A Pilot Study on Introducing Continuous Integration and Delivery into Undergraduate Software Engineering Courses

Brian P. Eddy, Norman Wilde, Nathan A. Cooper, Bhavyansh Mishra, Valeria S. Gamboa,
Keenal M. Shah, Adrian M. Deleon, Nikolai A. Shields
University of West Florida
Pensacola, Florida
Email: beddy, nwilde - @uwf.edu
nac33, bm37, vsg3, knp8, kms118, nas30, amd87 - @students.uwf.edu

***Abstract*—As continuous delivery and continuous integration practices become more prevalent in industry, the need for education in these areas grows. Introducing these topics introduces complexities due to the learning curve of the involved tools and the amount of time available for teaching these topics. Furthermore, there has been limited research into effective teaching practices for incorporating continuous integration and delivery concepts into traditional software engineering courses. In this paper, we discuss the results of an initial study of introducing a continuous delivery educational pipeline into an undergraduate software engineering course. The pipeline used was designed to help instructors introduce continuous integration and delivery into preexisting courses and allow students to visually understand the processes of continuous delivery and continuous integration.**

***Index Terms*—continuous integration, continuous delivery, undergraduate education, automated software engineering**

I. Introduction

Agile software engineering processes are developed around a set of common principles defined by the Agile Manifesto. The principles identified in the Agile Manifesto include, the early and continuous delivery of software to the customer and the ability to easily adapt to changing requirements. In order to support agile principles more easily, software engineers have adopted new practices that help to decrease the amount of time required to take software from the development team to the customers and users of the software. Furthermore, at the same time a high level of quality should be maintained with each release.

Continuous integration and continuous delivery are two related practices that help to reduce the amount of time needed to release software to customers while also maintaining a high level of quality. Continuous integration is a software development practice where members of a team integrate their work frequently, usually with each person integrating at least daily [1]. Continuous integration includes automated building and unit and integration testing of each version of the software system. Continous delivery is a software development practice where software is built in such a way that the software can be released to production at any time [2]. Continous integration is usually a part of the continuous delivery practice. As agile processes have become more widely adopted in industry, so have the practices of continuous integration and delivery. For this reason, it is important for students of a software engineering program to be exposed to the practices of continuous integration and delivery, however teaching these practices presents a set of challenges.

Both continuous integration and continuous delivery are facilitated through the use of pipelines of automated tools. Continuous integration and delivery pipelines can become extremely sophisticated and complex, with each tool and step in the pipeline's process presenting a new learning curve. This can make it quite difficult to teach the concepts of continuous integration and delivery in software engineering courses where the majority of the time in the class is devoted to other concepts. It can also be difficult for an instructor to learn each of the tools needed to build a complete pipeline. For this reason, our previous work focused on the design and implementation of a continous delivery pipeline that can be used by instructors to teach these concepts [3]. In this paper, we present the results of a study in which students complete software engineering change tasks using the continuous delivery pipeline. Our main goal was to understand whether using a continuous delivery pipeline with a visual dashboard and automated feedback could help improve a student's understanding of continuous integration and delivery. The study presented in this paper makes the following contributions:

- An initial study on methodologies and tools for teaching students the concepts of continuous integration and delivery and the related steps.
- Guidance and recommendations on how to create activities and labs that improve student understanding of continuous integration and delivery.
- Noticed challenges on teaching the concepts of continuous integration and delivery to undergraduate software engineering students.

The rest of this paper is organized as follows: Section II presents background information on continuous integration

2377-570X/17 $31.00 © 2017 IEEE
DOI 10.1109/CSEET.2017.18

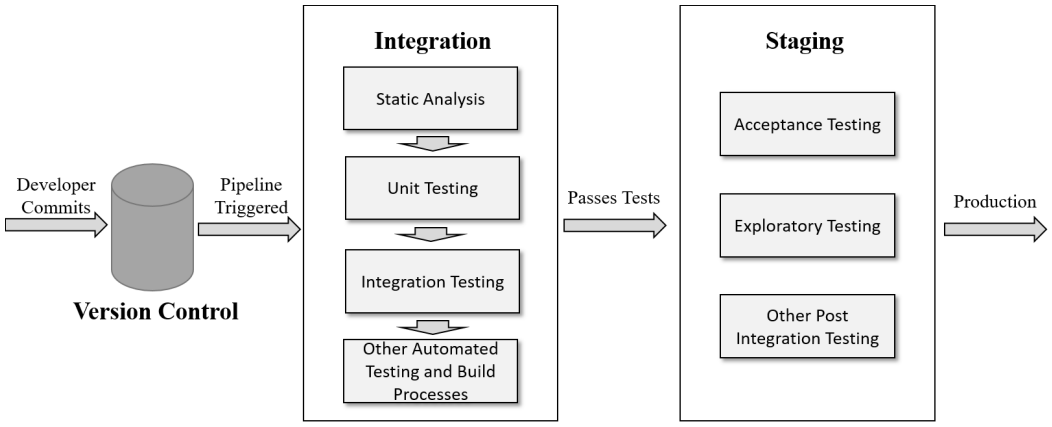

Fig. 1. Stages of a Simplified Continuous Delivery Process

and delivery, the pipeline used in the study, and related research, Section III discusses the design of our study, Section IV discusses the results of our study and presents recommendations for instructors teaching the concepts of continuous integration and delivery, Section V discusses future work planned by the authors, and Section VI concludes the paper.

II. Background

In this section, we present necessary background information on continuous integration and delivery, the specific pipeline used during our study, and related work.

A. Continuous Integration and Delivery

Figure 1 gives a simplified overview of the steps involved in a continuous delivery pipeline. When a developer commits changes to source code to a version control repository, the continuous delivery pipeline is triggered. The first few steps of the process are to perform continuous integration. Multiple steps are possible during the continuous integration process, and different pipelines handle the steps in different ways. We primarily focus on some of the most common steps during continuous integration which includes:

- **Static Analysis** - The static analysis step checks the source code without executing it by evaluating for well-known traps and pitfalls (e.g., security vulnerabilities) [4]. It may also check for adherence to or violations against coding standards or best practices.
- **Unit Testing** - The unit testing step tests individual units of code (e.g., methods) in isolation from the rest of the software system [5]. Because unit tests test each unit independent of other units of code, these types of tests allow for developers to easily identify where problems occur.
- **Integration Testing** - Unlike unit testing which focuses on each unit or module independently, integration testing combines two or more modules together for bugs that cut across multiple units [6].
- **Build Application** - The final build step depends on the type of software system. For a web application, this step may be optional. In the build step, any dependencies are bundled together with the source code to make the system ready to be downloaded and installed.

For the activity used in our study, we used a web application and therefore did not look at the build step. In addition, there may be other types of tests that are performed during continuous integration. Since the remaining types of tests may be optional or dependent on the type of system that is being developed, we do not discuss them here. The listed steps are the steps involved in continuous integration. When the tests pass, they may be sent to a staging environment or simply uploaded to another repository. Continuous delivery extends beyond the continuous integration process.

The focus of continuous integration is to maintain high quality code and to decrease the time and effort needed for testing. The main focus of continuous delivery is to go from a code change to a verified production ready change in as little amount of time and manual interaction with the process as possible. The continuous delivery process incorporates the testing and analysis steps of continuous integration, but also improve the pipeline with various tools for notification systems and deployment schemas to staging environments for the quality assurance (QA) testers.

The main requirements for a continuous delivery pipeline are as follows:

- Continuous integration system for automated testing
- An archiving system for versioning and performing roll backs when necessary
- A deployment process for moving a passing build to a staging environment before it is approved to be shipped off to production
- A notification system to update developers of passes and fails of a build
- A feedback system for QA testers to allow for discussion of changes that may need to be done before it is sent to production

The main form of testing performed during staging is acceptance testing which seeks to make sure that the software system meets the business requirements of the users and customers.

Continuous delivery is often confused with a similar process called continuous deployment. The main differences between the two is that continuous delivery has a manual step, usually some sort of QA testings, before a code change that successfully passes through the continuous delivery process is sent into production while continuous deployment is sent automatically to production and has actual users do the QA testing. Continuous deployment requires additional machanisms to allow users to provide feedback and to report bugs. We do not focus on continuous deployment in our study.

There are several studies that focus on continuous delivery use in industry. Success stories of continuous delivery being implemented of those like Neely and Stolt [7], Chen [8], and Gmeiner et al. [9] talk about the benefits of continuous delivery with quick feedback from users, bugs are found faster due to quicker release cycles, and features don't need to wait long periods of time to be implemented.

Building your software to work best with continuous delivery was discussed in Bellomo et al. [10], Austel et al. [11], and Chen [12] with the idea of software architectural design for continuous delivery. Bellomo et al. attempted to validate whether architectural design could significantly impact how well software systems can adjust to continuous delivery. Chen discusses some of what is called Architecturally Significant Requirements (ASR) which are design decisions that have a benefit for implementing continuous delivery. Austel et al. describes a research project to create a Platform as a Service (PaaS) with roles, tools and processes needed for having composite, low coupling, software systems that will benefit the greatest with continuous delivery.

Bae and Kim [13] and Soni [14] discuss different designs and frameworks for full CI/CD pipelines. Bae and Kim attempted to design a continuous integration and delivery framework for IoT devices all of which could have different underlying architectures requiring most communication to be performed through strong interfaces. Soni's approach was to design an entire end to end continuous integration and delivery pipeline using tools like Jenkins [15] for continuous integration management with builds, git [16] or svn [17] for source control, and other tools for performing automated testing and feedback systems.

Dunne et al. [18] looked at bug bounties for prioritizing continuous delivery resources. The study involved analyzing one company's bug reports generated both internally and externally and classifying them based on severity. The authors found that minor bugs that were functional bugs were generally found by users meaning that whatever continuous delivery steps in place were not catching those types of bugs and perhaps using users and bug hunters and using continuous delivery resources to target different more critical bugs would be more beneficial.

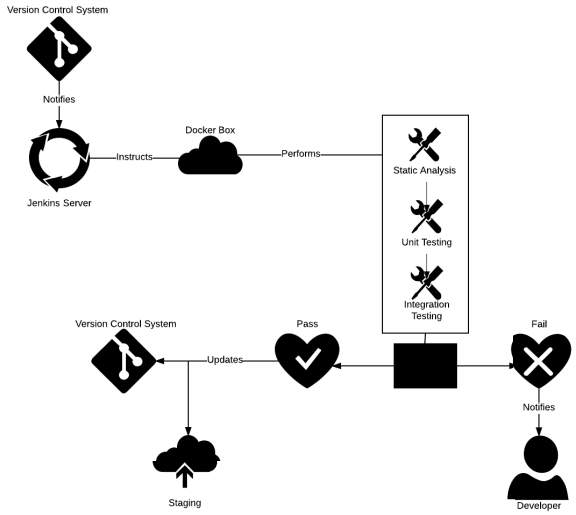

Fig. 2. The work flow of the continuous delivery educational pipeline tool

B. Continuous Delivery Education Pipeline (CDEP)

The continuous delivery educational pipeline (CDEP) is a set of tools that allow students to learn the fundamentals of continuous integration and continuous delivery and the work flow of such a system [3]. It was created to demonstrate the usefulness of such tools and provide instructors who are interested in teaching these practices to students in a laboratory style environment. The design of the system is meant to be simple to lower the learning curve for instructors while also allowing for multiple students to work with the system concurrently.

It has two major components: the Jenkins server and the Docker box. The Jenkins server is responsible for orchestrating the pipeline using a continuous integration tool called Jenkins. The tasks Jenkins performs are retrieving code changes from version control (e.g., Github), performing static analysis and tests on the retrieved code, and, if the code passes all the tests, moving the code change into a staging environment for acceptance testing. The Docker box uses a software container tool called Docker which allows for lightweight containers to be easily created and managed. Docker containers are used to create multiple environments for each student, including separate environments for static analysis, unit testing, and integration testing, as well as containers for storing an activity's database and a staging server for acceptance testing. Using containers allows for each student to have their own copy of each environment in the pipeline, for multiple build processes to be ongoing at the same time, and for each step of the process to be highly customizable.

The workflow of the CDEP system is shown in Figure 2. The process when the version control system (VCS) notifies our Jenkins server through webhooks that a code change has been made and is ready for integration. The next step is

to instruct the Docker box server to provision the needed analysis and testing environments to perform their individual tasks on the new code change. Once the code change has successfully passed through the static analysis, unit testing, and integration testing phases, Jenkins instructs the Docker box to update the VCS staging branch with the new changes. A new staging environment is then created and the updated application deployed to it for acceptance testing. At each step in the process, detailed feedback reports about success and failure are provided to the student through a dashboard. If any of the tests failed then the process immediately terminates and a report is available to the user through the dashboard. The student can then choose to fix the issue and recommit new changes. Since each student has their own set of containers, a failure in one student's activity has no effect on the completion of other students' activities. The current version of CDEP handles all of the described tasks, however new changes are still ongoing.

C. Related Work

Research into how to teach students the topics of continuous integration and delivery is limited. Research that has looked at such issues has typically focused on integrating such concepts as a major component in a semester long course such as the work by Krushche and Alperowitz [19]. Krushche and Alperowitz required the use of a continous delivery pipeline in a multi-customer project based software engineering course. In the course used by Krushche and Alperowitz, a considerable amount of time and resources could be devoted to continuous integration and delivery.

Süß and Billingsley did significant work with trying to reduce the requirements and resources needed in order to allow students to gain hands on experience with different industry standards, and practices [20]. Their premise was to use legacy code as a code base that students would maintain and add to over a semester long course. This would give students an accurate environment that is similar to ones they would experience in actual industry. In addition, they used continuous integration practices to provide automated feedback to the students over the course of the semester. However, their focus was on using these practices to provide feedback as opposed to teaching students about the practices themselves.

Christensen [21] looked at how to teach DevOps principles in a cloud computing course. Christensen identified that traditional software engineering courses typically focus on the beginning stages of software engineering and have less emphasis on the later stages of deployment and maintenance. Over a seven week course, students would make changes to a cloud based application and Docker was used to learn infrastructure logic.

These approaches have their own merit and advantages such as students gaining practice over a significant amount of time with these tools, team oriented projects, and processes. However, most of these approaches require continuous integration and delivery to be a primary focus of the courses and do not address the problem facing instructors of how to implement these types of practices into a more traditional software engineering course. Specialized courses on these topics may not always be possible due to either lack of resources or lack of time to devote to the concepts. The continuous delivery pipeline for education used in our study allows students to learn about industry standards in terms of software development in a concise and visual way and can be incorporated if time or resources are limited.

III. Study Design

In this section we discuss the design of our study.

A. Definition and Context

We conducted an empirical study to evaluate whether using a continuous delivery pipeline with visual and informed feedback in a laboratory setting could help improve a student's comprehension of continuous integration and delivery processes. Our primary interest was in improving a student's understanding of the steps involved in continuous integration and delivery and not in improving their understanding of a specific tool or tool chain.

In the study we asked students to complete a software change task which involved adding features to an existing web application. As the focus was on their understanding of continuous integration and delivery and not on their coding ability, they were provided with detailed instructions for how to make the change and additional details on each step they were performing. They were given an hour to complete the activity and data was collected by the researchers before and after the study.

1) Students and Demographics: There were 16 students involved in the study (S1-S16). All were software engineering undergraduate students with at least two years of programming experience. Before the study, each student completed a demographic form. The purpose of the demographic information was to help establish the experience of the students on the types of tools they would be exposed to through the laboratory environment. The information collected by the demographic form included:

- Year of study
- Highest level of development experience
- Tasks performed in a professional environment
- Technologies in which there is current familiarity

The course used in the study was a senior level software engineering course at the University of West Florida. Therefore, of the 16 students who completed the study, 94% (15) of the students were senior students with the remaining student being a junior.

Figure 3 indicates the highest level of experience students have in developing software systems. All students were assumed to have experience with class projects from prior undergraduate classes, however we also asked for experience working on individual paid projects, as interns in a software development company, or as part time or full time employees in a software development company. The highest responses were for students who had only completed some class projects

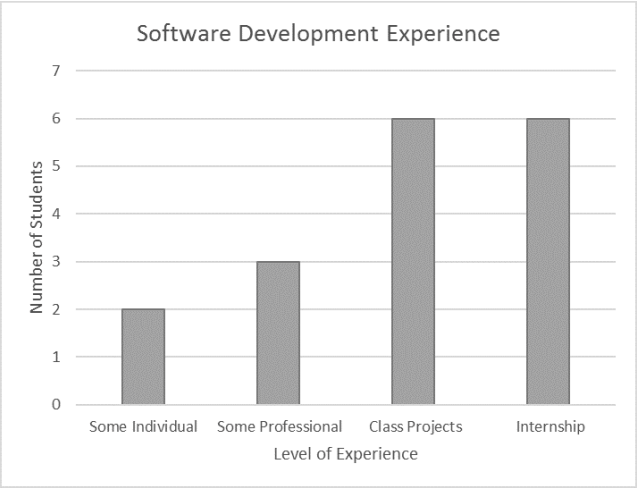

Fig. 3. Highest Level of Software Development Experience

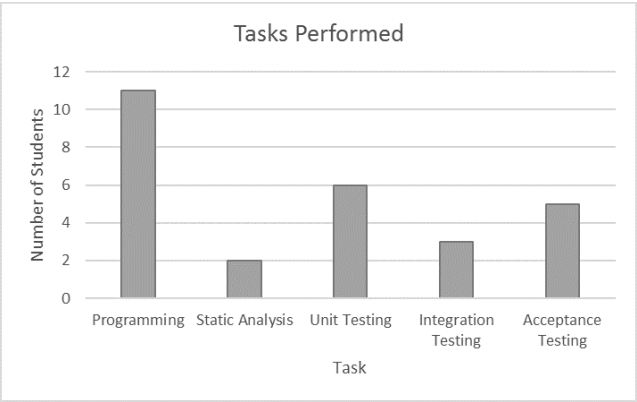

Fig. 4. Experience with Continuous Delivery Tasks

or had participated in an internship. Of the 16 students involved in the study, only three of the students had worked as either part time or full time employees of a software development company.

The students were asked about their experience with continuous delivery related tasks performed professionally. Of the 16 students in the student, 11 of the students have performed some basic professional programming either individually or as a part of a company. In Figure 4, it can be seen that the majority of students lacked some form of testing or static analysis experience. This is possibly due to classes that involve intensive programming concepts required for their degrees. Another reason to encourage continuous delivery education is to improve testing concepts. A course module in continuous delivery could help the students grasp the concepts of software testing.

In addition to previous professional experience, we were also interested in the types of technologies that students had been exposed to. According to the results of this question, students were most familiar with IDEs and version control systems. We also found that multiple students had exposure to unit testing frameworks, however experience in other types of software development tools was lacking. As a follow up to this question, we asked students to list the tools they were familiar with. Many tools were listed, but the tools students were most familiar with included NetBeans, jGrasp, and Eclipse, all of which are IDEs with jGrasp being an IDE meant for educational purposes. While the focus of our study is on the steps involved in continuous integration and continuous delivery, an additional benefit of using a pipeline is the exposure to other types of tools.

2) Study Materials: Prior to the study, one of the researchers developed a PHP web application to be used for the study. The web application was an online electronics store which stored information on products in a MySQL database. The web application included unit tests and integration tests written in the PHPUnit framework. The source code for the web application and the tests as well as documentation were stored in a repository on GitHub. Each student was provided with their own fork of the repository, therefore giving each student their own copy of the application. As part of the activity used in the study, students were required to use git through the command line interface.

Before and after the study, students were asked to complete a survey to gather information about how comfortable the students were with the concepts of continuous integration and delivery and the involved tasks. The surveys collected the following information:

- Understanding of version control, branching, static analysis, unit testing, integration testing, continuous integration and delivery tasks, and the usefulness of continuous integration and delivery based on a six point scale.
- Concepts that the students believed the pipeline and activity helped them to understand.
- Remaining questions that the students have on the concepts contained within the activity.
- Possible improvements to the pipeline or the activity that the students believed would help them better understand the concepts.

The first point was expressed as a six point Liekert scale with 1 being the lowest value and 6 being the highest value.

Using the command line interface for git, students would create a local repository on their machine of the example application. Then, following the instructions for the lab, they would make small changes to either the source code of the system or to the automated tests. There were two parts to the lab. The first part of the lab had students editing PHPUnit test cases. The student would edit a test case and then push their code to the remote repository. A webhook on the remote repository would trigger Jenkins to pull the updated code. Then on the Docker box, scripts would run through unit testing, integration testing, and deployment to a production server. Feedback to the students existed in two forms. First, students could monitor the deployment through the dashboard available in Jenkins. If a failure occured, they could fix the issue and push out a new change which would trigger the process from

the beginning. Second, upon a successful deployment, students would be able to instantly access the newly deployed website. This gave students the ability to directly witness the pipeline in action and to also observe the intermediate results from each stage of the pipeline.

The second part of the activity asked them to implement functionality that would allow users of the website see details related to a product in the online store. Students would need to develop appropriate tests along with the new functionality. Creating new functionality allowed the students to understand the support provided by the pipeline and the use of automated testing. Both parts of the activity were meant to give students a different perspective on how the continuous delivery pipeline worked.

3) Setting and Study Procedure: Students were part of the Software Engineering Management course at the University of West Florida. This is a 15 week undergraduate course focusing on topics of both traditional and agile software project management. Students learned concepts of scoping, planning, monitoring and tracking, estimation and cost analysis, as well as quality assurance, cultural challenges of change, and making decisions regarding technical debt. As part of the course, students learned concepts related to deploying the system to the customer. The course was taught by one of the authors, however the study was carried out and conducted by the other researchers. The study was conducted in a classroom with each student having their own computer.

Before beginning the study, students were given a brief explanation on the topics of continuous integration and delivery and were given the lab's instructions. In order to keep students anonymous, each student was given a random number that associated them with their data and their repository. Students were asked to complete the pre-survey and the demographics collection form. Then they were given the activity and an hour to complete the steps involved. After a subject completed the activity, they were given a post survey to fill out regardless of whether or not students were able to complete the activity in the hour allotted. The students were then debriefed on the focus on the study and were asked to sign an informed consent form if they agreed to let their data be used in the study. Given an original count of 19 students within the course, 16 students agreed to participate in the study.

B. Research Questions

The questions of interest are as follows:

- RQ1 Does using a continuous delivery pipeline in a laboratory environment help students better understand continuous delivery related concepts?
- RQ2 What concepts showed the greatest increase in understanding after using the continuous delivery pipeline?
- RQ3 What concepts require a deeper discussion in addition to exposure of the continuous delivery pipeline?
- RQ4 What are the problems associated with using a continuous delivery pipeline for teaching in a laboratory environment?

RQ1 is the primary question of the study and seeks to identify whether using a continuous delivery pipeline and lab activities can help students better understand the concepts of continuous integration and delivery. In RQ2, we tried to identify the concepts that had the greatest increase in understanding from the pre and post surveys. The answer to RQ2 could help understand what types of tasks similar learning activities best help with. RQ3 looked at concepts where students still struggled and that may require additional support, such as additional lectures or activities. Finally, RQ4 focuses on identifying unexpected problems that were encountered during the activity.

C. Statistical Testing and Hypotheses

To determine whether there is a significant difference between the students' responses of the pre and post surveys, we form Wilcoxon signed-rank tests corresponding to each continuous delivery concept. The Wilcoxon signed-rank test is the nonparametric analog of the student's t-test and is used when distributions are not assumed to be normal. For each Wilcoxon signed-rank test, we do not presuppose the directionality of the difference between the pre and post surveys. Therefore, each hypothesis test is two-tailed. For each test, we formulate a null hypothesis to evaluate whether there is a significant difference before and after the study. If, after testing the null hypothesis, we find that we can reject the null hypothesis with a high confidence ($\alpha = 0.05$), we accept an alternative hypothesis, which corresponds to there being a significant difference between the pre and post surveys for that concept.

An example null hypothesis:

$H_0 : Static_{pre} = Static_{post}$
: There is **no significant difference** in the student's understanding of static analysis between before and after the study.

The corresponding alternative hypothesis:

$H_0 : Static_{pre} \neq Static_{post}$
: There is **a significant difference** in the student's understanding of static analysis between before and after the study.

The remaining null and alternative hypotheses are analogous.

D. Threats to Validity

We have identified three primary threats to validity. First, it is possible that the pipeline used within the study does not adequately represent the concepts we wish for the students to learn. To ensure that our pipeline reflects modern industry practices, we validated the pipeline with the help of external industry professionals. Therefore, we have reasonable confidence that the pipeline correctly demonstrates the described concepts. Second, as with most educational research, we run the risk of students not answering sincerely out of fear that reporting undesired results may hinder them in the class. To mitigate this issue, students were not graded on the study or

Student	Version Control		Branching		Static Analysis		Unit Testing		Integration Testing		CI & D Tasks		CI & D Usefulness		Activity Helped
	Pre	Post	Pre	Post	Pre	Post	Pre	Post	Pre	Post	Pre	Post	Pre	Post	
S1	5	5	5	5	2	4	4	4	4	5	4	5	4	5	4
S2	6	4	5	4	2	2	5	4	5	5	3	3	3	3	2
S3	6	6	6	6	4	4	6	6	5	5	4	5	5	5	4
S4	5	6	3	5	3	3	5	5	5	6	3	6	4	6	6
S5	6	6	5	6	3	5	6	5	6	5	5	5	3	5	5
S6	4	3	4	5	2	4	4	4	4	4	3	5	4	5	5
S7	4	6	5	6	2	3	5	4	5	5	4	5	5	5	6
S8	4	4	4	4	1	3	4	5	4	5	4	5	4	5	5
S9	4	5	4	5	4	5	5	5	5	5	3	5	3	5	5
S10	5	6	5	6	3	2	1	2	1	2	3	3	3	4	3
S11	3	5	5	5	3	6	6	6	4	6	4	5	3	6	6
S12	6	6	6	6	3	3	6	6	6	6	5	5	5	5	4
S13	4	6	4	6	1	1	4	6	4	6	4	6	6	6	6
S14	6	6	6	6	3	2	5	3	6	3	6	6	6	6	6
S15	4	5	4	5	5	3	4	3	3	4	4	3	3	4	4
S16	4	4	6	3	3	3	5	4	3	4	4	4	4	4	4

TABLE I
STUDENT SURVEY RESPONSES

the corresponding activity, furthermore the instructor of the course was not involved in the collection of results and all results reported to him were completely anonymized. Finally, there is the threat that students may not have an accurate gauge of their understanding on a topic. For this reason, our goal is not to get an absolute score of how well a student understands the concept, but to increase awareness about the use, purpose, and importance of each concept in the study. We therefore believe that the pre and post surveys are sufficient for our intended purpose.

IV. RESULTS AND DISCUSSION

In this section we discuss the results of our study and try to provide additional guidance for instructors teaching the studied concepts. Table I presents the responses of each student for the pre and post surveys.

A. RQ1 Does using a continuous delivery pipeline in a laboratory environment help students better understand continuous delivery related concepts?

Figure 5 shows the distribution of students ratings of whether the activity and the pipeline improved their understanding of continuous integration and delivery. From the figure, it can be seen that a majority of the students in the study found that the pipeline and activity were useful to them in understanding the concepts. Both S3 and S4 spoke highly of the pipeline's ability to help them visualize the steps in the process. According to S4, "Now that I have physically witnessed a pipeline in action, I can see how it is useful to development. Being able to see tests passing and failing, along with manually altering the source code has helped me understand pipelines 100 times better." S9, S13, and S14 stated that they had heard of continuous pipelines before, but seeing the pipeline in action allowed them to solidify their notion

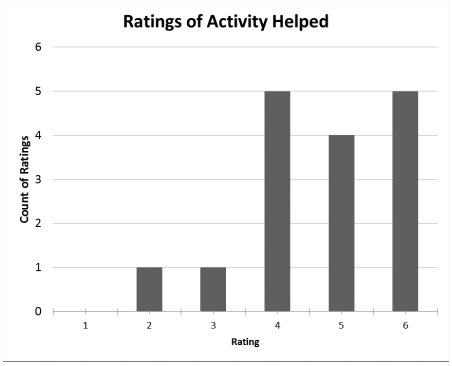

Fig. 5. Ratings of the Usefulness of the Activity

of the topic. While these results were promising, there were two students who still struggled after the study. We found that S2 had difficulties with the study due to the fact that they were not familiar with how to interact with the version control tools from the command line as their previous experience was with using GUI based tools. Based on S2's response, we identified that experience with command line tools should be a prerequisite to the activity and lab. S10 was borderline on their experience, while they found that they understood many of the concepts better overall, they still struggled with a few significant parts, primarily in their understanding of static analysis.

B. RQ2 What concepts showed the greatest increase in understanding after using the continuous delivery pipeline?

To better understand the answer to this question, we look at the results from multiple angles. We start by looking at the overall percentage change in each concept. The results of

Concept	Decreased	Unchanged	Maxed	Increased
Version Control	12	19	25	44
Branching	12	19	19	50
Static Analysis	18	38	0	44
Unit Testing	37	25	19	19
Integration Testing	12	32	6	50
CI & D Tasks	6	32	6	56
CI & D Usefulness	0	32	12	56

TABLE II
PERCENTAGE OF DECREASED, UNCHANGED, AND INCREASED RATINGS

this can be seen in Table II. From this table, it is apparent that the concepts with the greatest percentage of students who increased their understanding was for the overall continuous integration and delivery process and the usefulness of these processes. If we look at the individual concepts, we see that branching and integration testing have the second highest percentages. However, it is important to note that some subjects claimed an increased understanding of version control despite rating their understanding of version control as a 6 in the pre-survey. Maxed is the percentage of students who rated themselves as having the maximum understanding in both the pre and post surveys and could therefore not demonstrate an increase. While most concepts had a higher percentage for increased than any other column, unit testing was an exception. Unit testing actually showed a decrease in understanding. We believe this decrease is due to students learning how to write basic unit tests in other courses, but not seeing the usage of unit testing in a larger software system. Not seeing unit testing in context led to an inaccurate representation of their understanding of the concept of unit testing. Once realizing the additional complexities involved in unit testing, students corrected their scores in the post surveys. Support for this observation comes from the point that while the scores decreased, many of the comments reported a better understanding of automated tests.

In addition to looking at the percentages of change, we also look at the changes in the distributions of the ratings.

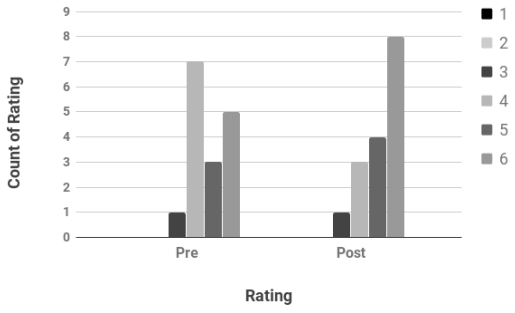

Fig. 6. Ratings of the Understanding of Version Control

In Figure 6, the distribution of ratings for the concept of version control systems is shown. The results of the pre-survey for version control showed that the majority of students rated their understanding of version control at a 4 before the study, however after the study, the majority of students rated their understanding at a 5 or 6 with a 3 or 4 being in the minority. We use the increase in ratings as a positive indicator that the activity helped students to better understand version control systems even though a majority of students had prior experience with such systems.

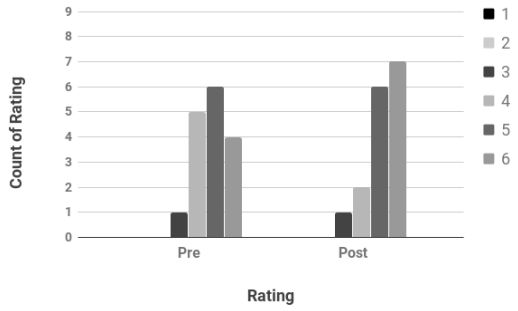

Fig. 7. Ratings of the Understanding of Branching

Figure 7 shows the distribution for the concept of branching in a distributed version control system. As with version control, the distribution after the study showed that a majority of students rated their understanding as a 5 or 6 with a minority rating themselves at a 3 or 4.

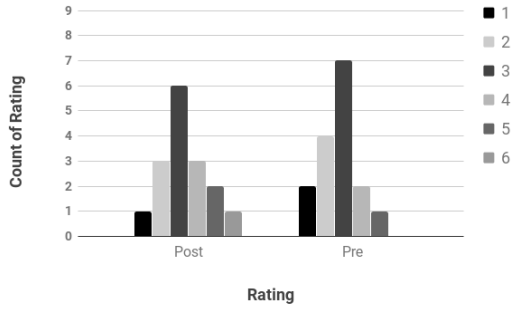

Fig. 8. Ratings of the Understanding of Static Analysis

Figure 8 shows the distributions for static analysis. Static analysis had one of the smallest degrees of change with the largest number of students that rated themselves the same before and after the study. While there was a small improvement in the distribution, overall it did not appear to be significant.

Figure 9 shows the distributions for unit testing. As discussed earlier, some students rated themselves lower in the post-survey than in the pre-survey, however the minimum rating did improve. Overall, a large portion of the distribution remained unchanged and as discussed earlier, we do not necessarily believe that the lower student ratings are a result of less understanding as this would be contradicted by the students' comments, but as a result of correcting their understanding

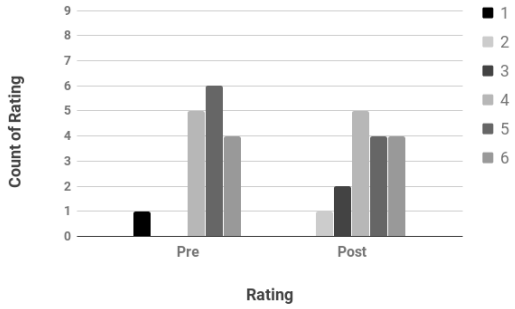

Fig. 9. Ratings of the Understanding of Unit Testing

and realizing that unit testing is a more complicated process than originally believed.

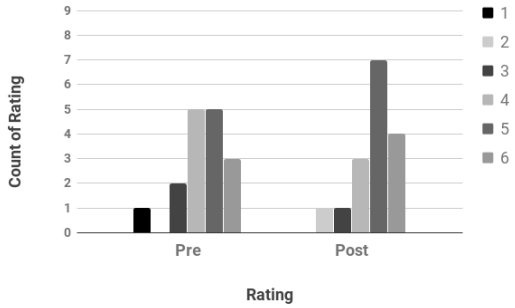

Fig. 10. Ratings of the Understanding of Integration Testing

Figure 10 shows the changes in the distribution for integration testing. Overall, integration testing showed good improvement. In the pre-survey, the most frequent ratings were 4 and 5, however after the study students reported ratings of 5 and 6 more frequently.

Figures 11 and 12 look at students' overall understanding of continuous integration and delivery. These distributions showed the largest improvements of all the distributions.

We looked for statistically significant differences between the pre and post survey results. For this purpose, we used the Wilcoxon signed-rank test. Statistically significant differences were found in both the overall understanding of the tasks and of the usefulness of the practices.

C. RQ3 What concepts require a deeper discussion in addition to exposure of the continuous delivery pipeline?

The majority of concepts resulted in an improvement of understanding, however there are two exceptions in unit testing and static analysis. Students coming from a software engineering program should have a good understanding of how unit testing works. One of the problems with teaching unit testing is if a student learns the basics of a unit testing framework or tool, but does not learn how to divide a software system for testing and to isolate the individual units using mocks, stubs, and fakes. This is one of the more complicated parts

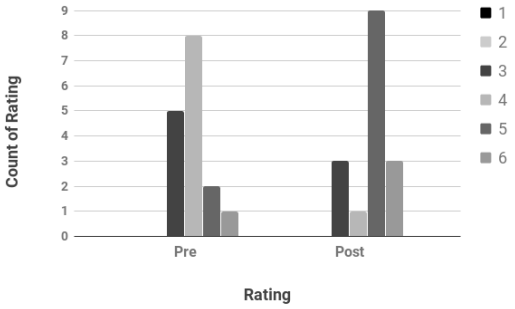

Fig. 11. Ratings of the Understanding of CI & D Tasks

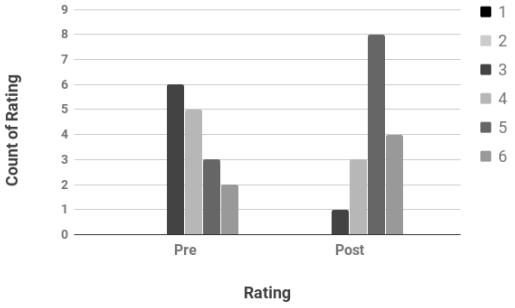

Fig. 12. Ratings of the Understanding of CI & D Usefulness

of learning unit testing and the degree of isolation is one of the things that sets unit testing apart from integration testing. This is what we believe may have caused some discrepancies in a student's understanding during the study.

We believe that students had a harder time understanding static analysis due to the lack of failures when static analysis detects possible issues. Instead of failing the pipeline as is done with testing, the static analysis stage generates a detailed report of possible issues for the student to look over. With the limited amount of time provided in the lab students did not look too in depth at the generated reports. For this reason, special attention should be paid to the static analysis stage during the activity to ensure that students do not overlook this information.

D. RQ4 What are the problems associated with using a continuous delivery pipeline for teaching in a laboratory environment?

One of the primary issues that occurred during the study was the lack of experience of some students in using version control from the command line. This inexperience caused a dip in productivity as students without experience with the command line interface took longer to complete the activity. These students were the most likely to not complete the assignment. There are apparent ways to help mitigate this issue. The first is for students to gain familiarity with the command line interface for version control before beginning the activity. The second is to provide detailed instructions

within the activity for both the command line interface and the graphical interface.

Some students struggled with getting started and performing the intial setup of the activity. Before the two main parts of the activity, the activity instructions go over details about how to setup the environment and ensure that the pipeline is working as expected so that the steps in the activity can be followed. There were some struggles with the initial steps, however after the initial steps were complete, students were able to follow the instructions for the remainder of the activity. It is advised that when setting up for independent labs, that the instructor leads the class through the setup process. Otherwise, the instructor will have to work one on one with students to help them start the activity which limits the amount of time they have to complete the activity.

In addition to the setup that must be performed by the students, instructors should make note of the amount of time that it will take for them to setup for the lab. If the instructor is setting up each student's repository themselves, this can be a significant time investment without automated scripts to help in the process. Furthermore, depending on the tools that the instructor is using to run the activity, they should be aware that there may be an involved learning curve. They will need to familiarize themselves with the technologies before the start of the activity.

V. Future Work

Based on the results of our study, we believe that there are multiple benefits to using visual feedback and laboratories for teaching continuous integration and delivery. In future research, we plan to expand our current study by improving CDEP and our activities with information gathered during the pilot study, increasing the number of students studied, and increasing the range of students studied. Additionally, the focus of the study was on the concepts of continuous integration and delivery and not on the tools used to build such a pipeline. Future research will look at methods for instructing students on the usage of such tools.

Futhermore, in the future we intend to develop additional educational tools that help to teach the processes and steps discussed throughout this paper. These tools will use a variety of visualizations and feedback mechanisms to help students understand these modern software development practices. In the future, we intend to release all activities and tools that we develop for use by educators at other colleges and universities.

VI. Conclusion

We looked at using a continuous delivery pipeline to teach students the concepts of continuous integration and delivery within a lab environment. We presented an initial study on such an approach and showed promising results. We identified concepts in which the pipeline was most effective, where students still struggled, and areas for improvement to our approach. Further research is still required to better understand how to best teach these practices to students.

Acknowledgment

This research was funded in part by the Nystul Foundation at the University of West Florida.

References

[1] M. Fowler. (2006) Continuous integration. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[2] ——. (2013) Continuous delivery. [Online]. Available: https://martinfowler.com/bliki/ContinuousDelivery.html

[3] B. P. Eddy, N. Wilde, N. A. Cooper, B. Mishra, V. S. Gamboa, K. N. Patel, and K. M. Shah, "Cdep: Continuous delivery educational pipeline." Kennesaw, GA, USA: ACM Southeast, 2017. [Online]. Available: http://dx.doi.org/10.1145/3077286.3077301

[4] P. Louridas, "Static code analysis," *IEEE Software*, vol. 23, no. 4, pp. 58–61, July 2006.

[5] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, no. 4, pp. 22–29, July 2006.

[6] P. C. Jorgensen, *Software testing: a craftsmans approach*. CRC press, 2016.

[7] S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," in *2013 Agile Conference*, Aug 2013, pp. 121–128.

[8] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, Mar 2015.

[9] J. Gmeiner, R. Ramler, and J. Haslinger, "Automated testing in the continuous delivery pipeline: A case study of an online company," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2015, pp. 1–6.

[10] S. Bellomo, N. Ernst, R. Nord, and R. Kazman, "Toward design decisions to enable deployability: Empirical study of three projects reaching for the continuous delivery holy grail," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 702–707.

[11] P. Austel, H. Chen, T. Mikalsen, I. Rouvellou, U. Sharma, I. Silva-Lepe, and R. Subramanian, "Continuous delivery of composite solutions: A case for collaborative software defined paas environments," in *Proceedings of the 2Nd International Workshop on Software-Defined Ecosystems*, ser. BigSystem '15. New York, NY, USA: ACM, 2015, pp. 3–6. [Online]. Available: http://doi.acm.org/10.1145/2756594.2756595

[12] L. Chen, "Towards architecting for continuous delivery," in *2015 12th Working IEEE/IFIP Conference on Software Architecture*, May 2015, pp. 131–134.

[13] J. Bae and J. Kim, "An experimental continuous delivery framework for smartx-mini iot-cloud playground," in *2016 International Conference on Information Networking (ICOIN)*, Jan 2016, pp. 348–350.

[14] M. Soni, "End to end automation on cloud with build pipeline: The case for devops in insurance industry, continuous integration, continuous testing, and continuous delivery," in *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, Nov 2015, pp. 85–89.

[15] Jenkins, "Jenkins," 2011–2016. [Online]. Available: https://jenkins.io/

[16] L. Torvalds, "Git," 2005–2016. [Online]. Available: https://git-scm.com/

[17] Apache Software Foundation, "Subversion," 2000–2016. [Online]. Available: https://subversion.apache.org/

[18] J. Dunne, D. Malone, and J. Flood, "Social testing: A framework to support adoption of continuous delivery by small medium enterprises," in *2015 Second International Conference on Computer Science, Computer Engineering, and Social Media (CSCESM)*, Sept 2015, pp. 49–54.

[19] S. Krusche and L. Alperowitz, "Introduction of continuous delivery in multi-customer project courses," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 335–343. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591163

[20] J. G. S and W. Billingsley, "Using continuous integration of code and content to teach software engineering with limited resources," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 1175–1184.

[21] H. B. 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*. New York, NY, USA: ACM, 2016, pp. 174–179.