Effect of Continuous Integration on Build Health in Undergraduate Team Projects

Effect of Continuous Integration on Release Quality in Undergraduate Team Projects

Suzanne M. Embury and Christopher Page

School of Computer Science, University of Manchester,
Manchester M13 9PL, UK
Suzanne.M.Embury@manchester.ac.uk

Abstract. We present the results of an analysis of the changing patterns of build health quality across 3 cohorts of undergraduate students, in a compulsory software engineering course unit. In the course unit, student teams were asked to make changes to a large open source software system, and to maintain clean release builds as they did so. Release build health (in terms of compiling code and passing unit tests) was explicitly included in the marking scheme for the coursework.
We set up a continuous integration server to keep track of student build health. Initially, this was used only by TAs in marking student work, but for later cohorts we provided access to continuous integration results from the early stages of each exercise. This has provided us with data on the changing patterns of student build health, with differing access to the CI server, giving an insight into how students learn these skills and the effects of allowing them access to CI results.
We found evidence of a clear improvement in ability to manage build health when CI facilities are made available, but that some student teams were not making use of the facilities to much effect. The improvement effect was strongest on the build health of release builds, corresponding to the area of greatest marks in the marking scheme. The CI results also proved to be very valuable for academic staff, in making the problems with student builds visible.

Keywords: continuous integration, build health, release quality, software engineering education

1 Introduction

In recent years, the School of Computer Science at the University of Manchester has undertaken an extensive revision of software engineering teaching at undergraduate level. The focus was the level 2 compulsory course units in software engineering, taken by between 200 and 270 students each academic year. The course team for these units has a challenging goal. Many of our students have had little or no programming experience before joining us in their first year. After completing the second year, many will go on to undertake a year-long internship in industry, acting as professional software engineers and often working on mission critical developments. The software engineering course unit must somehow bridge this gap, in just 4 contact hours per week.

To meet this goal, we designed a syllabus based on the use of an industrial strength toolkit, focussed on the kinds of brown-field software development tasks that form the bedrock of much software engineering practice. Turning our back on the more traditional document-oriented build-a-project-from-scratch approach, we asked our students to work with a large open source software system consisting of thousands of classes and many thousands of files. Students are asked to fix bugs, add features and refactor code to meet new non-functional requirements, while managing the quality of the code using an extensive test suite, code review, automated build tools and a continuous integration server.

The use of a continuous integration (CI) server has proven to be a key element of this approach. As well as providing students with a very useful skill set, the build health information provided by the CI server has given us an insight into how students learn about managing the quality of their builds, and the effects of introducing these tools on learning. In this paper, we describe how we have gathered data on build health from student coding teams across 3 cohorts, covering the work of around 700 students and around 10,000 builds. We use the data to compare how teams with access to continuous integration tools differ in their ability to release clean code (and to keep their development branch clean) with teams with reduced or no access to CI build results. The results indicate that embedding build health into marking schemes is not enough in itself to encourage students to maintain clean builds, even on key deliverables such as released code. However, our results suggest that CI facilities can be significant in helping students to understand the importance of ensuring the code committed to team repositories compiles and passes all automated tests. The results were also a very useful diagnostic tool for staff, in understanding how much ground we had to make up in our courses in order to help students develop the discipline and habits needed to deliver clean code in a professional manner.

The remainder of this paper is organised as follows. In Section 2, we examine how CI tools have been used in undergraduate teaching, as reported in the research literature. We then describe how we have used CI at the University of Manchester to support our undergraduate course units in software engineering (Section 3). The experimental design is outlined (Section 4) along with details of the data gathering pipeline we used to infer build health for cohorts where CI was not in place at the time of teaching (Section 5). The results of our analysis are presented (Section 6) and threats to validity are discussed (Section 7). Finally, we conclude and suggest some directions for future work (Section 8).

2 Literature Survey

With the increase in the use of CI (and related technologies such as continuous delivery) in industrial practice, there have been corresponding attempts by academic faculty to include CI within relevant course units. A common approach is to embed CI tools within a software engineering project, with the aim of giving students experience of working with this important class of software engineering tool [Wil01,LD11,MTU17]. Significantly, Süβ and Billingsley demonstrated the ability of CI tooling to allow a small number of academic staff and teaching assistants to teach a project-oriented software engineering course for far more students than would have been practicable without it [SB12].

Others have set up CI infrastructure for use across multiple courses or projects. The work of Pedrazzini is an example [Ped10], in which the TeamCity[1] CI tool is configured by staff for use by students, who then tailoring the build script to suit the specific project being worked on. Pedrazzini notes the learning benefits obtains, including subject specific learning outcomes (such as increased understanding of release management, and of the value of regression testing) and transferable knowledge (such as the ability to track personal improvement over time using the CI reports).

Some proposals for the introduction of CI tools into undergraduate curricula have been motivated additionally by their potential to facilitate the automation of assessment. Heckman and King report on the *Canary Framework*, an infrastructure based on Eclipse, GitHub and Jenkins for automatically assessing software development exercises [HK18]. They found scaling benefits (particularly significant decreases in the time needed to mark work), and also note that the combination of software tools used in their project provides the opportunity to carry out learner analytics. For example, they looked at the relationship between submission times, submission frequency and grades, and found useful correlations. They mention requiring students to keep the release branch clean, though details of how this is converted into marks or assessment are not given.

As witnessed by some of the other papers presented at the DEVOP'18 workshop, academics are increasingly turning their attention to approaches to teaching DevOps and related concepts [Chr16]. Eddy *et al.* proposed a pipeline using Jenkins and Docker to teach the concepts of continuous integration and continuous delivery [EWC$^+$17]. They evaluated the resulting teaching materials with a cohort of 16 senior students, by taking them through an example set of exercises, and asking them to complete a survey afterwards. The results showed that the students appreciated the additional concepts they were able to learn through the approach, but also pointed to some areas for improvement.

We were able to find very few attempts to assess the effects of providing CI tools to undergraduates in terms of learning outcomes or skills gained. Most reports of courses using CI do not attempt to evaluate the usefulness of what was done, or else they rely on collating the results of surveys of learners, shortly after being exposed to the technology. Some attempts to be more systematic exist, however, such as the work of Billingsley and Steel, who compared two successive cohorts of a course unit, when improvements were made for the teaching of the second cohort [BS13]. Amongst the metrics compared between the cohorts are: number of commits made per week, and number of comments per issue tracker ticket. Bowyer and Hughes used a CI server as the basis for a course teaching test-driven development [Bec03]. They extracted a number of metrics from the CI system: proportion of time with a failed build and number of overnight failing builds are two that are mentioned explicitly. However, these metrics are extracted

[1] `/www.jetbrains.com/teamcity/`

for the purposes of assessing students in their proposal, and not to evaluate the usefulness of the course. Another very interesting proposal is the Prof. CI system, proposed by Matthies *et al.* [MTU17], in which CI tools replace the more usual browser-based coding environments. Prof. CI is intended primarily to teach TDD, and while the authors have compared data from 2 cohorts in order to understand the strengths and weaknesses of the approach, they focussed on metrics relating to the number and quality of tests written, rather than on build health.

To the best of our knowledge, no work has yet attempted to assess students' ability to manage their build health, with and without the use of CI tools.

3 Continuous Integration at Manchester

Our first semester second year software engineering course unit asks students to make multiple changes to a large open-source code base, using a number of best practice tools and techniques to ensure the code that is released has no obvious flaws. After consultation with the members of our School's Industry Club[2], we selected the following toolset, to be used by students during the course unit:

- A distributed version control system (Git)
- An issue tracker (GitLab issue tracker)
- An automated test tool (JUnit)
- A code coverage tool (JaCoCo)
- A code review tool (GitLab)
- An automated build tool (Ant, chosen because used by the OSS we are using)
- A continuous integration server (Jenkins[3]

In order to manage the risks inherent in introducing so many new technologies for use by large numbers of students, we brought them into use incrementally (following a lean startup model [Rei11]). The CI server gave particular concerns, since we needed to integrate the Jenkins authentication with the University's own authentication system. If we did not set the security parameters correctly for each job, then the work of one team could be visible to other teams, raising the prospect that work could be copied or shared inappropriately. With such large numbers of students to manage, it was necessary to automate the set-up process using scripts. It is not practical to check that the set up is correct for individual students, so any errors in the scripts could compromise the exercise. We therefore elected to introduce the use of the CI system gradually, across three consecutive cohorts. This allowed us time to discover security and other set up problems before giving full access to students.

Introduction of the CI tools into the coursework took place as follows:

[2] `http://www.cs.manchester.ac.uk/employability/industry-club/`
[3] `jenkins.io`

- Cohort 1: for the first cohort, we set up CI builds for each student team, but did not make these visible to the undergraduates. They were used only by the course staff and teaching assistants, to help them assign marks to teams for their build health, and to provide feedback on build health problems.
- Cohort 2a: for the next cohort of students covered by this study, we set up the same pattern of build jobs for each student team, and allowed them to view the results just before the deadlines. This gave teams the option of making final changes to fix major build health problems before marking. These builds were also used by teaching assistants during the marking process.
- Cohort 3a: for the final cohort included in this study, we provided access to the CI build job results to all teams, from early in each coursework exercise. Most of the builds we provided are performing a continuous build and test function, but for this cohort we also provided true continuous integration, by setting up builds that merged feature branches with the development branch and reported on the result.

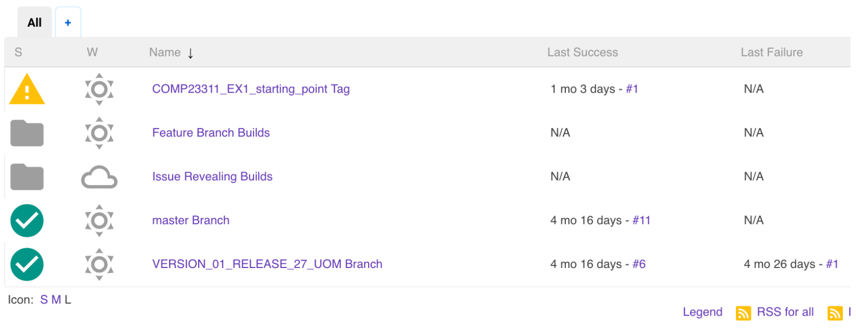
For each team and for each coursework exercise, we set up a number of builds:

- A build of a tag which marks the starting commit for an exercise. This helps teams to see if there are problems with the build at the start of the exercise, that they should fix before beginning work on the exercise itself. (Otherwise, problems from earlier exercises can leach through to the current exercise, and cause problems for the final release build.)
- A build of the release tag for the exercise. Students were asked to make sure this build was compiled cleanly and passed all tests. Marks were lost if this was not the case.
- A build of the development branch. Students were asked to keep this build compiling cleanly and passing the tests, as far as possible but, since (for many of our students) this was the first time they were required to consider build health, they were not penalised for falling short of this provided release build health was not impacted.
- A build of each feature branch. Students were asked to make sure feature branches compiled cleanly and passed all the tests, but were only penalised for falling short of this in terms of marks at the point where feature branches are merged into the development branch.

We use the Jenkins continuous integration server in our course units[4]. Figure 1 shows an example of the build jobs that are set up for an exercise[5]. Figure 2 shows an example of the feature branch builds for an exercise with defined feature branches. Later exercises ask students to manage their own feature branches; in this case, a single build job is created that builds on any push to any feature branch with a prefix set by the exercise.

[4] `jenkins.io`

[5] The Issue Revealing Builds folder contains builds set up to check that students are working test-first. They are not relevant to the experiment carried out in this paper.

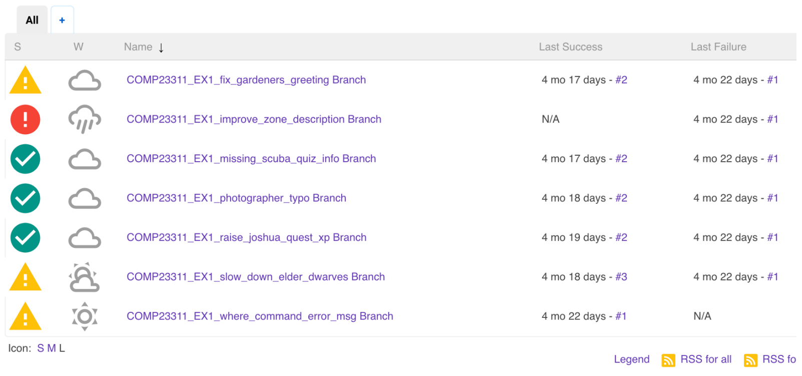

All +

S	W	Name ↓	Last Success	Last Failure
⚠	☀	COMP23311_EX1_starting_point Tag	1 mo 3 days - #1	N/A
📁	☀	Feature Branch Builds	N/A	N/A
📁	☁	Issue Revealing Builds	N/A	N/A
✓	☀	master Branch	4 mo 16 days - #11	N/A
✓	☀	VERSION_01_RELEASE_27_UOM Branch	4 mo 16 days - #6	4 mo 26 days - #1

Icon: S M L

Legend RSS for all I

Fig. 1. Example Build Jobs Set Up for An Exercise


All +

S	W	Name ↓	Last Success	Last Failure
⚠	☁	COMP23311_EX1_fix_gardeners_greeting Branch	4 mo 17 days - #2	4 mo 22 days - #1
❗	🌧	COMP23311_EX1_improve_zone_description Branch	N/A	4 mo 22 days - #1
✓	☁	COMP23311_EX1_missing_scuba_quiz_info Branch	4 mo 17 days - #2	4 mo 22 days - #1
✓	☁	COMP23311_EX1_photographer_typo Branch	4 mo 18 days - #2	4 mo 22 days - #1
✓	☁	COMP23311_EX1_raise_joshua_quest_xp Branch	4 mo 19 days - #2	4 mo 22 days - #1
⚠	⛅	COMP23311_EX1_slow_down_elder_dwarves Branch	4 mo 18 days - #3	4 mo 22 days - #1
⚠	☀	COMP23311_EX1_where_command_error_msg Branch	4 mo 22 days - #1	N/A

Icon: S M L

Legend RSS for all RSS fo

Fig. 2. Example Feature Branch Build Jobs

The icons to the left of the build jobs in this figure show the status of the most recent build. The red icon indicates a *failed* build. This means that some error prevented the build job from even creating any executable code. Compilation errors are the most common cause of this type of build status amongst our students. The yellow icon indicates an *unstable* build. Executable code was created and executed, but some quality indicator has flagged up a problem. For our students, this quality indicator is failure of one or more of the automated tests. Finally, the green icon indicates a *successful* build; an executable could be created and all tests and quality checks passed when run against it.

4 Experiment Design

It will be seen that the incremental introduction of CI tools into our course unit provides the set up for a *natural experiment* into the effects of giving CI facilities to UG students. In all three cohorts, release build health was factored into the marking scheme for the exercise (counting for around 10% of the marks for each of the three exercises). Build health of the development branch and feature branches at the point of merging was also included, though to a lesser degree. Student teams therefore already had a strong reason for doing their best to manage the health of their builds. However, teams in the earlier cohorts had to do this by remembering to check their code health by running the build script and tests on their own local repository before committing code to their Git repository. Students were asked to use an IDE for managing their code, and therefore should have had automatically generated warnings about compilation errors in their code, but other build errors and failing tests could only be found if students remembered to run the (simple to use and fully configured) automated build script before committing code to Git.

In the latest cohort of the three covered by this paper, our CI server was set up to check build health on every push to the remote repository, and to provide student teams with a clear report on their build health at each stage. These students had to remember to log in and check the CI server results after pushing code, but had the advantage that build results from pushes by all individuals were visible to the whole team. Only one team member had to remember to check the results regularly for the whole team to have early warning of problems. The second cohort had access to the CI reports just before their deadline, giving them a final opportunity to manage the health of their release builds only.

By examining the different build health patterns across these cohorts, we hoped to be able to see correlations between the degree of access to CI build reports, and to understand whether the (not insignificant) effort involved in configuring and running the CI server for classes of more than 200 students was delivering useful educational benefits or not.

5 Data Gathering Pipeline

To understand the effects of introducing CI facilities for our students, we had to be able to compare the detailed build health records of cohorts that had access to CI with those that did not. Accessing the build health of teams in the third cohort was straightforward. The CI server we were using provided an easy-to-use API from which we could extract information about build jobs and their statuses programmatically. But for the cohorts that did not have CI in place throughout, we needed a way to reconstruct the build health reports these teams *would have seen* if CI had been running for their teams from the beginning.

Figure 3 shows the sequence of steps involved in extracting the build results for analysis. We first work out which commits would have been built by the CI server, if it had been set up at the time. These are the commits at the tip of each branch that was pushed to the remote Git repository by any team member. For some of our cohorts, we were able to extract this information from the *activity logs* maintained automatically by our School's GitLab[6] server. These logs record details of all the major operations performed on project managed by the server,

[6] `gitlab.cs.man.ac.uk`

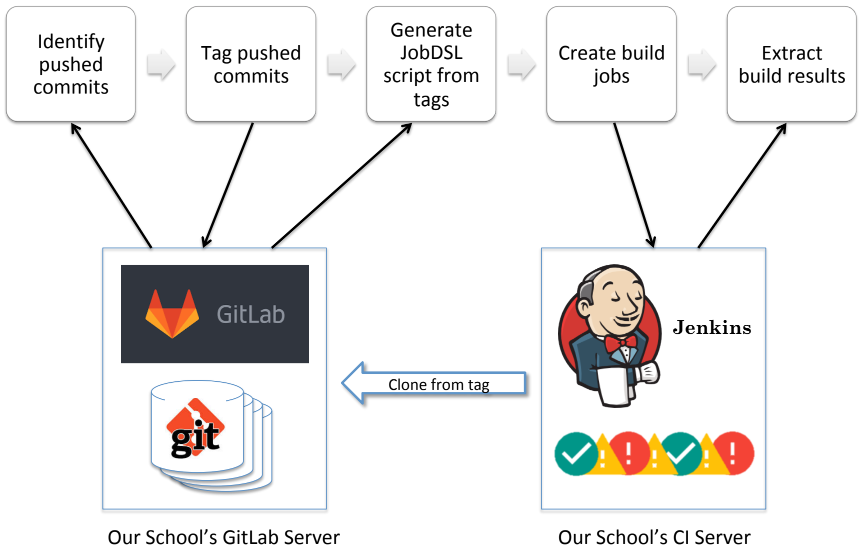


Fig. 3. The data extraction pipeline used to extract build health results from all cohorts.

including giving details of pushes to the repository. The activity log includes a time stamp for the push and the *HEAD* commit for the branch immediately after the push. But for some of the older cohorts, these logs were lost, as a result of a major system upgrade. By chance, the activity logs for the second cohort had been cached by the software we use for automatically marking part of the students' work. For the first cohort, we were able to reconstruct the pushes using timestamps on the objects in the Git database.

Having identified the pushed commits for all cohorts, we used a back-end script to create tags at each such commit. We used tags with the general form "SELA/<commit SHA>" to uniquely identify the point of each push. (SELA,here, stands for "Software Engineering Learner Analytics"). When the tags were created, we used a second script to read each repository, searching for the tags, and to create a script for each one using the Jenkins Job DSL[7]. This provides a means of creating a large number of Jenkins build jobs automatically, using a configuration-as-code approach. The Jenkins Job DSL script we generated created one build job for each commit across all cohorts with a tag with the "SELA/" prefix.

Since there were a great many builds to run (more than 20,000 in total, including builds for some cohorts not mentioned in this paper, due to space

[7] `jenkinsci.github.io/job-dsl-plugin`

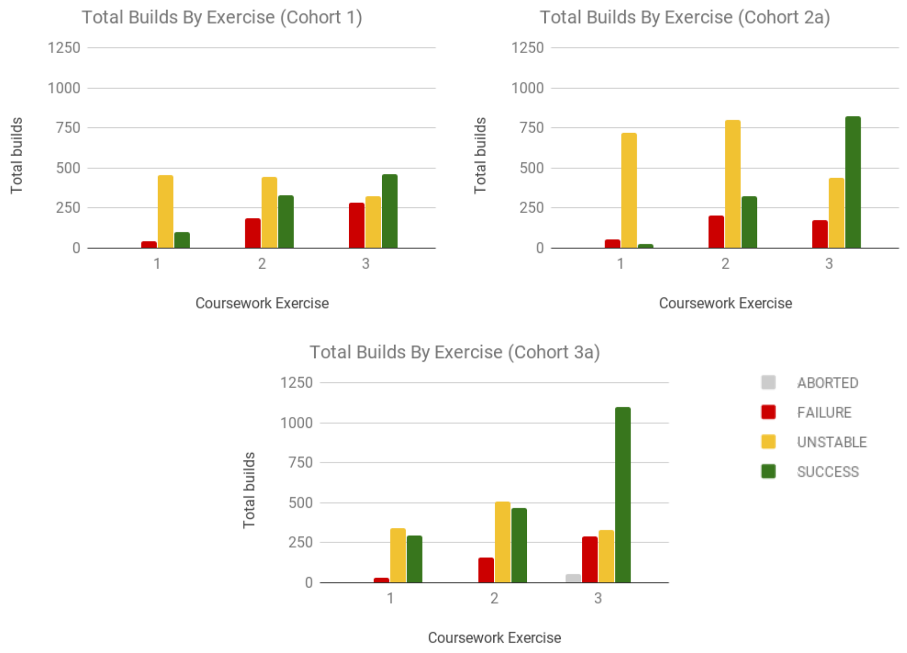


Fig. 4. Total number of builds of each build status, grouped by exercise, for the main cohorts.

limitations), we ran the builds in stages. Once the builds for a cohort were complete, we ran a further script to extract the build results from the CI server (using the provided API) and into a spreadsheet for analysis.

6 Results

6.1 Ability to Manage Overall Build Health

We first examined the results to see how well teams in each cohort were able to manage the health of their builds overall. Figure 4 shows the total number of builds created by teams in each cohort, for each exercise. Builds are separated out into the various possible build results: primarily, failed builds, unstable builds and successful builds[8].

Looking across the three cohorts, we can see several trends:

[8] The small number of aborted builds in the final exercise for cohort 3 result from a feature of the exercise set, which caused GUI code which passed the tests successfully on a desktop machine to fail them when run on the headless server running Jenkins.

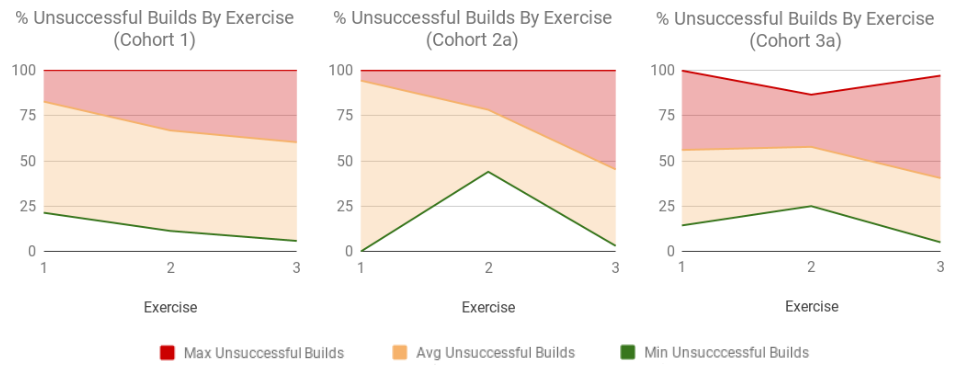


Fig. 5. Percentage of unsuccessful builds, grouped by exercise, for the main cohorts.

- The number of pushes being made increases across the semester, for all cohorts. This is most likely a factor of the exercises being attempted. In the first exercise, teams are asked to fix a number of small bugs, while in the second exercise they add small but complete features to the code base. In the final exercise, they re-factor a significant section of the code base. That is, the scale of the exercise set increases across the semester, leading naturally to increased commits and pushes. Increased student confidence with the use of Git for team coding could also be a factor.
- The proportion of successful builds increases throughout the semester in all cohorts, but the effect is most noticeable in the cohorts with at least some access to CI, and is most pronounced for the cohort with access to CI throughout.
- A significant number of failed and unstable builds remain, even in the final exercises and in cohorts will full access to CI. This makes sense because CI tools can only detect failing builds when they have reached the remote Git repository. They make detecting such builds quick and easy, but they don't prevent them in the first place. And students have to remember and choose to access the CI build system. For the final cohort, we offered to set up e-mail notification for teams on unsuccessful builds, but only a handful of teams took us up on this.

6.2 Spread of Ability to Manage Overall Build Health

The figures for the total builds made of each status, given in Figure 4, give an overview of how the cohorts were managing their build health across the semester. What they don't show is the spread of abilities across the cohort. Some teams may be managing their build health well, while others may be doing a much poorer job. We wanted to understand the spread of ability in this respect across our cohorts, to compare how the average teams were performing against the best teams, and against those teams that were struggling the most.

To understand this, we created the charts in Figure 5. These show, for each cohort, information about the percentage of builds created by individual teams that were unsuccessful. Here, we define *unsuccessful* as meaning any aborted, failed or unstable build. The middle, orange, line in the charts shows the average percentage of unsuccessful builds across all the teams in a cohort. So, for cohort 1, in exercise 1, on average, around 80% of builds were unsuccessful. By exercise 3, this had dropped to approximately 60% of builds being unsuccessful.

The green line at the bottom shows the percentage of unsuccessful builds for the team that was managing their build health the best, and the red line at the top shows the percentage of unsuccessful builds for the team that was faring the worst in this respect. It will be noted that in all three cohorts, some teams made only unsuccessful builds.

Looking at the trends visible across the three cohorts, we can see an improvement for each cohort as the semester progresses: the average percentage of unsuccessful builds drops as the exercises progress, and students get more used to managing their build health (with or without CI). There is an improvement in the cohorts that had access to CI, but perhaps not as much as we had hoped. The average number of unsuccessful builds is around 50% or lower for cohort 3, across all exercises, but that is still a lot of broken builds. And the best team is arguably not doing much better than the best team in cohort 1.

So, these figures show a small improvement in the ability to manage build health, but not the revolution in practice that we might have expected from the introduction of full CI.

6.3 Ability to Manage Release Build Health

The results presented so far show a somewhat mixed picture. CI tools seem to be helping our students to manage their build health, but there are still many unsuccessful builds being created. When we confine our attention to the health of the code released by students at the end of each exercise, however, the picture is somewhat clearer. The results in figure 6 show the total number of release builds for each build status, per exercise, per cohort. So, for example, for exercise 1, 6 cohort 1 teams released code that failed to compile, while 12 cohort 1 teams released code that failed some tests and 14 teams managed to release code that compiled and passed the quality checks.

When comparing across the cohorts in this case, we can see a marked improvement when CI facilities are made available compared with when they are not. While all cohorts included some teams that released broken code, the proportion of teams that were able to release a clean code base by the end of the semester was markedly improved when CI was available. For the third cohort, this was the case even from the first coursework exercise, while second cohort teams (who had access to CI only at the end of each exercise) appear to have taken longer to learn about the need to manage release code quality, and how the CI server can assist with this. However, by the end of the semester for the

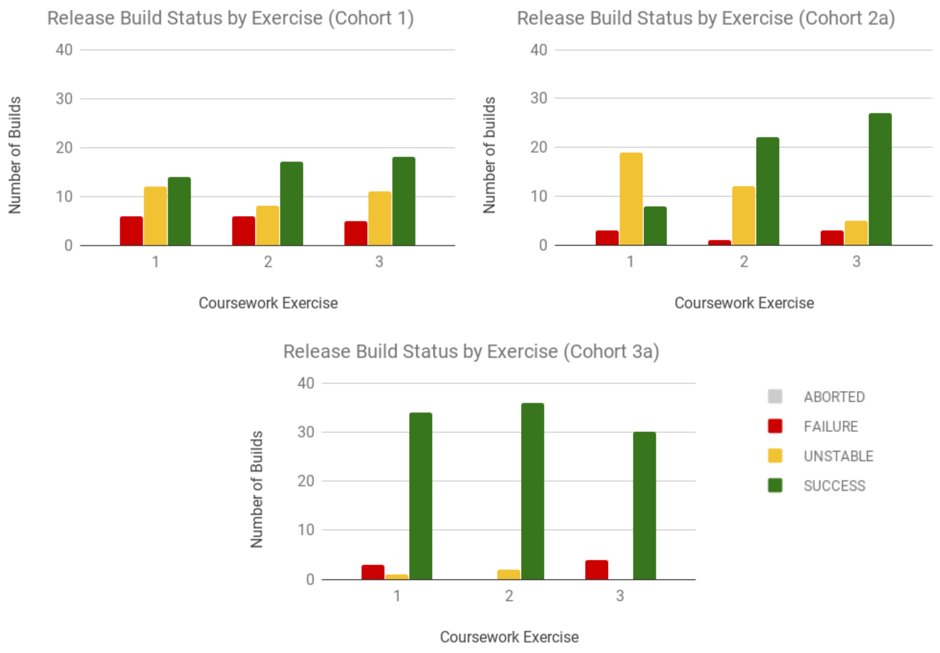


Fig. 6. Total release builds for each build status, grouped by exercise, for the main cohorts.

second cohort, the majority of teams have learnt how to release compiling code that passes all the tests[9].

7 Threats to Validity

This paper describes an opportunistic attempt to extract lessons from the data sets that happened to be available, rather than a properly designed controlled study. The difficulties of running such studies in an educational context are well known. As Holmboe, McIver and George have stated:

> "[...] there are obvious difficulties in empirically evaluating [teaching innovations] aside from the expense of running two concurrent courses and comparing results, such techniques would be ethically dubious, potentially disadvantaging students in one course or the other. Where comparisons can be done across different years, the number of changes be-

[9] As part of the assessment process, we monitored whether teams disabled or deleted test cases, and found no teams that were doing this in an obviously fraudulent way. The builds released genuinely did pass the 1800 plus unit tests describing the required behaviour of the system.

tween the courses makes it difficult, if not impossible, to evaluate the effect of individual changes." ([HMG01], p.4)

A great many such differences occurred between the cohorts examined in this study, of which the most significant are probably:

– Differences in cohort sizes (ranging from 200 to 270), which go some way towards explaining the differences in number of pushes between cohorts.
– Differences in the exercises set. Since the use of Git means that any student can take and share a full copy of their team's solution to the exercise, we have assigned a different set of bugs to be fixed and features to be shared in each academic year the course unit is taught. A different aspect of the code was also chosen to be refactored. While we attempted to set work of a similar size ain each academic year, these differences could explain the differences in the number of pushes and the difficulty of making tests pass. (It was certainly a factor in affecting the number of aborted builds.)
– Differences in the open source code base used as the basis for the exercise. Since we aim to give students experience of working on a live code base, we update the open source code base used for our exercises every year. This means that each year we teach on a slightly larger and more complex code base, with new features added that are likely to be a little unstable in ways that are hard for us to predict. (Indeed, we exploit this feature in order to find the bugs that the teams will fix.) Significantly for our analysis, each year the set of automated tests is slightly different, with different fragilities and defect finding powers. The high number of students releasing unstable builds in cohort 2a, for example, may have been affected by a particularly erratic test behaviour that was present in the code base used for that year.
– Differences in teaching approaches used. We learnt a lot about how to teach the use of these and related tools over the course of the three academic sessions covered by this paper, and have made significant changes to (and hopefully improvement of) our teaching materials and approaches. This could have affected our students' confidence with using Git as a tool for team coding. This could certainly be a partial explanation for the disproportionate increase in the number of pushes being made by later cohorts. How far students' increasing confidence and familiarity with the CI tools is a factor in this confidence with Git is hard to untangle with any precision.

8 Conclusions and Future Work

In this paper, we have examined data gathered on the health of builds produced by software teams, working to make changes to a large open source code base. For all cohorts, build health was a significant part of the marking scheme for the exercises the students undertook. We compared the statuses of builds produced by teams with no access to CI, limited access to CI and full access to CI. We were able to observe an improvement in the overall ability manage build health when CI facilities are provided, though the effect was only strongly visible in the release builds for the cohorts.

A significant unexpected benefit of introducing CI facilities into our course unit was that it raised the visibility of the fact that our students were struggling to form the habits needed to regularly commit clean code. For example, we had one team that never made a single clean build across the whole semester, and which pushed code 22 times for their solution to one exercise without ever pushing code that compiled. While not quite at this level, a significant number of other teams were discovered to be having difficulty in controlling the quality of the code they committed to their repository. This issue was formerly invisible to our students and (more importantly) to the teaching team. Now that we are aware of the issue, we can work on developing our teaching approaches to help guide students into better habits in this respect.

There are a number of further analyses we could carry out with the data we have collected from the cohorts in this study. We have been making use of the Jenkins Build Failure Analyser plug-in[10] to provide a more detailed characterisation of the causes of failing builds. This plug-in scans the console log for each build and classifies the build according to the types of failure it finds matches for in the log. It can therefore distinguish, for example, a failed build caused by a compilation error in code from a failed build caused by a run-time exception, from a failed build caused by a missing file. Using the data gathered by this plug-in, we will be able to perform intra- and inter-cohort analyses of the errors being made that lead to failing builds, and can design teaching materials that guide students to be aware of these common pitfalls.

We are also exploring ways in which we have configure our CI server to give students better tools for recovering from failed builds, and for preventing them in the first place. We will experiment with setting up e-mail notification of unsuccessful builds to all teams, rather than only for those that request it, as at present. The effect of this must be carefully monitored, however, as we would not want students to be deterred from pushing code because they are afraid of their errors being broadcast to their team. Other options are to design a GitLab sandbox, in which individual team members can apply their local commits to the team remote repository hypothetically, to examine their effects, before pushing them to the team repository. This would also allow CI to be applied to the hypothetical commits and merges, before any poor quality code reaches the other members of the team.

Acknowledgements

We are grateful to all the University of Manchester students who have taken the course units on which the work in this paper was based, for their willingness to engage with new approaches to teaching and for their (sometimes robust) feedback on our course units. We also thank the other members of the academic

[10] `https://github.com/jenkinsci/build-failure-analyzer-plugin`

teaching teams we work with for their help in keeping the course units running. A portion of the work reported in this paper was funded by the Higher Education Funding Council for England, under the aegis of the UK Institute of Coding[11].

References

[Bec03] Kent Beck. *Test-Driven Development: by Example.* Addison-Wesley Professional, 2003.

[BS13] William Billingsley and Jim Steel. A Comparison of Two Iterations of a Software Studio Course Based on Continuous Integration. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, pages 213–218. ACM, 2013.

[Chr16] Henrik Bærbak Christensen. Teaching DevOps and Cloud Computing using a Cognitive Apprenticeship and Story-Telling Approach. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 174–179. ACM, 2016.

[EWC+17] Brian P Eddy, Norman Wilde, Nathan A Cooper, Bhavyansh Mishra, Valeria S Gamboa, Keenal M Shah, Adrian M Deleon, and Nikolai A Shields. A Pilot Study on Introducing Continuous Integration and Delivery into Undergraduate Software Engineering Courses. In *Proceedings of 30th IEEE Conference on Software Engineering Education and Training (CSEE&T)*, pages 47–56. IEEE, 2017.

[HK18] Sarah Heckman and Jason King. Developing Software Engineering Skills using Real Tools for Automated Grading. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 794–799. ACM, 2018.

[HMG01] Christian Holmboe, Linda McIver, and Carlisle E George. Research Agenda for Computer Science Education. In *Proceedings of 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG-13)*, volume 13, 2001.

[LD11] Baochuan Lu and Tim DeClue. Teaching Agile Methodology in a Software Engineering Capstone Course. *Journal of Computing Sciences in Colleges*, 26(5):293–299, 2011.

[MTU17] Christoph Matthies, Arian Treffer, and Matthias Uflacker. Prof. CI: Employing Continuous Integration Services and Github Workflows to Teach Test-Driven Development. In *Frontiers in Education Conference (FIE)*, pages 1–8. IEEE, 2017.

[Ped10] Sandro Pedrazzini. Exploiting the Advantages of Continuous Integration in Software Engineering Learning Projects. *Koli Calling*, page 35, 2010.

[Rei11] Eric Reis. *The Lean Startup.* 2011.

[SB12] Jörn Guy Süβ and William Billingsley. Using Continuous Integration of Code and Content to Teach Software Engineering with Limited Resources. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1175–1184. IEEE Press, 2012.

[Wil01] Dwight Wilson. Teaching XP: a Case Study. In *XP Universe*, 2001.

[11] `instituteofcoding.org`, `ioc.manchester.ac.uk`