The Impact of a Remote Live-Coding Pedagogy on Student Programming Processes, Grades, and Lecture Questions Asked

Anshul Shah
ayshah@ucsd.edu
University of California, San Diego

Vardhan Agarwal
v7agarwa@ucsd.edu
University of California, San Diego

Michael Granado
magranado@ucsd.edu
University of California, San Diego

John Driscoll
jjdrisco@ucsd.edu
University of California, San Diego

Emma Hogan
emhogan@ucsd.edu
University of California, San Diego

Leo Porter
leporter@ucsd.edu
University of California, San Diego

William Griswold
bgriswold@ucsd.edu
University of California, San Diego

Adalbert Gerald Soosai Raj
asoosairaj@ucsd.edu
University of California, San Diego

ABSTRACT

Live coding—a pedagogical technique in which an instructor plans, writes, and executes code in front of a class—is generally considered a best practice when teaching programming. However, only a few studies have evaluated the effect of live coding on student learning in a controlled experiment and most of the literature relating to live coding identifies students' *perceived* benefits of live-coding examples. In order to empirically evaluate the impact of live coding, we designed a controlled experiment in a CS1 course taught in Python at a large public university. In the two remote lecture sections for the course, one was taught using live-coding examples and the other was taught using static-code examples. Throughout the term, we collected code snapshots from students' programming assignments, students' grades, and the questions that they asked during the remote lectures. We then applied a set of process-oriented programming metrics to students' programming data to compare students' adherence to effective programming processes in the two learning groups and categorized each question asked in lectures following an open-coding approach. Our results revealed a general lack of difference between the two groups across programming processes, grades, and lecture questions asked. However, our experiment uncovered minimal effects in favor of the live-coding group indicating improved programming processes but lower performance on assignments and grades. Our results suggest an overall insignificant impact of the style of presenting code examples, though we reflect on the threats to validity in our study that should be addressed in future work.

CCS CONCEPTS

• **Social and professional topics → Computing Education**.

KEYWORDS

live coding, programming processes, incremental development, debugging, grades, lecture experience

ACM Reference Format:
Anshul Shah, Vardhan Agarwal, Michael Granado, John Driscoll, Emma Hogan, Leo Porter, William Griswold, and Adalbert Gerald Soosai Raj. 2023. The Impact of a Remote Live-Coding Pedagogy on Student Programming Processes, Grades, and Lecture Questions Asked. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023), July 8–12, 2023, Turku, Finland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587102.3588846

ITiCSE 2023, July 8–12, 2023, Turku, Finland
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0138-2/23/07.
https://doi.org/10.1145/3587102.3588846

1 INTRODUCTION

Live coding is a pedagogical technique in which an instructor writes code in real-time in front of students in a class [26]. In contrast to static-code examples, live-coding examples allow the instructor to demonstrate coding concepts and techniques in a dynamic and interactive way. Multiple prior studies have shown benefits of live coding, such as improved learning outcomes [6, 25] and reduced extraneous cognitive load during lectures [22]. Specifically, students and instructors have also reported perceived benefits to the *programming process*, such as engaging in incremental coding [3, 23, 25], improved debugging skills [4, 19], and better testing skills [3, 16].

Despite live-coding being recommended as a best practice for teaching programming [5], a literature review on live coding [26] revealed a lack of work that evaluates the empirical impact of live coding on student learning and programming processes. To bridge this gap, we conducted a quasi-experimental study during the Spring 2022 term at UC San Diego, a research-intensive public university in the United States, where a CS1 course was taught *remotely* with live-coding examples in one section and with static-code examples in another section. The purpose of our study is to provide an empirical examination of unresolved questions from prior work, relating to students' programming processes, learning outcomes, and lecture experiences.

Throughout the term, as students completed weekly programming assignments, we collected snapshots of their code every time they ran it. We also collected students' grades on each assignment, exam, and other course activities. Finally, we obtained the text of each question students asked in lecture via chat, enabled by the remote technology used for lectures. Together, these sources of data allowed us to empirically evaluate whether a live-coding pedagogy impacts students' programming processes and the types of questions that students asked during lectures. Specifically, we ask the following research questions:

(1) How does a remote, live-coding pedagogy impact students' programming processes, such as adherence to incremental development and using debugging techniques?
(2) How does a remote, live-coding pedagogy impact student performance on assignments and exams?
(3) How does a remote, live-coding pedagogy impact the types of questions that students ask during lectures?

2 RELATED WORK

2.1 Prior Live Coding Controlled Studies

Live coding has been empirically evaluated over the past couple of decades, with numerous studies evaluating the impact of live coding on student perceptions. However, a literature review of live coding [26] indicated that only three prior studies have used an experimental approach to assess the impact of live coding on student *learning* [22, 25, 29].

In 2013, Rubin conducted the first controlled study that compared the impact of live-coding examples and static-code examples on students' grades on assignments, exams, and a project [25]. The experimental design carefully controlled for the effect of the live-coding examples by keeping all other conditions consistent between the two groups, including the instructor, course content, and assignments. The results indicated no significant difference in assignment and exam grades, but students in the live-coding group earned significantly higher grades on the final course project.

A 2016 Tan et al. used a pre- and post-test approach to measure the improvement of students' conceptual understanding and program implementation skills after 18 weeks of asynchronous live-coding lecture videos [29]. The results of the study showed that conceptual understanding did not improve from the intervention, but the program implementation skills did show improvement. However, since the study did not use a control and treatment group, the results do not show the advantages of live-coding examples over static-code examples.

In 2020, Soosai Raj et al. conducted a randomized, controlled experiment to measure improvement in student learning [22]. The study was designed such that the only difference between the two groups was in the presentation of coding examples—the live-coding group only saw code examples in which the instructor wrote the code from scratch. The study found that students in the static-code group showed slightly higher improvement between pre- and post-tests than students in the live-coding group, although the difference was not statistically significant. This result suggests that live coding may not improve student outcomes in traditional assessments such as exams that involve skills besides code writing (e.g., code tracing).

Although the results of these studies are inconclusive, we aim to replicate these attempts to evaluate the learning impact of a live-coding pedagogy. Further, these prior works are limited to evaluating learning in terms of student grades; however, they do not shed light on the impact of live coding on the specific programming processes that students use.

2.2 Programming Process Metrics

In order to assess the impact of a live-coding pedagogy on student programming processes, we leverage pre-existing metrics that measure how effectively students adhere to ideal processes, such as incremental development and effective debugging techniques. In this section, we describe the process-oriented metrics that are relevant to our research questions.

2.2.1 *Incremental Development.*

The Measure of Incremental Development (MID) developed by Shah et al. measures a student's adherence to incremental development based on whether a student added manageable chunks of code [27]. It was developed for CS1 programming tasks and is designed to be agnostic to the size of the task. The metric rewards a student for writing code in smaller chunks and not experiencing excessive struggle after large code additions. The recent work by Charitsis et al. similarly aims to quantify program decomposition, using Natural Language Processing [8]. We note, however, that program decomposition can be slightly different from incremental development. The metric by Charitsis et al. defines program decomposition by the progression of *functions* written by students, whereas the MID is agnostic to the number of functions written.

2.2.2 *Debugging.*

Since Jadud introduced the Error Quotient in 2006 [13], numerous metrics have been developed that relate to students' debugging processes and frequency of encountering errors [2, 7, 15, 30, 31]. The metrics developed by Kazerouni et al. reward students for starting the assignment early with regard to the assignment deadline [14], which is not the skill we aim to measure for this experiment. The Normalized Programming State Model [7] and the Watwin Score [31] rely on features such as time spent working and semantic correctness of the code, which are expensive features to obtain. However, the Repeated Error Density (RED) [2] is not dependent on the size and programming language of the task and accounts for cases when students encounter repeated errors [30]. A lower score on the RED indicates that the student rarely encountered the error or resolved the error message quickly.

2.3 In-Lecture Effects of Live Coding

There is currently little research that evaluates the impact of live-coding examples on students during lecture. One of the few works in this area compared the cognitive load of viewing static-code and live-coding examples [22], which revealed students experience less cognitive load when viewing live-coding example. Although the work from Soosai Raj et al. evaluated the effect of bilingual instruction on the questions asked during lecture, we were moved by the methods applied in that study. Soosai Raj et al. noted the questions asked by students during lectures and grouped the questions into categories based on Bloom's taxonomy [24]. Although this previous analysis was outside the context of a live-coding pedagogy, we plan to conduct a similar analysis by comparing the types of questions between the treatment and control group in our study, since our remote pedagogy enabled easy access to questions asked by students.

3 STUDY DESIGN

3.1 Participants

Our study was conducted in a CS1 course at UC San Diego, which is a large public R1 university. It was approved by the UC San Diego IRB, and the IRB number is 201792. Of the 199 students that consented to participate in the study, 91 students were in the live-coding group and 108 students were in the static-code group. Students were not randomly assigned to groups; they self-selected into one of two lectures that were held at 9:30 AM or 11 AM on Tuesdays and Thursdays each week. At the time of enrolling, they did not know that there would be one live-coding lecture and one static-code lecture.

We asked students to complete a pre-course survey when the term started. In the survey, students self-reported their current year in university, race, and prior experience with programming. Students from each of the four years—from first year to fourth year—were represented roughly evenly in the two groups. In the live-coding group, 80% of students reported not having prior programming experience, compared to 75% in the static-code group. In the live-coding group, 50% of students self-identified as Asian, 20.5% identified as Latinx, and 17% identified as White. In the static-code group, 47.6% of students self-identified as Asian, 2.9% identified as Latinx, and 13.3% identified as White. Across both lectures, less than 10 students self-identified as any other race.

3.2 Experimental Design

We conducted our study over one academic term in Spring 2022. We used an experimental setup similar to that in Rubin's initial controlled study [25]. Each week, students attended two remote 80-minute lectures, a mandatory 50-minute in-person lab, and an optional 50-minute in-person discussion section. Students also had the option to attend office hours, either online or in-person, hosted by instructional assistants or the instructor.

Both groups were taught by the same instructor, who has experience teaching a CS1 course with both static-code and live-coding examples. The slides used in each lecture were identical for both groups, except for the 4-6 code examples per lecture that were shown to students. Importantly, students from both groups could access previous lecture slides, which included the static code snippets presented in class. Moreover, both the live-coding group and the static-code group were explicitly taught about incremental development, debugging techniques, and how to write test cases in a lecture halfway through the term. We elaborate on the impact of this lecture in Section 6.2.

The only difference between the treatment and control group in the remote lectures was the presentation of the code examples. In line with previous studies that used live-coding examples [26], no pre-written code was presented to students when the instructor used live coding. Instead, the instructor started with an empty Python file and wrote the code from scratch. While live coding in front of the class, the instructor intentionally engaged in effective programming processes, such as incremental development and using print statements for debugging. For example, when showing longer code examples, the instructor broke the example down into smaller chunks and compiled the code after writing each smaller part. Additionally, the instructor occasionally made syntactic or semantic errors and demonstrated how to use print statements to locate them. Because of this unique aspect of the live-coding lecture, we hypothesize the students in the live-coding group may have acquired and used these implicit skills in their own programming tasks. Some sample code snippets that were developed during the live-coding portion of the lecture can be found at the following link: https://bit.ly/code-examples-live-vs-static.

In the static-code group, the instructor showed a slide that had pre-written Python code on it. The code snippets were identical to the code written during the live-coding group. However, instead of writing the code in an IDE, the instructor annotated the code snippet by drawing memory diagrams, listing values of variables during execution, or other notes that may be helpful for student comprehension.

The controlled parts of the experiment included all other aspects of the course, such as the required lab sections, optional discussion sections, and office hours. Lab sections typically involved students working in pairs on short programming tasks that covered lecture content and did not provide an opportunity for a presentation of code examples. In each discussion section, three teaching assistants spent 50 minutes reviewing recent lecture material and had students work through roughly 3 to 5 code tracing questions. However, we were not able to control the office hour interactions, as the 30 teaching assistants may have had different teaching styles in the 1-on-1 interactions with students.

4 METHODS

4.1 RQ1: Programming Processes

Throughout the term, students completed 8 programming assignments (PAs), each of which had two to three programming tasks. Students were given one week to work on each PA and were allowed to use pair programming [18]. The PAs covered all of the content taught in our CS1 course: basic syntax, conditionals, functions, `for` loops, `while` loops, image manipulation using 2-D lists of tuples as images, dictionaries, and reading files. Students completed the PAs on EdStem [9], a platform that includes an online integrated development environment (IDE). EdStem provided our research team with snapshots at runtime across all assignments for each student who consented to participate in the research, consistent with our human subjects research protocol. In total, we collected approximately 150,000 snapshots across the 8 assignments.

Although we assigned 18 total programming tasks across 8 PAs, not every task lent itself to analysis. In some tasks, students were not asked to write functions and were given significant scaffolding. Similarly, some tasks in later assignments were only one function long or did not require complex logic. Instead, we wanted to analyze the longer, more complex programming tasks to collect a fuller representation of their programming processes. Therefore, we removed PAs 1 and 2 from our analysis, since they both included scaffolding and did not require students to write any functions. A member of the research team then selected one task from each PA that required more functions or needed more complex logic to implement than the other task(s). The exact instructions and content of each programming task in our final data can be found at https://bit.ly/programming-tasks.

Once we decided on the six tasks to analyze, we applied a suite of process-oriented metrics to measure incremental development and debugging skills. Due to their flexibility and relevance to our research questions, we applied the Measure of Incremental Development (MID) [27], which computes a student's adherence to incremental development, and the Repeated Error Density (RED) [2], which represents the amount a student struggled to fix a specific error. The MID was trained and evaluated on a similar data set to our programming tasks [27] and the RED is agnostic to the language or size of the program being analyzed [30]. Since other metrics required an input such as time spent in the IDE [7, 31], which we could not collect, or require unit tests to be written [15], which was outside the scope of our assignments, we could not apply them. Conversely, the MID and RED metrics only require snapshots at the time of compilation, which can be readily collected by online IDEs. We also applied a custom metric: the proportion of snapshots that include a print statement within a function. While this metric is not empirically evaluated and does not fully represent debugging skills, we chose this specific metric because the instructor used print-statement debugging during the live-coding lectures only. Therefore, this metric may reveal whether students implicitly picked up the programming processes demonstrated during live-coding lectures.

4.2 RQ2: Student Grades

We recorded the grades of all students[1] throughout the term, including those from weekly programming assignments, lab work, and reading quizzes on the Stepik e-textbook [28]. The labs and reading quizzes were required for students and were graded based on correctness, with unlimited attempts allowed until the deadline. Additionally, we collected scores from the midterm and final exams, as well as the overall course grades.

We evaluated student performance on weekly programming assignments (PAs), a midterm exam, a final exam, and overall grade (which included points from weekly lab and reading quizzes). For each student, we removed the lowest score out of the 8 PA scores as per course policy and calculated the average PA score.

4.3 RQ3: Lecture Questions Asked

One of the unique affordances of our remote experimental setting was access to all of the questions students asked via the Zoom chat. Teaching assistants monitored the chat and relayed questions to the professor that they thought would be useful to the class. Otherwise, teaching assistants responded in the chat. We have access to 427 questions asked during the 20 lectures. Of the 427 questions, 206 were asked by the live-coding group and 221 were asked by the static-code group. Since questions about course logistics are irrelevant to the impact of code examples on student learning, they were excluded, resulting in a set of 406 questions.

We used an open-coding ("affinity diagramming") approach [11] to categorize the student questions based on common characteristics. In order to categorize questions in an unbiased manner, the researchers were made blind to whether the questions were from the live-coding or static-code lecture by combining and randomly ordering the questions across all lectures.

[1]Note that the sample size for our student grades data is higher than the sample size for our other analyses. This occurred because more students consented to releasing their grades data from the course than to releasing their programming process data.

Table 1: Final code book achieved through open coding and deliberation

Label	Description: "Questions about..."
Conceptual	- how an element of programming works in general (not specific to the current program) - practical applications or real world scenarios
Syntax	- why a programming character or phrase is needed in an example - what certain Python terms mean
Result Explanation	- how/why a specific output was produced or why a result was correct - an idea of why a certain result occurred - how variables (names or values) change
Process	- the motivation for writing part of the code - why a programming element is used to further the current program - why a variable was given a certain name or assigned a specific value - where a portion of code was written
What If	- a hypothetical scenario (these are along the lines of "What if we ...")

Two researchers independently coded the first 60 student questions, creating their own initial code books. Those two code books were compared and combined to create a common code book. The researchers then began an iterative process of individually categorizing 50 unseen student questions according to the new code book, comparing results, discussing, and updating properties in the code book. After each iteration, the inter-rater reliability was measured and checked against an 80% agreement threshold. Following the third iteration, the researchers agreed on 41 (82%) of the 50 responses, resulting in a Cohen's kappa statistic of 0.77. After this point, the two researchers evenly divided the remaining questions and independently coded them according to the final code book, shown in Table 1.

5 RESULTS

5.1 RQ1: Programming Process Results

We conducted two-sample t-tests [21] to compare the MID, RED, and our custom metric between the two groups. In each application of the t-test, we had a sample size of well over 25 [17] and similar distributions between groups. We used an α value of 0.05 as our significance threshold for all tests, and applied a Holm-Bonferroni correction for tests with multiple comparisons on the same topic [12]. Table 2 shows the mean, standard deviation (SD), t-statistic (t), p-value (p), and Cohen's effect size (d) of all t-tests conducted. The Cohen's effect size indicates the standardized mean difference between the two groups (e.g., an effect size of 0.2 denotes that the mean in one group was 0.2 standard deviations higher than the mean in another group) [10].

Table 2: Comparison of programming process metrics between live coding (n = 90) and static code group (n = 107)

		Summary Statistics				
Metric	**Group**	**Mean**	**SD**	**t-stat**	**p**	**d**
MID	Live	1.75	0.98	-0.08	0.93	-0.01
	Static	1.77	1.07			
RED of TypeError	Live	0.32	0.41	-0.41	0.68	-0.06
	Static	0.34	0.49			
RED of NameError	Live	0.14	0.25	-0.17	0.87	-0.02
	Static	0.15	0.23			
RED of SyntaxError	Live	0.21	0.30	-0.97	0.33	-0.14
	Static	0.27	0.46			
Ratio of Prints	Live	0.19	0.19	0.84	0.40	0.12
	Static	0.17	0.15			

5.1.1 *Incremental Development.* We compared the adherence to incremental development between the two learning groups using the Measure of Incremental Development (MID). In Table 2, the first row summarizes the results of a two-sample t-test [21] of the overall MID between the two groups across all six programming tasks in our data set. When interpreting the MID, a lower value indicates greater adherence to incremental development. The average MID of the live-coding group was lower than the static-code group, and the effect size minimally favors the live-coding group. However, the high p-value suggests that the difference is not statistically significant. In fact, when we compared the MID values across *each* assignment between PA3 and PA8, none of the comparisons were statistically significant after we applied the Holm-Bonferroni [12] correction to our α values.

5.1.2 *Debugging.* We conducted two-sample t-tests on the Repeated Error Density (RED) values across 5 different error types: Type Errors, Name Errors, Syntax Errors, Value Errors, Index Errors, and Key Errors (only found in PA7 and PA8). In this table, a *lower* score indicates a *lower frequency of that error* occurring throughout a student's development process for that PA, which generally indicates better debugging skills. Table 2 displays the RED values for the three most common types of errors we saw among the PAs by a significant margin: Type Errors, Name Errors, and Syntax Errors. The results show a small effect size in favor of the live-coding group across all three types of errors, though the differences are not statistically significant. Further, the two-sample t-tests across *all error types and all assignments* revealed no statistically significant difference in the RED value between the two groups.

Table 2 also shows the average proportion of snapshots that include a print statement inside of a function. For this metric, a *higher* value indicates *more frequent use of print statements* inside a function. Similar to the results of the MID and RED metrics, we found a small effect of 0.12 in favor of the live-coding group, although the results were not statistically significant.

5.2 RQ2: Student Grades Results

We conducted two-sample t-tests to compare students' grades. We found that the mean scores of the static-code group were slightly higher than those of the live-coding group for assignments, both exams, and overall grades. Indeed, the effect sizes for each item in Table 3 are favorable for the static-code group, though the p-values are too large to identify any statistically significant differences.

Table 3: Comparison of grades between live coding and static code learning groups

			Grade (out of 100)				
Item	**Group**	**N**	**Mean**	**SD**	**t**	**p**	**d**
PAs	Live	115	82.50	21.75	-0.72	0.47	-0.09
	Static	126	84.38	18.58			
Midterm Exam	Live	115	85.05	21.75	-0.70	0.48	-0.09
	Static	126	86.90	18.63			
Final Exam	Live	115	73.19	27.44	-0.87	0.38	-0.11
	Static	126	76.19	25.64			
Overall	Live	115	82.94	20.12	-0.77	0.44	-0.10
	Static	126	84.83	17.80			

Table 4: Comparison of the types of questions asked between the live coding lectures and static code lectures

	Frequency of Label	
Label	**Live Coding**	**Static Coding**
Process	32.9% (n = 61)	33.9% (n = 75)
Result Explanation	18.9% (n = 35)	18.0% (n = 40)
What if	7.5% (n = 14)	13.1% (n = 29)
Conceptual	24.8% (n = 46)	19.0% (n = 42)
Syntax	15.6% (n = 29)	15.8% (n = 35)
Total	100% (n = 185)	100% (n = 221)

5.3 RQ3: Lecture Questions Results

Table 4 displays the frequency of each label in our code book across all 406 questions asked between the two groups. Notably, the live-coding group asked more "Conceptual" questions than the static-code group, but asked fewer "What if" questions. In order to test for an association between the type of code example and the types of questions asked, we conducted a chi-squared test [20]. Our test returned a chi-square statistic of 4.60, which has a p-value of 0.33. Therefore, with an α value of 0.05, there was no relationship detected between the types of questions asked by either group.

6 DISCUSSION

6.1 Findings

Our results are unable to confirm the perception among instructors and students that live-coding examples improve students' programming processes [3, 4, 16, 19] due to the lack of significant differences in the programming metrics between the two groups. One interpretation of the minimal effect we detected is that the style of code examples in lecture does not have a significant impact on how students program. Within any given week of the CS1 course at our large, public university, students attended two lectures, one lab section, one optional discussion section, and optional tutor hours. They also read one chapter from their interactive, online textbook [28], completed all the shorter-form programming activities in the textbook, and wrote two to three longer functions in their programming assignment. Among all these weekly activities, the code examples that are displayed in class make up only a fraction of lectures, which themselves are only a fraction of the time spent on learning material related to the class. Though the p-values were ultimately insignificant, it is noteworthy that in all five metrics in Table 2, the direction of the effect is in favor of the live-coding group, though the effect size is minimal.

Our results on the impact of live coding on students' grades also revealed no significant differences, though the effect size for all four items in Table 3 were minimally in favor of the static-code group. The lack of a significant difference across exam or assignment scores aligns with Rubin's study [25]. On our exams, students had to demonstrate declarative knowledge, such as code tracing and identifying correct syntax, more so than procedural knowledge, such as how to develop code. Since the lecture content was exactly the same for the experimental and treatment groups, we suspect that students in both groups acquired similar declarative knowledge. This reasoning also holds true for assignment scores—both groups had access to the same code snippets regardless of whether the snippets were live coded or presented as static code. Since assignment grades were given based on the correctness of the final state of the students' submitted code, they only needed to demonstrate a correct implementation of the logic, which both groups could obtain from revisiting the lecture slides.

Our analysis regarding the types of questions asked between the two lecture groups also detected no significant relationship between the type of questions asked and the lecture group that students were in. One notable takeaway, however, is that the static-code group asked more questions across the entire quarter. Unfortunately, the interpretation for such a finding is ambiguous since it could either be the case that students were more engaged in the static-code lectures, so they wanted to ask more questions, or it may be that students were more confused in the static code lectures, so they needed to ask more questions. Similarly, we note that there was nearly double the rate of "What if" questions asked by the static-code group. Although the results were not statistically significant, we believe this finding may lend some evidence to the advantages of the dynamic aspect of live coding in showing students hypothetical changes to the code, thereby reducing "What if" questions.

6.2 Threats to Validity

The largest threat to validity occurred in the collection of our programming process data. Specifically, the process data was noisy in two ways. First, students were allowed to use pair programming while completing their assignments, even though both students would have to submit the code separately on EdStem. Although students could only work with a partner within their lecture section, which meant there was limited contamination between the two groups, they still could work together on one computer and copy-paste the finished code onto the other partner's computer. Second, students could get help from teaching assistants on the assignments, which means that the development patterns we observe may also be a result of assistance from teaching assistants on how to design and approach a solution. Both of these potential confounds threaten the reliability of our data.

Another major threat to validity in our experiment is that we taught students about incremental development and print statement debugging in a lecture during Week 5 of the course. In this lecture, the instructor explicitly showed students in both groups an example of incremental development and how to use print statements to validate a program's logic. As a result, it is hard to determine whether the programming processes we observed are a result of the implicit skills they may have picked up during the lecture examples or of the explicit instruction about these skills in the Week 5 lecture.

6.3 Limitations

A significant limitation of our study is that both the static-code and live-coding pedagogy were administered remotely. Since nearly every student in the lectures had their cameras turned off, we do not know to what degree students were actually engaged during the lecture and paying attention during the code examples. Therefore, although many universities are using blended or hybrid learning models after the COVID-19 pandemic [1], the results of a study that implements a live-coding pedagogy in a traditional classroom setting may shed further light on the true impact of live coding.

A second major limitation is that we administered live coding in a CS1 course, so our findings may not extend to more advanced courses. A large majority of our students did not have any prior coding experience before taking the course. It may be the case that first-time programmers do not pick up meaningful process-oriented skills at this stage in their learning. Therefore, we urge a replication of this work in contexts outside of a remote, CS1 course.

7 CONCLUSION

In our quasi experiment that compares a remote, live-coding pedagogy to a remote, static-code pedagogy, we ultimately found no statistically significant differences between the control and treatment group on programming processes, grades, or lecture questions asked. One explanation for our lack of statistically significant results is that students' programming processes on assignments and students' grades on exams are minimally impacted by the style of code examples in lecture. Despite the lack of statistical significance, the sizes and directions of the effect sizes indicate that the live-coding group exhibited slightly better adherence to programming processes, but the static-code group earned slightly better scores on assignments and exams. Given that live coding is hailed as a best practice to teach programming, future work should continue to investigate the empirical impacts of the pedagogy so that instructors may know in which contexts, courses, and modalities to use a live-coding pedagogy.

ACKNOWLEDGMENTS

This work was supported in part by NSF award 2044473.

REFERENCES

[1] Amreen Bashir, Shahreen Bashir, Karan Rana, Peter Lambert, and Ann Vernallis. 2021. Post-COVID-19 Adaptations; the Shifts Towards Online Learning, Hybrid Course Delivery and the Implications for Biosciences Courses in the Higher Education Setting. *Frontiers in Education* 6 (2021), 1–13. https://doi.org/10.3389/feduc.2021.711619

[2] Brett A. Becker. 2016. A New Metric to Quantify Repeated Compiler Errors for Novice Programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (Arequipa, Peru) *(ITiCSE '16)*. Association for Computing Machinery, New York, NY, USA, 296–301. https://doi.org/10.1145/2899415.2899463

[3] Jens Bennedsen and Michael E. Caspersen. 2005. Revealing the Programming Process. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (St. Louis, Missouri, USA) *(SIGCSE '05)*. Association for Computing Machinery, New York, NY, USA, 186–190. https://doi.org/10.1145/1047344.1047413

[4] Naomi R. Boyer, Sara Langevin, and Alessio Gaspar. 2008. Self Direction & Constructivism in Programming Education. In *Proceedings of the 9th ACM SIGITE Conference on Information Technology Education* (Cincinnati, OH, USA) *(SIGITE '08)*. Association for Computing Machinery, New York, NY, USA, 89–94. https://doi.org/10.1145/1414558.1414585

[5] Neil C. C. Brown and Greg Wilson. 2018. Ten quick tips for teaching programming. *PLOS Computational Biology* 14, 4 (04 2018), 1–8. https://doi.org/10.1371/journal.pcbi.1006023

[6] Russel E. Bruhn and Philip J. Burton. 2003. An Approach to Teaching Java Using Computers. *SIGCSE Bull.* 35, 4 (Dec 2003), 94–99. https://doi.org/10.1145/960492.960537

[7] Adam S. Carter, Christopher D. Hundhausen, and Olusola Adesope. 2015. The Normalized Programming State Model: Predicting Student Performance in Computing Courses Based on Programming Behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15)*. Association for Computing Machinery, New York, NY, USA, 141–150. https://doi.org/10.1145/2787622.2787710

[8] Charis Charitsis, Chris Piech, and John C. Mitchell. 2022. Using NLP to Quantify Program Decomposition in CS1. In *Proceedings of the Ninth ACM Conference on Learning @ Scale* (New York City, NY, USA) *(L@S '22)*. Association for Computing Machinery, New York, NY, USA, 113–120. https://doi.org/10.1145/3491140.3528272

[9] Edstem. 2023. *Edstem*. https://edstem.org/

[10] David C. Funder and Daniel J. Ozer. 2019. Evaluating Effect Size in Psychological Research: Sense and Nonsense. *Advances in Methods and Practices in Psychological Science* 2, 2 (2019), 156–168. https://doi.org/10.1177/2515245919847202

[11] Gunnar Harboe, Jonas Minke, Ioana Ilea, and Elaine M. Huang. 2012. Computer Support for Collaborative Data Analysis: Augmenting Paper Affinity Diagrams. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work* (Seattle, Washington, USA) *(CSCW '12)*. Association for Computing Machinery, New York, NY, USA, 1179–1182. https://doi.org/10.1145/2145204.2145379

[12] Winston Haynes. 2013. *Holm's Method*. Springer New York, New York, NY, 902–902. https://doi.org/10.1007/978-1-4419-9863-7_1214

[13] Matthew C. Jadud. 2006. Methods and Tools for Exploring Novice Compilation Behaviour. In *Proceedings of the Second International Workshop on Computing Education Research* (Canterbury, United Kingdom) *(ICER '06)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/1151588.1151600

[14] Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. 2017. Quantifying Incremental Development Practices and Their Relationship to Procrastination. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 191–199. https://doi.org/10.1145/3105726.3106180

[15] Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant. 2019. Assessing Incremental Testing Practices and Their Impact on Project Outcomes. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 407–413. https://doi.org/10.1145/3287324.3287366

[16] Michael Kölling and David J. Barnes. 2004. Enhancing Apprentice-Based Learning of Java. *SIGCSE Bull.* 36, 1 (mar 2004), 286–290. https://doi.org/10.1145/1028174.971403

[17] Saskia le Cessie, Jelle J Goeman, and Olaf M Dekkers. 2020. Who is afraid of non-normal data? Choosing between parametric and non-parametric tests. *European Journal of Endocrinology* 182, 2 (2020), E1 – E3. https://doi.org/10.1530/EJE-19-0922

[18] Nachiappan Nagappan, Laurie Williams, Miriam Ferzli, Eric Wiebe, Kai Yang, Carol Miller, and Suzanne Balik. 2003. Improving the CS1 Experience with Pair Programming. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (Reno, Navada, USA) *(SIGCSE '03)*. Association for Computing Machinery, New York, NY, USA, 359–362. https://doi.org/10.1145/611892.612006

[19] John Paxton. 2002. Live Programming as a Lecture Technique. *J. Comput. Sci. Coll.* 18, 2 (dec 2002), 51–56.

[20] Karl Pearson. 1900. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50, 302 (July 1900), 157–175. https://doi.org/10.1080/14786440009463897

[21] Harry O. Posten. 1984. *Robustness of the Two-Sample T-Test*. Springer Netherlands, Dordrecht, 92–99. https://doi.org/10.1007/978-94-009-6528-7_23

[22] Adalbert Gerald Soosai Raj, Pan Gu, Eda Zhang, Arokia Xavier Annie R, Jim Williams, Richard Halverson, and Jignesh M. Patel. 2020. Live-Coding vs Static Code Examples: Which is Better with Respect to Student Learning and Cognitive Load?. In *Proceedings of the Twenty-Second Australasian Computing Education Conference* (Melbourne, VIC, Australia) *(ACE'20)*. Association for Computing Machinery, New York, NY, USA, 152–159. https://doi.org/10.1145/3373165.3373182

[23] Adalbert Gerald Soosai Raj, Jignesh M. Patel, Richard Halverson, and Erica Rosenfeld Halverson. 2018. Role of Live-Coding in Learning Introductory Programming. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '18)*. Association for Computing Machinery, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/3279720.3279725

[24] Adalbert Gerald Soosai Raj, Hanqi Zhang, Viren Abhyankar, Saswati Mukerjee, Eda Zhang, Jim Williams, Richard Halverson, and Jignesh M. Patel. 2019. Impact of Bilingual CS Education on Student Learning and Engagement in a Data Structures Course. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '19)*. Association for Computing Machinery, New York, NY, USA, Article 24, 10 pages. https://doi.org/10.1145/3364510.3364518

[25] Marc J. Rubin. 2013. The Effectiveness of Live-Coding to Teach Introductory Programming. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 651–656. https://doi.org/10.1145/2445196.2445388

[26] Ana Selvaraj, Eda Zhang, Leo Porter, and Adalbert Gerald Soosai Raj. 2021. Live Coding: A Review of the Literature. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 164–170. https://doi.org/10.1145/3430665.3456382

[27] Anshul Shah, Michael Granado, Mrinal Sharma, John Driscoll, Leo Porter, William Griswold, and Adalbert Gerald Soosai Raj. 2023. Understanding and Measuring Incremental Development in CS1. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education* (Toronto, ON, Canada) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569880

[28] Stepik. 2023. *Stepik*. https://stepik.org/course/84164

[29] Sheng-Rong Tan, Yu-Tzu Lin, and Jia-Sin Liou. 2016. Teaching by demonstration: programming instruction by using live-coding videos. In *Proceedings of EdMedia + Innovate Learning 2016*. Association for the Advancement of Computing in Education (AACE), Vancouver, BC, Canada, 1294–1298. https://www.learntechlib.org/p/173121

[30] Maureen M. Villamor. 2020. A Review on Process-oriented Approaches for Analyzing Novice Solutions to Programming Problems. *Research and Practice in Technology Enhanced Learning* 15, 1 (Apr 2020), 8. https://doi.org/10.1186/s41039-020-00130-y

[31] Christopher Watson, Frederick W.B. Li, and Jamie L. Godwin. 2013. Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*. 319–323. https://doi.org/10.1109/ICALT.2013.99