Continuous Delivery of Personalized Assessment and Feedback in Agile Software Engineering Projects

Xiaoying Bai
Tsinghua University
baixy@tsinghua.edu.cn

Mingjie Li
Tsinghua University
li-mj14@mails.tsinghua.edu.cn

Dan Pei
Tsinghua University
peidan@tsinghua.edu.cn

Shanshan Li
Tsinghua University
lishanshan@tsinghua.edu.cn

Deming Ye
Tsinghua University
ydm14@mails.tsinghua.edu.cn

ABSTRACT

In recent years, Agile development has been adopted in project practices of Software Engineering (SE) courses. However, it is a great challenge to provide timely assessment and feedback to project teams and individual students with a frequency that catches up with iterative, incremental, and cooperative software development with continuous deliveries. Conventional project reviews are mostly dependent upon instructors and teaching assistants in a manual reviewing/mentoring approach, which are simply not scalable.

In this paper, we argue that agile projects warrant a "continuous delivery" of personalized assessment and feedback. To this end, we propose an online-offline combined approach and built a system upon GitLab. An online platform was built by integrating DevOps tool chains so that personalized reports and assessments are delivered automatically to the teams/students, which serve as the very efficient trigger and basis for the close and targeted offline interactions between students and TAs: daily discussion over instant messaging and weekly in person meeting. This system has been in operation since 2014 for an undergraduate SE course, with over 500 students participating in over 130 project teams in total. Our results show that such a continuous assessment/feedback delivery system is very effective in educating Agile projects in SE courses.

KEYWORDS

Software Engineering Course, Project, Agile, DevOps, Assessment

ACM Reference Format:
Xiaoying Bai, Mingjie Li, Dan Pei, Shanshan Li, and Deming Ye. 2018. Continuous Delivery of Personalized Assessment and Feedback in Agile Software Engineering Projects. In *ICSE-SEET'18: 40th International Conference on Software Engineering: Software Engineering Education and Training Track, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3183377.3183387

ICSE-SEET'18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5660-2/18/05…$15.00
https://doi.org/10.1145/3183377.3183387

1 INTRODUCTION

With the rise of Web 2.0 and Software-as-a-Service, software paradigm has shifted considerably in recent years. The Web 2.0 principle, "forever beta", has been a normal status of most Internet-based applications, which means that software are frequently updated online and released. Accordingly, conventional development style, that has clear well-planned and staged process of requirements-design-implementation-testing cycles, is giving way to the Agile style, which values small and frequent incremental delivery, test-driven development, continuous integration, and adaptability to changes. As a result, this paradigm shift in modern software industry requires the reform of college courses [2, 4, 6].

While Agile development has been widely adopted in industry, it remains a great challenge to educate Agile projects in Software Engineering (SE) courses due to the following three reasons.

- Students need timely feedback that catches up with iterative, incremental, and cooperative Agile software development with continuous deliveries [1, 7, 9]. Traditional assessment and feedback are in person review/mentoring approach based on regular meetings, which is too infrequent for the students to learn Agile development.
- Personalized feedback and assessment are necessary for project teams and individual students. However, it is both time and effort consuming to develop the reports manually. For example, in our course with ∼150 students and ∼10 diverse projects from the real world, each of the ∼10 teaching assistants (TAs) covers one project of ∼4 separate teams and ∼16 students. It's infeasible for each TA to manually provide personalized reports on a daily basis.
- There lack objective and widely-accepted metrics [1] to evaluate the performance of teams and students in terms of collaboration and communication, such as task management, commit management, and branching management, as opposed to just the final delivered products. Thus, scoring and ranking are always difficult

for SE team projects, especially for quantitative process quality measurement and personal assessment [1].

To address the above challenges, we argue that Agile projects in SE course warrant an automatic "continuous delivery" of personalized feedback to teams and students. In this paper, we make an important first step by proposing a combined online-offline approach: 1) An online system is built upon GitLab with a customized Scrum process and several open source plug-in tools. Whenever code updates are committed to GitLab repository, static code quality review, build, coverage-based testing, and docker delivery are automatically triggered immediately. 2) A project team is organized offline with students, TA, and customer representative. TAs can review the reports and proactively contact the students for any discovered problems. With the automatic online assessment, both the daily discussion over instant messaging and weekly in person meeting of TA/customers/students are very focused and targeted.

The main contributions of the paper can be summarized as follows:

- We propose an online-offline combined approach for educating Agile projects in SE course, an important first step towards continuous delivery of personalized assessment and feedback.
- We build a project management platform by integrating the tool chains following DevOps practices, which strengthens project communication and collaboration by streamlining the development and delivery process with automatic tool chain.
- Based on the DevOps repository, we define several objective metrics for issue management, commit management, and branching management, to facilitate quality analysis from different perspectives.

Our proposed system has been in operation since Fall 2014, with over 500 students participating in over 130 project teams in total. Our preliminary results show that the proposed approach helps to train students for professional SE skills, which we believe are applicable to other Agile projects of SE courses in general.

The rest of the paper is organized as follows. Section 2 briefly introduces course design including projects organization and the Scrum process, and presents the DevOps platform. Section 3 demonstrates the metrics and measurements for assessment. Section 4 reports our operational experiences and results. Finally, Section 5 concludes paper and discusses future work.

2 AGILE PROJECT AND PRACTICE DESIGN

We now briefly introduce our course design including project organization, the customized Scrum process, the automatic DevOps platform, and the combined online-offline approach.

2.1 Project Organization

Like SE education in general, to teach and to learn Agile, it is necessary to strengthen operations to cultivate students' practical capabilities by hand-on experiences [2, 6]. Team project is a generally adopted method such that students can understand SE principles by practices. With projects from "real world", students learn not only practical engineering solutions, but also the context, constraints, and social aspects of SE. By completing a project, students experience end-to-end system development, including the design of different modules and layers from user interface to computation logic to persistent data storage to network communication, and the tradeoffs to satisfy functional as well as non-functional requirements.

When picking project topics, we intentionally add some complexity from the real world. For example, one project is a follow-up (or phase 2) of a project from the previous year, in which case the current project team needs to work on legacy codebase from the original project team. In another case, two sub-projects (*e.g.*, one for frontend and one for backbend) form a big project, and one project team picks frontend and another picks backend. In such cases, students learn to collaborate with other teams to compose and integrate their individual subsystems.

To accomplish the project, students are organized in development teams, with 3-5 members in each team. Students in a team take different roles and responsibilities, such as leader, architect, and quality controller. They are encouraged to work closely with each other and with customers. They learn to balance the workload among team members to stimulate contribution from each participant. For each project topic, we invite customer representatives to provide domain knowledge, and 4^{th} year or graduate students as TAs. Students, customer representative(s), and a TA form a team called Win-Win team in order to advocate a culture such that teammates all hold a stake in the team, and the team wins or loses as a whole. The structure of project organization is shown in Figure 1.

2.2 Agile Process

As a 3^{rd} year undergraduate course, it could be very hard for students with only basic programming skills to design a system as a whole from the beginning. In the lecture, students learn progressively comprehensive use of various SE methods and techniques, such as application frameworks, development tools, requirements analysis techniques, and team building skills [8]. A carefully designed process is thus necessary to guide students to build up the system step-by-step. Through the process, students gain the skills to break down requirements into manageable small tasks, and to decompose system into decoupled modules.

A customized Agile Scrum process is adopted in our SE course project design. The development process is structured into five iterative Sprint deliveries. Following the Scrum process, for each team, customer representatives, TA, and all students in the development team meet at every Sprint (2

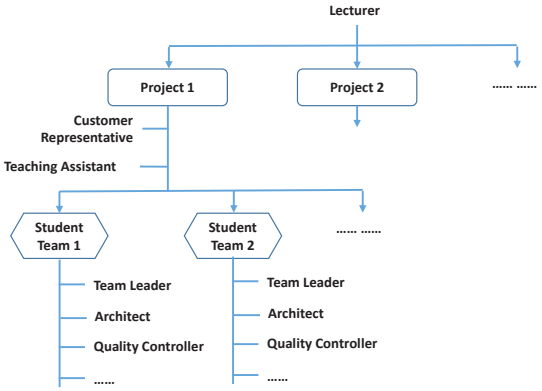


Figure 1: Project organization structure. Diversified projects are collected from different customers. Customer representatives and student assistants are invited for each project to help hand-on knowledge transferring. Students are organized into teams with 3-5 members taking various roles.

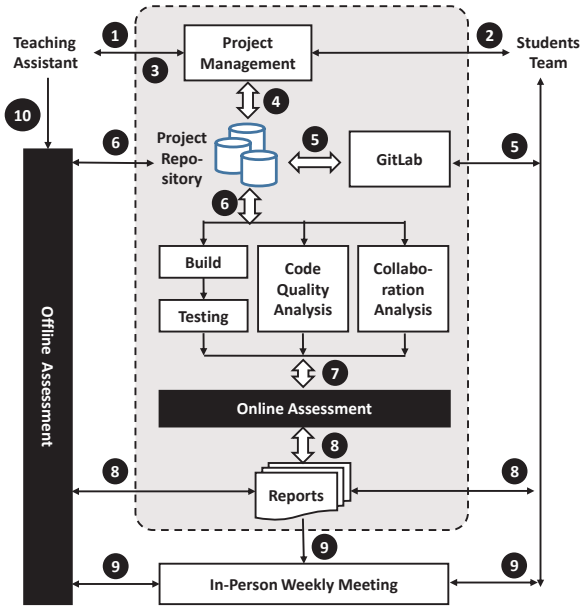


Figure 2: The DevOps platform which builds the tool chain for course project supervision.

weeks) while the TA meets with the team every week. At each Sprint, customer representatives direct each development team to implement a subset of requirements based on each team's individual progress. During the process, students practice with SE methods, techniques and tools which they learned in the class. Between the in person meetings, on a daily basis, students and TAs use instant messaging to discuss the problems discovered in the automated assessment and feedback reports generated by the project management platform.

The above process intends to build good engineering habits though repeated reinforcement, such as eliminating bad code smells, continuous version control, coverage-based unit testing, and so on. Systems are finally wrapped up, deployed on the specified cloud platform or customers' environment, and delivered with necessary readme and configuration files.

2.3 The Automatic DevOps Platform

The wide adoption of DevOps is inspired by modern SE methods and culture in the Internet era. To accommodate rapid changes, the Agile manifesto advocates early, frequent, and continuous delivery of working software to facilitate close cooperation between customers and developers throughout the prototype system. To support Agile practices, DevOps aims to provide an infrastructure of automatic tool chain, which reflects the interaction of development, quality assurance and operations. The tool chain in general consists of various tools to automate lifecycle project management, including source code management tools, build tools, testing tools, deployment tools, configuration tools, and monitoring tools [10]. As Gartner Report noted, "DevOps Philosophy Shapes a Culture", which is characterized by continuous delivery with continuous quality.

The DevOps platform for course project supervision was built by integrating various framework and tools, including GitLab[1] for source code management [5], Jenkins[2] for build management, Codeface[3] for Git repository analysis, and SonarQube[4] for code quality management. Build management framework like Jenkins can further integrate with plug-in tools, such as Docker[5], xUnit[6], and Selenium[7] to support automatic deployment and testing. The results from various tools are collected for assessment and reports from following three perspectives:

(1) Source code quality analysis, such as cyclomatic complexity, code style, bad smells, and code clone.
(2) Test coverage and bug reports.
(3) Collaboration analysis, such as contributions of each team member, version branching patterns, commit frequency, and the adequacy of comments.

2.4 Online-Offline Combined Assessment

Ideally, one would hope that these feedback and assessment are *fully* automated based on well-defined and objective metrics for every trackable action of teams and students. In reality, close TA-student interaction is still a must since the metrics are still far from perfect at this stage. Therefore, we

[1] https://www.gitlab.com
[2] https://jenkins.io
[3] https://siemens.github.io/codeface
[4] https://www.sonarqube.org
[5] https://www.docker.com
[6] https://xunit.github.io
[7] http://www.seleniumhq.org

apply a combined approach which incorporates both online automatic evaluation based on the DevOps platform, and the offline manual work based on regular Sprint meetings and in person discussions. Figure 2 shows the combined online-offline project supervision and assessment process, supported by the DevOps platform.

(1) TAs set up course projects.
(2) Students form teams and bid for the projects.
(3) TAs allocate projects to development teams.
(4) The code repository is automatically set up at GitLab for each team.
(5) Throughout the process, students collaborate based on GitLab source code management.
(6) Tools are configured to be integrated in a streamline, automatically triggered by predefined events. Code commits to GitLab automatically trigger the build, testing, and analysis actions.
(7) Whenever code updates are committed to GitLab repository, static code quality review, build, coverage-based testing, and docker delivery are automatically triggered immediately. Issues, commits, and branches are also tracked and assessed periodically. Assessment, with objective and well-defined metrics, are then automatically generated based on either existing tools or our proposed metrics.
(8) Scores and rankings are generated for teams/students immediately, and reports delivered to teams/students regularly with a maximal interval of 1 day. Problems are discovered quickly. Students/teams can either improve by their own or ask TA targeted questions. TAs can also review the reports and proactively contact the students for any discovered problems.
(9) TAs still manually grade each team and each student based on the demos and online/onsite discussions, with the automatic reports as the important input.
(10) At the end of the semester, TAs combine the weekly grades to give a final grade, taking into consideration the feedback from customer representatives on the final project demo.

3 ASSESSMENT METRICS

Both team and individual performance are monitored throughout the project process from various perspectives. Even though we strongly encourage students to share weal and woe in the collaboration, there are still individuals who perform especially well or poorly in the team. Assessment tracks every student's performance and contribution in the team. Each individual student gets bonus or penalty points on top of team scores. Specifically, assessment metrics are defined from following 5 perspectives.

- Issue (Task). GitLab issue management is used for task allocation, scheduling, and tracking. Metrics are defined to evaluate the workload of each Sprint and each team member, the duration and delay of task completion time, etc.
- Commit. Code should be committed with proper size and frequency, and clear message. The metric aims to encourage students to form good commit habits.
- Branching. The formation of branching in the repository network reflects collaboration situation. There are many discussions of well-formed branching patterns, and the principles that developers should follow in collaborative development. Metrics we defined for branching guidelines include merge your own code, early branching, and merge often.
- Test Coverage. Using container and xUnit tools, unit testing is automatically exercised after commits, and coverage statistics are collected and reported. The objective is to enhance code quality by enforcing test-driven development practices.
- Code Quality. Static analysis tools like SonarQube facilitate code quality analysis based on pre-defined rules. Reports are provided with identified potential bugs, vulnerabilities, bad smells, and duplications. Students are thus encouraged to follow good coding styles.

For test coverage and code quality, we directly integrate the reports of third-party tools. The rest of the section focuses on the first three perspectives, *i.e.*, the assessment of issue, commit, and branching.

3.1 Issue(Task)

Students are required to use GitLab issue management for backlog task management in Scrum process. Issues are thus considered as tasks with various tags annotating task types, status, and priorities. Issue board is used to facilitate task planning and organization, and to visualize progress by tracing issues open/close status. Students need to adhere to following task management guidelines.

- Each member should be allocated with appropriate amount of work. A team should schedule and balance the workload among team members, to avoid overloading or underloading.
- Each task should be finished within a reasonable amount of time. A task is marked as started when the corresponding issue is opened, and done when the issue is closed. Each task is defined with an estimated duration time, usually 1 to 2 days in the project Sprint backlog. A delay may be caused by unrealistic estimation, too-large task granularity, or technical obstacles, which should be warned for improvements.

Statistics are collected for teams as well as individuals. Statistics for team evaluation include the load balance of issues and time among team members. Delayed issues are reported separately to attract attentions. Similarly, working and idle time are also reported to facilitate future workload allocation.

A metric is defined as follows to measure project progress of each team. Given a team of n team members and project duration T, $PROG$ measures the percentage of effective time in the duration of all team members. Ineffective time is counted based on two parts:

- Over-slack time ($SLACK$), that is, the time of the period without any task assignments that has a length longer than expected;
- Delay time ($DELAY$), that is, the extra time of the tasks whose durations are beyond the scheduled time.

$$PROG = max\{\frac{nT - \sum_{i=1}^{n}(SLACK_i + DELAY_i)}{nT}, 0\} \quad (1)$$

Suppose T_SLACK is the threshold of tolerable slack time. The over-slack time for a team member i, $SLACK_i$, is calculated as follows.

$$SLACK_i = \sum_{j=1}^{m_i} Free_{i,j} \times f_{i,j}$$

where

- $\{Free_{i,j}|j = 1, .., m_i\}$ is the set of time intervals that a team member i has no task assignments;
- $f_{i,j}$ is the tolerable factor of the slack time interval $Free_{i,j}$, defined as follows:

$$f_{i,j} = \begin{cases} 0 & Free_{i,j} \leq T\_SLACK \\ 1 & Free_{i,j} > T\_SLACK \end{cases}$$

Suppose $\{TASK_{i,k}|k = 1, ..., s_i\}$ is the set of tasks assigned to a team member i in the project duration, $TASK\_C_{i,k}$ is the actual completion time and $TASK\_S_{i,k}$ is the scheduled time of a task $TASK_{i,k}$, the total delay time for i, $DELAY_i$, is calculated as follows.

$$DELAY_i = \sum_{k=1}^{s_i} max\{TASK\_C_{i,k} - TASK\_S_{i,k}, 0\}$$

3.2 Commit

Observations on student projects showed that size, comments, and frequency are three kinds of common problems for code commits. Beginners tend to push large modifications after finishing a complicated task, and may push libraries together with their modifications, such as jquery.js with 10 thousand lines of code. Abnormal commit size is an indicator of such abnormal operations.

In addition, each commit needs to contain messages with enough information and be clearly specified, which is often omitted by students. Most of commit messages are casual ones such as "update", "adding a file", and "fix a bug". Insufficiently documented comments are useless for teammates to understand the modification. It is necessary to form the habit of providing concrete and specific commit messages.

Moreover, there are also abnormal commit frequencies and quantities. For example, a team may make a large number of commits frequently short before weekly team meeting, and then very few during the week. Furthermore, the number of commits among team members may be imbalanced. In some extreme cases, the team leader may undertake all the commits for team members.

Taking these problems into consideration, we defined the metrics to assess the commit quality and enforce good commit style. Suppose $\{cm_{i,j}|j = 1, ..., c_i\}$ is the set of commits by a team member i, each commit, $cm_{i,j}$, is evaluated by three factors.

- $mod_{i,j}$ for the reasonable modification size of each commit. To avoid over-commit, a threshold is defined to control the size of each commit in terms of added/deleted lines of code. A commit is reasonable only if its size is within the defined threshold.
- $msg_{i,j}$ for the reasonable message length of each commit. A commit should be adequately described of its updates in its message. The length threshold is a basic indicator. If the comment message of a commit is too short, it could be a potential insufficient description.
- $freq_{i,j}$ for the reasonable frequency of commits. The intervals between two consecutive commits are restricted by upper-bound and lower-bound threshold, to discourage excessive frequent commits in short time, or over scattered commits which could be an indicator of inactiveness.

The metric is thus calculated as follows.

$$COMMIT_i = \sum_{j=1}^{c_i} mod_{i,j} \times msg_{i,j} \times freq_{i,j} \quad (2)$$

Where

$$mod_{i,j} = \begin{cases} 1 & MLN_{i,j} \leq T\_MOD \\ 0 & MLN_{i,j} > T\_MOD \end{cases}$$

where $MLN_{i,j}$ is the size of the j^{th} commit of team member i in term of the number of added/deleted lines of code, and T_MOD is the threshold of reasonable size of a commit.

$$msg_{i,j} = \begin{cases} 0 & MsgLN_{i,j} < T\_MSG \\ 1 & MsgLN_{i,j} \geq T\_MSG \end{cases}$$

where $MsgLN_{i,j}$ is the length of message of the j^{th} commit of team member i, and T_MSG is the threshold of reasonable length of a commit message.

$$freq_{i,j} = \begin{cases} 0 & INTV_{i,j} < L\_FQ \\ 1 & L\_FQ \leq INTV_{i,j} < U\_FQ \\ \frac{E\_FQ - INTV_{i,j}}{E\_FQ - U\_FQ} & U\_FQ \leq INTV_{i,j} < E\_FQ \\ 0 & INTV_{i,j} \geq E\_FQ \end{cases}$$

where $INTV_{i,j}$ is the time interval of the j^{th} commit of team member i, L_FQ and U_FQ are the lower bound and upper bound of acceptable time respectively, and E_FQ is the extended interval with progressively decreased score.

Furthermore, we also assess the team as a whole in terms of workload variance among team members using standard deviation of commits numbers, as shown in Equation (3). The result is restricted in $[0, 1]$, where the normalized standard deviation is compared to distribution $(1, 0, 0, 0)$ with standard deviation 0.5.

$$COMMIT\_TEAM = \frac{0.5 - \frac{1}{n\overline{C}}\sqrt{\frac{\sum_{i=1}^{n}(COMMIT_i - \overline{C})^2}{n-1}}}{0.5} \quad (3)$$

Where

$$\overline{C} = \frac{1}{n}\sum_{i=1}^{n} COMMIT_i$$

3.3 Branching

Branching is an effective tool for educating collaborative development. Students learn by practice how to divide work, parallelize development, merge code, and deal with conflicts. Appleton et al. analyzed various branching patterns for parallel software development [3]. It pointed out that even though branching is widely used in version control systems, the policies and guidelines may not be well followed, which could result in misuses of branching and merging mechanisms that break parallel development. This is usually a serious problem for educating version control to junior students. In this course, we define measurement to assess if a project's branching style conforms to guidelines from three perspectives.

- Merge Your Own Code (MYOC). One can only merge his or her own code with another branch to guarantee the familiarity with what to be done, which is named as Merge Your Own Code (MYOC). In a course of 2016 semester, 22 teams out of all 42 teams were detected MYOC violations on 63 merges. Among these violations, 36 merges were executed by team members that did not show up in recent commits, while 26 by members that never committed in the merged branches.
- Early branching. We encourage students to place each task in one branch, and each branch is dedicated to one task. In case a branch has multiple independent tasks, it is suggested to divide the branch (and tasks) and parallelize the development. This also helps to decompose a large task into multiple subtasks which are distributed to multiple development branches.
- Merge often. This is to enforce staged delivery to have at least one release in each Sprint iteration. Merge with conflicts is one of challenges to learn in collaborative development, and it is necessary to train students with efficient merge and conflict resolution capabilities.

3.3.1 MYOC. MYOC metric is defined to evaluate the activeness of a student's collaboration by counting how often he/she initiates the merge. For team member i in a team, suppose TM_i is the total merges of his/her branches, and M_i is the number of merges by him/her on his/her branches, then the ratio between the two numbers, $MYOCI_i = \frac{M_i}{TM_i}$, indicates the activeness of i in the collaborative development.

3.3.2 Early Branching. To provide advices of early branching, we evaluate the correlations among the commits in a branch. For two commits c_i and c_j, if the files they update overlap, these two commits are considered correlated, $R(c_i, c_j)$. Commits that are not correlated in any sense are recommended to be divided into different branches as they may address different tasks.

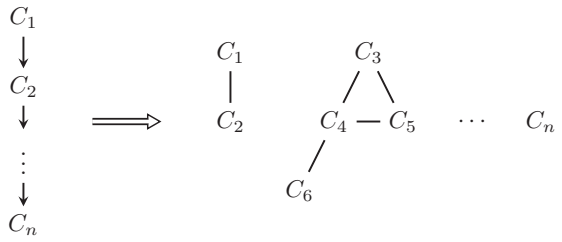


Figure 3: An example of early branching analysis. The set of commits $\{C_1, C_2, ..., C_n\}$ are in one branch. Analysis shows that $\{C_1, C_2\}$ and $\{C_3, C_4, C_5, C_6\}$ are two separated sets of correlated commits, which are candidates for dividing into two branches.

More specifically, a graph is built to analyze the correlations among commits, which is defined as $G =< C, R >$ where C is set of nodes and $R = C \times C$ is the set of edges. Each commit is represented as a node in the graph. Two nodes of commits are linked by an edge if they are correlated, which is measured by the percentage of overlapped files they update. The linked nodes of commits form a connected subgraph. In case there exists multiple subgraphs, each subgraph is a candidate for a new branch.

Figure 3 shows an example of early branching analysis. Suppose that a branch contains a set of commits $\{C_i | i = 1, 2, ..., n\}$. By analyzing the updates of these commits, it detects the following correlations between commits, $\{R(C_1, C_2)$, $R(C_3, C_4)$, $R(C_3, C_5)$, $R(C_4, C_5)$, $R(C_4, C_6)\}$. Using the graph tool, it identifies two independent subgraphs $G_1 =< \{C_1, C_2\}$, $\{R(C_1, C_2)\} >$, and $G_2 =< \{C_3, C_4, C_5, C_6\}$, $\{R(C_3, C_4)$, $R(C_3, C_5)$, $R(C_4, C_5)$, $R(C_4, C_6)\} >$. The commits in G_1 and G_2 respectively are potential updates for different tasks, and thus the candidates for different branches. It thus suggests to divide the large task branch into smaller ones.

Given a Sprint period, suppose that the number of all branches is BR and the number of well-formed branches is BR_W which conform to the Early Branching guideline, the goodness of branching is measured by the percentage of well-formed branches over all branches as follows.

$$EB = \frac{BR\_W}{BR}$$

3.3.3 Merge Often. We take a Sprint iteration as a release cycle. Hence, it requires that before the Sprint milestone, all the branches in the iteration need to be merged to the master branch. Figure 4 shows an example of merge often pattern. Three branches bifurcate from commit B_2, which merge back to the branch within a Sprint iteration.

Given a project with S Sprint iterations, suppose the number of Sprints that follow the Merge-Often (MO) guideline

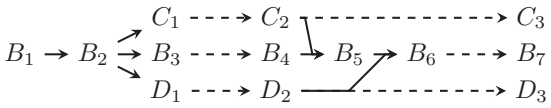


Figure 4: An example of merge often pattern. The branches are encouraged to be merged back in a Sprint iteration as a release.

is S_MO, the conformation to the guideline is defined as follows.

$$MO = \frac{S\_MO}{S}$$

4 OPERATIONAL EXPERIENCES AND RESULTS

4.1 Implementation

A prototype system was built based on GitLab. Our platform so far has focused on Java and Python projects. For Java projects, it can support automatic build using Ant[8]; code analysis and quality evaluation using Checkstyle[9], Findbugs[10], Pmd[11], Simian[12], and Sonarqube; unit testing using JUnit[13]; and Web application testing using Selenium. For Python projects, it can support code-based quality evaluation using Pylint[14], Pep8[15], Clonedigger[16], and Sloccount[17]; unit testing and coverage analysis using Nosetests[18].

Various statistics are collected throughout the project process. Figure 5 shows some example reports produced by the system, including the assessment of tasks, commits, Merge Your Own Code (MYOC), Early Branching (EB), Merge Often (MO), and overall evaluation. Each assessment is ranked at 4 levels: wonderful, good, bad, and awful, represented by 4 different colors respectively. The pie charts show the statistics of all projects in the class from different aspects.

To motivate peer competitions, rankings of individuals and teams are also reported. Figure 6 shows the Personal Commits Ranking (PCR) statistics, including total commits, commits by week, and commits by month.

Figure 7 shows the metrics for a project. The radar chart shows the multi-dimensional quality evaluation, including issues (tasks), commits, MYOC, EB and MO for the whole team; and individual EB and MYOC for the current user. It can further show the detailed information of each metric as well as the explanation and demonstration of the guidelines.

[8]http://ant.apache.org
[9]http://checkstyle.sourceforge.net
[10]http://findbugs.sourceforge.net
[11]https://pmd.github.io
[12]http://www.harukizaemon.com/simian
[13]http://junit.org
[14]https://www.pylint.org
[15]https://www.python.org/dev/peps/pep-0008
[16]http://clonedigger.sourceforge.net
[17]https://www.dwheeler.com/sloccount
[18]https://nose.readthedocs.io

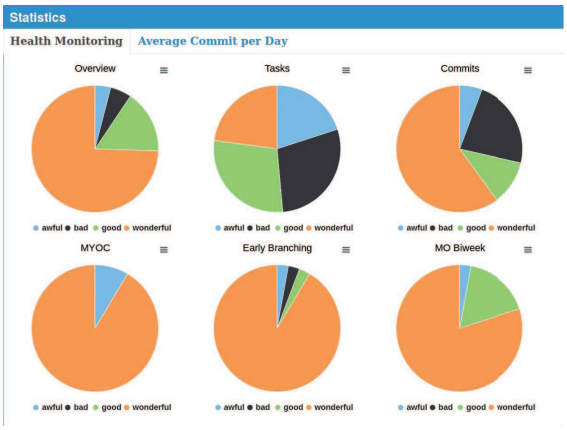


Figure 5: Project status statistics of the all projects in the class. The pie charts show the statistics of projects at each level for each metric.

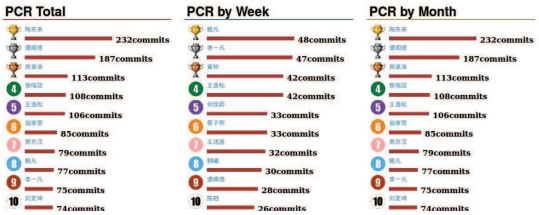


Figure 6: Personal commits ranking (PCR), which lists the top 10 students with the most commits.

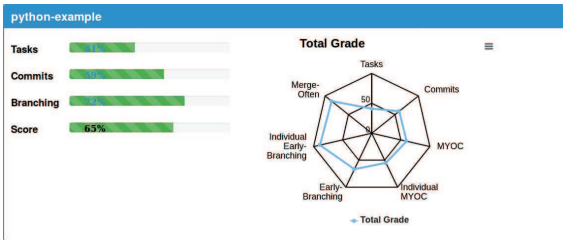


Figure 7: Project quality metrics for individual team and student. The radar chart shows evaluations from seven perspectives for the current student.

4.2 Students' Performance

In this paper, we reported the data collected from the DevOps platform for 2017 Fall semester during the 64 days of project development, following a customized Scrum process with bi-weekly Sprint iterations. There are 141 students enrolled in the course, organized into 33 teams participating 11 projects. Altogether 68 code repositories were created in GitLab, with 3076 issues and 14506 commits throughout the project lifecycle.

To analyze students' performance, Table 1 shows the parameter settings of the metrics.

Metric	Parameter	Explanation	Setting
Issue(Task)	T_SLACK	The threshold of tolerable slack time.	3 days
Commit	T_MOD	The threshold of the size of updated code in a commit.	200 LOC
Commit	T_MSG	The threshold of the length of a commit message.	20 characters
Commit	L_FQ	The lower bound of commit frequency.	30 minutes
Commit	U_FQ	The upper bound of commit frequency.	1 day
Commit	E_FQ	The extended interval of commit frequency with progressively decreased score.	3 days

Table 1: The Parameter Settings of Metrics

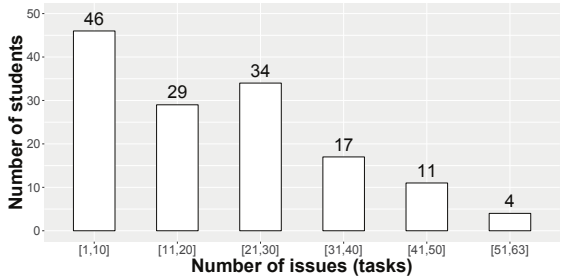


Figure 8: The distribution of workload among students. The X-axis is the intervals of number of issues(tasks) and Y-axis identifies the number of students at each interval.

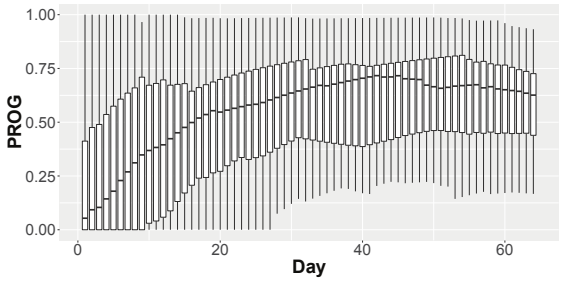


Figure 9: Assessment of task planning in terms of the percentage of effective time throughout the project process, as defined by the metric PROG in Section 3.1.

4.2.1 Issue(Task). We encouraged students to use issue management for planning and progress tracing, as discussed in Section 3.1. At each Sprint during the Scrum process, the team leader discussed with all members, TA and customer representatives to identify the set of tasks of the Sprint backlog, recorded tasks in the system as GitLab issues, and allocated them to each team member. Students were expected to gradually improve their capabilities of project planning, workload estimation, and task division of collaborative development.

Figure 8 shows the statistics of total issues allocated to students. About 5% issues were identified but not allocated, which were mostly discarded features. All students were allocated tasks with at lest 1 issue. We divided the number of issues into six intervals, as an approximation of the workload at different levels from light to heavy. It showed that most of the students were allocated with less than 30 issues through the process of 5 iterations/10 weeks, with an average of 1-3 issues per week. We also encouraged the students to trace the issues to commits, that is, from planned task to code implementation. However, statistics showed that only about 45% students associated over 50% of their commits to their issues. Most of the commits were not tagged with issues, and most of the students were not well accustomed to this guideline yet. We intent to improve this in future work, so that we can have more insights and advices on the granularity of issues (that is, the workload of tasks) in terms of the size of code.

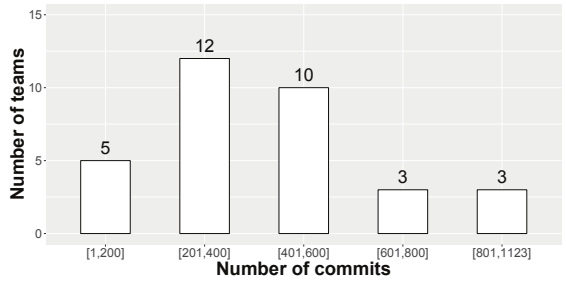
Figure 9 shows statistics of the metric PROG in Section 3.1, the assessment of the percentage of time with well-planned tasks. We are glad to see that the average value increased throughout the semester, that is, most teams had less slack time and fewer delayed issues as projects progressed, which could be a sign of their improved planning and management abilities.

4.2.2 Commit. We analyzed the number of commits as well as the quality of commits. Students were expected to gain good engineering habits from following perspectives: 1) Adequate code version control, which is evaluated by the moderate frequency and size of code commits to the GitLab version control repository. 2) Sufficient documents so that the update of each commit could be well recorded and communicated to teammates.

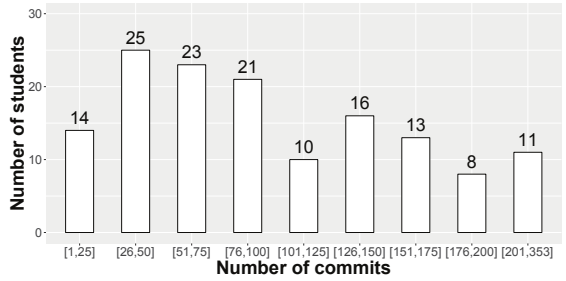
Figure 10 shows the statistics of total number of commits by project teams (Figure 10(a)), and by individuals (Figure 10(b)). We can see that active students/groups committed as many as ten times of those inactive ones. We could set up a bottom line (for example, at lest 1 commit per student per day during the development period), and pay more attentions to those most inactive students (for example, students with less than 50 commits) in the offline in person discussions.

Figure 11 shows the statistics of metric COMMIT in Section 3.2, a combined quality evaluation of commits. The improvements are obvious in general in terms of commit frequency, size, and documents. However, there are still scattered outliers. The process monitoring mechanism could be improved so that those outliers are warned immediately.

4.2.3 Merge Your Own Code. The metric MYOC in Section 3.3.1 aims to guide students to be responsible for their



(a) Commits by team



(b) Commits by Individual

Figure 10: The number of commits by team and by individual.

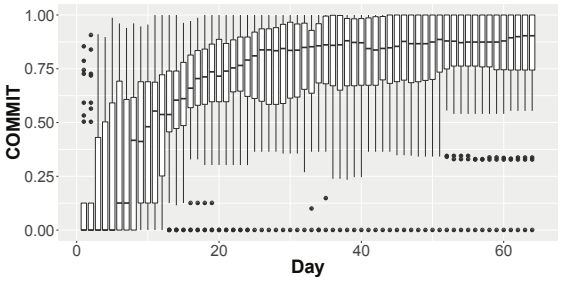


Figure 11: Assessment of the quality of commits for each team, in terms of commit frequency, size, and documents, as defined by the metric COMMIT in Section 3.2.

own code and to actively resolve merge conflicts in collaborative development. Figure 12 shows that once enforced, most of students quickly master the skills to follow the guideline. In the future, we will further look into the details of merge conflicts to provide instructions for timely and effective merge.

4.2.4 Early Branching. Based on our practices of educating SE projects throughout these years, we observed that branching is always a difficulty for novices in collaborative development. Students usually feel confused of when to create a branch and then merge back, and are unwilling to manage branches. The metric EB as defined in Section 3.3.2 intends

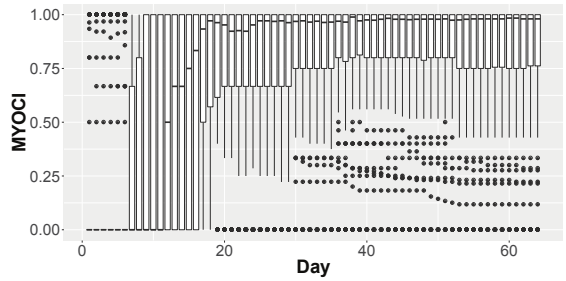


Figure 12: Assessment of the goodness of merge habits following the guideline of Merge Your Own Code, as defined by the metric MYOC in Section 3.3.1.

to encourage branches by features, and provide advices on candidates for branches.

Figure 13 shows the statistics of active branches of each team throughout the weeks. It shows that students gained the skills of the breakdown of tasks and implementations into branches for parallel development. Most of the teams can adequately manage a number of branches at each iteration, avoiding over-branching at the same time.

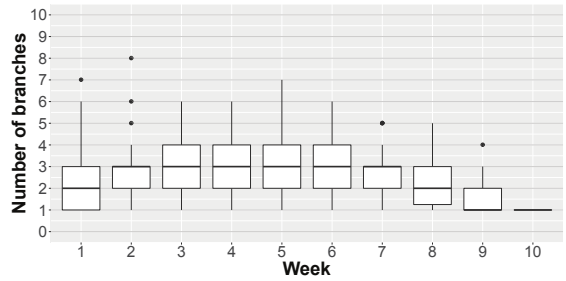


Figure 13: The number of active branches for each team.

We further analyzed how well the tasks are sufficiently divided, as measured by the metric EB in Section 3.3.2. To identify the correlations between commits, we set the threshold of overlapping as 20% such that two commits in a Sprint are considered correlated if their overlapped files are above 20%, and a link is identified between the two nodes of the commits in the correlation graph. Figure 14 shows that at the beginning, most of the students did not use branching and there were considerable overlaps between team members. Such chaos during the first two weeks were greatly improved once the guideline was enforced.

5 DISCUSSION AND FUTURE WORK

Teaching Agile development warrants continuous delivery of personalized assessment and feedback to project teams and individual students, with a frequency that catches up with iterative, incremental, and cooperative software development with continuous deliveries. Towards this direction,

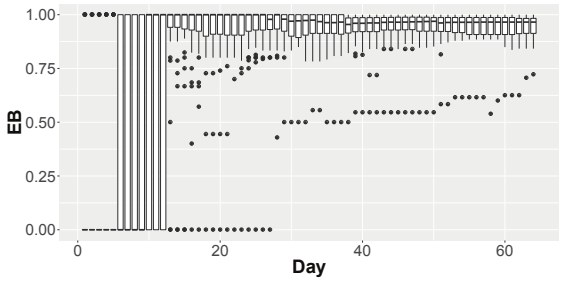


Figure 14: Assessment of the goodness of branching habits following the guideline of Early Branching, as defined by the metric EB in Section 3.3.2.

as an important first step, we propose an efficient two-tiered approach that frees TA from rudimentary and straightforward problems. 1) Whenever code updates are committed to GitLab repository, personalized reports, with objective and well-defined metrics and rankings, are then automatically generated and delivered to the teams and students. In this way, majority of problems, especially those rudimentary and straightforward ones, are discovered timely. 2) With the reports, the discussions between students and TAs are in a very targeted manner, daily via instant messaging and weekly in in-person project meetings, making the best of TAs' limited time. The system has been in operations since Fall 2014, and our preliminary results indicate this system is very effective in improving students' SE skills and reducing TAs' workload. We believe that this two-tiered approach is generally applicable to Agile projects in SE courses.

A key issue in the proposed approach is the properness of the metrics. Project management and team collaboration are, to a large extent, based on experiences and personnel characteristics. In real industry environment, they are greatly affected by organizational culture and may differ a lot from one project/organization to another. It is hard to define unified guidelines and assessment. However, in the education environment, we need disciplines to systematically train students the necessary engineering skills so that they can be prepared to cope with future large and complex software development in the real-world.

The paper reported our first attempt to address the needs. Aware of the differences between industry and education, we tried to identify the essential guidelines and valuable skills to train in class. Some of the metrics are suitable for beginners in a controlled course project context, but not for experienced engineers in an industry-scale project. The metrics, though far from perfect yet, are useful tools for process monitoring and assessment. They provide quantitative views of the students' behavior and progress, so that both students and TA/lectures can get clear understanding of the actual problems and the directions for improvements. The metrics can be continually adjusted and improved based on feedbacks from the class as well as industry.

The research is still in its early stage. It revealed some interesting problems that we could not know without close and in-depth progress tracking and data analysis. In the future, we plan to continue the efforts from following aspects: 1) To enhance quality assessment by incorporating reports from various tools such as testing, code analysis, and online monitoring, and build quality control throughout DevOps lifecycle; 2) To build more intelligence into our platform for quality results synthesis, analysis, assessment, and warning.

ACKNOWLEDGMENTS

We would like to thank all the teachers and students who support the course development. Special thanks to Dr. Wolfgang Mauerer and his team for sharing with us Codeface tools, their knowledge and experiences.

REFERENCES

[1] Lukas Alperowitz, Dora Dzvonyar, and Bernd Bruegge. 2016. Metrics in Agile project courses. *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16* (2016). https://doi.org/10.1145/2889160.2889183

[2] Craig Anslow and Frank Maurer. 2015. An Experience Report at Teaching a Group Based Agile Software Development Project Course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 500–505. https://doi.org/10.1145/2676723.2677284

[3] Brad Appleton, Stephen P. Berczuk, Ralph Cabrera, and Robert Orenstein. 1998. Streamed lines: Branching patterns for parallel software development. *PLoP* (1998).

[4] Robert Chatley and Tony Field. 2017. Lean Learning - Applying Lean Techniques to Improve Software Engineering Education. *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)* (May 2017). https://doi.org/10.1109/icse-seet.2017.5

[5] Joseph Feliciano, Margaret-Anne Storey, and Alexey Zagalsky. 2016. Student experiences using GitHub in software engineering courses. *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16* (2016). https://doi.org/10.1145/2889160.2889195

[6] Phillip A. Laplante. 2006. An Agile, Graduate, Software Studio Course. *IEEE Transactions on Education* 49, 4 (Nov 2006), 417–419. https://doi.org/10.1109/te.2006.879790

[7] Maíra Rejane Marques. 2016. Monitoring: An Intervention to Improve Team Results in Software Engineering Education. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA. https://doi.org/10.1145/2839509.2851054

[8] Ana M. Moreno, María-Isabel Sánchez-Segura, Fuensanta Medina-Domínguez, Lawrence Peters, and Jonathan Araujo. 2016. Enriching traditional software engineering curricula with software project management knowledge. *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16* (2016). https://doi.org/10.1145/2889160.2889193

[9] Maria Paasivaara, Jari Vanhanen, Ville T. Heikkila, Casper Lassenius, Juha Itkonen, and Eero Laukkanen. 2017. Do High and Low Performing Student Teams Use Scrum Differently in Capstone Projects? *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)* (May 2017). https://doi.org/10.1109/icse-seet.2017.22

[10] Rayene Ben Rayana, Sylvain Killian, Nicolas Trangez, and Arnaud Calmettes. 2016. GitWaterFlow: a successful branching model and tooling, for achieving continuous delivery with multiple version branches. *Proceedings of the 4th International Workshop on Release Engineering - RELENG 2016* (2016). https://doi.org/10.1145/2993274.2993277