Defining Metrics for Continuous Delivery and Deployment Pipeline

Timo Lehtonen[1], Sampo Suonsyrjä[2], Terhi Kilamo[2], and Tommi Mikkonen[2]

[1]Solita Ltd, Tampere, Finland
timo.lehtonen@solita.fi
[2]Tampere University of Technology, Tampere, Finland
sampo.suonsyrja@tut.fi, terhi.kilamo@tut.fi, tommi.mikkonen@tut.fi

Abstract. Continuous delivery is a software development practice where new features are made available to end users as soon as they have been implemented and tested. In such a setting, a key technical piece of infrastructure is the development pipeline that consists of various tools and databases, where features flow from development to deployment and then further to use. Metrics, unlike those conventionally used in software development, are needed to help define the performance of the development pipeline. In this paper, we address metrics that are suited for supporting continuous delivery and deployment through a descriptive and exploratory single case study on a project of a mid-sized Finnish software company, Solita Plc. As concrete data, we use data from project "Lupapiste", a web site for managing municipal authorizations and permissions.

Keywords: Agile measurements, continuous deployment, lean software development.

1 Introduction

Software development, as we know it today, is a demanding area of business with its fast-changing customer requirements, pressures of an ever shorter time-to-market, and unpredictability of market [1]. Lean principles, such as "Decide as late as possible", have been seen as an attractive way to answer to these demands by academics [2]. With the shift towards modern continuous deployment pipelines, releasing new software versions early and often has become a concrete option also for an ever growing number of practitioners.

As companies, such as Facebook, Atlassian, IBM, Adobe, Tesla, and Microsoft, are going towards continuous deployment [1], we should also find ways to measure its performance. The importance of measuring the flow in lean software development was identified already in 2010 by [3], but with the emergence of continuous deployment pipelines, the actual implementation of the Lean principles has already changed dramatically [4]. Further on, measuring has been a critical part of Lean manufacturing long before it was applied to software development [5]. However, the digital nature of software development's approach

to Lean (ie. continuous deployment pipelines) is creating an environment, where every step of the process can be traced and thus measured in a way that was not possible before. Therefore, the need for a contemporary analysis of what should be tracked in a continuous deployment pipeline is obvious to us.

In this paper, we address metrics that are suited for supporting continuous delivery and deployment through a descriptive and exploratory single case study on a project of a mid-sized Finnish software company, Solita Plc (`http://www.solita.fi`). As case studies investigate the contemporary phenomena in their authentic context, where the boundaries between the studied phenomenon and its context are not clearly separable [6], we use concrete data from project "Lupapiste", or "Permission desk" (`https://www.lupapiste.fi`), a web site for managing municipal authorizations and permissions. The precise research questions we address are the following:

> **RQ1:** Which relevant data for practical metrics are automatically created when using a state-of-the-art deployment pipeline?
> **RQ2:** How should the pipeline or associated process be modified to support the metrics that escape the data that is presently available?
> **RQ3:** What kind of new metrics based on automatically generated data could produce valuable information to the development team?

The study is based on quantitative data and descriptions of the development processes and the pipeline collected from the developer team. Empirical data of the case project was collected from information systems used in the project, including a distributed version control system (Mercurial VCS) and a monitoring system (Splunk).

The rest of this paper is structured as follows. In Section 2, we address the background of this research. In Section 3, we introduce our case project based on which the research has been conducted. In Section 4, we propose metrics for continuous delivery and deployment pipeline. In Section 5, we discuss the results of case study and provide an extended discussion regarding our observations. In Section 6 we draw some final conclusions.

2 Background and Related Work

Agile methods – such as Scrum, Kanban and XP to name a few examples – have become increasingly popular approaches to software development. With Agile, the traditional ways of measuring software development related issues can be vague. The outcome of traditional measures may become dubious to the extent of becoming irrelevant. Consequently, one of the main principles of Agile Software Development is "working software over measuring the progress" [7].

However, not all measuring can be automatically considered unnecessary. Measuring is definitely an effective tool for example for improving Agile Software Development processes [8], which in turn will eventually lead to better software. A principle of Lean is to cut down waste that processes produce as well as parts of the processes that do not provide added value [9]. To this end, one should first

recognize the current state of a process [3]. This can be assisted with metrics and visualizations, for instance. Therefore, one role for the deployment pipeline is to act as a manifestation of the software development process and to allow the utilization of suitable metrics for the entire flow from writing code to customers using the eventual implementation [10].

Overall, the goal of software metrics is to identify and measure the essential parameters that affect software development [11]. Mishra and Omorodion [11] have listed several reasons for using metrics in software development. These include making business decision, determining success, changing the behavior of teammates, increasing satisfaction, and improving decision making process.

2.1 Continuous Delivery and Deployment

Continuous delivery is a software development practise that supports the lean principles of "deliver as fast as possible" and "empower the team". In it the software is kept deployable to the staging and production environments at any time [12, 13]. Continuous delivery is preceded by continuous integration [14, 15] where the development team integrates its work frequently on a daily basis. This leads to a faster feedback cycle and to benefits such as increased productivity and improved communication [15–17]. Similarly, "the final ladder " — continuous deployment — requires continuous delivery. So, continuous deployment [18, 19] takes one step further from delivery. In it software is automatically deployed as it gets done and tested. Taking continuous deployment to the extreme would mean deployment of new features directly to the end users several times a day [20, 21]. Whether software is deployed all the way to production, or to a staging environment is somewhat matter of opinion [18, 22] but a reasonable way to differentiate between delivery and deployment in continuous software development is the release of software to end users. Delivery maintains a continuously deployable software, deployment makes the new software available in the production environment.

Regardless of actual deployment, continuous software development requires a deployment pipeline (Figure 1) [10], which uses an automated set of tools from code to delivery. The role of these tools is to make sure each stakeholder gets a timely access to the things they need. In addition, the pipeline provides a feedback loop to each of the stakeholders from all stages of the delivery process. An automated system is not about software going into production without any operator supervision. The point of the automated pipeline is that as the software progresses through it, different stages can be triggered for example by operations and test teams by the click of a button.

2.2 Agile Metrics

In [8] the authors categorize agile metrics used in industry into metrics relating to iteration planning and tracking, motivation and improvement, identifying process problems, pre-release and post-release quality, and changes in the processes or tools. The metrics for iteration planning offered help with prioritization

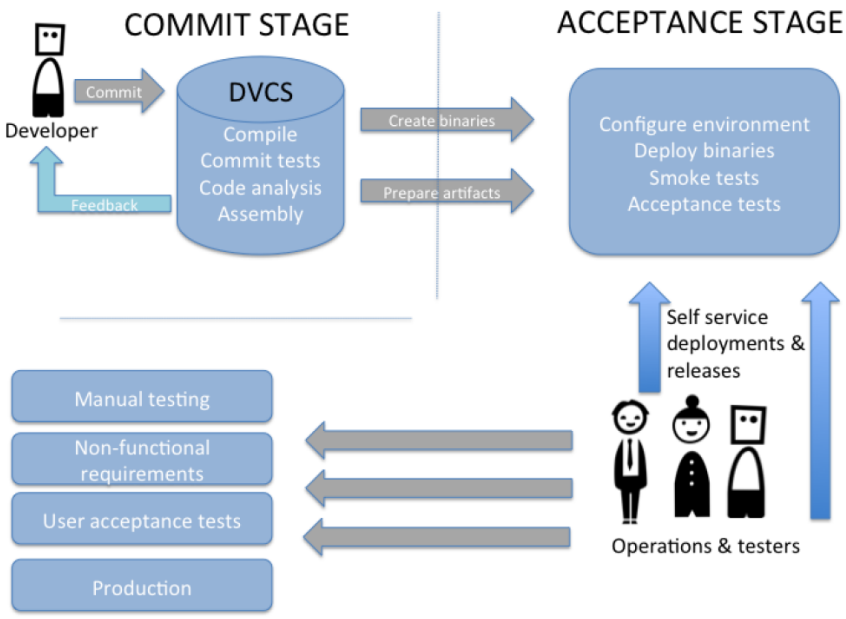

Fig. 1. Anatomy of a Deployment Pipeline according to [10].

of features. These include estimation metrics for measuring the size of features, the revenue a customer is willing to pay for a feature, and velocity of the team in completing a feature development. Iteration tracking include progress metrics such as the number of completed web pages, story completion percentage, and again velocity metrics. In the category of motivation and improvement, approaches such as visualizing the build status and showing the number of defects in monitors were found to lead into faster build and fix times. Using metrics such as lead time and story implementation flow assist in identifying waste and in describing how efficiently a story is completed compared to the estimate. Pre-release quality metrics were found to be used for making sure the product is tested sufficiently and for avoiding integration fails. Post-release quality metrics measure attributes such as customer satisfaction and customer responsiveness. These can be evaluated for example with the number of defects sent by customers, change requests from customers, and customer's willingness to recommend the product to other potential customers. For the final category of metrics for changes in processes or tools, sprint readiness and story flow metrics were found to change company policies to having target values for metrics.

In [11] a more general approach in categorization of agile metrics is used. The authors define the core agile metrics to include product, resource, process, and project metrics. Of these, the product metrics deal with size, architecture,

structure, quality, and complexity metrics. Resource metrics are concerned with personnel, software, hardware, and performance metrics. Process metrics deal with maturity, management, and life cycle metrics, and project metrics with earned business value, cost, time, quality, risk, and so on. Each of these sub-metrics can define a range of additional metrics such as velocity, running tested feature, story points, scope creep, function points, earned business value, return on investment, effort estimates, and downtime. The researchers also point out that teams should invent metrics as they need such, and not use a metric simply because it is commonly used – this might result in data that has no value in the development.

Kunz et al. [23] claim that especially source-code based product metrics increase quality and productivity in agile software development. As examples, the researchers present Number of Name-Parts of a method (NNP), Number of Characters (NC), Number of Comment-Lines (CL), Number of Local Variables (NLV), Number of Created Objects (NCO), and Number of Referring Objects (NRO). All in all, the researchers emphasize the early observation of quality to keep the software stable through the development process.

In their 2009 book [9] the Poppendiecks emphasize the customer-centricity in metrics. They present examples of these including time-to-market for product development, end-to-end response time for customer requests, success of a product in the marketplace, business benefits attributable to a new system, customer time-to-market, and impact of escaped defects.

2.3 Lean Metrics

As lean methods have been developed originally for manufacturing, there are obviously collections of corresponding metrics. For instance, the following has been proposed [3]: Day-by-the-Hour (DbtH) measures the quantity produced over the hours worked. This should correspond to the same rate of customer need. Capacity utilization (CU) is the amount of work in progress (WIP) over the capacity (C) of the process. An ideal rate is 1. On-time delivery (OTD) is presented as the number of late deliveries over the number of deliveries ordered. Moreover, such metrics or signals that help the involved people to see the whole, are mentioned in [24].

Petersen and Wohlin [3] present cost efficiency (CE), value efficiency (VE), and descriptive statistics as measurements for analyzing the flow in software development. A possible way of measuring CE is dividing lines of code (LOC) by person hours (PH). However, they point out how this cost perspective is insufficient as value is assumed to be created always by investment. The increase in LOC is not always value-added as knowledge workers are not machines. On the other hand, VE = (V(output) - V(input)) / time window. V(output) represents the final product, and V(input) the investment to be made. This type of measuring takes value creation explicitly into account, and therefore it can be a more suitable option.

Overall, according to van Hilst and Fernandez [25] there are two different approaches to evaluating efficiency of a process considering Lean ideals. These

views apply models from queuing theory, in which steps of a process are seen as a series of queues. Work advances from queue to queue as it flows through the process, and process performance is then analyzed in terms of starvation(empty queues) and bottlenecks(queues with backlogs). The first approach is to look at a workstation and examine the flow of work building up or passing through. At the same time, the activities on the workstation are studied to see how they either add value or impede the flow. On the contrary, the second approach follows a unit of work as it passes through the whole process. In that case, the velocity of this unit is studied. Considering these two approaches, van Hilst and Fernandez [25] describe two metrics: Time-in-process and work-in-process. Work-in-process is corresponding with the first approach as it describes the amount of work present in an intermediate state at a given point in time. The second approach is measured with time-in-process describing the time needed for a unit of work to pass through the process. For an optimal flow, both of these need to be minimized.

Finally, Modig [24] takes an even deeper look into measuring flow efficiency. This metric focuses on the unit, which is produced by an organization, (*flow unit*) and its flow through different workstations. Flow efficiency describes how much a flow unit is processed in a specific time frame. Higher flow efficiency is often better from the flow units point of view. For instance, if a resource processes the flow unit for one hour, and then the flow unit is placed to a queue of two hours, and then another resource starts to process it for three hours, the flow efficiency is 4 / 6 = 66%. If the length of the queue is shortened to for example half an hour, the flow efficiency is higher (4 / 4.5 = 89%).

3 Case Lupapiste

An industrial single case study was conducted to investigate measuring a state-of-the-art pipeline within a two months time frame of actual development work. The case project, and its deployment pipeline are introduced in the following.

3.1 Description of the Case Project

The application "Lupapiste", freely translated "Permission Desk", is a place for the citizens and companies in Finland to apply for permissions related to the built environment, available at `https://www.lupapiste.fi`. The project was started in 2012, and the supplier of the system is Solita Plc., a mid-sized Finnish ICT company. The end users of the system consist of various stakeholders, with various interests. The Environmental Ministry of Finland owns the project code and acts as a customer in some new functionalities needed to the system.

At the time of research (Fall 2015), the project team consisted of seven developers, a user experience (UX) designer and a project manager that are co-located in a single workspace at the supplier. On the management level there are four more persons in different roles. The team is cross-functional and has also DevOps [26] capabilities. Some team members have an ownership of certain parts of the

system, but the knowledge is actively transferred inside the team by changing the areas continuously and for example by applying agile practices like pair reviewing of code to spread out the knowledge in a continuous manner. The team takes use of a custom Lean Software Development process that includes features from agile Scrum-based processes with lean heritage. The process is ongoing and has no sprints, but milestone deadlines for certain functionalities are set by the product owner team, which consists of project management personnel of the supplier and the formal customer of the project. Furthermore, agile practices, like daily meetings, have been combined with lean practices and tools, like a Kanban board.

3.2 Deployment Pipeline of the Project

The pipeline of the case project has several environments (see. Figure 2) – a personal local development environment (Local), the shared development environment (Dev), a testing environment (Test), a quality assurance environment (QA) and the production environment (Production). Each of these environments serve different needs, and deployments to the different environments are managed through the version control system. Therefore, it automatically provides accurate data and meta data to measure the pipeline, which we have already proposed in an earlier paper [27]. The actual timestamps of deployments are stored in the meta data of the version control system branches.

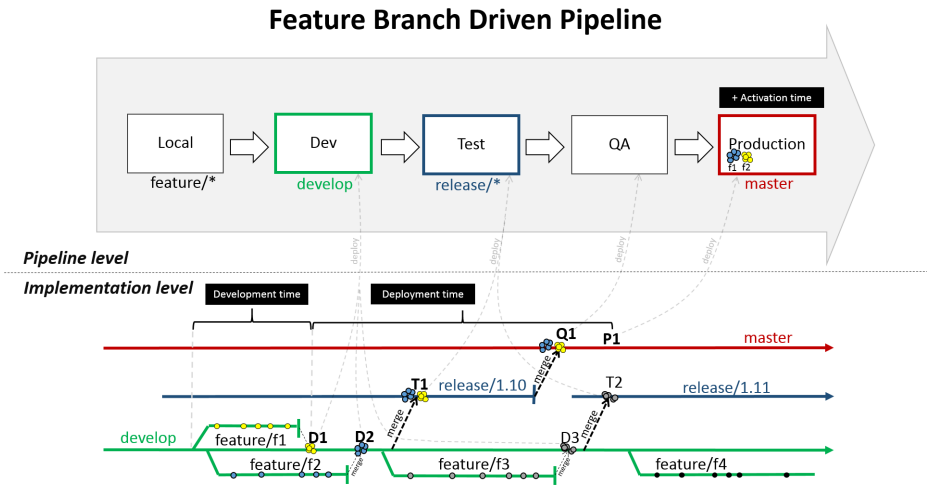

Fig. 2. Deployments to the pipeline environments are triggered by push-events to the distributed VCS. Features f1 and f2 have been merged and pushed to the develop-branch (triggering deployments D1 and D2 to the Dev-environment), then to the Test-environment (deployment T1 to Test-environment) and production environment (P1).

The team uses a VCS-driven solution to manage the deployments to the environments of the pipeline. The team applies the Driessen branching model [28], which utilizes feature branches. Figure 2 presents the connection between the branches and the deployments to the various environments of the pipeline. When the development of a new feature starts, a developer opens a new feature branch and starts developing the feature in the Local environment by committing changes to the new branch. The developer may push the local changes to the version control system from time to time, but no CI jobs are executed in this phase. When the development of the feature is ready, the feature branch is closed and the changes are merged to the develop-branch. When the changes are pushed to the version control system, a stream of CI jobs for deploying a new version to the Dev-environment is triggered automatically (deployments D1, D2 and D3 in Figure 2). The CI jobs build the software, migrate the database, and deploy and test the software.

The deployment to the Test environment is accomplished by merging the develop-branch to a release-branch. Once again, when the changes to a release branch are pushed to the version control system, a stream of CI jobs for building the software, migrating the database, and deploying and testing the software in the Test environment is triggered (deployments T1 and T2). For instance, deployment t1 in Figure 2 was triggered by a push to branch release/1.10, which contained features f1 and f2. Similarly, the production deployment happens by closing the release branch, which is then merged to the master-branch. The new version to be released can then be deployed to the QA (deployment Q1) and production environments (deployment P1) with a single click from the CI server.

In Figure 2, feature f1 flows from the development to production in deployments D1, T2 and P1. Feature f2 flows in deployments D2, T1 and P1. Feature f3 has flown to the test-environment in deployments D3 and T2. In order to deploy feature f3 to the production environment, release branch release/1.11 should be closed and merged to the master branch, which then would be manually released with a single click from the CI system.

Figure 3 presents the correspondence of the branches in the version control system and the CI jobs on the radiator screen in the team workspace. If a CI job fails, the team is immediately knowledgeable of the problems. Moreover, the current status of the functional end-to-end tests running in the Dev-environment is visible to the team.

In case of urgent problems in the production environment, the branching model also allows creation of a hotfix branch. Figure 3 represents a situation where urgent problems occurred after a deployment to the production environment. The automated tests had passed, but the login button was invisible on the front page because of layout problems. In this special case, a hotfix branch was then opened, the layout problems were fixed, the branch was merged to the master branch, and when the changes were pushed and a CI job was triggered manually, the problem was fixed and the users could continue logging in to the system.

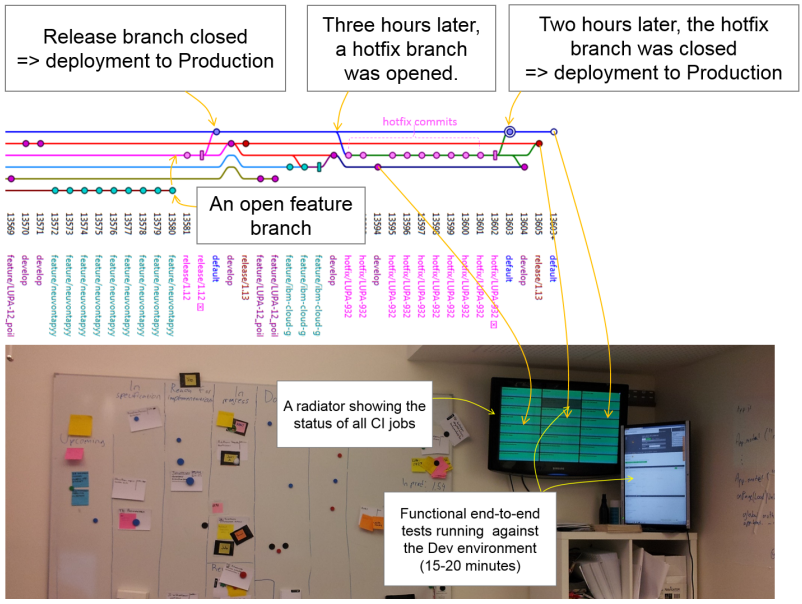

Fig. 3. An actual usage sample of the branching model and its correspondence to the CI jobs.

4 Defining Metrics for Pipeline

In this section, we define several new metrics which describe the properties of the deployment pipeline. The goal of the metrics is to provide valuable information for the team for improving the performance of the pipeline. With them, it is possible to detect bottlenecks, indicate and consequently eliminate, waste, and find process improvements.

We divided the metrics into two categories. First, *Metrics on the Implementation Level* dependent of the toolset and practices used to implement the pipeline. Second, *Metrics on the Pipeline Level* are metrics that are independent of the actual implementation of the pipeline. The metrics in the two categories are discussed in more detail in the following.

4.1 Metrics on the Implementation Level

The availability of data to calculate flow and throughput depends on the implementation of the pipeline and the actual tools and practices used. In essence, development, deployment and activation time must be available for each feature, discussed in more detail in the following.

– *Development time*, or the time it takes for the team to implement a new feature. The development time of a single feature can be measured in our case project, as each new feature is a new branch in the version management

system. The starting time for the new feature is simply the time when the branch is created, and completion time is when the branch is merged with the master branch. See Figure 2 for an example of development time of feature/f1. It is the time from opening the feature branch until D1. In an earlier paper [27] we measured the value of this metric during a three month period. The value was typically one or two days, but for larger features, it was even 12 working days.

- *Deployment time*, or the time it takes to deploy a new feature to production use when its implementation has been completed. There are two dimensions to this metric. One is the execution time of the tools needed for the actual deployment (e.g. seconds or minutes), and the other is the decision process to deploy the features, if additional product management activities, for example acceptance testing, are associated with the deployment (e.g. hours or days). See Figure 2 for an example of deployment time of feature/f1 – the time from D1 to P1. In [27], we measured a mean value of nine working days during a three month period.
- *Activation time*, or the time it takes before the first user activates a new feature after its deployment. Activation time can only be measured for features that are specific enough to be visible in production logs. At times, however, this can be somewhat complicated. For instance when a new layout is introduced, the first time the system is activated can be considered as the first use of the feature. See Figure 2 for an example of activation time of features found in the production log. It is the time from P1 to the first use caught from the production logs. The mean activation time in [27] was three working days while the median was one working day.

Another viewpoint to the time a feature spends in the deployment pipeline, is to count the age of the features that have been done, but are still waiting for production environment deployment. The following metric is based on measuring the current time spent on the pipeline:

- *Oldest done feature (ODF)*, or the time that a single feature has been in development done state, but is still waiting for deployment to the production environment in some of the environments of the deployment pipeline. The metric is dependent on Definition of Done (DoD) [29]. In our case project, this data is available from the meta data of the feature branches: a feature branch closed, but not merged to a closed release branch. At the time of research (Autumn 2015), the value of ODF in the case project is currently six days and the weekly release schedule has kept the value in less than one week constantly.

4.2 Metrics on the Pipeline Level

In the context of continuous delivery and deployment, the throughput of the pipeline used to deliver features to the end user is an important metric. Out of

the existing metrics, flow efficiency, proposed in [24], best captures the spirit of the pipeline. We propose the following new metrics to this category.

- *Features Per Month (FPM)*, or the number of new features that have flown through the pipeline during a month. The metric is based on Day-by-the-Hour (DbtH), which measures quantity produced over hours worked [3]. In the case project, the data for this metric can be collected from the implementation level data (number of feature branches closed and merged to a release branch that has been closed). Apparently, this metric can be measured in many other implementation settings, for example in a project that does not use feature branches. For example, issue management system data or version commit messages following a certain convention, are possible sources for this data. At the time of research, the value of this metric for the case project is 27 FPM during the last three months, which is more than one feature per working day.
- *Releases Per Month (RPM)*, or the number of releases during one month. Long term change of this metric provides information on changes in the release cycle. In the case project, this data is available both in the version control system and the CI server logs. At the time of research, the value of this metric for the case project is 7 RPM, which is one or two releases per week.
- *Fastest Possible Feature Lead Time*, or the actual time the feature spends in the build and test phase on the pipeline. In our case project, there is latency which origins from the use of feature branches and separate build processes for each branch. The code is compiled and packaged multiple times during the different phases of the pipeline. A build promotion approach in e.g. [30], where the code is build only once and the same binary is deployed to all environments, the lead time may be shorter. At the time of research, the value for this metric is two hours (quick integration and unit tests running some minutes in the commit stage and functional end-to-end browser tests running one hour in the pipeline environments). As a shortcut for urgent issues, the team can also use a hotfix branch, which allows making a quick fix in minutes.

5 Results

Working in close cooperation with industry to answer to our research questions has given us important insights over industry tools, processes and needs. Next, we will revisit our original research questions, and give a short discussion regarding our observations.

5.1 Research Questions Revisited

Considering the metrics defined above in the light of data available in version control system has given us high confidence that these metrics can be gathered

in a straightforward fashion, with certain exceptions. However, tools are needed to automate data collection process, and to help visualizing the results [31]. The exact answers to research questions are the following.

> **RQ1:** Which relevant data for practical metrics are automatically created when using a state-of-the-art deployment pipeline?

Data can be collected regarding development, deployment and activation time from the tool chain that is used by the project. For the two former, data is precise, but requires following certain conventions, such as creating feature branches in the version data base for new features – a feature which is not supported by all version control systems. Regarding feature activation, the situation is less clear, since for numerous features it is not obvious when they are truly activated. When referring to a new function in the system, such as a widget in the screen for instance, the activation produces identifiable traces, whereas a change in layout or in libraries used are harder to pinpoint. Therefore, to summarize, with version control and usage monitoring system data, it is also possible to address the numbers of features in development, deployment, and activation, although for the latter with only some limitations and interpretations.

Regarding practicality, we feel that any team performing continuous delivery and deployment should place focus on metrics listed above. Based on discussions with the team developing Lupapiste, visualizing the data regarding features on the pipeline was found very useful, and exposing developers to it actually led to faster deployment and to less uncompleted work in the pipeline.

> **RQ2:** How should the pipeline or associated process be modified to support the metrics that escape the data that is available?

While actions related to actual development are automatically stored in the version control system, end users' actions are not. Therefore, better support for feature activation is needed. This can not be solved with tools only but require project-specific practices. For instance, additional code could be inserted to record activation of newly written code, or aspect-oriented techniques could be used to trace the execution of new functions as proposed in [32].

In the present setup, there is no link to product management activities. In other words, the pipeline only supports developers, not product management. More work and an improved tool chain is therefore needed, which is also related to the above discussion regarding feature activation. However, it can be questioned if this falls within the scope of the pipeline, or should be considered separately as a part of product management activities.

> **RQ3:** What kind of new metrics based on automatically generated data could produce valuable information to the development team

We presented data collection methods for collecting data for new metrics regarding the deployment pipeline. We proposed multiple new metrics for the deployment pipeline. For example, metric *Oldest Done Feature (ODF)*, which the

team of the case project found especially potentially useful, could be applied for measuring the current state of the deployment pipeline. Exposing such a metric to the team for example on the radiator screen in the team workspace, could improve the release cycle of the project.

We measured the values for the new metrics proposed for the case project. The team was producing more than one feature a day and making releases at least once a week. The *Oldest Done Feature (ODF)* at the time of research was only six days old. According to these metrics, the features are flowing fluently from development to the production environment.

5.2 Observations

To begin with, the deployment to production may have extra latency even in a state-of-the-art deployment pipeline. For instance, in the case project, many of the features suffered from a long latency of even weeks or months between the time the feature was done till the time when it was deployed to the production. The team was shortly interviewed about the obstacles why the features were not deployed to the production environment earlier. The obstacles were often related to features that had been merged to the develop-branch, which then accompanied the develop branch to a state where it was not possible to deploy anymore. For instance in one case, a key functionality was broken and the fix needed data from a design process.

The time after a new feature that has been deployed to the production environment and is waiting for users to use the feature, can be regarded as waste. To eliminate such, the users of the system must be informed regarding newly deployed features, and they also have to have the skills to use them. Because the users in the case project are the municipal authority users nationwide, an announcement sent by email as new features are introduced. Moreover, a wizard, which would tell about the new features, for example after login or in the context of the features, could help the users to find the new functionality.

We discussed about the proposed new metrics with the development team of the case project. *Oldest Done Feature* was found as the most useful metric that could possibly help the team to improve the flow of the pipeline. The team even considered that this kind of metric could be shown on the radiator screen – if the oldest feature is for example two weeks old, the radiator could indicate the problem in the pipeline. However, the actual usage of such a metric is not straightforward. There are times, when the develop branch is not deployable because of, for example, a major refactoring. In this kind of circumstances this kind of metric may disturb the team.

6 Conclusions

A metric should be used for a purpose. A modern deployment pipeline paves a highway for the features to flow from the development work to actual usage in the production environment. The tools on the pipeline produce a lot of data

regarding the development and deployment activities of the new features. We analyzed the tools and practices of an industrial single case study in order to identify which data are automatically created by the several tools of the deployment pipeline. The results show that data for many new useful metrics is automatically generated.

Based on this data, we defined several new metrics for describing the properties of the deployment pipeline. For instance, the metrics proposed can be applied to analyze the performance and the present status of the pipeline. The goal of metrics is to provide valuable information to the team to improve processes and the pipeline. Applying the metrics in a continuous delivery project setting can help to achieve this.

Acknowledgements

This work is a part of the Digile Need for Speed project (`http://www.n4s.fi/en/`), which is partly funded by the Finnish Funding Agency for Innovation Tekes (`http://www.tekes.fi/en/tekes/`). Persons in Figure 1 are designed by Paulo S Ferreira from `thenounproject.com`.

References

1. G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software Technology*, vol. 57, pp. 21–31, 2015.
2. M. Poppendieck and T. Poppendieck, *Lean software development: an agile toolkit*. Addison-Wesley Professional, 2003.
3. K. Petersen and C. Wohlin, "Measuring the flow in lean software development," *Software: Practice and experience*, vol. 41, no. 9, pp. 975–996, 2011.
4. M. Fowler, "Agileversuslean," http://martinfowler.com/bliki/AgileVersusLean.html, 2008, retrieved: November 2014.
5. R. Shah and P. T. Ward, "Defining and developing measures of lean production," *Journal of operations management*, vol. 25, no. 4, pp. 785–805, 2007.
6. R. K. Yin, *Case study research: Design and methods*. Sage publications, 2014.
7. K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "The agile manifesto," http://agilemanifesto.org, 2001, retrieved: November 2014.
8. E. Kupiainen, M. V. Mäntylä, and J. Itkonen, "Why are industrial agile teams using metrics and how do they use them?" in *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. ACM, 2014, pp. 23–29.
9. M. Poppendieck and T. Poppendieck, *Leading lean software development: Results are not the point*. Pearson Education, 2009.
10. J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
11. S. Misra and M. Omorodion, "Survey on agile metrics and their inter-relationship with other traditional development metrics," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 6, pp. 1–3, 2011.

12. S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," in *Agile Conference (AGILE)*, Aug 2013, pp. 121–128.
13. M. Fowler, "Continuous delivery," http://martinfowler.com/bliki/ContinuousDelivery.html, retrieved: November 2014.
14. D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, pp. 48–59, 2014.
15. M. Fowler, "Continuous integration," http://martinfowler.com/bliki/ContinuousDelivery.html, retrieved: November 2014.
16. J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
17. A. Miller, "A hundred days of continuous integration," in *Agile, 2008. AGILE '08. Conference*, Aug 2008, pp. 289–293.
18. J. Humble, "Continuous delivery vs continuous deployment," http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/, retrieved: November 2014.
19. J. Humble, C. Read, and D. North, "The deployment production line," in *Agile Conference*. IEEE, 2006, pp. 6–pp.
20. D. Feitelson, E. Frachtenberg, and K. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, p. 1, 2013.
21. J. Humble, "Continuous delivery vs continuous deployment," http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/, 2010, retrieved: November 2014.
22. B. Fitzgerald and K.-J. Stol, "Continuous software engineering and beyond: trends and challenges," in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. ACM, 2014, pp. 1–9.
23. M. Kunz, R. R. Dumke, and N. Zenker, "Software metrics for agile software development," in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. IEEE, 2008, pp. 673–678.
24. N. Modig and P. Åhlström, *This is lean: Resolving the efficiency paradox*. Rheologica, 2012.
25. M. Van Hilst and E. B. Fernandez, "A pattern system of underlying theories for process improvement," in *Proceedings of the 17th Conference on Pattern Languages of Programs*. ACM, 2010, p. 8.
26. P. Debois, "Devops: A software revolution in the making," *Cutter IT Journal*, vol. 24, no. 8, 2011.
27. T. Lehtonen, T. Kilamo, S. Suonsyrjä, and T. Mikkonen, "Lean, rapid, and wasteless: Minimizing lead time from development done to production use," in *Submitted to publication*.
28. V. Driessen, "A succesful git brancing model." http://nvie.com/posts/a-successful-git-branching-model/, retrieved: November 2014.
29. K. Schwaber and M. Beedle, "Agile software development with scrum. 2001," *Upper Saddle River, NJ*, 2003.
30. L. Chen, "Continuous delivery: Huge benefits, but challenges too," *Software, IEEE*, vol. 32, no. 2, pp. 50–54, 2015.
31. A.-L. Mattila, T. Lehtonen, K. Systä, H. Terho, and T. Mikkonen, "Mashing up software management, development, and usage data," in *ICSE Workshop on Rapid and COntinuous Software Engineering*, 2015.
32. S. Suonsyrjä and T. Mikkonen, "Designing an unobtrusive analytics framework for java applications," in *Accepted to IWSM Mensura 2015, to appear*.