Chapter 5

Continuous Integration Is Not About Build Systems

This chapter is published as: Mårtensson, T., Hammarström P. and Bosch, J. (2017). Continuous integration is not about build systems. 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017, pp. 1-9.

Abstract: Keeping the build fast is often stated as an important prerequisite for continuous integration, and is also widely discussed in literature. But which importance does the capacity of the build system actually have in relation to developer behavior? Based on metrics and interview results from a large-scale industry project, we present the factors that according to the developers themselves affect how often they deliver software to the mainline. We show that the developer will deliver less frequently if the delivery processes is time-consuming, if it’s too complicated to deliver or if there is no evident value in delivering often to the mainline. Behind these three main themes, we also present a range of sub-categories such as architecture, test activities and administration. The build system capacity is one of several factors which, if not considered, could result in undesired continuous integration behaviors – but other factors should be seen as at least as important.

5.1 Introduction

A build system with the capacity to support frequent integration builds is one of the prerequisites for continuous integration and related practices. For example, the importance of keeping the build fast is stated in Martin Fowler’s popular article about continuous integration (Fowler 2006). In our previous work (Mårtensson et al. 2016), we have also discussed long build times as one of several factors that could constrain a full implementation of continuous integration. Different approaches have been proposed for the structure of the continuous integration pipeline: the integration build could be nightly (Goodman and Elbaz 2008, Trimble and Webster 2013) or instant (Goodman and Elbaz 2008, Humble and Farley 2010) to provide developers with fast feedback. Beck is an advocate of fast feedback, stating “Automatically build the whole system and run all of the tests in ten minutes. A build that takes longer than ten minutes will be used much less often, missing the opportunity for feedback.” (Beck 1999)

Build duration (time to compile and run tests) in different projects have been examined in related work (Downs et al. 2010, Woskowski 2012, Yüksel et al. 2009). Some publications also highlight how build duration affects continuous integration, as too long duration means that “continuous integration starts to break down” (Owen Rogers 2004) or that the build time must be quick enough to “allow the CI server to keep up with the changes and return feedback to the software engineers while their memory of (Dössinger et al. 2012).

Based on our experiences, we argue that continuous integration implementations in industry only partly depend on technological solutions. Following this reasoning, we
phrase the following research question: *Which additional factors other than the build system play a role when applying continuous integration in industry, and which importance has actually the capacity of the build system?*

The contribution of this paper is three-fold. First, it provides metrics, interview results and experiences from a large-scale industry project before and after the introduction of a new build system. Second, based on interview results it discusses which factors that affect how often developers deliver software to the mainline. Third, it sheds light on how developers rate the importance of the build system in relation to the developers’ continuous integration behaviors. The remainder of this paper is organized as follows. The next section describes the research method. Subsequently, in Section 5.3 we present the result from the pre-study for a new build system. In Section 5.4, we present the results from the study after the build system was introduced, and compare them with the results from the pre-study. Section 5.5 discusses threats to validity, followed by conclusions in Section 5.6.

### 5.2 Research Method

In order to discuss the research question, we compare quantitative and qualitative data from two studies, conducted before and after the introduction of a new build system in a large-scale software development project.

#### 5.2.1 The Case Study Company

The case study company is developing airborne systems and their support systems. The main product is a fighter aircraft, which has been developed in several variants. The next major upgrade will include both major changes in the hardware systems (sensors, fuel system, landing gear etc.) and a completely new software architecture.

Continuous integration practices such as automated testing, private builds and integration build servers are applied in the development of the software for the aircraft computer systems. Test activities are both automated and manual (especially testing related to HMI aspects). The development teams deliver software to a common mainline. Testing is conducted in simulated environments, rigs and test aircraft.

#### 5.2.2 The Studies Before and After the New Build System

Our first study was conducted as the pre-study for either an upgrade or a replacement of an existing build system. The results from the pre-study was used to define a project that should develop a new build system. Most of the teams migrated to the new build system during the first six months after it was introduced, but it took more than three years before all teams were migrated. The second study was conducted when all teams used the new build system, and compared the status before and after the new build system was introduced.

Quantitative data were in both studies collected from the integration team, which was responsible for integrating the software delivered by the developers into the
mainline. A series of interviews was also held in both the pre-study for a new build system and the study after the build system was introduced. The interviews in the pre-study had the characteristics of an unstructured (informal) interview, giving the interviewee the opportunity to speak freely in their own terms about a limited set of questions. The interviews in the study after the new build system was introduced were conducted as semi-structured interviews, using an interview guide with pre-defined specific questions. The interview results were analyzed based on thematic coding analysis as described by Robson and McCartan (2016). Extracts from the transcribed interview responses were coded and collated into themes. A thematic network was then constructed, resulting in a thematic map.

The interview groups were to a great extent the same individuals in both studies. In the cases where this was not possible (due to that some individuals had changed jobs or were not available for the second interview) other interviewees were hand-picked to represent the same roles as the participants from the pre-study which were not included.

5.3 Working with the Old Build System

5.3.1 The Software System in the Case Study

The software system in this case study is a highly integrated system, where processes execute in software partitions hosted by a number of computers with bespoke hardware. Communication between the partitions is handled only in safe time slots. The system is also a hard real-time system, meaning that every process must absolutely hit every deadline in the execution pattern.

The execution pattern for the software processes in this complex system is constructed and statically linked as part of the build process. Therefore, parts of the build system for this type of product must be developed in-house, or at least be customized.

5.3.2 Pre-study for a New Build System

In response to feedback from developers regarding long build-times, a pre-study was initiated aiming at improving the build system and the integration process. The pre-study collected quantitative data from the integration team as a measurement of the build system capacity, and also collected qualitative data in a series of interviews.

A simple metric was used to measure the build system capacity: the average number of integration builds per day (measured over a longer period of time). Development was distributed to around 20 teams, which were working in team branches in synchronized three week sprints. The metrics showed that the build system did not have enough capacity to handle even one delivery of software into the mainline from each team within a single sprint. The build system was clearly the bottleneck that prevented the teams from delivering software more often to the mainline, and therefore the major constraint for the organization during planning of when the team’s software deliveries should be integrated in the mainline.
Fifteen individual interviews were held with participants representing different stakeholders: Two project managers, seven developers, one individual representing the IS/IT organization and five engineers from the build and integration team. The interviews were designed as in-depth interviews lasting for one to one and a half hours, and were conducted face to face. In the first part of the interview, the interviewee was asked to describe the integration and build process. In the second part the interviewee was asked which factors the interviewee thought affected how often developers deliver to the mainline. Finally, the interviewee was asked for ideas on how the build system could be improved. The interview results were summarized in the following three bullets:

- The build system was considered to have a low degree of transparency
- The main reason for not delivering more frequently was that build and integration took too long
- Dependencies and merge conflicts were described as a major problem

The interviews clearly showed that the interviewees considered the build system to have a very low degree of transparency, and that it was very difficult to understand the integration process. The main reason seemed to be that the build system was constructed with a hierarchy of make scripts that were built on top of each other. As it seemed, no one could clearly describe dependencies between different parts of the software, or tell how a new build would be produced step by step. One interviewee answered the question about how the build system worked with “I do not know – magic, perhaps?”

The interviewees pointed out that the main reason for not delivering software more frequently was that build and integration took too long. As the build system did not distinguish between different types of changes in the software, the whole system was rebuilt for every build. Therefore, every single build took hours to complete. On top of this, the build system also required a lot of manual steps and temporary adaptations. This caused many builds to fail, and the build had to be restarted. This was described as a problem that caused much frustration.

Long time for build and integration implies a long time before the user gets feedback: does the software that was delivered work as expected when integrated in the mainline? The importance of finding problems as fast as possible is visualized by Boehm’s cost curve (Boehm 1976), which shows that longer feedback loops means a higher cost for the project. In addition to using agile methods, model-based development can also be used to shorten the feedback time. Experiences from the same case study company as in this paper have been presented in related work (Andersson et al. 2013): when using model-based development, the system’s maturity increases faster due to early use of modeling and simulations (Figure 12). That is, problems (defects in the system) are found at an earlier stage.
As the teams delivered their software to the mainline so seldom, a delivery very often revealed difficult merge conflicts which took a long time to handle. The strategy was to solve the merge conflict before integrating the next delivery, which extended the throughput time significantly as deliveries were not handled in parallel. The workflow with the old build system is shown in Figure 13. A lot of frustration was showing during the interviews. For example, one interviewee described the problems with difficult merge conflicts and an extensive integration process with the words “integration angst”.

Another result from the pre-study was that the execution of tests were included in the build system, and no longer seen completely as separate activities which were part...
of other systems and processes. In parallel with the project that developed the new build system, other initiatives established an integration pipeline, which included a range of test activities with the purpose to gradually increase confidence in the build’s quality. Within the context of continuous delivery (Humble and Farley 2010), this is referred to as the deployment pipeline (shown in Figure 14).

![Deployment Pipeline Diagram]

**Figure 14:** Trade-offs in the deployment pipeline.

The pre-study was also a starting-point for several initiatives related to better planning and coordination between the teams in the project, including visualization of dependencies from one team to another of deliveries in particular time-slots. The project that developed the new build system was followed by a long period of time where the build system was rolled out to the teams. Most of the teams transferred to the new build system within six months, but it took as long as three years before all the teams were using the new build system.

### 5.4 Working with the New Build System

#### 5.4.1 The New Build System

The new build system was designed to build incrementally, which means that only affected parts of the software are rebuilt for a new build. All builds that not affect the interfaces or scheduling use only a fraction of the previous build time. This means that if dependencies between components are removed, this will also result in faster builds. The build system also handles all dependencies instead of depending on manual input and temporary adaptations. All compiler flags were also centralized instead of the previous solution which was to use a separate make file for each module.

Collection of data from the build logs showed that the capacity of the new build system clearly transcended the old build system. The measured capacity of integration builds per day easily reached the goal to be able to handle at least one delivery from each team within a single sprint. In addition to that, the data showed a significant over-capacity, showing that the capacity of the build system was quite likely not the main impediment that prevents more frequent deliveries to the mainline. A series of interviews were held in order to capture different stakeholders’ views of the new build system. The interviews queried the interviewees about the transparency of the build...
system, about the frequency of software deliveries and about continuous integration behaviors.

Ten individual interviews were held with a group of stakeholders sampled to be as similar as possible to the interview group in the pre-study: One project manager, five developers and four engineers from the build and integration team. All of the interviews were conducted face to face. Five of the interviewees were also included in the interview group in the pre-study. The other five were sampled to represent the same roles as the participants from the pre-study which were not included. The reasons for not including all interviewees from the pre-study interviews were that the interviewees had changed jobs (to a new assignment not matching this study) or general availability to participate in the study.

5.4.2 Transparency of the Integration Process

The first question in the interview was “How would you rate how much you know about the steps in the integration process (purpose, included activities, activity sequencing etc.)?” A list of the steps in the integration process (as defined by the integration team) was provided to the interviewee, and the interviewee was asked to set a rating on a Likert scale from 1 (“I know almost nothing”) to 5 (“I have a really good understanding”). The responses from the interviewees are summarized in Table 11.

<table>
<thead>
<tr>
<th>Interviewee</th>
<th>Role</th>
<th>All steps</th>
<th>“Internal” steps</th>
<th>“External” steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Integration</td>
<td>4.3</td>
<td>4.4</td>
<td>4.0</td>
</tr>
<tr>
<td>2</td>
<td>Developer</td>
<td>2.8</td>
<td>2.4</td>
<td>3.3</td>
</tr>
<tr>
<td>3</td>
<td>Integration</td>
<td>4.1</td>
<td>5.0</td>
<td>2.7</td>
</tr>
<tr>
<td>4</td>
<td>Developer</td>
<td>1.9</td>
<td>1.8</td>
<td>2.0</td>
</tr>
<tr>
<td>5</td>
<td>Developer</td>
<td>3.8</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>6</td>
<td>Integration</td>
<td>3.1</td>
<td>3.6</td>
<td>2.3</td>
</tr>
<tr>
<td>7</td>
<td>Integration</td>
<td>3.8</td>
<td>3.8</td>
<td>3.7</td>
</tr>
<tr>
<td>8</td>
<td>Developer</td>
<td>3.0</td>
<td>3.2</td>
<td>2.7</td>
</tr>
<tr>
<td>9</td>
<td>Developer</td>
<td>3.4</td>
<td>2.6</td>
<td>4.0</td>
</tr>
<tr>
<td>10</td>
<td>Developer</td>
<td>2.9</td>
<td>2.0</td>
<td>4.3</td>
</tr>
</tbody>
</table>

Table 11: Average rating per interviewee on “How would you rate how much you know about the steps in the integration process (purpose, included activities, activity sequencing etc.)?”

The interviewees average rating for all steps in the integration process spans from 1.9 to 4.3, showing a wide span within the group. In the same way, the ratings of knowledge about a single step also spanned from one to five for most of the steps in the integration process. A distinction could be seen in the ratings correlated to the role of the interviewee. Therefore, interviewees working as developers or project managers (for a software development project) are here labeled “developer”. In the same way, interviewees working in the build and integration team are here labeled “integration”.

77
Another trend in the ratings is the distinction between steps handled by the build and integration team (here labeled “internal step”) and the other steps in the integration process (here labeled “external step”).

Table 12 shows the average rating for all interviewees and all steps in the integration process, as well as the average value for the “internal” steps and the “external” steps. Table 12 shows that the average rating for all interviewees and all steps in the integration process is 3.3. This could be interpreted as that the interviewees’ overall rating on how much they know about the steps in the integration process is somewhere in the middle between the answers “I know almost nothing” to “I have a really good understanding”. Not surprisingly, the “developers” rating of the “external steps” are higher than of the “internal” steps, which is natural as developers are not directly involved in the “internal steps”. Mirroring this, the “integration” group’s rating of “internal steps” are higher than the rating of the “external steps”. This could be seen as more surprising, as the build and integration team is the owner of all steps in the integration process. However, to discuss correlations or causality based on the ratings within sub-groups demands a larger interview group than what’s included in this study and can be a plausible area for further work.

<table>
<thead>
<tr>
<th>Interviewees</th>
<th>All steps</th>
<th>“Internal” steps</th>
<th>“External” steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>3.3</td>
<td>3.2</td>
<td>3.4</td>
</tr>
<tr>
<td>Developers</td>
<td>2.9</td>
<td>2.5</td>
<td>3.6</td>
</tr>
<tr>
<td>Integration</td>
<td>3.8</td>
<td>4.2</td>
<td>3.2</td>
</tr>
</tbody>
</table>

Table 12: Average rating for all interviewees and all steps in the integration process.

5.4.3 Delivery Frequency

The interviewee was then queried about the frequency of deliveries to mainline. The questions were phrased to include both delivering directly to mainline and delivering to a branch (feature branch or team branch) which then delivers to mainline. The questions regarding delivery frequency were:

- How often do you commit your software to the mainline (either directly or from a team or feature branch)?
  - Same as I do now
  - More frequently
  - Less frequently
- How often do you want to commit your software to the mainline (either directly or from a team or feature branch)?
- How often do you think developers on average commit their software to the mainline (either directly or from a team or feature branch)?
- How often do you think developers on average should commit to the mainline?
  - Same as they do now
  - More frequently
  - Less frequently
The interviewees from the build and integration team did not deliver software themselves, and were therefore asked to answer only the last two questions.

In the same way as we have seen in our previous work (Ståhl et al. 2017b), the developers in this study worked in team branches and did not deliver directly to the mainline, which result in that software is delivered to the mainline at a lower frequency. The interviewees’ responses on how often they commit software to the mainline was with one exception around three weeks which means one delivery per sprint. One interviewee responded that he sometimes delivered to the mainline with an interval as long as several months, but clarified that this referred to new functionality and that bug fixes were delivered “immediately”. The responses on how often the interviewee thought developers on average delivered were in all cases equal to the response on how often the interviewee delivered.

Four of the developers wanted to deliver with the same frequency as they did now, two wanted to deliver “much more frequently”. One of the interviewees answered that developers in average should deliver “same as they do now”. Seven of the interviewees answered that developers on average should deliver more frequently, or “much more frequently”. Interestingly enough, the general opinion seems to be that other developers should deliver more often than the interviewee should do. However, one interviewee answered that developers should deliver “less often”, and added the explanation that “otherwise they will introduce errors and deliver things that don’t work”. One interviewee refrained to answer the question.

5.4.4 Continuous Integration Behaviors

The perhaps most interesting question in the interview was “Which factors do you think affect how often developers deliver to mainline?” Extracts from the interview responses on this question were coded and collated into themes. A thematic network was constructed, resulting in a thematic map with three main themes which summarize the interviewee responses.

The responses tended to shift from “which factors do you think affect how often developers deliver to mainline” to “which factors prevent the developer from delivering more often”. The three main themes which are the result of the thematic coding analysis (also shown in Figure 15) are summarized as:

- The developer delivers less frequently if the delivery process is time-consuming
- The developer delivers less frequently if it’s too complicated to deliver
- The developer delivers less frequently if there is no evident value in delivering often to the mainline

Ten out of ten interviewees provided statements that were connected to the factor “the delivery process is time-consuming”. The other factors were also widely supported: Extracts from responses from six interviewees were connected to “it’s too complicated to deliver” and five of the interviews to “no evident value in delivering often to the mainline”.

79
Figure 15: The main factors that prevents the developer from delivering more often.

The factor “the delivery process is time-consuming” consists of a number of sub-categories (shown in Figure 16). Most supported is “the delivery process is time-consuming due to test activities”, which includes comments from seven interviewees. The interviewees especially pointed out manual test activities, or activities where the developer must be standing by to do manual steps to interpret test results or to go from one test activity to another. The CM system (software repository) was mentioned by five of the interviewees. Manual steps when using the CM system were described as a problem, in the same way as previously described with test activities. “It’s easy to make mistakes”, as one interviewee stated.

Administration with boards and paperwork were mentioned by four interviewees. It should be noted that the view of paperwork as an impediment was contradicted by one interviewee, who stated that “the paperwork is not something that is a problem”. The build system (build time) was described as an impediment by three interviewees, and other tools were mentioned by two interviewees. A bit surprisingly, the integration
team’s way of working was described as a problem by two of the interviewees. Another two interviewee’s mentioned only general comments like “it takes too long to deliver”, but did not specify why. Several of the interviewees, on the other hand, presented a range of problem which were connected to several themes (e.g. CM system, build system and test activities).

The second factor, “it’s too complicated to deliver”, consists of four sub-categories (shown in Figure 17). Most supported is “it’s too complicated to deliver due to the tools”. The comments (often stated with much frustration) were related to the CM system, except for one comment about the build system: “When you do something wrong… The error messages are too hard to interpret. Then you correct something and it starts all over again.”

![Figure 17: The sub-categories within the factor “The developer delivers less frequently if it’s too complicated to deliver”.](image)

“It’s too complicated to deliver due to the delivery process” includes statements from four of the interviewees, with several interesting comments, for example: “It’s complicated to find the document that specifies what to do”. Other comments described uncertainty about the prerequisites for a delivery and how the integration team works. Four interviewees described problems related to architecture and dependencies, especially shown as rebase and integration problems. One interviewee even said “As people have to adapt to changed interfaces, they cannot focus on their own software, which slows down development.” Tests were mentioned by two interviewees, of which one clearly stated that all manual steps in the test activities must be automated.

The third factor “no evident value in delivering often to the mainline” (shown in Figure 18) include comments from only five interviewees. However, we believe that the interview responses on previous questions, describing a way of working with long-lived branches and delivering with a low frequency to the mainline supports “no evident value” as an important factor. We argue that the fact that the interviewees responded that developers on average should deliver more frequently than the interviewee should do could also be interpreted as that the interviewee are seeing no or little value for
themselves in delivering often to the mainline (but still a value for teams working with other types of functions or sub-systems).

Figure 18: The sub-categories within the factor “The developer delivers less frequently if there is no evident value in delivering often to the mainline”.

“No evident value in delivering often to the mainline” is either related to “no value for the team” or “no value according to project management”. Quite interesting is that developers working with infrastructure functions declared that there was “limited value in interacting with other teams”, which was contradicted by a developer working with top-level applications who stated “[…] we don’t need to deliver often – but if I was delivering infrastructure, I would deliver often”. As we have discussed in previous work (Mårtensson et al. 2016), a large number of technology fields in a product may foster silo behaviors. Then the developers see their sub-system as “our system”, and treats the complete system as a secondary concern. This attitude might be the reason why developers deliver frequently to branches, but delivers much less frequently from the branches to mainline (as there is “no evident value in delivering often to the mainline”).

Two interviewees gave similar comments: “someone [from management] must ask for the developer’s delivery” and “if project management requests more frequent deliveries, there will be more deliveries”. One interviewee described that it is how you split the work into pieces that can be developed and tested that is the most important factor for how often you deliver as a developer, and that this is something that must be requested by management.

5.4.5 The Importance of the Build System

The final question in the interview guide was “How would you rate how important the build system is in relation to the developers’ continuous integration behavior?” The interviewee could answer with one of the following alternatives:

- A: Characteristics of the build system is the only thing that affects how often developers commit software
- B: Characteristics of the build system is the most important thing that affects how often developers commit software
- C: Characteristics of the build system is one of several factors that affect how often developers commit software
- D: Characteristics of the build system is a factor that to some extent affects how often developers commit software
- E: Characteristics of the build system is not a factor that affects how often developers commit software

All interviewee responded “C”, which corresponds well with the responses on the previous question “which factors do you think affect how often developers deliver to mainline”. The build system places itself in the middle of the list of factors behind that the delivery process is time-consuming. The build system is also one of the factors behind the comments on both “it’s too complicated to deliver due to the delivery process” and “it’s too complicated to deliver due to the tools”, but is clearly not the most important factor.

The two-factor theory (also known as Herzberg’s motivation-hygiene theory and the dual-factor theory) states that there are certain factors in the workplace that cause job satisfaction, while a separate set of factors cause dissatisfaction (Herzberg et al. 1959). The two-factor theory distinguishes between the following two sets of factors:

- Motivators, for example challenging work, opportunity to do something meaningful, involvement in decision making
- Hygiene factors, for example job security, salary, work conditions

According to the two-factor theory there are four possible combinations, which are shown in Figure 19.

![Figure 19: The four combinations in the two-factor theory.](image)

We argue that the build system in a development environment should be seen merely as a hygiene factor, and is seen by the developers as only part of reasonable work conditions. The capacity of the build system can never itself work as a motivator, making developers to deliver more frequently to mainline. An interesting area of further work would be to identify and investigate the factors that truly motivates developers to deliver often.
5.4.6 Vicious Circles

Different factors might also be connected and strengthen each other. Three examples of vicious circles were found during the interviews (shown in Figure 20). The first vicious circle is about that a developer that finds the delivery process and/or the tools hard to understand tends to deliver less frequently, which means that the developer will not learn the process and/or the tool. To quote one of the interviewees: “It’s complicated to check in and deliver to [the CM system]. People tend to avoid using [the CM system], which makes you less familiar with the tool”. Another interviewee stated “I think the developers who deliver less frequently are the ones who think that the process is complicated. So the factor is knowledge about the delivery process. If you don’t know how to deliver, you tend to do it less frequently.”

The second vicious circle is about merge conflicts which tend to be even more complicated if you choose not to deliver (and handle the merge conflicts). The interviewees’ comments reveal a good knowledge about this problem: “There are often integration problems. This is because we deliver so seldom.” One interviewee went as far as to say that the situation has not improved with the new build system: “If everything does not go well, you end up doing things over and over again. We have the same situation as with the old build system.”

The third vicious circle is a related topic: If the developer delivers with a low frequency, software from many other systems are delivered in-between. This means that a larger part of the product has to be rebuilt when the developer finally delivers, resulting in a longer build-time. However, as the correlations presented in Figure 20 are based on statements from one or two interviewees, this should be seen only as a discussion about interesting topics for further work.

Figure 20: Vicious circles identified during the interviews.
5.5 Threats to Validity

This section discusses threats to construct validity, internal validity and external validity.

5.5.1 Threats to Construct Validity

In this paper we present interview results from the second study, indicating that the build system is one of several factors that affect how often developers deliver software. One way of reasoning is that the build system always is the most important factor, and the reason that this is not showing in our interview results is because the build system has been improved so much that the previous problems are now solved.

Our position is that the interviewees’ responses clearly show that the build system should be considered as one of several factors that affect developer behaviors. We do not rate the importance of the identified factors, or claim which of the factors that is the most important. As long as this is kept in mind, we do not believe that the reasoning stated above poses any significant threat to construct validity.

5.5.2 Threats to Internal Validity

Of the twelve threats to internal validity listed by Cook, Campbell and Day (1979), we consider Mortality, Selection, Ambiguity about causal direction and Compensatory rivalry relevant to this work:

- **Mortality**: Only part of the interview group used in the pre-study was reused in the interviews at the second study. As we focus on the roles of the interviewees and not on the specific individuals, we do not see this as threat to validity.
- **Selection**: The interviewees were purposively sampled to represent a mix of developer and integrators, both with experience from the build system and the integration process. Considering the rationale of these samplings, we consider this threat to be mitigated. It could be noted that, as we describe in Section 5.4.3, in one case one interviewee refrained to answer one of the questions, i.e. the responder was in this case self-selecting.
- **Ambiguity about causal direction**: While we in this study discuss correlation, we are very careful about making statements regarding causation. Statements that include cause and effect are collated from the interview results, and not introduced in the interpretation of the data. Due to this, we consider this threat to be mitigated.
- **Compensatory rivalry**: When performing interviews and comparing scores or performance, the threat of compensatory rivalry must always be considered. In this study, this applies in particular to the question where the interviewee rates his/her own knowledge about the integration process. As the data (presented in Table 11) is not used as the only source to support the conclusions in Section 5.4, we consider this threat to be mitigated. In the same way, the question about how often developers deliver to mainline could cause defensiveness and blame-avoidance. Therefore, the question was designed to focus on “factors that affect how often developers deliver”
and not to focus on the developer’s behavior. Generally, the questions were also
designed to be opened-ended to avoid any type of bias and ensure answers that were
open and accurate. However, our experiences from previous work is that we found
the interviewed engineers more prone to self-criticism than to self-praise.

5.5.3 Threats to External Validity

The study is based on one case study, concerning a single company. It is conceivable
that the findings from this study are only valid for the one company, or the one industry
segment. For this reason we have presented detailed information about both the case
study company and the software system in the case study company. As we present these
limitations clearly to the reader, we argue that this threat has been mitigated.

5.6 Conclusion

In this paper, we have presented metrics and interview results from two studies in a
large-scale industry project, before and after the introduction of a new build system.
The comparison of the results from our two studies (in Section 5.4) shows that the
project for the new build system generally succeeded and reached its objectives. The
collected metrics show that the new build system is no longer the bottleneck, and the
capacity of the build system is therefore quite likely not the main impediment that
prevents a developer from delivering more frequently (as described in Section 5.4.1).

This is confirmed by the interview results (presented in Section 5.4.4), which instead
points at a range of other factors. The interviews in our second study also show that
transparency is generally no longer described as a major problem. This is evident in the
interview responses both from the ratings of the interviewees’ understanding of the
integration process (presented in Section 5.4.2), and from the interviewees’ opinions
on what factors that affect how often a developer delivers to mainline (described in
Section 5.4.4).

Our research question was to investigate which additional factors other than the build
system play a role when applying continuous integration in industry, and which
importance the capacity of the build system actually has. As we have seen in the
interview results from this study (presented in Section 5.4.4 and 5.4.5), the build system
capacity is one of several factors which, if not considered, could result in that
(according to the developers) the delivery process is time-consuming. In addition to
this, we have also identified a range of other topics (presented in Section 5.4.4) which
we have grouped into three main factors.

The three factors found in this study which affect continuous integration behaviors
(the factors that make developers deliver less frequently) are:

- The delivery process is time-consuming
- It’s too complicated to deliver
- No evident value to deliver often to the mainline
Build system capacity should be considered as an important factor – but other factors should be seen as at least as important.

5.6.1 Further Work

In addition to the results presented in the analysis and the conclusions, we believe that this study opens up several interesting areas of further work. Much is yet to study in the field of what the developers see as continuous integration impediments (which could be related to both the organization and the characteristics of the product). Further work could expand the analysis of this paper to a study that includes several companies in multiple industry segments, and continue the work to describe impediment areas together with strategies for how the impediments should be eliminated or evaded.

Another topic for further work is to further examine the mechanisms laying behind the vicious circles which are presented in Figure 20. Are the factors presented in the figure truly correlated, and if so – are there an unambiguous causal direction?

The interview results showed significant differences between the responses from the two groups (labeled “developer” and “integration”). An area for further work could be to further investigate how the transparency of a build and integration process is perceived by different stakeholder groups.

In this paper we have only briefly touched upon the area what really motivates the developers to deliver frequently to mainline. Another interesting area of further work could be to analyze both the technical and cultural factors that are the causes behind that a developer wants to deliver frequently to a functional branch or a team branch, but finds much less value in delivering their software to the mainline for the complete system.

Acknowledgement

The authors would like to thank all the participating engineers for their insights, patience and willingness to share their experiences and data with us.