A Hundred Days of Continuous Integration

Ade Miller
Microsoft Corporation
ade.miller@microsoft.com

Abstract

Many agile teams use Continuous Integration (CI). It is one of the Extreme Programming practices and has been broadly adopted by the community [1]. Just how effective is it? Does the effort of maintaining the CI server and fixing build breaks save time compared to a lengthier check-in process that attempts to never break the build? While much anecdotal evidence exists as to the benefits of CI there is very little in the way of data to support this. How do you convince teams and management that it's worth adopting and how best to do it? This report outlines our experience with CI in a distributed team environment and attempts to answer these questions.

1. Background

The data presented here was collected from the Team Foundation Server and continuous integration (CI) servers used on the Web Service Software Factory: Modeling Edition (Service Factory) project (http://www.msdn.com/servicefactory). Service Factory was a small project run by Microsoft's patterns & practices group and staffed by about a dozen people. It ran for nine months during 2007, delivering the final release in November of that year.

The Service Factory development team was based in Redmond, WA where two developers were located. A further three developers were located offshore in different locations and time zones in South America. These offshore developers periodically visited Redmond to work on site with the rest of the team. We also worked with a test team located in India.

Over a period of about a hundred working days (approximately 4000 hours of development time) we monitored the number of check-ins and the associated CI server build failures. During this time the project went from being in full flight, delivering features in 2-3 week iterations, to executing a three week stabilization phase and shipping.

On average developers checked in once a day. Offshore developers had to deal with network latencies and checked in less frequently; batching up work into single changesets. The distributed nature of the team made pair programming difficult if not impossible. We tried to achieve shared code ownership through code reviews for each check-in. This, combined with time differences, also led to some batching of work and reduced the check-in frequency.

Our CI server compiled the code and ran unit tests and FxCop (http://blogs.msdn.com/fxcop/) analysis every time a check-in took place. We also had the same server compiling MSI installers, testing them once or twice a day, and doing a further build each night to generate static analysis data using NDepend (www.ndepend.com) and unit test code coverage metrics.

2. What broke the build?

Over the analysis period the team checked in 551 times which resulted in 515 builds[1] with 69 build breaks (13% of check-ins) including six breaks that left the server down over night.

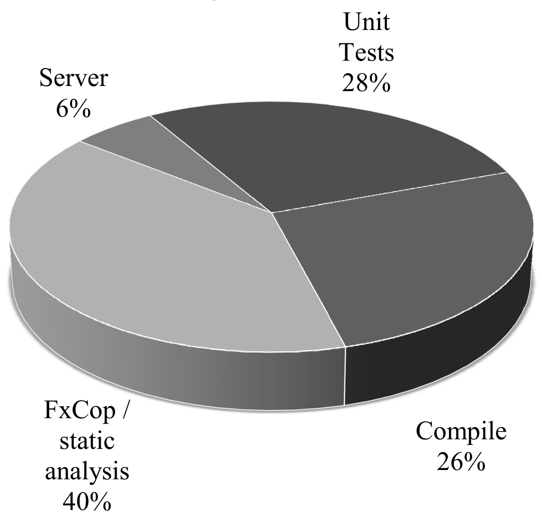

Figure 1: Causes of build breaks

A build break was defined as one or more consecutive failed CI builds. For example, a build

[1] Our CI server was configured to run each time it detected a new changeset but not to queue and run builds for each changeset. This accounts for there being slightly more check-ins than builds.

break is followed up by a fix check-in that fails to correct the issue resulting in another failed build. This counts as a single break. The length of the break is taken to be the time difference between the first failed build and the next successful build.

High level root cause analysis of these breaks revealed four main causes, as shown in Figure 1.

Compile – A developer checked in code that failed to compile, typically due to files missing from the changeset but present on the local machine.

Unit Tests – A developer checked in code that broke unit tests.

FxCop / static analysis – A developer checked in code that failed the static analysis bar.

Server – Our server was a virtual machine running on Windows Server 2003. We didn't anticipate how large our CI builds would become, eventually requiring more disk and memory than the server had available. This caused the CI server to fail, in several cases overnight.

It's also worth noting that some members of the development team were not particularly familiar with our CI practices at the start of the project and our distributed nature made coaching them that much more challenging. We might well improve our 13% failure rate on check-ins on a subsequent project. As Troy Maginnis points out broken CI builds aren't necessarily a sign of an unhealthy team but apathy around getting the build back to green definitely is [2].

3. Which were the worst breaks?

How long did most breaks cause the server to be down and which types of breaks took the most time to fix? Figure 2 shows the distribution of breaks by duration. This is important because once the build is broken other developers cannot check in their work, effectively blocking the team's progress.

As is clear from Figure 2 the great majority of build breaks were fixed within an hour. We had seven breaks that left the server broken out of Redmond business hours (overnight). Several of the other lengthy breaks were related to server issues (see Figure 3).

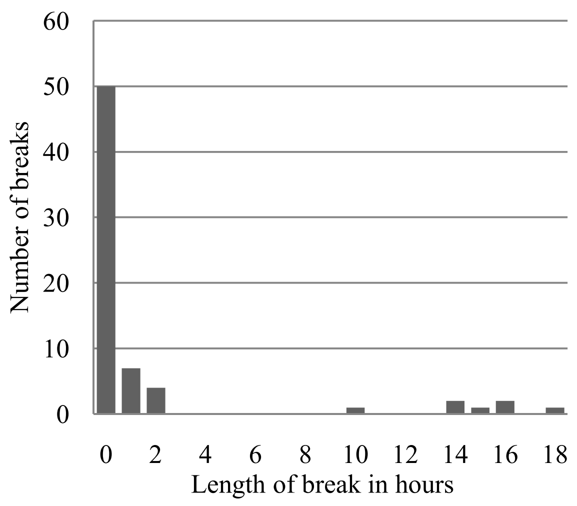

Figure 2: Length of build breaks

Our process around CI server breaks was as follows. After an initial investigation, one developer would fix the build while the rest of the team continued with their work but did not attempt further check-ins. Excluding out of Redmond business hours breaks, the average time to fix a CI issue was 42 minutes. This includes the time to submit the fix and have it verified by the CI server—approximately 20 minutes. So a typical CI issue fix took less than an hour. This is short enough that breaks typically did not block other developers significantly.

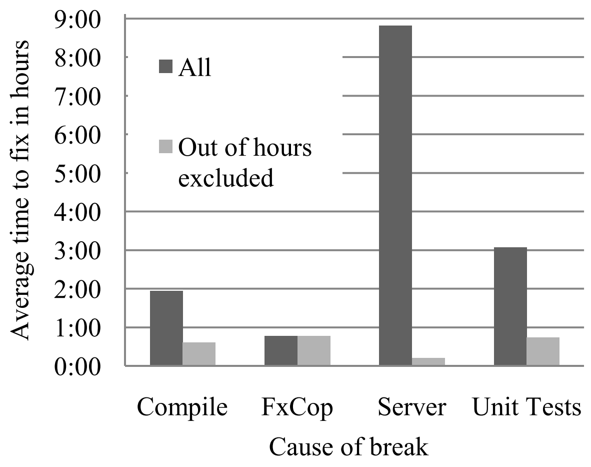

Figure 3: Average time to fix a build break

4. How much did CI cost?

From the data we collected it is possible to infer the cost of using a CI driven check-in system over the course of the 108 working days and 551 check-ins. Let's assume that developers compile the code, run appropriate unit tests against their change, and possibly run some static analysis prior to submitting their change. Developers also have to get a code review for all but trivial changes. It is assumed that they waited for feedback on the review in parallel with other work or were pair programming. The overhead for reviews is not considered as it would occur with or without adopting a CI process.

Given these assumptions a reasonable estimate, based on our experiences, for the time spent by a developer doing these tasks is twenty minutes; ten minutes doing a full compilation and running tests, and ten minutes reviewing changes, writing check-in notes and e-mails, etc.

Using the data from the CI server (presented in Sections 2 and 3) we also know that the average build break took a developer about 45 minutes to fix and that there were 69 build breaks.

Table 1: CI process overhead

CI Server setup and maintenance time and build script authoring (est.)	50
Total time spent checking in 20 minutes (est.) × 551	165.3
Total time spent fixing build breaks 45 minutes × 69	51.75
Total overhead (hours)	**267**

Table 1 shows the total cost to the project associated with checking in and fixing build breaks was approximately 267 hours, 7% of the total effort.

5. What might an alternative have cost?

Let's consider a *hypothetical* heavyweight check-in process *for this team*.

> For each check-in a developer is required to compile, run all unit tests, the installation tests, and static analysis tools on a clean build machine.

In other words developers do all the work the CI server is doing before each check-in. This is really the only alternative directly equivalent to CI in terms of ensuring the same quality of the code base and product under development. Other alternatives are discussed in Section 6.

Such an approach for this team would take at least 50 minutes; ten minutes doing a full compilation and running some tests on the developer's system, ten minutes to create a changeset and unpack it on the clean build machine, 20 minutes to compile the code and run all tests and some static analysis (based on the time taken by the CI server), plus an additional ten minutes for reviewing changes, writing check-in notes and e-mails, etc. Let's assume the best case and say that *no* build breaks ever happened and the clean build machine never needed to be rebuilt because the check-in process was so perfect. As shown in Table 2 the lowest possible total cost would then be 464 hours, 12% of the total development effort.

This is the best possible case for our alternative process. Build breaks will occur and need to be fixed. This estimate also allocates only a minimal time to set up and maintain a clean machine for builds.

Table 2: Alternative process overhead

Clean build machine setup and maintenance time (est.)	5
Total time spent checking in 50 minutes (est.) × 551	459
Total time spent fixing build breaks	0
Total projected overhead (hours)	**464**

Even this optimistic analysis of the alternative suggests that it would result in at least another 200 (5%) hours being spent on integrating changes rather than generating customer value.

6. Other check-in alternatives

Are there are other approaches one might take to reduce check-in overhead? Developers could optimize the process by amortizing the check-in overhead across several tasks or bugs. In this scenario the developers check in work related to several tasks or bugs in a single changeset, typically checking in once every few days but maybe keeping work on their machine for more than a week. Further overhead reduction can also be achieved by amortizing check in work across developers by having an "Open Check-in Window." During this window developers can check in without any validation and one developer runs the validation, identifies breaking changes, and coordinates any fixes.

Indeed, by batching up work and doing fewer check-ins developers could reduce the overhead to be on a par with the CI process. Unfortunately, these alternatives have several negative consequences:

- When (not if) the build breaks, fixing it will be harder because the changeset that needs to be examined is much larger and may have multiple authors.
- Checking in less frequently increases the possibility of merge conflicts and therefore causes additional work.

- Checking in less frequently reduces visibility into the current state of the code. Unchecked code is essentially hidden—it is invisible to stakeholders looking at the latest build.
- Code reviews on large changesets take more time and are harder to do effectively.
- Usually during any product stabilization phase developers are typically only allowed to check-in changesets for a single bug or design change, the batching strategy isn't available to them. This can significantly slow up your release process.

So while there are other alternative approaches that reduce the check-in overhead most of them do so at the expense of maintaining the quality of the code base and product under development. Reduced quality has further knock-on costs, not only to the product but also to the team's time. Developers spend more time debugging and maintaining their code thereby decreasing the team's velocity.

7. Conclusions

The actual cost of using the CI approach on this project was at least 40% less that the hypothetical cost of a check-in process that doesn't leverage CI but still maintains the same level of code base quality. Given the relatively small size of the product being developed and the low cost of doing a complete build this study actually gives us a number for the smallest saving likely from deploying a CI process. Experience with larger teams on larger product, specifically Visual Studio Tools for Office [3] suggests that the case presented here actually downplays the advantages of CI. CI shortened the check-in process for Visual Studio Tools for Office developers from over two hours to less than one. This represents a potentially much larger saving than seen here.

Given all of the above, teams moving to a CI driven process can expect to achieve at least a 40% reduction in check-in overhead when compared to a check-in process that maintains the same level of code base and product quality.

8. Best practices and lessons learned

In the course of running the Service Factory project we learned something about maximizing our investment in CI and using it with highly distributed teams.

We developed a *Defense in Depth* approach to CI, adding more and more testing and analysis to the CI server as we came across issues. Initially our CI server compiled the code and ran unit tests. Over time we added FxCop and NDepend for static analysis, source tree layout checking, code coverage tracking, MSI testing and partial installer tests. Each time we increased the depth of our defenses against a drop in product quality. In some cases we added defenses to address some specific problem we had been having.

Defense in depth requires high end hardware for the build server so that the team can keep adding new tests and analysis without slowing it down. We ended up running several different CI builds largely because running everything in one build became prohibitively slow and we wanted the check-in build to run quickly.

Think of the CI server as a way of taking the grunt work out of checking in code. Machines are good at executing repeated tests, people aren't. To that end we adopted the following rules:

- Treat warnings as errors. Modify your build scripts to fail on warnings.
- If you break the build, you fix the build. Don't make the people who are co-located with the CI server responsible for fixing it.
- Don't check in and go home. This leaves the remaining developers to clean up the mess or potentially be blocked for the remainder of their day.

Agile development's everyone in the team room philosophy is seriously challenged by geographically and temporally distributed teams [4]. Our team's heavy reliance on CI further highlighted the impact of these challenges. The following recommendations are a result of our experiences:

- **Co-locate as much as possible:** If this is not possible then plan for team members to spend time traveling and working at the main or offshore site, especially at the start of a project—the first few iterations—when key architectural/design decisions will be made.
- **Align the team locations:** The more offshore locations your team has the higher the communication barriers become. When using offshore team members group them into sub-teams aligned by feature.
- **Time zones add further tax:** One of the hardest aspects of distributed collaboration is working across time zones. Try to minimize the time shift between the team locations as much as possible and try to establish core working hours that all members can adhere to.
- **Co-locate by feature not discipline:** Distributed sub-teams should work together on features rather

than be distributed by discipline; for example, have an offshore team working on a feature not an offshore PM or Test team. Splitting your team by discipline increases the boundaries between activities, it's the old developers build it and throw it at the testers, only worse.

- **Onshore representative for offshore team:** Have someone in the team room responsible for being the offshore team's voice in the room. This isn't to get the offshore people off the hook for attending standup or using other practices to maximize communication. It's designed to help the remote team members stay synched up with key conversations they may have missed.
- **Pay the tax associated with distribution:** Co-location of your team allows them to communicate rapidly with minimal formal processes. If you have distributed team members be prepared to pay the tax associated with this. For example, your specifications will need to be much more complete to offset communication that would have occurred in a team room—you'll be writing much more complete story cards for example. More pre-work will also be required for meetings, for example getting user stories in shape prior to iteration planning so that the team can review and ask clarifying questions via e-mail rather than during the meeting.
- **Improve communication where possible:** Get good communication going from the outset of the project. Make sure everyone has access to the right communications tools; conference phones, instant messenger or IRC, hands free phone headsets, a camera for taking whiteboard pictures and sufficient network bandwidth to use them effectively.
- **Focus on team consistency:** It takes time to build good working relationships, especially on distributed teams. Try and minimize staff churn on teams so that this is not lost.
- **Add additional nightly builds for time zones:** We ended up having two builds one at 1 pm and another at 6 pm PST. The 1 pm build gave us time to fix any installer issues before the 6 pm build—which had to be ready for the start of the India-based test team's day.

Currently patterns & practices projects are working towards real time trend analysis of code coverage and static analysis data.

9. Notes

Ade Miller (ade.miller@microsoft.com) is currently the Development Lead for Microsoft's patterns & practices group. He writes about his experiences in agile software development and other related topics on his blog, #2782 (http://www.ademiller.com/tech/). The information and recommendations in this article represent his personal views and do not necessarily represent the view of his employer, Microsoft Corporation.

I'd like to thank Adam Barr, who asked the questions that prompted me to do the analysis. I'd also like to thank numerous people who gave their feedback on the original drafts of this paper, most notably Alan Ridlehoover. I'd also like to thank RoAnn Corbisier for editing this and previous papers.

10. References

[1] Duvall, Paul, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Ris*k, Addison Wesley, Boston, 2007.

[2] Magennis, Troy. "Continuous Integration and Automated Builds at Enterprise Scale" Online posting. Nov. 2007. <http://aspiring-technology.com/blogs/troym/archive/2007/11/26/DoesContinuousIntegrationScale.aspx>.

[3] Miller, Ade and Eric Carter. "Agility and the Inconceivably Large." Agile 2007, 13-17 Aug, Washington DC, 2007.

[4] Kniberg, Henrik. *Scrum and XP from the Trenches: How we do Scrum*, C4Media Inc., 2007.