. Now, the CD pipeline automatically sets up the environments. Operations engineers used to expend a few days' effort to release an application to production. Now, it can be done with a developer's click of a button.
Reliable releases	Releases have become reliable. The high level of stress and uncertainty associated with release activities has gone. Some teams even make releases at peak times, which they could never have considered before.
Improved product quality	The number of open bugs for applications has decreased by more than 90%. Previously, approximately 30% of the workforce was fixing bugs. Now, usually nobody is working on customer found bugs. Bugs are so rare that the teams no longer need a sophisticated bug-tracking system.
Improved customer satisfaction	Previously, distrust and tension existed between the users' department and the software development teams, owing to quality and release issues. The managers commented that the relationship has improved enormously.

However, due to their similarities, the strategies reported in this paper likely also apply to the adoption of Continuous Deployment.

2.3. CD at Paddy Power

Paddy Power's journey to CD started four years ago, led by a dedicated team of eight people (the CD team). This team is primarily responsible for building the Continuous Delivery Platform (our tool chain to support continuous delivery), guiding and supporting the development teams in moving their applications to continuous delivery. Due to the considerable diversity of the teams, the guiding and supporting activities vary significantly from team to team. The author is part of this CD team. The CD team controls neither the development teams nor their applications; thus, it can provide only guidance and support and leave the development teams themselves to actually make the necessary changes to their development practices and applications.

The changes that a development team needs to make also vary significantly from team to team. Some example changes include increasing code coverage by implementing automated tests so that the development team can be confident that when a code change passes all the stages of the CD pipeline it is ready to release; modifying an application to increase its testability so that automated tests can be written with an acceptable level of cost; increasing an application's modifiability, deployability, etc. as discussed in Chen (2015b); breaking big features down into smaller ones so that each can be finished in a short time (e.g., a week).

At present, we have implemented CD for 60 different applications. For each of these applications, a developer making a code commit automatically triggers a CD pipeline execution. The updated code undergoes a series of stages that check the quality of the code change. If the change fails to pass at any of the pipeline stages, the pipeline is aborted, and the developers are notified so they can fix the problems. If the updated code passes all the stages, it can be released to production with the click of a button (Chen, 2015a).

As a result of the CD implementation, we have achieved huge benefits, as summarized in Table 1 (these were reported in an earlier paper (Chen, 2015a)).

However, our journey to CD was not smooth. We encountered many challenges along the way. For example, implementing CD requires support from different stakeholders, but obtaining buy-in from them is quite difficult. CD implementation requires a considerable and continuous investment over a long period of time (Savor et al., 2016), but gaining sustained support in a dynamic complex enterprise environment is challenging. Moving complex applications to CD can take months or even years because of the number of changes that need to be made to the team's mindset and practices as well as to the application itself. Maintaining the team's momentum toward CD adoption under such circumstances is challenging.

When we were working through these fundamental challenges, we looked to the research literature for possible solutions. Unfortunately, we were unable to find information that specifically addressed how to overcome these CD adoption challenges. It would have been helpful if such guidance had been available because it could have saved us a great deal of time and prevented us from taking wrong turns during our exploration and quest for the right solutions.

I think other fellow practitioners that are in the process or plan to implement CD may have similar needs and will likely encounter similar challenges. Therefore, I decided to present the strategies we learned along our path to implementing CD. In the next section, I will describe each of the six strategies. Henceforward, this paper refers to these strategies as S1 through S6.

3. Strategies to overcome CD adoption challenges

3.1. Selling CD as a painkiller (S1)

Adopting CD requires making changes in many areas. Apart from automating build, test, and deployment activities, introducing CD also requires changes to architecture activities (Chen, 2015b), software development practices, organizational structure and culture, and so forth.

To make such substantial changes, we need support from a wide range of stakeholders including top management, the infrastructure team, the system engineering team, the security team, and the operations team.

However, gaining buy-in from all these groups is an enormous challenge because all of them have their own day-to-day work and their own goals and priorities, all of which can seemingly be different from or even conflict with the goal of our team—the team attempting to implement CD.

One strategy that we found useful is selling CD as a painkiller. Using this strategy, we identify each stakeholder's pain points and identify the ones that CD can help to solve. When we introduce CD to this stakeholder, we focus on explaining how and why CD can help to solve the identified pain points.

After the stakeholders realize how CD can help them with their pain points, they often become proponents of CD adoption. I provide two examples below.

Fig. 2. An example of a Kanban board with many stories piled up in the "ready for acceptance" column due to the unavailability of testing environments.

3.1.1. An example with software development teams

Software development teams form an important stakeholder in CD implementation. This is because, eventually, it is the software development teams who are responsible for following CD practices and using the CD pipelines. If we do not obtain buy-in from the software development teams, the CD adoption will fail.

It may seem easy to obtain buy-in from the software development teams, but in reality it is not straightforward. For example, our software development teams were often under time pressure. For some teams, overtime work was not uncommon. To squeeze time from their already busy schedules, we needed to give them a very strong motivation.

The first software development department where we implemented CD had approximately 70 employees. A software development team's size depends on the application's size and complexity. The teams range from two to twelve people; most teams have four to eight people. We analyzed the pain points they were experiencing and identified one major pain point that CD could solve that concerned testing environments. In this department, developers on average spent 20% of their time setting up or fixing their testing environments. This was a very boring and frustrating task.

In addition, because the testing environments were shared among teams, a team often needed to wait for one or two weeks for a testing environment to be released by other teams. For new projects requiring new machines, the teams often had to wait for a month before they could obtain the required testing environments from the system engineering and infrastructure teams.

Moreover, the unavailability of testing environments often cause user stories to pile up in the "ready for acceptance" column of the teams' Kanban boards (Anderson, 2010) (Fig. 2), causing further pain.

While these stories were waiting in the "ready for acceptance" column, developers would usually pick up another story to work on. Then, when the testing environments finally became available, the developers had to switch back to that story in the "ready for acceptance" column. This context switch incurred extra cost and frustration.

The wait time spent in "ready for acceptance" mode also caused delays because the development team was unable to obtain early user feedback on the waiting stories. Note that the company also loses the value during the delay period that would otherwise be generated if those waiting user stories were released.

CD can eliminate the problems described above. On the CD platform, a development team can create a CD pipeline for each application with support from the CD team. The CD pipeline can automatically spin up all the testing environments within a few minutes. Thus, the developers would no longer need to wait for weeks to obtain testing environments for a new project, nor would they need to spend time setting up and fixing the testing environments.

When the above benefits of CD were explained to the development teams, the developers in that department were keen to move to CD and happy to schedule the time needed to make the changes required on their side.

3.1.2. An example with security team

The security team is an important stakeholder. In our company, before any change can go to production, it must be approved by the security team. The security team's authority is justified. When a security vulnerability goes to production and the system is compromised, the negative effects cannot be undone. We can roll software changes back, but we cannot roll back any security damage that has already occurred. Thus, the company does not allow any change to go to production without the security team's approval.

Without buy-in from security team, we would not be able to implement the crucial step of fast delivery to production. With CD, we needed to release at a much higher frequency, changing from no more than six releases a year to at least one release per week

per application. This increased release frequency required more security checks to be conducted. However, considering the large number of applications, the security team's existing practices could not handle the new increased workload. Thus, we needed to make changes—and to make changes in the security area, we needed the security team's buy-in.

We talked to security team and figured out their pain points. CD could solve two of these important pain points. The first pain point concerned ensuring security policy. Although one of these policies was that every change had to go through security checks before it could enter production, because the security scans were performed manually, due to human mistakes, there were situations in which the code that was actually sent to production was not the same code that had been checked by the security team. The following is an example.

A team decided to schedule a release. The source code was tagged in the version control system. The security team checked that code and approved it. However, immediately after the security check, the team made some more changes to the code. The changed code was tagged again in the version control system, and when the release was executed, the code with this new tag (which had not been checked by the security team) was released to production.

The above situation looks like an error that should never happen. However, when the number of applications the security team needs to approve is large (several hundreds) and the process is manual, human errors inevitably happen occasionally.

The security team also had another pain point. There were not enough security engineers to run security checks frequently—for each code commit. For most software applications, the security check was often the last step in the release cycle. When the development team was in a hurry to release code to production (e.g., to adhere to an already slipped deadline), because the security team did not allow them to release until the security checks were finished, developers viewed the security team as an impediment to that release. In such cases, the security team was blamed, which created tension between the security team and the software development teams.

With CD, the execution of security checks is automated. The automation is built into one of the early stages of the CD pipeline. Thus, every code commit goes through the security check. If a security vulnerability is identified, the developers must fix it before they can proceed any further. This ensures that the security policy is always followed.

In addition, the CD pipeline produces an audit trail of the changes made for each code commit. The audit trail includes when the security scan was conducted, what was checked, and what the results of checking were.

This also solves the second pain point. Because the security check is run early and frequently, when the application is to be released, all the security problems have already been identified and fixed. Consequently, the development teams no longer need to wait for the security check, and the security team no longer gets the blame because they no longer stand in the developers' way at the last minute, when the development teams are keen to make the release.

After communicating these advantages to the security team, they became interested in collaborating. Then, after the security team had experienced the benefits from the initial few teams, they became proponents of CD and began to suggest that other teams move to CD as well.

I found that the pain points often differ from one stakeholder to another. Therefore, each stakeholder should be treated individually. I also anticipate that the pain points will differ from one organization to another. So, I do not claim that the examples above are what fellow practitioners will experience or should follow. However, I believe the general strategy is likely to be applicable.

3.2. Establishing a dedicated team with multi-disciplinary members (S2)

Establishing a dedicated team with multi-disciplinary members can help to smooth the adoption journey. I will explain why a dedicated team is needed and why the team should contain multi-disciplinary members.

3.2.1. Dedicated team

Implementing CD requires a dedicated team for two reasons. First, implementing CD requires a big effort. Building the CD platform involves a major effort; moving each individual team to CD also requires a significant amount of effort, particularly in a large organization. We have been implementing CD with an eight-person team for four years, amounting to 32 person-years of effort.

Second, without a dedicated team, the CD implementation initiative often suffers from disruptions that make it difficult to make visible progress. For example, before establishing a dedicated CD team, some engineers had been trying to implement CD on a part-time basis. However, no real progress was made, because when the engineers were not dedicated to CD implementation, they were too easily pulled off-task to work on other value streams. Often, for example, the engineers were shifted to work on a late project to help maintain the deadline.

Consequently, we decided to establish a dedicated team. No matter how busy other teams were, the engineers in this team were not switched to other work. Only after this dedicated team was established, did we start to see steady progress in the CD implementation.

3.2.2. Multi-disciplinary members

Implementing CD requires a team with multi-disciplinary members—another lesson that we learned during our journey. When our team was initially established, all the team members had a software development background. However, we soon discovered that developing a CD platform needs not only developer skills but also system administration and operations skills. To implement the CD platform, we would need to bring in many new tools. System administrators are good at installing and maintaining these tools. A large component of the CD platform is the capability to automatically deploy an application, but automating the deployments needs a great deal of operating system configuration, application configuration, and network configuration knowledge. System administrators and operation engineers have deep knowledge in these areas, whereas developers often do not.

Because our team initially had only engineers with a software development background, we were forced to perform these tasks despite the fact that our skills did not match the tasks, which caused the CD adoption to take a longer time than necessary.

Apart from the skills mismatch for performing deployment tasks, communication with the system engineering team (which is responsible for system administration and operations), was not smooth. Because of our developer background we spoke a different language than system engineers. Moreover, we had a lack of appreciation for their concerns, which caused tension between both teams.

To solve this problem, we decided to add a team member who had many years of experience and deep knowledge in system administration and operations. After this member joined our team, we not only accelerated progress in those work areas but also eased the tension with the system engineering team.

When we started to move the more complex applications, whose development process was less amenable to CD, we found

that before these teams can effectively practice CD and obtain the benefits of our CD platform, their existing software development practices and processes had to be improved. Expertise in process improvement was very valuable in driving changes in the practices of the development teams, moving them toward the practices required by CD. Having learned from experience, we added a member with process improvement expertise.

Now, our team consists of members who have backgrounds in development, system administration, operations, and process improvement. This diversity not only provides us with the required skills to complete work in different areas but also eases communications with other related teams in the organization.

We have gone through a long process to reach our current multi-disciplinary team; therefore, I recommend that any organization planning to adopt CD should consider building a multi-disciplinary team from the beginning of the journey. The exact mixture of skills may vary from one company to another, but at minimum, I suggest including the development and operations disciplines.

3.3. Continuous delivery of continuous delivery (S3)

As mentioned above, implementing CD can require a significant effort. We have already spent 32 person-years over four years. To finish something of this size over a long period of time, we need sustained support.

However, getting sustained support in today's dynamic business environment is challenging. Over the last four years, our CEO has changed; our CTO has changed; our manager's manager has changed five times; our organization has been re-structured three times. Whenever such changes occurred, the value of the CD implementation initiative was questioned.

To maintain sustained support and prevent the CD implementation initiative from being killed, we could not afford to deliver the value of CD only after 32 person-years of effort had been spent. We needed to demonstrate the value of CD by delivering it early in the process. Although the total effort so far is 32 person-years, we began onboarding the first application to CD when we were only several months into the CD adoption journey.

With the benefits demonstrated early on, the CD implementation survived many business and organizational changes and maintained sustained support from the company. This support enabled us to develop more features of the CD platform to onboard additional project types. For example, our support for technology stacks has been extended from only Java to Ruby, PHP, Javascript, Golang (Donovan and Kernighan, 2015), Scala (Odersky et al., 2016), and .NET.

Apart from delivering value to the company early to maintain sustained support, this strategy can also help to reduce the risk and difficulty of CD platform adoption by development teams. We found that when introducing CD, the development teams needed to change the ways in which they typically worked. In other words, the way development teams work needs to be aligned with the new process and the new tools.

Using this strategy, the development teams gradually align their work habits to the new process and new tools in small steps each time we release a new feature of the CD platform. This gradual shift makes the alignment easier.

At the same time, it also provides us with the opportunity to get feedback from the application development teams, for example, to let us know whether a feature we added to the CD platform is useful, easy to use, and so on. We often make adjustments to the CD platform based on developers' feedback. These adjustments align the CD platform to the developers' needs and, thus, simplify its adoption by the development teams.

If we had started to onboard the development teams only after we had completed the entire CD platform, the development teams might have had to make lots of changes to align with the new platform. Large gaps between the way the teams typically working and the CD platform will hinder its adoption.

Thus, continuous delivery of continuous delivery not only helps to address the challenges in maintaining sustained support by the company but also reduces the risk and difficulty that software development teams will have when adopting the CD platform.

Is it possible to decompose the entire CD adoption work into smaller pieces in such a way that when we finish one piece we can gain immediate benefits from it? This is difficult but possible, according to our experience. The following are some dimensions along which one can slice the CD adoption work into smaller pieces.

Technology Stack: In a large organization, multiple programming languages are used. Each programming language has its own community which typically has developed other tools and technologies to create a full technology stack. The applications developed using different technology stacks use different tools and approaches for compiling, building/packaging, distributing, testing, configuring, deploying, etc. To support all these technology stacks used in a company requires significant effort.

We can build the support for the technology stacks incrementally. When we build the support for one technology stack, we can start onboarding applications using that technology stack to CD, and we can start gaining the benefits of CD from these applications.

Operating System Type: A large organization often uses different operating systems that can be quite different in terms of installation, authentication management, licensing management, scripting environment (e.g., Bash in Linux vs. Powershell in Windows) and so on. To support all these operating systems can require a significant effort.

We can build the support for various operating systems one by one. When we finish building the support for one operating system, we onboard applications running on that operating system, and we can start gaining the benefits of CD from these applications.

Environment (Pipeline Stage): In a large organization, a piece of software needs to be tested in several environments and, finally, be deployed to the production environment. Different environments have different requirements. For example, the requirement for an early-stage testing environment will be different from the requirement for the production environment. Building the supports for automatically provisioning and deploying to all these environments requires a significant effort.

We can build the support for these environments incrementally. When we build the support for one environment, we save the costs for manually setting up and maintaining that environment, which is an immediate benefit to the company.

Degree of Automation: Automating the provisioning of environments can be done in different degrees. A complete provisioning of an environment could include many things such as deploying an application to a virtual machine, creating the virtual machine to which the application needs to be deployed, creating the network that the virtual machine is part of, configuring the network access control list and firewall rules to allow the newly created virtual machine to communicate with other machines, and configuring the different layers of load balancers to make the newly installed application visible to its users. Fully automating all the above requires a significant effort and can take a long time because it not only requires technical changes but also changes to the way that an organization manages infrastructure.

We can start by automating the deployment of applications to pre-provisioned virtual machines. Although the provision of the

virtual machines is not automated, we still can begin to benefit from reducing the configuration drifts caused by manual deployments and save the effort of doing manual deployments. We can then proceed to automate virtual machine provision and gain the benefits of not having to create, configure, and maintain the virtual machines manually, finally, we can then automate network provision to gain the corresponding benefits—and so on and so forth.

The above are some heuristics that one can use to decompose the CD adoption work into smaller pieces each of which can be completed in a shorter time, while gaining the benefits from each completed piece immediately. However, one caveat to remember is that we need a clear blueprint to ensure that everything will fit together nicely at the end.

3.4. Starting with easy but important applications (S4)

Gaining sustained support is a big challenge, as I already mentioned above, and it is especially challenging in the early stages of the CD implementation. Many people in the organization tend to question CD's effectiveness, its suitability to the organization, the feasibility of its implementation, and whether the CD implementation work has a higher priority than many other work streams that also need people to work on them.

To justify the effort involved in CD implementation, avoid the adoption initiative being killed, and achieve sustained support, we have found another strategy to be quite useful. This strategy involves selecting the first few applications to migrate CD.

Choosing the initial applications is an important topic. In a large organization, both applications and teams can be very diverse. Applications can differ widely in size, complexity, architectural styles, and so forth. The teams differ in size, experience, development practices, culture, their relationships with users, and so on. There are often a large number (hundreds) of applications to select from.

If the wrong application is selected, it may be impossible to demonstrate sufficient CD value to justify the effort and priority required by the CD implementation initiative. Alternatively, selecting the wrong application may make it impossible to complete its migration to CD within the time frame expected by the important stakeholders. In either of these situations, achieving sustained support will become even more difficult.

Based on our experience, the best strategy is to start with the applications that are easy to migrate—but at the same time, to select those that are important to the business. I will explain the rationale below.

3.4.1. Easy to migrate

If the first application that we select to migrate to CD is difficult, it may take too long for us to complete the migration and demonstrate the value of CD. In the worst case, the migration may fail, which—apart from its impact on morale and momentum—will put the entire CD implementation initiative at risk.

Applications that are difficult to migrate to CD do exist in most large organizations. Migrating these applications often requires significant changes to both the application itself and the development team's practices. The changes to the application itself may include such things as architectural changes (Chen, 2015b) and test coverage improvements; the changes to the development team's practices can include changes in architecture practices, testing activities, build processes, and so on. Making these changes can require significant time and effort from the team.

For example, Paddy Power had a large monolithic application. To deploy the application, we needed to take the whole system down. Changing one component often required changes in many other components due to ripple effects. The automated test coverage was poor and many tests were flaky. The quality of the software was mainly ensured by a long round of manual testing at the end of each long release cycle. To move this application to CD, we needed to improve the architecture to satisfy the deployability, modifiability, and testability required by CD (Chen, 2015b). We also needed to increase the number of automated tests to a degree until the team would be confident about the application's production readiness when it passed the tests.

Making the above changes required huge efforts. This level of effort can impact the pace of delivery of features required by the business stakeholders. Juggling and balancing the CD implementation efforts and the pace of delivery of features is a challenge and sometimes required fighting for the priority of the tasks to make the changes relevant to the CD implementation. Thus, the time required to move this application to CD and showcase CD's benefits could be very long. During this long process, the teams could easily lose the momentum required to make the necessary changes for the move to CD. When we attempted to move this application to CD, progress was slow and maintaining momentum was difficult. We made little progress even over a period of more than 6 months.

If we had selected this application as the first application to move to CD, we would have faced challenges from the management team regarding the benefits and applicability of CD to our organization. The CD implementation initiative might have been killed off at an early stage. Fortunately, we began migrating this application only after having demonstrated the value of CD with several other projects, so we survived through this tough case.

As a heuristic, the applications that are easy to move to CD are often new applications with open-minded teams working on them. New applications do not contain as much legacy things and are more likely to use modern architectural patterns. It is generally easier for teams working on new applications to (re-) design the architecture in a way that is amenable to CD (Chen, 2015b). Moreover, an open-minded team can be easily influenced to adopt the required development practices. For example, we ask new teams to use Kanban, Test Driven Development (TDD) (Beck, 2002), Behavior Driven Development (BDD) (Wynne and Hellesoy, 2012), and other, similar development methodologies.

3.4.2. Important to business

The selected application needs to be important to the company. This helps to secure the required resources, demonstrate clear and unarguable value, and raise the visibility of CD in the organization.

An important application tends to attract resources. Moving an application to CD requires resources, e.g., we need people to spend time to make the necessary changes to the application and implement new development practices. Most organizations have more tasks than resources available to perform them; consequently, it is common to see projects competing for resources. An important application will have a higher probability of acquiring the resources required, whereas unimportant applications will often fail to gain the required resources.

We sometimes need to add additional features to the CD platform to support a new type of application. Such additions can require significant development effort and a sizable investment in infrastructure. For example, to move an application developed using .NET, we needed to make significant changes to our CD platform and make a sizable investment in infrastructure. This is because .NET is quite different from the other applications we already supported. First, .NET applications run on the Windows operating system, and Windows works quite differently from the Linux operating systems we were already supporting in terms of licenses, security, and so on. Thus, we needed to make significant changes to our CD platform. Windows operating systems also require more disk space; therefore, we needed to purchase additional hardware. However, because the application was critical to the business and had top priority, we were able to obtain the required resources.

Equally important, moving an important application to CD tends to generate value that is clearly more than the cost of CD implementation. We have applications that generate millions in revenue monthly for the company. We also have applications that generate tens of thousands in revenue monthly. CD can significantly reduce the time to market and improve product quality (Chen, 2015a). Therefore, moving important applications to CD demonstrates the significant value of CD adoption. Anybody can easily conclude that the value generated is clearly much higher than the cost of CD adoption. While moving less important applications to CD may still generate more value than the cost of CD adoption, the value will be less apparent and less dramatic than moving important applications. Being able to demonstrate clear value generated by CD adoption is important to convince top management to continue to invest in CD adoption.

An important application tends to have higher visibility in the organization. So, when this application moves to CD, the success and the value gained from CD adoption will be more visible to the rest of the company and can help drive interest in having more teams move to CD. Thus, migrating important applications helps to create momentum for CD adoption across the whole company. If the application is not important to the company, even if it is successfully moved to CD, people may not care, leaving the CD implementation team as a small hidden corner of the organization.

However, the strategy of starting with easy but important applications does not mean ignoring, avoiding, or not caring about other applications. Instead, it is about onboarding the applications in the right sequence to obtain sustained support for the CD adoption initiative and maximize the value of CD implementation.

After we had moved several applications to CD, the value of CD became concrete and tangible in the company. Motivated by these demonstrated benefits, other teams that were not initially amenable to CD began to show interest in adopting it and became willing to make the changes that they would not have been willing to make several months earlier. By that time, the management team had also already noted the benefits of CD, creating a growing motivation for moving additional applications to CD. This is a better point at which to tackle the tough applications—those that are difficult to move to CD.

3.5. Visual CD pipeline skeleton (S5)

Applications that are difficulty to move to CD do exist in most large organizations, and there are often many of them. We have a large number of such tough applications as well, but we cannot simply ignore them in the process of CD adoption. Although it may not make sense to move all these tough applications to CD, many are critical to the company and, therefore, it is worthwhile to migrate them to CD so that new application releases can enter the market faster and provide competitive advantages for the company.

As discussed earlier, some applications are difficult to move to CD because they require substantial changes to both the development team's current practices and to the software applications themselves, such as increasing code coverage by automated tests so that the development team can be confident that when a code change passes all the stages of the CD pipeline it is ready to release, modifying an application to increase its testability so that automated tests can be written with an acceptable level of cost, and increasing an application's modifiability, deployability, etc. as discussed in Chen (2015b). For these applications, a complete migration to CD can take months or even years to complete.

It is worth noting that our team does not own the applications; we cannot make changes to these applications. Instead, we need each application's development team to make the needed changes. Because we do not have control over the code for these applications, we can only influence and facilitate the improvements.

Under this circumstance, a major challenge is to maintain the application development teams' momentum for CD adoption over a long period of time. We have sometimes seen teams start a migration to CD with strong motivation, but that momentum gradually wanes over time.

One strategy we found useful is to give each team a visual CD pipeline skeleton that shows the full CD pipeline view—but not all the stages of the pipeline are implemented.

Fig. 3 shows an example. The first row in the figure represents the CD pipeline. Each gray box in the row represents a stage in the pipeline. The pipeline runs from left to right. The transition from one stage to another is automatic except for the transition to user acceptance and the transition to production. Each code change triggers an execution of the pipeline. Each execution of the pipeline is represented as a row under the first row. (For brevity, I listed only three executions.) Each box in the row indicates the status of a stage. The initial color is blue, which indicates the stage has not been executed. When the code change passes a stage of the pipeline, the box that represents the stage turns green. When the code change fails to pass the stage, the box that represents the stage turns red.

In this example, the team does not have automated acceptance tests and performance tests. The application's deployment cannot be fully automated either due to a problem in their architecture. Making improvements in the above areas can take a long time. We give the team a visual pipeline skeleton that has the same look and feel as the final pipeline, but the acceptance and performance stages do not do anything (you can see it takes less than one second to complete). Similarly, the user acceptance and production stages have not yet been implemented.

Although the pipeline is not complete, it helps to maintain the team's CD adoption momentum. Its visual nature makes the abstract concept of CD tangible. The pipeline sets a clear and visible vision and roadmap so that all the stakeholders can track the progress of ongoing improvements.

The pipeline is a thing that developers see every day, and it acts as a reminder. Every time the developers see the pipeline view, the gaps in their CD adoption are clear. For example, they can easily see that the acceptance and performance stages are empty. We have often seen developers start discussions about how they could fill the gaps by writing automated acceptance tests and performance tests.

Because the pipeline is already in place, the teams can begin enjoying the benefits of any improvements they make toward CD immediately. For example, whenever they write some automated tests and add them, the pipeline will begin to automatically execute the tests after each code commit. The immediate feedback from these benefits will further encourage the developers to write more automated tests, helping to maintain the CD adoption momentum.

3.6. Expert drop (S6)

Another strategy to address tough applications is expert drop. With expert drop, we assign an expert in CD adoption to join the team facing difficulties in migrating their tough application to CD. This expert will help to drive the changes from within the team in areas such as team practices, team culture, and applications architecture to help ease the migration.

The person in this expert role is a senior member of a team, possesses deep knowledge about CD, and has been through the full CD migration process. As mentioned earlier, by the time we started to onboard the tough applications, we had already successfully moved several important applications to CD; therefore,

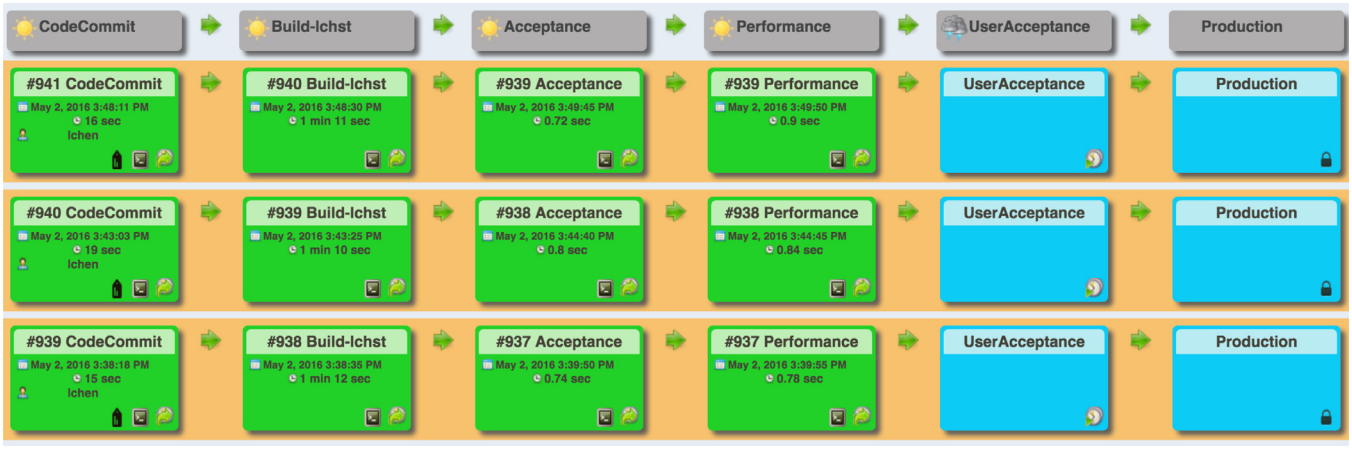

Fig. 3. An example visual CD pipeline skeleton.

the experts were selected from the teams who had already gone through the migration process.

The use of expert drop is important for some applications; trying to implement the required changes from outside a team may not always work. For example, to accelerate the adoption of CD by application development teams, the top management team started an initiative. A group of experts was formed who would visit the teams and help them to make the required improvements. When a team was selected, training sessions were organized to teach the team the "correct" way of working. Every day, there was a stand-up meeting with all the experts and the team members to push the improvements forward.

This group of experts had been helping one team with a tough application make changes for several months. However, the team members were resistant. Little progress had been achieved. When we talked to the members of the team, we found that they could not understand why they needed to change; they were happy with their old way of working. They also commented that the external experts did not understand their problems.

That led us to change the strategy. Rather than pushing the team to change using a group of external experts, we assigned an expert to join the team as a senior member. Then, the expert started to drive the changes from within the team.

Because he was a part of the team on a day-to-day basis, he discussed, communicated, and interacted with the members of the team. In the process, he obtained deeper insights into the problems the team faced, allowing him to explain to the team members why they needed to change their existing work processes in a way that the team members understood. He was also able to explain various CD practices in the context of a specific problem faced by the team.

The results were positive. The team members began to embrace CD. They started to exhibit interest in moving some CD adoption tasks forward. Gradually, the momentum to make changes was built. The release frequency increased, and the requirements and testing practices became more aligned with the requirements of CD.

Why did the team members collaborate with the senior member when she was a member of the team but were reluctant to collaborate with her when she was an external expert? Team culture is one factor. Overall our organization promotes collaboration, which is one of the company's five core values. However, in a large organization, in a real industrial setting, teams with culture problems can exist. Although the number of such teams is small, when we aim at implementing CD across the whole organization, we must deal with them.

Team culture is just one aspect of the difficulties. In some cases, due to the complexities and diversities of the situations, the experts, looking from outside the team, could not correctly and deeply understand the problems. Thus, the advice they gave may have sounded good but were not actually feasible to implement.

The strategy of expert drop to drive changes from within the team can be useful for migrating tough applications, especially when imposing the migration from outside the team does not work well. Using this strategy, we have successfully migrated some very tough applications to CD, some of which had already failed an initial migration attempt using an external expert group.

4. Challenges for research

What we know now about CD adoption is only the tip of the iceberg. There are many more specific challenges to address. Below, I identify some important challenges that need researchers' help. These challenges are important because they are likely to be faced by many practitioners trying to adopt CD. Any improvements (research breakthroughs) in these areas could result in significant benefits to practitioners.

4.1. Organizational challenges

I mentioned organizational challenges in my earlier paper (Chen, 2015a). However, apart from the points discussed in Chen (2015a), I recently noticed another area in which evidence-based results are important. Organizational structures can be a significant barrier when implementing CD (Chen, 2015a). We must change the existing organizational structure to create successful implementations. However, there is no target organizational structure that is evidence-based and validated. Therefore, new organizational structures are often based solely on opinion. As an example, our previous CTO re-structured the organization in one way. A new CTO took the job recently and had a different opinion. He re-structured the organization in yet another way—in fact, he reverted some of the changes the previous CTO had implemented.

Although there is a large corpus of studies on organizational structures (Organizational Structure, 2016), we still need evidence-based results regarding the connections between CD and organizational structures. For example, what are the possible structures that suit CD? What are the advantages and disadvantages of each? Which structures are most suitable for which situations? What are the organizational design guidelines? We need this information to make evidence-based decisions.

4.2. Process

I also mentioned process challenges in my earlier work (Chen, 2015a). However, to the best of my knowledge, these challenges have not received sufficient attention from researchers. Therefore, here I provide additional rationale on why I think processes are an important topic of study.

Most large organizations have many processes that must be followed to release a unit of software to production. For example, in our organization, we needed to create a change ticket, place the change request on the agenda of the next Change Advisory Board (CAB) meeting (Rob, 2007), present the change at the meeting, receive CAB approval, confirm the deployment window, and so forth.

This is particularly problematic at very large enterprises with long histories. For example, I have met practitioners from these large enterprises who report that they needed to provide more than one hundred different documents and follow numerous processes to release any code change to production. Worse, these processes could take several weeks or even months. Such lengthy processes prevent companies from adopting CD. We must change the existing processes.

However, changing these processes is very difficult because they were originally put in place to ensure quality and security. Many of them were introduced to prevent previous incidents from happening again. In fact, many organizations employ people whose sole job is to ensure that everyone follows the official processes.

Nevertheless, to achieve CD, we must change the processes. Although some practitioners have reported anecdotal stories, no systematic approach or strategies are available for other practitioners to follow. This is a gap that remains to be filled.

4.3. Tools

Implementing CD requires a new toolchain because, as reported by several studies, existing tools are inhibitory in achieving CD (Olsson et al., 2012; Rissanen and Münch, 2015; Marschall, 2007; Zhu et al., 2015; Debbiche et al., 2014; Volker et al., 2013). We call this new toolchain the "CD Platform".

This platform is a fairly complex software system. It must integrate with a large number of other tools, including version control systems, artifact repositories, release management systems, application and machine configuration management systems, deployment tools, identity management tools, and so on. It also needs to have the following quality:

- Reliable, because all the developers will rely on it to carry out their work.
- Secure, because production systems will be in danger if it is subverted (Bass et al., 2015).
- Scalable, to support the continually increasing number of applications in an organization.
- Easy to use, so that a team can implement a new deployment pipeline in a self-service manner; otherwise the CD team will likely be unable to support a large number of application development teams.

However, compared to research efforts spent on tools for testing, requirements, architecture, etc., tools for CD have not attracted enough attention from researchers. As a practitioner, if I am going to buy a tool for CD, there is no evaluation framework that I can use to evaluate and select from the available tools and vendors. If I am going to build the tool in-house, there is no reference architecture available. When I design the pipeline stages, there are no guidelines or best practices to follow either. Therefore, I would like to draw more researchers' attention to this area.

4.4. Infrastructure

Implementing CD requires infrastructure support. By infrastructure, I mean computing power (e.g., CPU), storage, network, memory, etc., all of which are typically managed by the infrastructure team. Infrastructure has been mentioned as a challenge by some studies. For example, Claps et al. (2015) reported that the need for adding additional computing power, network bandwidth, and memory to CI servers is a challenge. They consider adding more resources as a strategy (Claps et al., 2015).

However, according to our experience, the preceding problems do not provide a full picture of the infrastructure challenges. The way the infrastructure is managed is likely to be an even greater challenge.

Traditionally most infrastructure management is largely manual. When a team needs a server, they make a request to the infrastructure team. The infrastructure team creates a Virtual Machine (VM) for them. The system engineering team installs and configures the appropriate operating system on that VM. The network team adds the VM onto the network. At this stage, the new VM does not have access to other machines on the network because, by default, communications are blocked by the firewall for security reasons. The team then creates a change request to update the firewall rules. The security team reviews and approves the change request. The network team executes the change request. The VM is then handed over to the application operations team, which installs and configures the application on the VM. The system engineering team configures the load balancer to make the application on the VM available to external customers. Because many teams are involved in this process, the whole process can take several weeks or a month. This method of managing infrastructure is a hindrance to companies attempting to achieve CD.

To support CD, ideally we need the capability to automatically provision the required infrastructure. In other words, we need to change to Infrastructure as a Service (IaaS) (Mell and Grance, 2011).

However, making a company's entire infrastructure available as a service is a big transformation for any organization, and the transformation itself can take a very long time (e.g., a couple of years). For example, in our company, due to the large cash investment required, the decision was made very carefully. Selecting the technologies, investigating the feasibility, and evaluating different vendors took a long time. When a decision was finally made, implementing it also required a long time. Apart from the technical implementation, the workflow of infrastructure management also needed to be updated.

According to informal conversations with attendees at the DevOps Showcase Dublin conference (http://conferences.unicom.co.uk/fundamentals-of-devops/), the Delivery of Things World conference (http://deliveryofthingsworld.com/en/), and the QCon London conference (https://qconlondon.com/), I found that for many companies, infrastructure management methods are not ideal for CD. Many organizations have begun to discuss or even implement IaaS, but not all of them have actually turned their entire infrastructure into a service that can be managed and consumed by APIs.

Even moving to a public cloud such as AWS may not be as straightforward as one might wish. For example, as reported by an attendee of the DevOps Showcase Dublin conference, an Irish ecommerce company had to move their services from AWS back to their internal data center because the application required significant amounts of I/O Operations (IOPs). The price for IOPs on AWS was too expensive; running the services in their own data center proved to be a much cheaper option.

Consequently, implementing CD is a problem when the desired infrastructure support is not ready. This is an important topic to address because many companies that would like to adopt CD are in similar situations.

However, there are no studied approaches, strategies, or guidelines in the literature to address this problem. Part of our future work is to derive and formulate some guidelines and strategies from the experiences of companies that have managed to overcome their infrastructure problems.

4.5. Legacy systems

According to our experience, there are probably more legacy systems than green field projects in the industry (Bennett, 1995). These legacy systems' architectures are usually not amenable to CD (Chen, 2015b). Moreover, the teams working on these legacy systems usually have "legacy" culture, practices, and mindsets that would need to be changed to migrate the applications to CD. Unfortunately, making these changes is very difficult.

Considering the large number of such applications in the industry, studying the strategies and analyzing the approaches for moving legacy systems to CD is very important because any improvements in this area could have significant practical impacts. However, little, if any, work has been reported that specifically seeks to understand and address this important type of software (Rodríguez et al., 2016).

4.6. Architecting for continuous delivery

We found that application architecture can be a key barrier for CD adoption. However, discussions of CD have primarily focused on build, test, and deployment automation; insufficient attention has been paid to application architecture. More research is needed to provide deeper understanding and guidance to help practitioners (re-) architect their applications for CD (Chen, 2015b).

4.7. Continuous testing of non-functional requirements

While unit tests and acceptance tests have been extensively discussed and widely practiced in CD, testing non-functional requirements has received considerably less attention. However, unit tests and acceptance tests are mainly intended to ensure functional requirements.

Performing only functional testing might be acceptable for some applications that are insensitive and tolerant of non-functional requirements. However, there are many applications that are strict and sensitive to non-functional requirements. For example, we have many applications that have strict performance requirements measured in milliseconds.

There are extra challenges in testing non-functional requirements. For example, with functional requirements we can easily determine whether a test passes or fails. However, with some non-functional requirements, defining the criteria for passing or failing a test is less straightforward. For example, with a performance test, the results can vary from one test to another due to unavoidable variations in the testing environments, such as network conditions.

Testing non-functional requirements may require a larger scale testing environment than functional testing. For example, scalability testing often needs a larger scale environment than acceptance testing.

Automating tests for non-functional requirements can be quite different from writing tests for functional requirements. For example, when testing reliability, the tests need to create different failure scenarios such as killing some application nodes or creating network partitions between nodes, etc.

Thus, testing of non-functional requirements in the context of CD is an important challenge. I believe this is another interesting area for researchers to explore.

4.8. Test execution optimization

When practicing CD, software releases occur much more frequently. Before CD, an application might typically have fewer than six releases per year. Under CD, an application releases at least once a week and sometimes multiple times per day.

To ensure the quality of each release, tests are executed more frequently. Unit tests are executed for every code commit. Acceptance tests are executed for each code commit that passes the unit tests.

This significantly increased frequency of test execution makes test optimization especially important and useful. If even a small saving can be achieved for each test execution, when multiplied by the large number of executions, the total savings quickly become significant.

Moreover, the potential saving we could get from each test execution is significantly bigger in a CD context than in a traditional context. Before CD, when we released every six months, the number of changes that each release included was large. In each test execution, to cover all the changes, it is likely that many test cases needed to be executed. In contrast, under CD, the number of changes in each release is small. To cover this significantly smaller number of changes, it is likely that only a much smaller set of tests needs to be executed. However, we still execute the tests in the old way—in other words, we still execute all the test cases even when only one line of code has been changed. Thus, there is plenty of room for test optimization under CD.

Researchers have proposed some test selection techniques and test prioritization techniques, but these proposals have not been widely adopted in industry (Singh et al., 2012; Engström et al., 2010; Gligoric, 2015). We need to identify the gaps and develop new approaches or improve the existing approaches to fill those gaps.

In addition, existing test selection or test prioritization techniques mainly focus on optimizing an individual test execution. The high frequency of test executions in CD could make optimizing across multiple adjacent test executions a viable option to explore. For example, by knowing that we normally run tests ten times a day, is it possible to reduce a single test case's repetition across all these ten test executions?

Finally, as the popularity of CD increases, test optimization techniques also become more important in practice. I would like to draw more researchers' attention to test optimization as well.

5. Related work

In this section, I highlight the differences of this paper in relation to existing literature and to my previous work.

5.1. Differences from existing strategies

Many works have reported challenges and solutions in adopting CD (Leppanen et al., 2015; Claps et al., 2015; Olsson et al., 2012; Karvonen et al., 2015; Rissanen and Münch, 2015; Marschall, 2007; Zhu et al., 2015; Debbiche et al., 2014; Puneet, 2011; Noureddine and Foutse, 2014; Laukkanen et al., 2015; Rogers, 2004; Sekitoleko et al., 2014; Krusche and Alperowitz, 2014; Adams et al., 2015; Souza et al., 2015; Feitelson et al., 2013; Rahman et al., 2015; Neely and Stolt, 2013), including a very recent systematic literature review (Laukkanen et al., 2017). However, none of the existing literature on CD has included the strategies I reported in this paper.

Nevertheless, I believe these strategies are useful because we found that putting some of the existing strategies into action is challenging in its own right. For example, Claps et al. (2015) reported that it is important to "ensure that top management implements a strategy to push the need to implement the CD process" as a strategy to adopt CD. However, putting this strategy into

Table 2
Summary of strategies to overcome adoption challenges.

Strategy	Description
Selling CD as a painkiller	Identify each stakeholder's pain points that CD can solve, and sell CD as a painkiller to that stakeholder. This strategy helps to achieve buy-in from the wide range of stakeholders that a CD implementation requires.
Dedicated team with multi-disciplinary members	Without a dedicated team, it can be hard to progress because employees are often assigned to work on other value streams. A multi-disciplinary team not only provides the wide range of skills required for CD implementation but also smooths the communication with related teams.
Continuous delivery of continuous delivery	Organize the implementation of CD in a way that delivers value to the company as early as possible, onboarding more projects gradually, in small increments and eventually rolling out CD across the whole organization. This strategy helps justify the investment required by making concrete benefits visible along the way. Visible benefits, in turn, help to achieve the sustained company support and investment required to survive the long and tough journey to CD.
Starting with easy but important applications	When selecting the first few applications to migrate to CD, choose the ones that are easy to migrate but that are important to the business. Being easy to migrate helps to demonstrate the benefits of CD quickly, which can prevent the implementation initiative from being killed. Being important to the business helps to secure the required resources, demonstrates clear and unarguable value, and raises the visibility of CD in the organization.
Visual CD pipeline skeleton	Give a team a visual CD pipeline skeleton that has the full CD pipeline view but with empty stages for those they cannot implement yet. This helps to build up a CD mindset and maintain the momentum for CD adoption. The pipeline skeleton is especially useful when the team's migration to CD requires a large effort and mindset changes over a long period of time.
Expert drop	Assign a CD expert to join tough projects as a senior member of the development team. Having the expert on the team helps to build the motivation and momentum to move to CD from inside the team. It also helps to maintain momentum when the migration requires a large effort and a long period of time.

action is a challenge because the CD implementation team is typically not in a position to order top management to implement any strategy to push the CD implementation process forward. Instead, the real challenge for us was how to acquire their buy-in and sustained support, so that we could convince them to carry out that action.

As another example, Claps et al. (2015) also reported providing more resources to a product's CI servers as a strategy to adopt CD. However, acquiring more resources was a true challenge for us. Without buy-in by the related stakeholders and their sustained support, we were unable to obtain the required resources.

As a third example, several papers have reported moving to more flexible and modular designs as a strategy to adopt CD (Claps et al., 2015; Olsson et al., 2012; Debbiche et al., 2014; Noureddine and Foutse, 2014; Sekitoleko et al., 2014; Laukkanen et al., 2017). However, moving a legacy system to a flexible and modular design is a major challenge and may require significant effort over a long period of time. The CD implementation team does not own the applications and cannot make changes to them. Instead, our challenge was to convince the responsible teams to make the changes and find ways to maintain their momentum until those changes were complete. The strategies reported in this paper aim at addressing these challenges so that we can make the required changes happen.

5.2. Differences to change management literature

Adopting CD is a change process. Change management has been studied and a large corpus of change management literature exists (Todnem By, 2005; Change Management, 2016).

For some of the strategies I reported, a generalized form can be found in the general change management literature. For example, we could say a more generalized form for S3 (continuous delivery of continuous delivery) is the strategy of making changes incrementally rather than all at once. Thus, the strategy may seem too general.

However, I have described these strategies in the specific context of CD adoption. I also provided experience-based advice specific to CD adoption. For example, in S2 (establishing a dedicated team with multi-disciplinary members), I provided experience-based advice on what kinds of disciplines should be included in the multi-disciplinary team and why we need each of those disciplines. For another example, in S3 (continuous delivery of continuous delivery), we provided experience-based heuristics on how to decompose the whole CD adoption effort into smaller pieces in such a way that we can complete each piece in a short time and gain the benefits from that piece immediately. As a third example, in S4 (starting with easy but important applications), I provided heuristics on what applications are easy to migrate.

Some strategies I reported are specific to CD adoption. For example, S5 (visual CD pipeline skeleton), for which we need a CD pipeline to use it.

Thus, although literature on change management exists, it does not cover the specifics of introducing CD to an organization. I believe other practitioners will find the CD-specific details I provided in this paper useful.

5.3. Differences from existing reported challenges

The strategies reported in this paper aim at addressing the challenges of obtaining buy-in from a wide range of stakeholders whose support is required when implementing CD, gaining sustained support to continue the often lengthy and tough journey toward CD, and maintaining the momentum of application teams when their CD migrations require huge efforts over long periods. These challenges are not reported in the recent Systematic Literature Review (SLR) (Laukkanen et al., 2017), which systematically reviewed the existing literature to identity problems and solutions in adopting CD.

I also outlined a few further research challenges that are not reported in the above-mentioned SLR, such as process challenges, CD platform development challenges, and infrastructure management challenges. For those challenges that were mentioned before, in this paper I provided extra new descriptions that enrich our understanding of the challenges.

5.4. Differences from my previous work

My previous work (Chen, 2015a) did not include any of the strategies reported in this paper. In terms of challenges, four of the further challenges for research were not included in my previous paper. For those challenges that were mentioned in (Chen, 2015a), I provided new and additional descriptions in this paper.

In addition, this work was based on four year's longitudinal first-hand experiences in implementing CD across an entire large organization. The first-hand experiences allowed us to feel the real pain and develop a richer and deeper understanding. Over a long time helped us to encounter challenges that are revealed only at the far end of the journey to CD (e.g., processes were not seen as challenges initially). Implementing CD across the entire large organization helped us to see challenges relating to team and application varieties (e.g., there are very tough applications and teams). Our context is different from that of other papers and has helped us to identify some of the challenges that have not been reported before.

6. Summary

Continuous Delivery (CD) has become a popular software development approach. Motivated by the huge benefits reported (Chen, 2015a; Leppanen et al., 2015), many practitioners want to implement CD in their organizations (Perforce Software Inc., 2015). However, implementing CD can be very challenging (Chen, 2015a; Leppanen et al., 2015; Claps et al., 2015).

To help overcome CD adoption challenges, this paper reported six strategies derived from our four-year journey to implement CD at a multi-billion-euro company. These strategies are summarized in Table 2.

I hope these strategies can help fellow practitioners to overcome similar challenges and reap the huge benefits of CD (Chen, 2015a; Leppanen et al., 2015). The information also contributes toward building a body of knowledge concerning CD adoption.

For many practitioners, CD is a clear goal, but achieving that goal is difficult. What we currently know about CD adoption is just the tip of the iceberg. There are many more specific challenges to address, such as the challenges discussed in this paper regarding organizational structure, processes, tools, infrastructure, legacy systems, architecting for CD, continuous testing of non-functional requirements, and test execution optimization. I invite more researchers to investigate these increasingly important topics.

Acknowledgments

I thank Dave Farley of Continuous Delivery Ltd for reviewing these strategies and my colleagues at Paddy Power for their support. I also thank the reviewers for their thoughtful comments and suggestions to improve this paper. This paper is based on my experience at Paddy Power. It represents only my own views and does not necessarily reflect those of the employer.

References

Adams, B., Bellomo, S., Bird, C., Marshall-Keim, T., Khomh, F., Moir, K., 2015. The practice and future of release engineering: a roundtable with three release engineers. IEEE Software 32, 42–49.

Anderson, D.J., 2010. Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press.

Bass, L., Holz, R., Rimba, P., Tran, A.B., Zhu, L., 2015. Securing a deployment pipeline. Presented at the 2015 IEEE/ACM 3rd International Workshop on Release Engineering (RELENG).

Beck, K., 2002. Test Driven Development: By Example. Addison-Wesley Professional, Boston, MA.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., et al., 2001. Manifesto for agile software development.

Bennett, K., 1995. Legacy systems: coping with success. IEEE Software 12, 19–23.

Change Management (2016). Available: https://en.wikipedia.org/wiki/Change_management (Accessed 30.10.16).

Chen, L., 2015a. Continuous delivery: huge benefits, but challenges too. IEEE Software 32, 50–54.

Chen, L., 2015b. Towards architecting for continuous delivery. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 131–134.

Claps, G.G., Berntsson Svensson, R., Aurum, A., 2015. On the journey to continuous deployment: technical and social challenges along the way. Inform. Software Tech. 57, 21–31.

Debbiche, A., Dienér, M., Berntsson Svensson, R., 2014. Challenges when adopting continuous integration: a case study. In: Jedlitschka, A., Kuvaja, P., Kuhrmann, M., Männistö, T., Münch, J., Raatikainen, M. (Eds.). In: Product-Focused Software Process Improvement, vol. 8892. Springer International Publishing, pp. 17–32.

Donovan, A., Kernighan, B.W., 2015. The Go Programming Language. Addison-Wesley.

Engström, E., Runeson, P., Skoglund, M., 2010. A systematic review on regression test selection techniques. Inform. Software Tech. 52, 14–30.

Feitelson, D.G., Frachtenberg, E., Beck, K.L., 2013. Development and deployment at Facebook. IEEE Internet Comput. 17, 8–17.

Gligoric, M.Z., 2015. Regression Test Selection: Theory and Practice. University of Illinois at Urbana-Champaign.

Humble, J., Farley, D., 2010. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley, New York.

Karvonen, T., Lwakatare, L.E., Sauvola, T., Bosch, J., Olsson, H.H., Kuvaja, P., et al., 2015. Hitting the target: practices for moving toward innovation experiment systems. In: Fernandes, M.J., Machado, J.R., Wnuk, K. (Eds.), Software Business: 6th international conference, ICSOB 2015, Braga, Portugal, June 10-12, 2015, proceedings. Cham: Springer International Publishing, pp. 117–131.

Krusche, S., Alperowitz, L., 2014. Introduction of continuous delivery in multi-customer project courses. In: Presented at the Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014). Hyderabad, India.

Laukkanen, E., Itkonen, J., Lassenius, C., 2017. Problems, causes and solutions when adopting continuous delivery—a systematic literature review. Inf. Software Technol. 82, 55–79 2//.

Laukkanen, E., Paasivaara, M., Arvonen, T., 2015. Stakeholder perceptions of the adoption of continuous integration – a case study. In: 2015 Agile Conference (AGILE), pp. 11–20.

Leppanen, M., Makinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mantyla, M.V., et al., 2015. The highways and country roads to continuous deployment. IEEE Software 32, 64–72.

Marschall, M., 2007. Transforming a six month release cycle to continuous flow. In: 2007 Agile Conference (AGILE), pp. 395–400.

Mell, P., Grance, T., 2011. The NIST definition of cloud computing. Computer Security Division, Information Technology Laboratory. National Institute of Standards and Technology, Gathersburg, MD.

Neely, S., Stolt, S., 2013. Continuous delivery? Easy! Just change everything (Well, maybe it is not that easy). In: 2013 Agile Conference (AGILE), pp. 121–128.

Noureddine, K., Foutse, K., 2014. Factors impacting rapid releases: an industrial case study. In: Presented at the Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14). Torino, Italy.

O'Dell, C., Skelton, M., 2016. Continuous Delivery with Windows and .NET. O'Reilly.

Odersky, M., Spoon, L., Venners, B., 2016. Programming in Scala: A Comprehensive Step-by-Step Guide, third ed. Artima Press.

Olsson, H.H., Alahyari, H., Bosch, J., 2012. Climbing the "Stairway to Heaven" – a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: 2012 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 392–399.

Organizational Structure. Available: https://en.wikipedia.org/wiki/Organizational_structure. (Accessed 30.10.16).

Perforce Software Inc, 2015. Continuous Delivery: The New Normal for Software Development Available: http://info.perforce.com/continuous-delivery-survey-report.html.

Puneet, A., 2011. Continuous SCRUM: agile management of SAAS products. In: Presented at the Proceedings of the 4th India Software Engineering Conference (ISEC '11). Thiruvananthapuram, Kerala, India.

Rahman, A.A.U., Helms, E., Williams, L., Parnin, C., 2015. Synthesizing continuous deployment practices used in software development. In: 2015 Agile Conference (AGILE), pp. 1–10.

Rissanen, O., Münch, J., 2015. Transitioning towards continuous delivery in the B2B domain: a case study. In: Lassenius, C., Dingsøyr, T., Paasivaara, M. (Eds.), Agile Processes in Software Engineering and Extreme Programming: 16th International Conference, XP 2015, Helsinki, Finland, May 25-29, 2015, Proceedings. Cham: Springer International Publishing, pp. 154–165.

Rob, A., 2007. Effective IT Service Management: To ITIL and Beyond!. Springer-Verlag, New York, Inc.

Rodríguez, P., Haghighatkhah, A., Lwakatare, L.E., Teppola, S., Suomalainen, T., Eskeli, J., et al., 2016. Continuous deployment of software intensive products and services: a systematic mapping study. J. Syst. Software (January).

Rogers, R.O., 2004. Scaling Continuous Integration. In: Eckstein, J., Baumeister, H. (Eds.), Extreme Programming and Agile Processes in Software Engineering: 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 68–76.

Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., Stumm, M., 2016. Continuous deployment at Facebook and OANDA. In: Presented at the Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16). Austin, Texas.

Sekitoleko, N., Evbota, F., Knauss, E., Sandberg, A., Chaudron, M., Olsson, H.H., 2014. Technical dependency challenges in large-scale agile software development. In: Cantone, G., Marchesi, M. (Eds.), Agile Processes in Software Engineering and Extreme Programming: 15th International Conference, XP 2014, Rome, Italy, May 26-30, 2014. Proceedings. Cham: Springer International Publishing, pp. 46–61.

Singh, Y., Kaur, A., Suri, B., Singhal, S., 2012. Systematic literature review on regression test prioritization techniques. Informatica 36.

Souza, R., Chavez, C., Bittencourt, R.A., 2015. Rapid releases and patch backouts: a software analytics approach. IEEE Software 32, 89–96.

Todnem By, R., 2005. Organisational change management: a critical review. J. Change Manage. 5, 369–380.

Volker, G., Christoph, H., Christian, J., 2013. Security of public continuous integration services. In: Presented at the Proceedings of the 9th International Symposium on Open Collaboration (WikiSym '13). Hong Kong, China.

Wynne, M., Hellesoy, A., 2012. The Cucumber Book: Behaviour-Driven Development For Testers and Developers. Pragmatic Bookshelf, Dallas, TX.

Zhu, L., Xu, D., Tran, A.B., Xu, X., Bass, L., Weber, I., et al., 2015. Achieving reliable high-frequency releases in cloud environments. IEEE Software 32, 73–80.

Lianping Chen is an independent researcher and consultant, currently works as chief software engineering expert of R&D System Engineering & Process Quality Department at Huawei Technologies. His interests include DevOps, continuous delivery, software requirements and architecture, and software product lines. Chen received an MS in software engineering from Northwestern Polytechnical University.