Experiences with auto-grading in a systems course

Dhananjai M. Rao
Computer Science and Software Engineerring (CSE) Department
Miami University
Oxford, OHIO 45056. USA
raodm@miamiOH.edu

***Abstract*—Rapidly growing computing-class sizes across college campuses are challenging instructional resources and staffing. We have developed an automatic testing and grading software system called Code Assessment Extension (CODE). It is integrated with our existing Canvas Learning Management System (LMS). This paper presents experiences from both student and instructor perspectives with using auto-grading in a junior-level, systems course with a heavy emphasis on programming in C++, with several challenges, including – ① this course was the first experience for both the students and the instructor in using any form of automatic grading, ② the students have limited experience with C++ programming, particularly in Linux, and ③ the course includes complex concepts on operating systems, multithreading, and networking. The paper presents quantitative results from 3,300 submissions (from 1 course, 1 semester, 6 programming assignments, 54 students, multiple submissions per-student per-assignment) and analysis of end-of-course surveys from 54 students. The inferences from the statistics strongly support the use of automatic grading systems such as CODE to enhance learning in programming-centric courses.**

***Index Terms*—Auto-grader, Programming, Automatic Testing, C++, Operating Systems, Networking, CODE**

I. Introduction and Motivation

Computer and information sciences (CIS) is a rapidly growing disciple, particularly at the baccalaureate level in the United States [7]. The longer-term trends in the number of students matriculating with undergraduate degrees in CIS is illustrated by Figure 1. The data in this chart has been obtained from the National Center for Education Statistics (NCES) [8]. As illustrated by the chart, currently, we are in the midst of a surge in enrollments causing heavy demand for computing courses across the nation [7]. Rapidly growing class sizes for computing across college campuses is challenging instructional resources and staffing [7]. The situation is further complicated by the challenges in hiring and retaining high-quality instructors [7]. These challenges have catalyzed the use of automatic testing tools to effectively teach programming as discussed in several recent publications [2], [6], [10], [9]. Several of these recent, closely related efforts are discussed in Section III.

A. Motivation for automated testing and grading

Consistent with national trends, we have also experienced significant growth in the number of majors in the Computer Science and Software Engineering (CSE) department. In just the last 3 years, our total number of students has almost doubled from 426 (Fall 2015) to 769 (Fall 2018). Moreover,

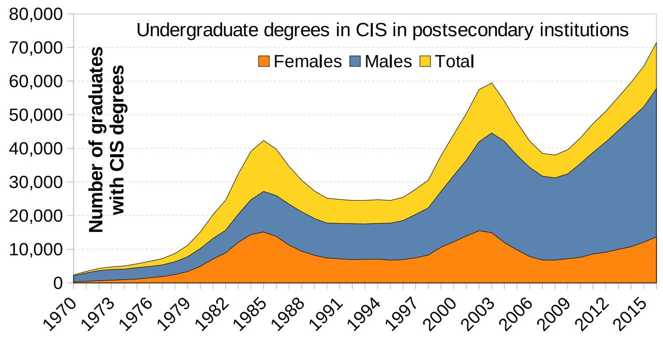


Fig. 1. Nationwide trends in undergraduate CIS enrollments [8]

a large number of non-majors also take courses offered by the CSE department. The rapid growth poses challenges in effectively teaching the substantially larger number of students. Accordingly, to ensure sustainable, high-quality outcomes for our students and instructors, we have developed an automatic testing and grading software system called Code Assessment Extension (CODE). It is integrated with our existing Canvas Learning Management System (LMS).

CODE has been designed to facilitate teaching and learning in programming-centric courses. It integrates with Canvas LMS, thereby preserving an environment that is familiar to both students and instructors. It automates the key processes of testing student submissions, including: compiling, style-checking, running functional tests, and annotating the source code to provide feedback. From a student perspective, CODE streamlines student experiences and provides timely (in about 20 seconds) feedback regarding various aspects of a solution. The student can use the feedback to refine their solution, thereby improving learning outcomes. From a faculty perspective, CODE eliminates many of the tedious tasks enabling instructors to focus on higher-level course outcomes, such as: conciseness of solution, design, quality, etc. Section II provides further details on the architecture and design of the CODE plug-in.

B. Pertinent education theories

The pedagogical underpinnings for the CODE plug-in stem from the interplay of two key educational theories, namely: behaviorism and constructivism. As the name suggests, "behaviorism" is a learning theory that focuses on eliciting desirable behaviors, typically through repetitive conditioning [1]. Specifically, Computer-based Training (CBT) builds on behaviorist theory to facilitate learning particularly at knowledge, comprehension, and application cognitive levels of Bloom's taxonomy. Our CODE plug-in falls into the category of behaviorist tools because it is designed to elicit several positive behaviors, including: ① regular testing via automated tests, ② maintaining good coding style via its style checkers, ③ improve structure and cohesion by limiting methods to 25 lines. The CODE plug-in also embodies the essence of constructivism in which students actively learn by solving a problem iteratively. The CODE plug-in runs functional tests and provides feedback to the students quickly (in ~20 seconds). The feedback enables the students to observe issues, form mental models, and enhance their solutions to troubleshoot reported errors. Combined with the aforementioned behavioral outcomes, the plug-in also facilitates learning at the application, analysis, and synthesis levels in Bloom's taxonomy.

C. Key contributions of this paper: Innovative Practice

Typically, auto-grading tools are used in introductory programming courses. However, this paper presents experiences with using auto-grading in a junior-level Systems-2 course (with computer science, software engineering, and computer engineering students) with a heavy emphasis on programming in C++, with several challenges, including – ❶ this course was the first experience for both the students and the instructor in using any form of automatic grading, ❷ the students have limited experience with C++ programming, particularly in Linux, and ❸ the course includes topics on operating systems, multithreading, and networking, which involve complex concepts.

Another conspicuous contribution of this paper is that it provides a comprehensive quantitative and qualitative analysis of experiences using automatic testing. Section V presents a detailed quantitative and qualitative analysis of data collected during the course. Statistical analysis of 3,300 submissions from 54 students (6 assignments, multiple submission per-student per-assignment) adds support for utilizing automatic-grading tools such as CODE to enhance teaching and learning in programming-centric courses. The experiences will benefit other educators (and the community) who are debating the use of auto-graders.

II. The Code Assessment Extension (CODE)

The Code Assessment Extension (CODE) has been developed to facilitate teaching and learning in programming-centric courses. CODE automates the process of compiling, style checking, testing, and annotating feedback for source code submitted by students. An overview of the software systems constituting CODE is illustrated in Figure 2. The core software system has been developed in PHP programming language using the popular Laravel framework. All interactions with the system are performed via web-pages served by an Apache web-server over HTTPS.

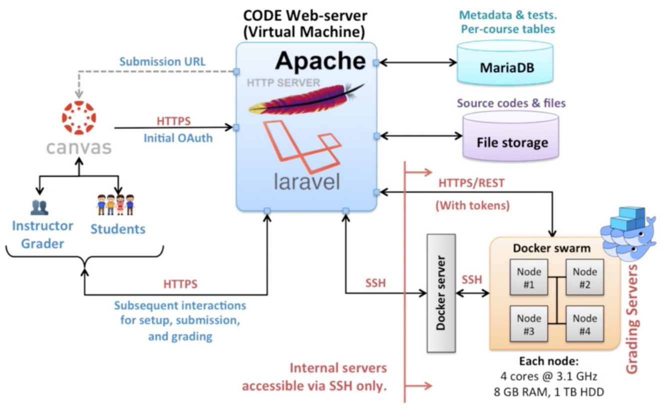


Fig. 2. Architectural overview of CODE

CODE has been designed to operate with our existing Canvas Learning Management System (LMS). Integration with Canvas has been accomplished using the Learning Tools Interoperability (LTI) capabilities of Canvas. In more generic terms, CODE can be viewed as a plug-in to Canvas. The primary motivation for this design approach is to reuse all the familiar features of Canvas for both students and instructors, including: assignment creation, assignment management, solution submission, grading via speedgrader, providing feedback, and managing grades via gradebook. Moreover, this design enables us to delegate initial authentication tasks to Canvas and CODE interacts with Canvas using OAuth tokens. Consequently, sensitive login credentials are not stored or managed by CODE, which alleviates a large number of security concerns.

The CODE plug-in is setup for all users by our IT department via suitable LTI configuration. Once setup, both instructors and students log onto CODE via Canvas. Instructors initiate interactions with CODE by creating an assignment in Canvas and using the `Website URL` as the submission type. This selection causes Canvas to permit the instructor and students to interact with CODE. The plug-in permits instructors to further configure assignment settings by selecting programming language, uploading necessary files, and configuring functional tests. Each test case includes optional inputs, command-line arguments, environment settings, and expected outputs. The expected outputs are used for validating student submissions. The assignment and test configurations are stored in a SQL database.

Students submit solutions to the assignment via CODE. The student submissions are stored on File storage and metadata about submission is stored in the SQL database. Next, the plug-in schedules a grading task to be run asynchronously via the backend Docker service as shown in Figure 2. The Docker service starts a Docker container that runs a dedicated `grader` process. The `grader` process downloads the student submissions, compiles it, runs tests, and stores the results back into the SQL database. Interactions between the `grader` process and the database are facilitated via a custom RESTful Application Program Interface (API). Once the grading process completes, the results are displayed back to the student. The student may then decide to finish the submission or start a new submission.

III. Related research

The challenges posed by the nationwide increase in enrollments for computing courses (see Figure 1) has catalyzed many institutions to seek innovative solutions to meet the demands. One of the well-established approaches is the use of automatic testing and grading software tools [12], [5]. Such "auto-grader" software, including our CODE plug-in, facilitate teaching and learning in programming-centric courses. Several such software systems have been reported in the literature [4], [5], [9], [10], [12]. This section presents a brief summary of some of the recent and closely related software tools. In addition, these publications also include references to similar tools. Readers are referred to the references in these closely related papers for a more comprehensive list of works.

Wilcox [11] presents a longitudinal study analyzing the effectiveness of automated testing in an introductory computing course. His grading software is comparable to our CODE plug-in but is primarily focused on Java programming language. From an analysis of over 1,000 student performance over 6 semesters, Wilcox concludes that automated testing improved overall average scores, reduced withdrawal rates, and reduced grading times [11]. A similar experience has been reported by Sherman *et al* with their auto-grading framework called "Bottlenose". They too observed an increase in submission rates, These two works provide strong support in favor of our CODE plug-in. Wilcox also presents a survey of testing strategies used by software for automatic grading of student programs [12]. Our CODE plug-in includes and enables several of the testing strategies, including: output comparison, stream control, object instantiation, instrumentation, reflection, and code analysis.

Peveler *et al* discuss the use of their auto-grading system called "Submitty" [10]. Similar to our CODE plug-in, Submitty also permits assessment of programs in different languages. However, unlike CODE that provides a simple interface to instructors to setup assignments and tests, Submitty requires a JSON input to be developed. Peveler *et al* also discuss a recent upgrade to Submitty in which they provide a performance comparison using Docker containers [9]. The use of Docker containers is similar to our approach, but with the difference that CODE uses a Docker swarm, which further reduces cost to spin-up containers.

Carbunescu *et al* discuss the issues involved in designing auto-grader for parallel code on the massively parallel XSEDE environment [2]. In contrast, our system is currently focused on running serial programs in different languages. Junival and colleagues propose and assess an automatic grading system called CPSGrader for cyber-physical systems [6]. Their work focuses on developing testing strategies in the context of programming robots. On the other hand, our CODE plug-in is purely focused on software testing, but in different programming languages.

Danutama and Liem [4] discuss their approach of integrating their auto-grading system into Moodle Learning Management System (LMS). They use a software bridge named "dispatcher" to loosely-couple Moodle and their autograder called LX. The "dispatcher" manages interactions between Moodle and LX. In contrast, CODE uses OAuth for integrating with Canvas LMS. Moreover, CODE uses a Docker swarm and Docker services for grading submissions.

IV. Course & Assessment Settings

The proposed Code Assessment Extension (CODE) plug-in has been used to teach a systems course (1 semester, *i.e.,* 15 weeks) with heavy emphasis on programming in C++. The topics in the course include core Operating Systems (OS) concepts, including multiprocessing, multithreading, and virtualization [3]. The concepts were applied in the context of networking and web-services to develop custom servers. This one semester-long course (15 weeks) consisted of 54 students with junior standing from 3 specializations, namely: computer science (42), software engineering (5), and computer engineering (7) students. The 3-credit hour course included 2-hours of lectures and one 2-hour laboratory session each week. The use of the CODE plug-in was introduced in the second week of the course as part of a lab exercise. Subsequent labs did not involve the use of the plug-in, primarily due to the nature and time constraints in lab sessions. However, all programming homework projects involved the use of the plug-in to submit solutions in the form of C++ source code.

The CODE plug-in was used for assessment of 6 separate homework in the course. Homework was not assigned during weeks around 2 midterm exams and during Thanksgiving break. The students had 7 days to complete each assignment that was due at midnight. However, the complexity of the assignments varied throughout the course, with complexity generally increasing as enumerated below:

1) File processing and operations on `unordered_map`
2) Printing process hierarchy from a file
3) Develop a custom shell using fork & exec
4) Custom Common Gateway Interface (CGI)
5) Multithreaded text file processing
6) Monitoring processes in virtual machine via `/proc`

In addition to a comprehensive description of the requirements, each homework included expected outputs from tests. Each homework also included "base test cases" – *i.e.,* minimal functionality that the program must accomplish in order to earn any points for the homework. These base test cases serve two key purposes. The first purpose is to motivate the students to learn the core concepts. In addition, they clarify that skeleton programs will not earn points.

The students were permitted an unlimited number of submission trials via the CODE plug-in until the stipulated deadline for each homework. The course used `NetBeans` as the recommended Integrated Development Environment (IDE). In addition, `NetBeans` and the CODE plug-in were configured to have identical settings to streamline student experience. The students were permitted to ask questions and clarifications at any time via online discussion forums. The students routinely solicited help from the instructor during office hours, for 4-hours each week. In addition, students also had access to 12 total hours of teaching assistant's office hours. Collectively, these resources were intended to facilitate learning and foster student success.

V. Results and Discussions

The CODE plug-in was used for automated assessment of student submissions for 6 homework assignments. In total, students submitted over 3,300 submissions as part of the 6 homework, with different number of submissions per-student for each homework. All of the submissions are retained in the CODE plug-in's database. The database also includes metadata about the submissions, including the time when the submissions were made. Figure 3 shows the distribution of the number of submissions as recorded in the database. There was one student who used 221 submissions for HW #5 and this data point is a conspicuous outlier and is not included in Figure 3.

As illustrated by the shape of the violin plots in Figure 3, most of the students required several submissions for each homework. The number of submissions for each student was clustered around the averages for each homework. However, several students required considerably more number of tries as indicated by the long tails on these distributions. The tails are longer for the latter assignments reflecting the trend of increasing complexity as the semester progresses. The total number of submissions (value for `N` in Figure 3) and the mean number of submissions-per-student reflect this trend, with the `Mean` increasing from 6.13 to 13.95. However, HW #4 was a larger term project and consequently, the average number of submissions (`Mean: 18.77`) was higher. The data in Figure 3 suggests that the average number of tries by the students reflected the complexity of the assignments. Possibly, this information can be used as a pseudometric to assess relative complexity of assignments.

A. Analysis of grades versus submission counts

An interesting question that arises in this context is the possibility of a correlation between the number of submissions versus the overall grade of students – *i.e.,* "Do students who perform well use more or fewer number of submissions per homework?" In order to explore this question, the degree of correlation between the total number of submission and the average of submissions-per-homework was analyzed. The correlogram in Figure 4 illustrates the results from this analysis. In this chart, the `Score` parameter corresponds to the final weighted score for each student in the course. The parameters `TotSubs` and `AvgSubs` corresponds to the total number of submissions (for all 6 homework) by each student and the average number of submissions-per-homework respectively.

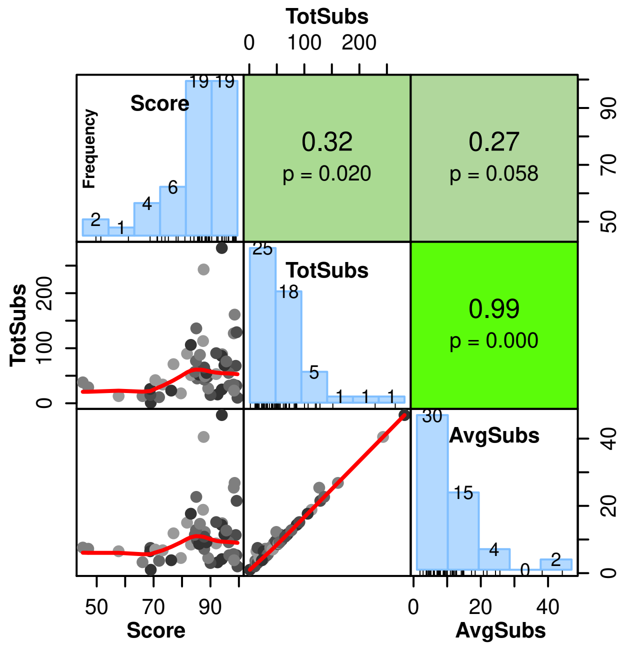


Fig. 4. Analysis of correlation between overall scores and number of submissions

The correlogram in Figure 4 shows only a weak correlation between the final scores of students and the submission parameters. The Pearson correlation coefficient between scores and the total number of submissions per student was the higher of the two at 0.32 (p=0.020). The correlation between scores and the average number of submissions-per-homework is weaker at 0.27 (p=0.058). The LOESS curve (▬ in Figure 4) for the `score` parameter shows two distinct regions with low corelation, *i.e.,* relatively flat line. These two regions correspond

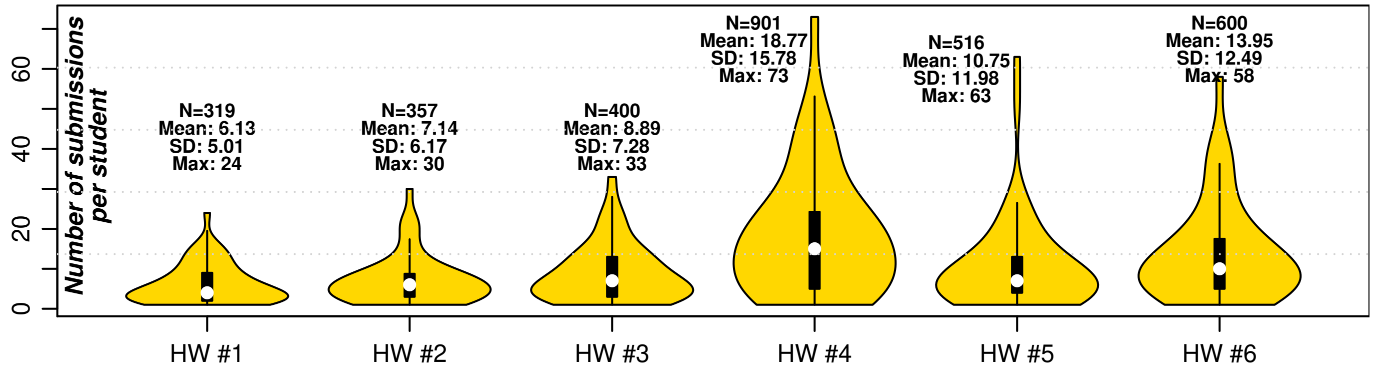


Fig. 3. Distribution of number of submissions for each homework with 1 outlier removed

to students with a lower (`score` < 70) and higher (`score` $>$ 85) scores. The observation suggests a piecewise trend, with students at either extreme requiring about the same number of trials. However, this trend could be an artifact of the skew in scores as illustrated by the `score` histogram. Most of the students performed well in the course (typical of higher-level course with mostly majors) with only a few underperforming students.

B. *Analysis of submission patterns*

A key characteristic that is of interest, particularly to instructors, is the work patterns of students. The submission timestamp metadata in the CODE database provides an insight into the work culture and habits of students at an institution. The charts in Figure 5 illustrate the submission patterns observed for the 6 homework assignments. In these charts, the x-axis shows the relative submission time with respect to the deadline – *i.e.,* the zero value on the x-axis is the submission deadline. Students had a week to work on each assignment. The blue curves (▬ in Figure 5) illustrate the submission trends for each homework. Note that the total number of submissions is not the same for each homework.

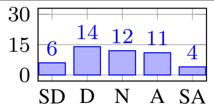
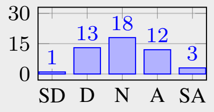
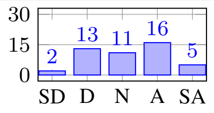
The charts for the six assignments show a pretty consistent pattern where most of the submissions are received in the day before the deadline. Across all six assignments, the average submission time was ~17.5 hours before the deadline, with over 62% of the submissions occurring in the last 8 hours. The submission patterns are consistent immaterial of the increasing complexity (see Figure 3) of the assignments. The students were informed about the increasing complexity but had no perceptible change in submission patterns. The data strongly suggest that students actively work on assignments just before the deadlines. Consequently, software systems such as the CODE plug-in need to be designed with an emphasis on managing peak loads.

C. *Analysis of start and finish times*

A key aspect that is stressed to ensure student success is to start working early on homework assignments as it provides the following benefits to students – ① it enables the students to plan and invest their time better, time, ② it permits the students to review selected topics for the assignment, and ③ solicit help from the instructor, TAs, or via online forum to troubleshoot issues. For each homework, students were reminded to start early to gain the aforementioned benefits. The statistics in Table I presents the student responses to these topics. The data was collected via an anonymous survey administered towards the end of the course. The data essentially reflects the students perception of their work habits.

Most of the students (~69%) disagreed or were neutral that they invested time to review material outside of class. Importantly, most of the students did not even review the short (~2 minutes) tutorial videos outside of class. This data suggests that students would need increased help with learning, applying, and retaining knowledge. Importantly, only about 32% of the students feel that they start work early

TABLE I
STUDENT FEEDBACK ON ASSIGNMENT EFFORTS FROM ANONYMOUS END-OF-COURSE SURVEY LEGEND: SD: STRONGLY DISAGREE, D: DISAGREE, N: NEUTRAL, A: AGREE, AND SA: STRONGLY AGREE.

Survey question	**Response (N=47)**
Every week, I review lecture slides and watch handout videos outside of class	
I start early on programming projects and plan my time	
I actively seek help from the instructor via online forums or during office hours. time	

on assignments. However, 45% of the students felt that they actively seek help from the instructor to troubleshoot issues. This is consistent with the instructor and TAs observations that there is a rush to seek help on the day assignments were due. Sadly, this trend did not change throughout the semester. Even students who underperformed in the course continued their pattern of delayed started despite being unsuccessful at completing earlier assignments.

1) Analysis of grades versus start and finish times: The observation of delayed starts (see Table I) and the large volume of submissions (see Figure 5) leads to the question of impacts on student grades. Consequently, we also pursued an analysis of a student's overall scores versus the timestamp of their first and last submission for each homework. We hypothesized that earlier start times would be a strong indicator of higher scores. The histogram in Figure 6 illustrates the submission times for the first and last submissions by each student. The histograms show average (from all 6 assignments) first and last submission times for all 50 students in the course. Each bar in the histogram corresponds to 5-hour intervals, with zero on x-axis corresponding to the submission deadline.

The histograms in Figure 6 shows that a large fraction of the students have the final submissions in the last 5 hours prior to the deadline. However, many students do have their first submissions coming in early, with ~20% of the first submissions occurring 48 hours before the deadline. The Correlogram in Figure 7(a) illustrates the correlation between the overall final scores versus the first and last submission times for each student. Both `score` vs. the `last` submission and `score` vs. the `first` submission show only a weak correlation of 0.37 (p-value of 0.007) and 0.39 (p-value of 0.004) respectively. In other words, the first and last submission times are not a strong indicator of overall student success. However, the LOESS curve (▬ in bottom-left corner of Figure 7(a)) for `score` vs. the `first` submission suggests a trend in the data.

The chart in Figure 7(b) shows a zoomed-in version of correlogram between `score` and time for the `First` submission. In this chart, the range of overall final scores has

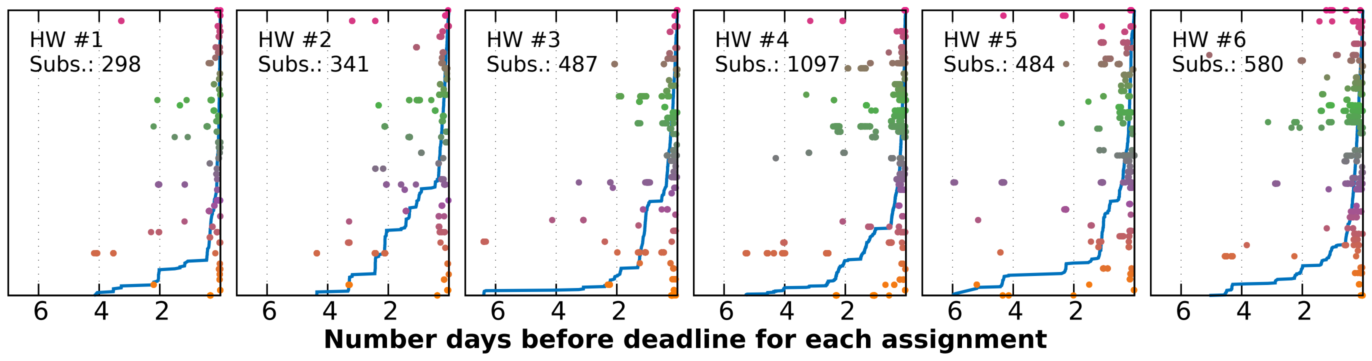


Fig. 5. Plot of submission times with reference to deadlines. Colors correspond to different students in the course. The y-axis has no unit but all submissions for a student are on one horizontal line. (`Subs.` indicates the number of submissions)

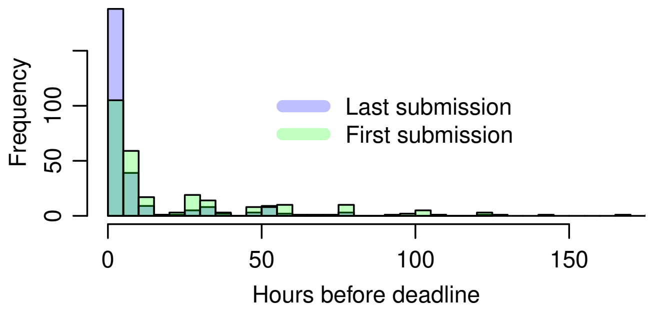


Fig. 6. Histogram of average submission times w.r.t deadlines

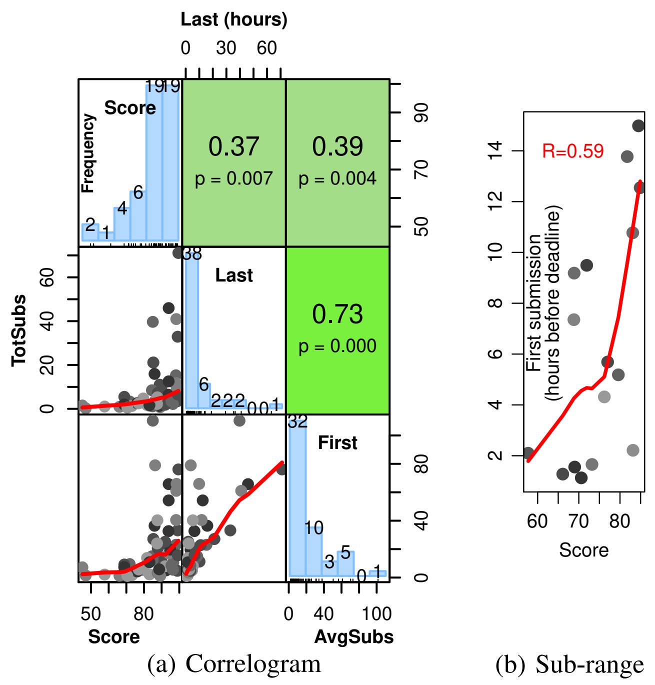


Fig. 7. Correlogram of scores vs. first & last submission times

been restricted to 50 <`score` <85. This range of scores restricts to students who have a B or lower grade in the course. This range of `scores` vs. time of the `First` submission shows a much stronger correlation of 0.59. This data indicates that the time for the first submission is a good predictor of student performance, for middle-tier students. In this context, it is important to note that we are discussing an interesting correlation and not causation – *i.e.,* we are not concluding that students underperform because they start late on assignments.

D. Analysis of student perceptions

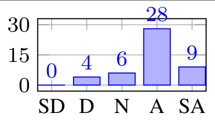
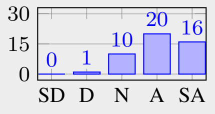
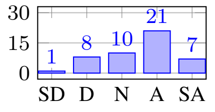
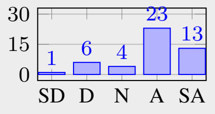
The data in Table II summarizes student perceptions about the effectiveness of automatic grading performed via the CODE plug-in. The feedback was obtained from students as part of the anonymous survey (same one used for data in Table I) conducted at the end of the course. The results for the last question in Table II clearly shows that a majority ($>$ 77%) of the students perceived that automatic testing had a strong positive effect on their learning. In addition, almost 80% of the students agreed that automatic testing helped to reduce errors in their program, with $<$ 10% disagreeing on this question. As expected, the majority of the students agreed that they liked getting early feedback, with only 2% of the students disagreeing on this question.

One minor concern that was observed on the student survey was regarding their reliance on automated testing. About 60% of the students indicated that they have come to rely on the tests run by the CODE plug-in to verify their solutions. On one hand, it can be argued that students have access to a tool and they are using it effectively. On the other hand, having students perform tests on their own is equally important. Striking a balance on this topic would be ideal.

VI. Conclusions

The rapid growth in demand for computing across college campuses is challenging instructional resources. The challenges have motivated the use of automatic testing tools, particularly in programming centric courses. This paper presented an automatic testing and grading software system called Code Assessment Extension (CODE). CODE has been developed as a plug-in to our current Canvas Learning Management System (LMS). Our plug-in has been used to teach a Systems

TABLE II
STUDENT FEEDBACK ON AUTOMATIC GRADING FROM ANONYMOUS END-OF-COURSE SURVEY LEGEND: SD: STRONGLY DISAGREE, D: DISAGREE, N: NEUTRAL, A: AGREE, AND SA: STRONGLY AGREE.

Survey question	**Response (N=47)**
I think automatic testing has helped reduce errors in my program	
I liked getting early feedback whether I have met most (if not all) program expectations via automated testing.	
I have come to rely on some of the automated testings to check if my program actually works as expected	
Overall I think the automatic testing has helped me improve my programs/solutions.	

2 course, which has a heavy emphasis on programming in C++. The paper provided a detailed analysis of quantitative metrics elicited from over 3,300 student submission. The paper also discussed qualitative feedback collected from the students. Collectively, our analysis supports the following inferences:

- Automatic testing is imperative because only 30% of the class agreed they regularly review course materials and consequently, rigorous testing and quick feedback is essential to motivate learning.
- In the anonymous survey, over 85% of the students indicated that they have a good understanding of testing and they were emphatic that they thoroughly test their programs prior to submission. Yet, on average, students used 10 submissions (median is 7) per program, indicating a dichotomy between student perceptions and student performance in practice. The data suggest that an automatic testing system is an important component to improve the quality of solutions.
- The average number of tries by the students reflected the complexity of the assignments. Possibly, this information can be used as a pseudometric to assess relative complexity of assignments.
- In the survey, ~70% of students indicated they start early on programs. Yet the submission patterns suggest that student start much later, often just the day assignments are due. Most of the testing activity occurred just a few hours before the deadline. This observation further adds support that automatic testing is an important tool as it can provide quick feedback to help students as they are rushing to submit solutions.
- In the survey, ~80% of the students had developed a positive opinion about automated testing. They indicated that they have come to rely on CODE to ensure their programs work as expected and to help reduce errors.
- CODE provides a convenient highlight-and-annotate feature to provide feedback to students. However, the instructor's experience suggests that students do not pay attention to feedback if it is collapsed in the User Interface (UI). Consequently, having non-collapsible UI may be for the better for feedback.
- A majority of the submissions are received in the last few hours before deadlines. Consequently, it is imperative that tools are carefully designed to handle peak loads rather than focusing on average characteristics.
- Students were introduced to CODE via a short video and they practiced using it in one lab session prior to using it for homework. The brief introduction to working with the plug-in was sufficient for students to learn to work with CODE, suggesting that the plug-in is relatively easy for students to learn and use.

The student surveys and statistics from student submissions show that automatic testing is an important component in a programming course. Combined with student surveys the statistics expose interesting discrepancies between student perceptions of self-effort and attitude versus their manifestations in practice. Faculty experience suggests that automatic testing helps to improve the overall quality of solutions and learning outcomes. Overall the students have a very positive opinion on automatic testing. Almost 80% of the students indicated that they have come to rely on automatic testing and agree it helps improve their learning. However, from an instructor perspective, this is a "double-edged sword" – having students rely too much on tools could be impacting longer-term learning outcomes – and a longitudinal study would be necessary to assess this concern.

Overall the results from a slew of quantitative and qualitative analysis provide strong, positive support for the importance and effectiveness of automatic testing enabled via the CODE plug-in. The plug-in required about 6 months of development time by a small team, with essentially just 1 developer, 1 accessibility specialist, and 1 part-time system administrator. We are planning on motivating further use of the CODE plug-in in other courses. We also envision to make it available as a Free and Open Source Software (FOSS) for use by other educational institutions.

SUPPLEMENTARY MATERIALS

Video demonstrations and tutorials for CODE, from both instructor and student perspectives, are available online at: https://code.cec.miamiOH.edu/code.

ACKNOWLEDGMENTS

We would like to thank faculty and staff in eLearning and Miami University IT for their help and support in certifying and deploying the CODE plug-in.

REFERENCES

[1] B. Bakhshinategh, O. R. Zaiane, S. ElAtia, and D. Ipperciel, "Educational data mining applications and tasks: A survey of the last 10 years," *Education and Information Technologies*, vol. 23, no. 1, pp. 537–553, Jan 2018.

[2] R. Carbunescu, A. Devarakonda, J. Demmel, S. Gordon, J. Alameda, and S. Mehringer, "Architecting an autograder for parallel code," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, ser. XSEDE'14. New York, NY, USA: ACM, 2014, pp. 68:1–68:8.

[3] CSE Department, "Removed for double blind review," 2019. [Online]. Available: Removedfordoubleblindreview

[4] K. Danutama and I. Liem, "Scalable autograder and lms integration," *Procedia Technology*, vol. 11, pp. 388 – 395, 2013, 4th International Conference on Electrical Engineering and Informatics, ICEEI 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2212017313003617

[5] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '10. New York, NY, USA: ACM, 2010, pp. 86–93. [Online]. Available: http://doi.acm.org/10.1145/1930464.1930480

[6] G. Juniwal, A. Donzé, J. C. Jensen, and S. A. Seshia, "CPSGrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory," in *2014 International Conference on Embedded Software (EMSOFT)*, Oct 2014, pp. 1–10.

[7] National Academies of Sciences Engineering & Medicine, *Assessing and Responding to the Growth of Computer Science Undergraduate Enrollments*. Washington, DC: The National Academies Press, 2018.

[8] NCES, "National center for education statistics (NCES): Degrees in computer and information sciences conferred by postsecondary institutions," 2018. [Online]. Available: https://nces.ed.gov/programs/digest/d18/tables/dt18_325.35.asp

[9] M. Peveler, E. Maicus, and B. Cutler, "Comparing jailed sandboxes vs containers within an autograding system," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '19. New York, NY, USA: ACM, 2019, pp. 139–145. [Online]. Available: http://doi.acm.org.proxy.lib.miamioh.edu/10.1145/3287324.3287507

[10] M. Peveler, J. Tyler, S. Breese, B. Cutler, and A. Milanova, "Submitty: An open source, highly-configurable platform for grading of programming assignments (abstract only)," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '17. New York, NY, USA: ACM, 2017, pp. 641–641.

[11] C. Wilcox, "The role of automation in undergraduate computer science education," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 90–95.

[12] C. Wilcox, "Testing strategies for the automated grading of student programs," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16. New York, NY, USA: ACM, 2016, pp. 437–442.