How well do student software engineering teams practice Continuous Integration?

Joshua Zeitsoff

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-183
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-183.html

October 9, 2020

Acknowledgement

I'd like to thank my advisor, Armando Fox, for his mentorship and guidance this year as well as throughout my time at ACELab. This master's report wouldn't be possible without your immeasurable support and patience.

I'd also like to thank my second reader, Michael Ball, for offering insightful feedback during the formulation and writing of my master's project.

Thank you to my Mom and Dad for your love, support, and encouragement each day.
We did this together.

Thank you to Sophia for supporting me and inspiring me to stay focused in the most challenging year yet.

Thank you to all the family and friends who shared this journey with me.

How well do student software engineering teams practice Continuous Integration?

by Joshua Isaac Zeitsoff

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Armando Fox
Research Advisor

August 14, 2020

(Date)

* * * * * * *

Professor Michael Ball
Second Reader

August 13, 2020

(Date)

How well do student software engineering teams practice Continuous Integration?

Joshua Zeitsoff
jzeitsoff@berkeley.edu
University of California, Berkeley
Berkeley, California

ABSTRACT

Student teams in software engineering courses are taught many processes to make collaboration easier, yet it is challenging to know if these processes are being followed when teamwork occurs out of view of the instructor. To gain a better understanding of what the teams are doing, we propose measuring the data exhaust generated by web-based project management, code management, and deployment management tools. To help students in our course coordinate their work with other developers, we recommend a simplified workflow based on tasks and tools, such as Pivotal Tracker, GitHub, and Heroku, common to the Agile/XP practice of Continuous Integration. We designed several tools to help us determine if students were following our recommended workflow. We analyze a current offering of a course in which students are taught our recommended workflow. Our results suggest that teams did follow the recommended workflow but at various levels of compliance; that many teams show substantial variation in compliance among individual team members; and that our recommended workflow should be expanded to explicitly address some student behaviors that we did not foresee.

1 INTRODUCTION

A common method of teaching the eXtreme Programming (XP) [4] variant of Agile software development [9] is through team-based projects. Students are put into teams of 4-6 and work on projects designing and creating a web application for several iterative cycles. Some courses may use internal projects and some may connect students with a real world customer who intends on using the software they create. We want to teach students how to be effective and disciplined developers when working in software engineering teams.

Students writing software as part of a team need to coordinate their work with other developers. Disciplined software engineering teams follow processes that make the task of coordination easier. A broad name for one important category of such processes is "continuous integration." Continuous Integration is the process by which developers working on different project tasks complete their code and tests, have it reviewed and approved by rest of team, and integrate it into the mainline code in a timely manner [8].

Continuous Integration is just one process that makes coordinating within teams easier. Students in software engineering courses are taught many different processes to make working in teams easier. One challenge instructors face is to be able to verify whether students are learning and following the processes that are being taught. In an ideal scenario, an instructor or teaching assistant would be able to directly observe all actions each student takes. The instructor could then come to a conclusion as to whether processes are being understood and followed. However, constant shadowing is not feasible in classes where students greatly outnumber course staff.

We would still like to know how closely students are following the processes being taught. Our idea is to leverage the tools used by software engineering teams for version control, project management, and deployment to understand what students are doing. These tools all produce data exhaust that can be used to paint a picture of the actions students are taking while writing software, even if these actions are occurring out of the view of an instructor.

One obstacle to using tool data generated by students is that processes taught to students are complex and involve the usage of several tools. Any attempt to understand what students are doing needs to connect the actions they are performing across different tools. We discuss several possible methods to connect actions from different services.

Using one method of connecting events across different services, automatic tool integration, we investigated if students in an upper-division software engineering course followed our recommended processes. Our findings suggest that many teams did follow our processes, although at different levels of compliance. Teams appearing to not follow our processes may have had varying levels of student compliance. Our results also showed that some student behavior was not anticipated in our recommended workflow and should be addressed in future courses.

Our contributions are as follows:

(1) We present a recommended workflow that focuses on connecting events across different services to help teach student software engineering teams how to follow the Agile practice of Continuous Integration.
(2) We discuss several different methods of connecting events across services and present results obtained by using one of these methods.
(3) We show how to analyze whether student teams from a course offering of a upper-division project-based software engineering course are following our recommended workflow.

The rest of the paper is organized as follows. Section 2 compares our approach with previous studies that introduced Continuous Integration in software engineering courses. Section 3 describes how we organize the tasks involved in Continuous Integration into a timeline of events occurring across different services. Section 4 discusses three different ways events across services could be correlated. Section 5 describes three tools we created to correlate events across different services. Section 6 outlines results from using one method to correlate events across different services for

Task	Tool & Action	Pivotal Tracker Story State
Writing code for a new feature	GitHub Branch Creation	started
Reviewing code amongst the team	GitHub Open Pull Request	finished
Merging code into the main branch	GitHub Merge Pull Request	delivered
Running tests	TravisCI Build	delivered
Deploying code to a staging environment	Heroku Release	delivered
Letting a customer see new feature	Heroku Staging Environment	delivered

Figure 1: GitHub is a tool used for version control, Heroku is a tool used for deployment and Pivotal Tracker is a tool used for project management. We are looking for associations between Pivotal Tracker story state changes and Continuous Integration activity that should accompany those state changes. This corresponds to columns 2 and 3. Rows of event types we want to correlate are highlighted.

a semester of a software engineering course. Section 7 discusses threats to validity and future work.

2 RELATED WORKS

An investigation of which Agile practices should be taught in software engineering courses [10] was based on a taxonomy of team practices as well as a field study with an established software company. When surveyed about the importance of various Agile practices, professional developers ranked Continuous Integration second most important, behind the practice of Retrospective. Their ranking supports our decision to measure if students are following the practice of Continuous Integration among other Agile practices.

Prior work introduced Continuous Integration and Test Driven Development concepts and tools to software engineering students [6]. Afterwards, students completed a week long project and were evaluated on their ability to integrate Continuous Integration and Test Driven Development into their development cycle based on tool data and qualitative assessments of their code base. In contrast, our intention is to use automated tooling to collect data and assess student compliance with recommended Continuous Integration practices.

Instructors have also introduced the practice of Continuous Integration as a means to teach software engineering to large courses of students working on the same code base with limited instructional staff [13]. Continuous Integration tools showing build failures, new code additions, and failing tests made it easy for the students and instructors to monitor class progress. However, they focused on using Continuous Integration as a means to teach a course with limited resources and not on teaching students about the practice of Continuous Integration and what processes students should be following. A comparison between that course and a newer version of the course [5] showed that most of the improvements were in student clarity regarding grading and student perception of the course, and not on explicit teaching of Continuous Integration.

More recently, some instructors have created their own Continuous Integration/Continuous Delivery (CI/CD) pipeline and tested whether its use helped students to better understand the practices of CI/CD [7]. 16 students rated their understanding of various CI/CD topics before and after using the pipeline individually for an hour, and results showed that their understanding of most topics improved. By contrast, our students typically work on a project in teams for four two-week iterations and we use tool data to determine if students are following our recommended Continuous Integration workflow.

3 RECOMMENDED CONTINUOUS INTEGRATION WORKFLOW

In Agile development, Continuous Integration is the process of rapidly integrating new code into the main line of development to allow developers to receive fast feedback from other team members and customers [8]. We focus on the events leading up to the moment of integration. These events are part of a complex process that involves several tasks.

In particular, our recommended Continuous Integration workflow is based on "branch per feature", in which each new software feature is entered into a project-management tool and is developed on its own code branch. In our course, students use the online project-management tool Pivotal Tracker, which represents each feature as a *story* [1] that describes the new feature's value to the customer or development team. In particular, a story is marked as "started" when a developer begins working on the associated code and tests, "finished" when the code and tests are ready to be reviewed by the rest of the team, and "delivered" when the reviewed code is merged into the main line of development and deployed for customer evaluation, in our case using the Heroku platform-as-a-service.

Using GitHub to manage a source code repository, new code can be written on a separate *branch* from the main branch. Developers create *Pull Requests* on GitHub when their code is ready to be merged into the main branch. *Pull Requests* can be reviewed and commented on by other members of the team. When a team approves of a new feature, the *Pull Request* can be merged into the main branch of code.

The list below shows how we expect the story lifecycle states to be interleaved with these code-development events; Figure 1 outlines the tasks performed as a part of a general workflow following Continuous Integration and the tools and corresponding Pivotal Tracker story state for each task.

- A developer "claims" a story in Pivotal Tracker
- The developer begins working on code and tests for the new feature, on its own branch

[1] https://www.pivotaltracker.com/help/articles/terminology/#story

- The developer completes the code and tests, and invites the team to review their work
- After review, the code is merged into the main branch
- The new feature is deployed to the customer for final approval

Figure 2 organizes these tasks and story states into a timeline showing which actions on GitHub and Heroku students should be taking at each stage in a Pivotal Tracker story's lifecycle.

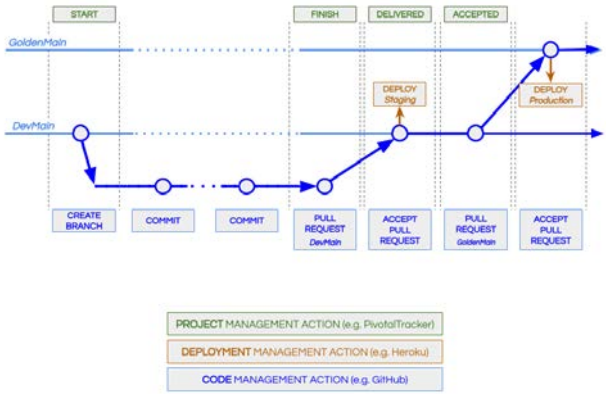

Figure 2: Recommended timeline of events as part of a Continuous Integration workflow

While there are other workflows that teams can follow, this simplified workflow has worked well for our students and is a realistic reflection of the tools they use and the actions they take. We chose this simplified version as it best allows us to focus on what the processes are and how we can measure them.

4 CORRELATING EVENTS ACROSS SERVICES

The workflow we described incorporates actions on both Pivotal Tracker and GitHub, which are completely separate tools on which the student account names may even be different. To reconstruct student teams' Continuous Integration actions, we must merge data together from different tools to generate a timeline of events, correctly interleaving the events in Figure 1. We describe three ways events can be correlated across services: automatic tool integrations, naming conventions, and correlation in time.

Automatic tool integrations. Pivotal Tracker provides functionality that enables certain events in GitHub to trigger story state changes in an associated Pivotal Tracker project [2]. Once the GitHub repository has been manually connected to a Tracker project, any commit messages that follow certain formatting rules and include the ID number of a Tracker story cause a change in the story's life cycle state, such as from "started" to "finished." However, this method of associating GitHub activity with specific Pivotal Tracker stories only works if the GitHub repository and Pivotal Tracker project have been set up this way from the beginning and if students consistently follow the rules for formatting commit messages.

Naming conventions. Another way to connect events across different services is to establish naming conventions. For example, we could require that students name their GitHub branches in a predictable way that includes the ID of the Pivotal Tracker story that the branch is associated with. This method only works if the naming convention can be consistently enforced.

Correlation in time. For historical projects that neither configured automatic tool integrations nor adhered to naming conventions, we can analyze how closely correlated in time different events were. For example, whenever a team had a Pivotal Tracker story marked as "started," the associated action is to create a branch on GitHub. If a story on Pivotal Tracker changed its state, such as from "unstarted" to "started," we could look within a short time window around this event to see if any Git branch had been created. If any such branch was found, we could reasonably conclude that it was created specifically for the nearest story that had its state changed to "started."

In the projects we studied, we were able to take advantage of automatic tool integration. Future work studying historical student projects may have to rely on one of the other methods.

5 TOOLS

5.1 Motivation

We had access to data from 3 semesters of an upper division (3rd and 4th year students) course on Agile software development at Berkeley. The course is aimed at students who want experience developing a medium-sized software application while working in a team. Enrollment typically reaches approximately 120 students, who are grouped into teams of 4-6. Each team collaborates with a customer, typically a nonprofit or on-campus organization, to create or extend a web application. Students work on their projects for the last 8 weeks of the course, which is broken down into four two-week iterations.

At the end of each iteration, students complete several different surveys evaluating themselves and their team for the past iteration. Students complete self-assessment surveys based on [11, 12] meant to gauge how well their team followed various Agile practices as well as set a goal for the next iteration. They also complete a peer-assessment survey allocating points to each member of the team based on their code and overall contributions for that iteration. Each student's email is collected when they fill out their response. Each team's customer also fills out a feedback survey at the end of each iteration, ranking the team on several questions related to their performance. Figure 3 lists an example self-assessment survey question.

Survey responses were the first data we examined to determine what teams were doing out of view of the instructor. Our goal was to standardize and compare survey responses across three offerings of the course so that all surveys could be analyzed together. This task was challenging for several reasons.

- Some questions had minor changes in wording across semesters, so we had to establish correspondence among questions.
- Response scales (e.g. 0-5 vs. 0-10) sometimes differed across semesters.

[2] https://www.pivotaltracker.com/help/articles/github_integration/

Score	Description
10	I sync with the main repo and close out my feature/bug branches daily or more often
8	I sync with the main repo regularly, but my feature/bug branches may be open for a few days
6	I sync with the main repo every few days; I resolve merge conflicts as needed when merging my changes
4	We work more or less independently but use merges and pull requests "as needed" to stay in sync
0	We do one big merge before trying to push new features to production at the end of the iteration

Figure 3: This is an example self-assessment rubric provided to teams for a question related to Continuous Integration. Some other question topics include Test Driven Development, Collective Ownership, and Pairing.

- The methods of collecting responses, and the data format of those responses, varied from semester to semester.
- The students' email addresses as reported in their responses did not always match the identities used on the course tools, such as GitHub, Tracker, and Heroku.

To address these challenges, standardizing the responses required a combination of manual massaging and automated processing using custom scripts.

As we standardized survey responses, we created a mapping of student and project identities. A mapping of each student and project's identity on each tool used, as well as in the survey responses, will make it easier to analyze survey and tool data together.

5.2 Pseudonymizer

To address the problem of student identifiers differing across services, we created an application that stores a mapping of both student and project identifiers for each service used in the course that we collect data from. The application is deployed as a proxy that sends and receives API requests between any of the services and a client, automatically performing the mappings. Figure 4 shows how our application lies in between clients and API services.

Without our application, each client would need to store every student and project's identifier for each API service. Instead, our application stores this information and provides each client with a unique and randomly generated identifier, or *pseudonym*, for each student and project. Our application is called the "pseudonymizer" since it generates pseudonyms for students and projects, while the unique and randomly generated identifiers are called pseudonymizer IDs.

If a client previously only had a student's real identifier for one service API, but the pseudonymizer was deployed with real student identifiers for several APIs and this client was given an API key for the pseudonymizer, it would be possible for this client to now make requests to any of the service APIs included at deployment of the pseudonymizer. However, clients must obtain an API key for the pseudonymizer in order to access any data about students.

The pseudonymizer uses special formatting in API requests sent to it by clients to determine which identifiers need to be translated. Figure 5 shows how a client would format a student or project's pseudonymizer ID in any request sent to the pseudonymizer. The pseudonymizer looks for specific keys in the JSONs returned by service APIs in order to make a best effort attempt to translate real student identifiers back into their pseudonymizer ID counterparts.

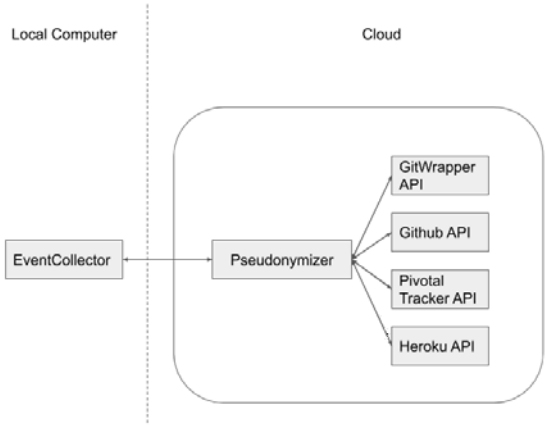

Figure 4: Our pseudonymizer tool forwards requests between clients and API services.

Any clients currently making API requests directly to service APIs would only need to change the request endpoint and parameters to integrate with the pseudonymizer. Figure 6 shows an example URL a client would make a request to before integrating with the pseudonymizer. Figure 7 shows the new URL a client would make a request to. The client should include the service API endpoint as a query parameter in the request to the pseudonymizer with real student identifiers replaced with their pseudonymizer ID counterparts. The pseudonymizer IDs in the URL have been specially formatted so that the pseudonymizer knows to translate them.

The pseudonymizer is hosted in the cloud and deployed with a mapping of student and project identifiers for service APIs.

5.3 GitWrapper

One of our initial goals was to analyze historical projects in which our only option was to use time correlation. We were challenged by the fact that the GitHub API only returns the previous 90 days' worth of activity, since the activity in the projects we were analyzing occurred much longer ago than that. To solve this challenge, we inspected the graph of Git commits to determine branch creation times, since it is common practice to write and commit code on a

API ID	Pseudonymizer ID	Formatted ID
example_github_user	abcd1234	%{user}%{Resolver}%{abcd1234}
example_github_project	efgh5678	%{project}%{Resolver}%{efgh5678}

Figure 5: Example Formatting of IDs in Request to Pseudonymizer

https://api.github.com/repos/example_github_repo/example_github_user/pulls

Figure 6: Example URL for Client without using Pseudonymizer

https://pseudo-pro.herokuapp.com?uri=https://api.github.com/repos/%{user}%{Resolver}%{abcd1234}/%{project}%{Resolver}%{efgh5678}/pulls

Figure 7: Example URL for Client using Pseudonymizer

branch relatively soon after creating it. We designed a service for the convenience of clients that provides the same API as GitHub for branches by using the commit history to extract information.

Our goal was to find the first commit on each branch in a repository and use the dates of those first commits as approximations of when each branch was created. There are two challenges to accomplishing this goal. The first challenge is finding all branches in a repository. The second is finding the first commit on each of those branches. Figure 8 shows an example Git branch structure that we will use for the remainder of this section.

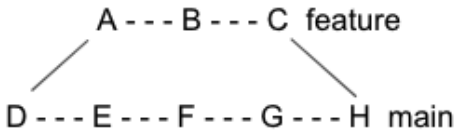

Figure 8: Example Branch Structure

Once branches are deleted, Git can no longer readily access them. However, merged branches can easily be found in the graph of Git commits by looking at the parents of merge commits. A merge commit is a commit generated when one branch is merged into another branch with diverged histories [1]. In Figure 8, commit "H" is an example merge commit with two parents. Commit "G" is the parent from the main branch and commit "C" is the parent from the feature branch. We leveraged the filtering options for the "git log" [2] command to obtain a list of all merge commits.

Given the commit at which a feature branch merged into the main branch, our challenge lay in finding the first commit on that feature branch. In Figure 8, the commit we want to find is represented by commit "A". Our plan was to list all commits on the feature branch from the parent of the merge commit to the last commit on the main branch before the feature branch diverged. In our example, this corresponds to listing all commits from commit "C" to commit "D".

Commit "D" can be found by identifying the best common ancestor between commit "H" and commit "C". Git has a command, "merge-base" [3] to find the best common ancestor between two commits. Given commit "D", we could list all commits from commit "C" back in time to commit "D" and return the creation date of the last commit in that list, commit "A". Repeating this process for each merge commit would give us an estimated creation date for each branch in a repository.

We recognize that our approach will not work for all branch structures, such as merging two branches without diverged histories [1]. However, anecdotally we have reason to believe that this approach covers the branching behavior of most students.

5.4 EventCollector

The EventCollector is a tool that we use built by collaborators. The EventCollector gathers data from service APIs and imputes a percentage of correlated events. A user can specify which services they want to correlate events from, which fields are of interest from the responses returned by APIs, and whether those events should be correlated using a time correlation window or naming conventions. This tool is described in more detail in a paper still in preparation written by our collaborators.

6 RESULTS

We analyzed projects from a summer 2020 offering of an Agile software engineering course aimed at juniors and seniors at a large US university. Small teams of students were each given the same starter code for a web application and asked to complete a set of features, some of which were common to all teams and some of which teams could choose among. A typical team consisted of three students, but a few teams contained only one or two students due to others dropping the course. Because the summer offering of the course was condensed in time, the course ran at a faster pace than it would during the academic year, so teams had a single ten-day sprint to complete all the features; during the semester, four two-week long iterations would have been more likely. Each team had a GitHub repository and a Pivotal Tracker project to track progress on the features.

Of 31 teams, four had to be removed from our data set because they showed no Pivotal Tracker activity. Two of these did not properly integrate with the course's research tools, and two projects had no Pivotal Tracker data during the iteration. This left us with

Type	Explanation	Compliant
0 branches	A "started" story that was connected to 0 GitHub branches	No
1 unique branch	A "started" story that was connected to 1 GitHub branch. This GitHub branch was **not** connected to any other "started" stories	Yes
1 non-unique branch	A "started" story that was connected to 1 GitHub branch. This GitHub branch was **also** connected to other "started" stories.	No

Figure 9: Different Story-Branch structures found in student behavior

27 teams who had Pivotal Tracker data during their 10 day long iteration.

Students were directed to documentation[3] describing how to use Pivotal Tracker's automatic integration feature with GitHub. This feature allows teams to link a specific Git branch to a Pivotal Tracker story, after which any Pull Requests opened or merged for that branch are displayed in Pivotal Tracker with the associated story. We examined if students followed each of the 3 pairs of events in Figure 1.

6.1 Does Starting a Story Correspond With Creating a Branch?

In the ideal version of our recommended workflow, each Tracker story is associated with exactly one branch on Git; the branch is created when work on the story is started, and the branch is merged when the story is delivered. In practice, however, we found that the correspondence between stories and branches was not always one-to-one. Figure 9 shows the three different types of branch structures we found. We decided that only stories with their own unique branch were compliant to our recommended process, and that other story-branch structures were not ideal behaviors.

Figure 10 shows that students did not create a Git branch for each story they marked as "started" on Pivotal Tracker. Each red dot on the graph represents one team; teams completed different numbers of stories. Stories that were connected with a Git branch were denoted as **correlated**. Teams following the recommended process have the same number of "started" stories as **correlated** stories, and therefore lie along the diagonal of the graph. However, many teams had more "started" stories than **correlated** stories.

We discovered that the reason for noncompliance was the presence of the other branch structures in Figure 9. Therefore, examining only "started" and **correlated** stories does not provide enough information about these alternative branching behaviors. We categorized the different story-branch structures seen for each team in Figure 11. Teams are sorted in order of how many compliant stories they had. A compliant story is a story with its own unique branch; this branch is not connected to any other stories. We notice that some teams did not manage to connect any of their "started" stories to Git branches, while some teams manage to have every "started" story connected to its own Git branch.

In addition to understanding the compliance of each team, we were also interested in the behaviors of individual students in the team. We measured each team's percentage of compliant stories, as

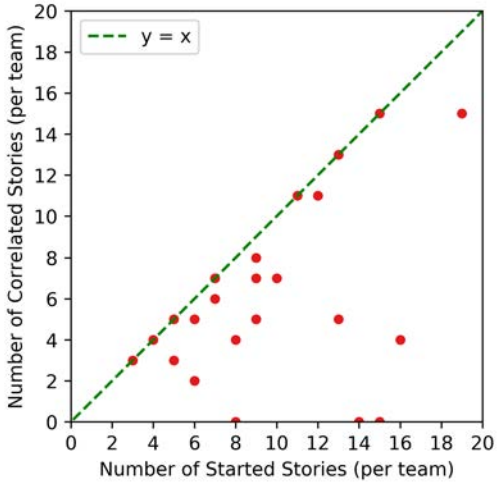

Figure 10: Teams did not link every "started" story with a Git branch. Each red dot represents a team.

well as the percentage of compliant stories for each student. A student's compliance percentage is determined by dividing the number of compliant stories they "started" by the total number of stories they "started". A team's compliance percentage is found by dividing the total number of compliant stories "started" by students within that team by the total number of stories "started" by students within that team. Figure 12 shows how each team's overall percentage of compliant stories compares to the students in that team. Teams are sorted by their percentage of compliant stories. This differs from Figure 11, which is in order of how many total compliant stories they had. While most-compliant teams had all members following the recommended process, behavior varied more widely for less-compliant teams. In particular, around one-fourth of teams include at least one member who was achieving 100% compliance, but the team's overall percentage was brought down by other members of the team.

One challenge affecting our analysis is one student might mark a story as "started", but a different student may create the branch for that story. We found that this occurred for 75 stories out of 246 total stories. Of these 75 stories, 30 of them were stories that belonged to branches with multiple stories. This suggests that a nontrivial amount (30.4%) of "started" stories were connected to a branch created by a different student, and of those stories, 40% were connected to a branch also connected to other stories.

[3]https://www.pivotaltracker.com/help/articles/github_integration/

Figure 11: Different types of branch structure for "started" stories for each team. Stories with their own unique branch are deemed compliant. Stories with 0 branches or a branch connected to other stories are non-compliant. Teams are sorted in order of how many compliant stories they had.

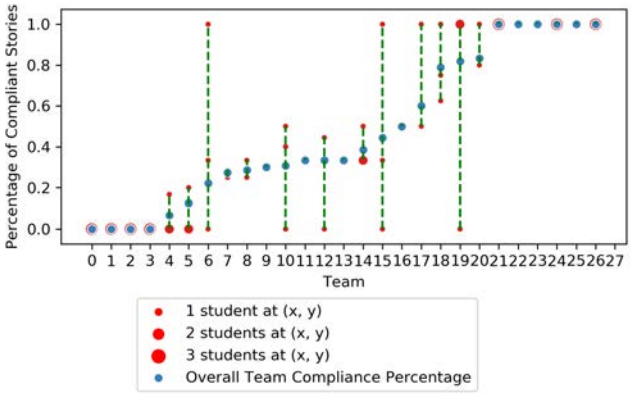

Figure 12: The percentage of compliant stories per student varied greatly within teams. Teams are sorted in order of their compliance percentage.

6.2 Does Finishing a Story Correspond With Opening a Pull Request?

In our recommended workflow, a story is marked "finished" when the code is ready to be reviewed, so the state transition to "finished" should be correlated with the opening of a new Pull Request in GitHub. As Figure 13 shows, many stories were marked as "finished" but were not connected with an opened Pull Request. Stories marked as "finished" might not be connected with an opened Pull Request for several reasons. If teams never connected the story with a Git branch, then there wouldn't be a branch to find a pull request from. Alternatively, a branch may have been merged directly, without opening a Pull Request to first solicit review from team members. Neither of these behaviors is compliant with our recommended workflow.

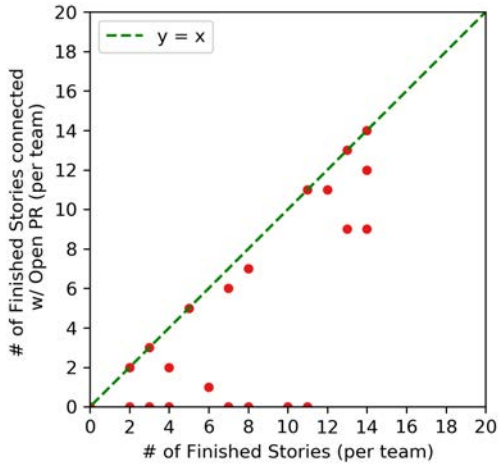

Figure 13: Number of stories that were marked as "finished" that were connected with an Open PR on GitHub. Each red dot represents a team.

6.3 Does Delivering a Story Correspond With Merging a Pull Request?

Similarly, Figure 14 shows that many stories were marked as "delivered" but were not connected with a merged Pull Request. Compliance with this step is even lower than for the previous two steps. Stories marked as "delivered" might not be connected with a merged Pull Request for reasons similar to "finished" stories not being connected with open Pull Requests. Additionally, "delivered" stories may have had PRs opened for them, but they were not able to be merged before the course ended. (Pull requests can remain open while the team discusses the new additions and any recommended fixes are made, and as the project occurred at the end of the course and teams only had a few days to complete it, it's possible that some opened PRs were not able to be approved by the team and merged before the course ended.)

7 DISCUSSION AND FUTURE WORK

A few observations from the results will guide our enhancement of both the recommended workflow and the method of monitoring it in future offerings of the course.

Examine student behaviors as well as team behaviors. Teams that appear marginally compliant may include students who are following the process diligently, and teams that appear more compliant may include non-compliant students. Besides ramifications for evaluating teams, this observation suggests there may be an opportunity to incentivize students on a team to influence each others' behavior more directly.

There are many ways not to follow an ideal process. Evaluating if students are following a recommended process is not a

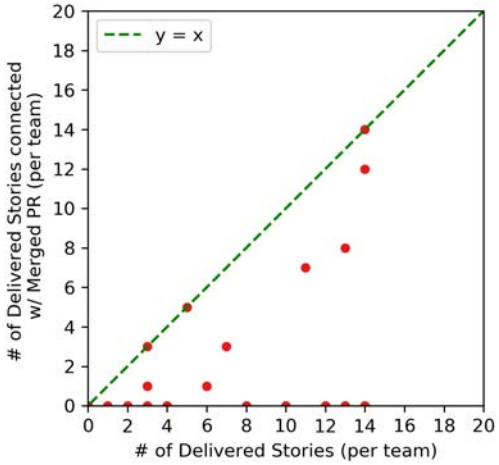

Figure 14: Number of stories marked as "delivered" that were connected with a Merged PR on GitHub. Each red dot represents a team.

simple binary classification problem. Our idealized workflow expected that students would create exactly one branch per story. But we observed many ways to deviate from this behavior, including stories that were marked as "started" multiple times, branches created for stories that were not yet marked as "started", and multiple "started" stories sharing the same branch. Manual inspection or interviews with students would be required in this case to understand why each behavior arose; for example, while working on their first story, a student may inadvertently write code that completes another story if the stories have overlapping logic. In future offerings of the course, our workflow should explicitly identify these behaviors as examples or non-examples of compliance and give specific guidance to avoid slipping into non-compliant behavior "accidentally."

More iterations may be better. Our data came from the summer offering of a course that is normally offered during the academic semester. This summer offering is 8 weeks long, while a fall or spring offering of this course lasts for 15 weeks. Due to the shortened time frame for summer courses, students only participated in a single ten-day long iteration; during a fall or spring offering, they would have participated in four two-week long iterations. Particularly interesting would be observing how and whether students on the same team who initially show varying levels of compliance would influence each others' behavior over time, and whether the team as a whole improves or worsens over time as a result.

8 CONCLUSION

Instructors of software engineering courses want to know if students are understanding the processes being taught. It is difficult to know definitively if students are understanding each process since student behavior cannot be directly observed. However, we propose that instructors can use the data exhaust from tools used by students to know what student teams are doing. We present a recommended workflow for student teams that integrates project, code, and build management tools to help students coordinate the difficult task of Continuous Integration. Since our workflow attempts to connect student actions across different tools, we list several methods of doing so and report on our results using the method of automatic tool integration. Our results indicate that teams did follow our recommended workflow but at various levels of compliance. Student compliance in teams at either end of the spectrum was uniform, but teams in the middle of the spectrum had varying levels of student adherence. Our recommended workflow should be expanded to explicitly address some student behaviors we did not expect. We hope that this work allows others to design processes similar to ours and use our tools to come to interesting conclusions of their own.

ACKNOWLEDGMENTS

Armando Fox provided primary guidance through this process and offered valuable perspective and experience from an instructor's point of view. Michael Ball offered additional insight and gave feedback on earlier drafts of this paper. Cesar Garcia Pascal, Alejandro Guerrero, Pablo Fernandez, and Antonio Ruiz-Cortez created the EventCollector system used to collect data and the recommended workflow proposed to students. Jack Boreczky helped design and develop the Pseudonymizer tool. Srujay Korlakunta facilitated our use of data from the summer version of our course. Jack Boreczky, Srujay Korlakunta, and Karthik Sreedhar also helped to categorize past survey responses. Rafael Fresno also completed initial work that laid the foundation for the EventCollector infrastructure used in this paper.

REFERENCES

[1] [n.d.]. . Retrieved August 11, 2020 from https://www.atlassian.com/git/tutorials/using-branches/git-merge#:~:text=A%20fast%2Dforward%20merge%20can,to%20the%20target%20branch%20tip
[2] [n.d.]. . Retrieved August 11, 2020 from https://git-scm.com/docs/git-log
[3] [n.d.]. . Retrieved August 11, 2020 from https://git-scm.com/docs/git-merge-base
[4] Kent Beck. 1999. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Longman Publishing Co., Inc.
[5] William Billingsley and Jim Steel. 2013. 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 (ITiCSE '13).* Association for Computing Machinery, New York, NY, USA, 213–218. https://doi.org/10.1145/2462476.2465592
[6] Jon Bowyer and Janet Hughes. 2006. Assessing Undergraduate Experience of Continuous Integration and Test-Driven Development. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06).* Association for Computing Machinery, New York, NY, USA, 691–694. https://doi.org/10.1145/1134285.1134393
[7] B. P. Eddy, N. Wilde, N. A. Cooper, B. Mishra, V. S. Gamboa, K. M. Shah, A. M. Deleon, and N. A. Shields. 2017. A Pilot Study on Introducing Continuous Integration and Delivery into Undergraduate Software Engineering Courses. In *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE T).* 47–56.
[8] Martin Fowler. 2006. *Continuous Integration.* Retrieved August 9, 2020 from https://martinfowler.com/articles/continuousIntegration.html
[9] Armando Fox and David Patterson. 2014. *Engineering Software as a Service: An Agile Approach Using Cloud Computing* (1st ed.). Strawberry Canyon LLC, San Francisco, CA.
[10] An Ju, Adnan Hemani, Yannis Dimitriadis, and Armando Fox. 2020. What Agile Processes Should We Use in Software Engineering Course Projects?. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20).* Association for Computing Machinery, New York, NY, USA, 643–649. https://doi.org/10.1145/3328778.3366864
[11] William Krebs. 2002. Turning the Knobs: A Coaching Pattern for XP through Agile Metrics. In *Extreme Programming and Agile Methods — XP/Agile Universe 2002,* Don Wells and Laurie Williams (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 60–69.

[12] Todd Sedano, Paul Ralph, and Cécile Péraire. 2017. Software Development Waste. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, 130–140. https://doi.org/10.1109/ICSE.2017.20

[13] J. G. Süß and W. Billingsley. 2012. Using continuous integration of code and content to teach software engineering with limited resources. In *2012 34th International Conference on Software Engineering (ICSE)*. 1175–1184.