An Empirical Evaluation of Live Coding in CS1

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

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

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

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

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

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

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

ABSTRACT

Background and Context. Live coding is a teaching method in which an instructor dynamically writes code in front of students in an effort to impart skills such as incremental development and debugging. By contrast, traditional, static-code examples typically involve an instructor annotating or explaining components of pre-written code. Despite recommendations to use live coding and a wealth of qualitative analyses that identify perceived learning benefits of it, there are a lack of empirical evaluations to confirm those learning benefits, especially with respect to students' programming processes.

Objectives. Our work aims to provide a holistic, empirical comparison of a live-coding pedagogy with a static-code one. We evaluated the impact of a live-coding pedagogy on three main areas: 1) students' adherence to effective programming processes, 2) their performance on exams and assignments, and 3) their lecture experiences, such as engagement during lecture and perceptions of code examples.

Method. In our treatment-control quasi-experimental setup, one lecture group saw live-coding examples while the other saw only static-code ones. Both lecture groups were taught by the same instructor, were taught the exact same content, and completed the same assignments and exams. We collected compilation-level programming process data, student performance on exam and homework questions, and feedback via a survey and course evaluations.

Findings. Our findings showed no statistically significant differences between the live-coding and static-code groups on programming process metrics related to incremental development, debugging, and productivity. Similarly, there was no difference between the groups on course performance on assignments and exams. Finally, student feedback suggests that more students in the live-coding group reported that lectures were too fast and failed to facilitate note-taking, potentially mitigating the perceived benefits of live coding.

Implications. Live coding alone may not lead to many of the perceived and intended benefits that prior work identifies, but future work may investigate how to realize these benefits while minimizing the drawbacks we identified.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; *Student assessment*.

KEYWORDS

live coding, programming processes, course performance, student perceptions, pedagogical techniques, Cognitive Apprenticeship

ACM Reference Format:
Anshul Shah, Emma Hogan, Vardhan Agarwal, John Driscoll, Leo Porter, William G. Griswold, and Adalbert Gerald Soosai Raj. 2023. An Empirical Evaluation of Live Coding in CS1. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1 (ICER '23 V1), August 07–11, 2023, Chicago, IL, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3568813.3600122

1 INTRODUCTION

A "live-coding" pedagogy typically involves an instructor programming in front of the class [42]. As opposed to an instructor annotating and explaining a pre-written, static piece of code, live coding aims to model heuristic programming strategies (i.e., the approaches and techniques to complete tasks [11]), such as incremental development [4, 25, 42] and debugging techniques [4, 7, 25, 36, 42]. A plethora of qualitative studies reveal that students and instructors *perceive* an improvement to students' programming processes from live-coding examples [4, 5, 25, 36, 42]. However, despite live coding being considered a recommended practice in teaching programming [7] and using a fundamentally different approach than traditional static-code examples, there are two glaring gaps in the literature related to live coding. First, very few experimental studies have compared a live-coding pedagogy to a static-code pedagogy.

ICER '23 V1, August 07–11, 2023, Chicago, IL, USA

ACM ISBN 978-1-4503-9976-0/23/08.
https://doi.org/10.1145/3568813.3600122

Second, little work has evaluated the impact of live coding on students' programming processes, such as incremental development and debugging.

Our goal in this work is to evaluate the impact of a live-coding pedagogy on students' adherence to effective programming processes and on student performance on assignments and exams. In an effort to understand the students' experience during the lectures, we also analyze student feedback on the code examples and survey responses about engagement in lecture. Specifically, we ask the following research questions:

(1) How do adherence to effective programming processes and programming productivity differ between students in live-coding and static-code pedagogies?
(2) How does course performance on exams and assignments, specifically on code tracing, code explaining, and code writing questions, differ between students in live-coding and static-code pedagogies?
(3) How does the lecture experience, in terms of engagement during lecture and perceptions of code examples, differ between students in live-coding and static-code pedagogies?

Despite a wealth of qualitative studies that identify perceived and intended benefits of live coding, our results suggest that perception may not match reality. Overall, our results generally showed no significant differences with respect to student adherence to effective programming processes and student performance on exams and assignments. However, students in the live-coding group reported *seeing* various programming processes (step-by-step development, debugging, and testing) in their lectures at a higher rate than the static-code group. Therefore, our inconclusive results may suggest that while live coding demonstrates effective programming strategies to students, it alone may not be enough to actually *impart* these skills to students. We also identify the main drawbacks of live coding that may help guide instructors that use it. More students in the live-coding group indicated that the lectures were too fast, did not hold their attention, and did not facilitate note-taking, indicating that live coding may not be optimal for every student. We conclude with a call for future work to explore techniques that help realize the perceived benefits of live coding while addressing its drawbacks.

2 THEORETICAL FRAMEWORK

2.1 Cognitive Apprenticeship: Modeling

Apprenticeship is a method of teaching that has been used for thousands of years in fields ranging from art to medicine to law [11]. Broadly, apprenticeship refers to the process of transferring knowledge from one person to another where the instructor shows the apprentice what to do and how to do it [11]. Whereas *traditional apprenticeship* typically involves one-on-one instruction and can impart knowledge of how to do tasks that can be learned through observation [19], the theory of *Cognitive Apprenticeship*, introduced by Collins et al. in 1989, involves the instructor making their *thinking* visible to students on reasoning-based tasks [11, 12, 14]. Collins et al. say that via Cognitive Apprenticeship, the expert should impart "domain knowledge" (specific concepts, facts, and procedures on the subject matter), "heuristic strategies" (techniques and approaches for accomplishing tasks), "control strategies" (metacognitive approaches for controlling the process of solving problems and carrying out tasks), and "learning strategies" (knowledge for learning domain knowledge, heuristic strategies, and control strategies) to learners [11]. In a programming context, *heuristic strategies* involve the processes and techniques to create a program, such as writing code, debugging errors, and testing for correctness.

Cognitive Apprenticeship encompasses six distinct teaching methods: modeling, coaching, scaffolding, articulation, reflection, and exploration [11]. Specifically, in the modeling method of Cognitive Apprenticeship, the instructor verbalizes their thought processes and explains the purpose behind specific choices while problem-solving [11]. For example, Collins et al. describe a modeling approach in which an instructor solves math problems in front of students during class to impart heuristic and control strategies [11]. Thus, modeling also draws upon key aspects of Albert Bandura's *observational learning*—the theory that humans learn from each other using observation, imitation, and modeling [2]. A learner taught with an observational learning model might first observe an instructor conducting some behavior and then later imitate the behavior if the learner associates a positive outcome with the behavior [2, 48].

Modeling is the most commonly mentioned theory in studies related to live coding because live coding typically involves an instructor doing a programming task in front of the students while explaining their problem-solving approach [42]. Instructors may also intentionally introduce bugs into the code and show how to debug them [4, 25, 42] or develop a program in small steps to illustrate the process of incremental development [4, 36] (i.e., *heuristic strategies* of programming). Therefore, when contextualizing our results, readers should note that our work is an experimental evaluation of a specific *modeling* method—live coding—on student learning. Though live coding itself has not been confirmed to empirically improve student learning, other Cognitive Apprenticeship methods have shown success in disciplines such as writing [16], mathematics [23], and even programming [15, 51].

2.2 Comprehension vs. Generation

In 2003, Robins et al. conducted a literature review of major trends in teaching and learning programming for novices [39]. The authors highlight the difference between program comprehension skills (demonstrating understanding on a pre-written snippet of code) and program generation skills (demonstrating ability to write code) among novices, noting that studies pertaining to students' comprehension skills are far more common in the computing education literature [39].

Brooks proposed a theory of program comprehension that frames program comprehension as a "top-down" approach in which novices examine an entire block of code and use their high-level domain knowledge to generate a hypothesis about the meaning of the code [6, 39]. With this high-level hypothesis, students then search for specific cues or markers in the code to confirm or reject their hypothesis [6]. Brooks posits that students vary in their comprehension strategies and programming knowledge, which in turn impacts their ability to understand a piece of code [6]. The model presented by Brooks has also been confirmed in many works that

show how differences in domain knowledge, task requirements, and comprehension strategies impact novices' program comprehension skills [39].

On the other hand, Rist presents a model of program generation for novices that represents the generative process as one in which students are continuously searching for the next *action*, or code snippet, to add based on the next subgoal in the development process [38]. Rist mentions that a novice's development pattern is typically composed of many such actions, requiring a student to constantly generate ideas for next steps in the programming process. This forward-looking process contrasts with the program comprehension model posed by Brooks, since students do not have a pre-written chunk of code as evidence to confirm their beliefs or instincts. Instead, students work from a "blank-slate" and have to build a program by retrieving and implementing various blocks of code [38].

Live-coding and static-code examples represent the fundamental contrast between generation and comprehension. In a live-coding pedagogy, an instructor aims to share their reasoning for adding certain pieces of code and verbalizing components of the generation process described by Rist. On the other hand, static-code examples that display a pre-written block of code to student embody the program comprehension model by Brooks. This fundamental difference in approach between live-coding and static-code examples may potentially lead to differences in student performance on generative tasks and comprehension tasks. For example, students taught with live coding that observe the step-by-step process of the instructor may engage in better incremental development practices and students taught with static-code examples that break down pre-written code may perform better on code tracing questions.

3 RELATED WORK

3.1 Prior Work to Evaluate Live Coding

A recent literature review of live coding revealed that the pedagogical approach has been studied for two decades, including several quasi-experimental studies and experience reports [42]. The majority of prior live-coding studies have focused on student learning measured by grades or on students' perceptions and experiences in live-coding lectures. However, these studies have not evaluated the impact of live coding on students' programming processes, which is a key learning goal of live coding [4, 7, 25]. These studies have demonstrated that live coding *seems* to benefit students' ability to follow effective programming processes, such as incremental development [36, 42], debugging [4, 5, 25, 33, 36], and testing [25, 36, 42] as determined by feedback, interviews, or surveys. By contrast, fewer studies have empirically compared the effects of live-coding to static-code examples on student learning.

One of the first empirical, comparative studies of live-coding and static-code examples was conducted by Rubin in an introductory computer science class [40]. The results showed that students in both groups had similar performance on assignments and exams. However, the live-coding group scored significantly higher on the large programming project, graded on correctness and code clarity. Notably, Rubin hypothesized that live coding may improve students' programming processes such as debugging and testing [40], but did not assess this in the study. Second, following Rubin's experimental design, Soosai Raj et al. conducted a study on 180 students in an advanced data structures course in India [35]. A key result from this study was that students in the live-coding group experienced *less* extraneous cognitive load during lectures compared to the static-code group [35]. Nonetheless, the study did not uncover any significant differences between the two groups with respect to student learning measured using a pre-test and post-test.

More recently, we utilized a similar experimental design to Rubin's [40], but conducted our analysis in a *remote* course taught in Python [43]. Perhaps the most similar work to the goals of the present work, our previous study compared students' programming processes, measured by applying a set of process-oriented metrics to the compilation-level histories of students' programming assignments [43]. The study found no differences between students' programming processes or overall performance on exams, yet suffered from several key threats to validity. First, the remote setting made it difficult to gauge attendance, so we are unsure how many students even saw the code examples. Second, the course exams were administered remotely in a non-proctored setting, enabling collaboration or cheating. Third, we leveraged programming process data from programming assignments on which students could engage in pair programming [31] and could receive help from course tutors or online sources [43].

In view of the above discussion, one of the key contributions of our work is to use reliable and representative data to empirically evaluate the impact of live coding on students' adherence to effective programming processes. We also extend the programming process analysis we conducted in earlier work [43] by comparing students' productivity on programming tasks in terms of time on task and correctness. Aside from programming process analyses, we also aim to provide a more holistic comparison of the two lecture styles on students' course performance and lecture experience than previous literature. The few comparative studies on the effects of live coding have reported coarse measures of performance, such as overall outcomes on assignments and exams [35, 40, 42, 43, 45, 49]. Given our theory-driven hypothesis (see Section 2) that the two lecture styles may impart different skills to students, there is a need for a deeper comparison between live and static examples on various types of coding questions (tracing, explaining, writing, etc). Last, our work builds upon the wealth of qualitative analyses of live coding by utilizing a combination of open-feedback and course evaluation data to compare students' perceived benefits *and drawbacks* between the two groups.

3.2 Programming Process Metrics

Relying on traditional assessments such as exams or the correctness of a final state of a program may not capture the differences in students' ability to use effective practices *during* the development process [26, 52]. Therefore, a key contribution of our study is an empirical investigation into the differences between students' *programming processes* in the two learning groups. In this section, we describe process-oriented metrics related to incremental development and debugging, since these have been cited as both learning goals [4, 25] and perceived benefits [36, 40, 42] of live coding.

3.2.1 Incremental Development. There are two metrics we found that aim to measure incremental development and program decomposition. The Measure of Incremental Development (MID), which we introduced in a previous work [44], aims to measure how well a student adhered to a process of adding manageable code chunks without experiencing excessive struggle after each addition of code. The metric was created to be agnostic to program size and aligns with instructor judgments of a student's adherence to incremental development at a rate of 80-85% [44]. Although this metric has not been evaluated on courses outside of CS1, the present study regards CS1 and analyzes a set of programming tasks similar in length and difficulty to those on which the metric was developed [44]. An adjacent metric related to program decomposition was developed by Charitsis et al., who leveraged natural language processing to gain insights into how a student decomposed the task into smaller chunks [9]. The model makes use of naming conventions of functions and variables to capture the components that a student adds to their program over time [9]. This metric is better suited to longer-form programming tasks that have many functions, since the model considers decomposition at the *function* level [9].

3.2.2 Debugging and Error Frequencies. Two outcomes of Jadud's early work to analyze student compilation behaviors were 1) the development of a metric called the Error Quotient (EQ), which characterizes how much a student struggles with a syntax error while programming [22], and 2) a deeper investigation into process data to capture various aspects of student debugging and error-handling behaviors [52]. Multiple works expanded upon the EQ and introduced ways to improve the predictive accuracy of the metric on course performance, such as the Watwin Score (WS) [53], Normalized Programming State Model (NPSM) [8], and the Repeated Error Density (RED) [3]. The WS uses the amount of time that a student took to fix an error to reward students for fixing errors more quickly [52, 53]. Similarly, the NPSM uses syntactic *and* semantic correctness, determined by the presence of a runtime exception [8, 52]. Finally, the Repeated Error Density (RED) measures the amount that a student struggled with a specific type of error; the RED achieved a higher predictive accuracy than Jadud's EQ on students' performance on programming tasks since it penalizes students more for encountering repeated errors of the same type [3, 52]. A score of 0 on the RED for a certain error means that either 1) the student never experienced that type of error or 2) the student resolved the error of that type after one compilation [3]. These metrics require various input data, such as the WS requiring time to resolve an error, the NPSM requiring the semantic correctness of a program state, and the RED requiring the type of error in the error message. Previous work found that the RED is less context-dependent than the EQ and NPSM and does not require time spent working on the IDE (which the WS metric requires) [52].

4 EXPERIMENTAL DESIGN

4.1 Course Setup

We ran a quasi-controlled experiment on a CS1 course at UC San Diego—a large, public, research-intensive university—in the Fall 2022 term. Our work was approved by the UCSD Institutional Review Board, Project #201792. Because of the size of the course, the 538 students in the course self-selected into three lecture sections, offered at various times in the day on Tuesdays and Thursdays. Two of the three lecture sections were taught by the same instructor, providing our research team with an opportunity to evaluate the effect of live coding in a treatment-control setup that controls for the instructor. At the time of registration, the students did not know one lecture would be taught with live-coding examples and one would be taught with static-code examples.

Each week, students attended two lectures, a lab, and an optional discussion section. They completed weekly programming assignments (PAs), worksheets, and reading quizzes. Table 1 contains the frequency and description of each key course component. The course was taught in Python and covers basic syntax, data types, conditionals, for loops, while loops, objects in memory, function calls, and image transformation (where images are represented as a 2-D list of RGB coordinates).

4.2 Treatment Condition

We chose a treatment-control design between the two lecture sections that were taught by the same instructor, who has taught CS1 extensively in the past using both static-code and live-coding examples. Both lecture sections covered the same material each week, included the same participation activities, and were recorded for students of that lecture section to watch later. Further, during both types of code examples, students watched the instructor explain the material and were free to take notes or follow along if they wanted. The only difference between the two lectures was the time of day (one was at 9:30AM and the other was at 11AM) and the presentation of the code examples.

In the earlier lecture section—the static-code group—the instructor showed students pre-written examples of code when demonstrating an implementation of a concept. When presenting these static-code examples, the instructor wrote annotations on the slides with static-code to help explain the static code snippet. For example, the instructor drew memory diagrams, underlined important lines of code, and listed the iterations of a loop. Though the specific annotation varied from example to example, the goal of the annotations was to enhance students' understanding of the code example. Figure 1(a) displays a static-code example and the instructor's annotations during a lecture about modifying lists within a function. In this example, the instructor helped students visualize the list object in memory and also drew a table to track the iterating variable `i` and the value of `nums[i]`.

In the other lecture section—the live-coding group—the instructor showed code examples by opening a Python file in an IDE and writing code in front of the students. As best as they could, the instructor aimed to follow the recommendations provided by Brown and Wilson [7]. Specifically, the instructor sometimes started with boilerplate code rather than a blank file to avoid having to type in code that was needed for the program but unrelated to the topic in the example [7]. While live coding, the instructor verbalized their thought process and explained their rationale for including certain lines of code. The instructor also exhibited effective development strategies, such as adding code in small, manageable chunks or using print statements to track variable updates. Similarly, the instructor sometimes deliberately ran into an error and demonstrated

Table 1: Key course components of the CS1 course

Component	Frequency	Description
Lectures	Twice per week	Students attend two 80-minute lectures and complete an active learning activity for participation credit (such as code challenges) and then discuss with their peers.
Programming Assignments	Once per week	Students individually complete 6 assignments which ask students to complete two independent programming tasks that cover the lecture material of the week.
Worksheets	Once per week	Students individually complete code tracing, explanation, writing questions in a one-week long assignment that covers the lecture material of the week.
Labs	Once per week	Students work in pairs in a 50-minute, in-person session to complete short, scaffolded programming tasks. Students receive credit for completing the tasks before the end of the week.
Reading Quizzes	Once per week	Students individually complete interactive activities in an online textbook [47]. Students can submit responses unlimited times until they are correct.
Midterm Exam	Once in the term	Students individually complete a proctored, in-person exam for 2 hours that covered the concepts taught in the first half of the term.
Final Exam	Once in the term	Students individually complete a proctored, in-person final exam for 3 hours that covered all concepts taught in course.
Discussion	Once per week (optional)	Students can attend a 50-minute, in-person session to review the lecture material for the week and preview the upcoming programming assignment.
Office Hours	Every day (optional)	Students can attend open hours hosted by course staff (TAs, tutors, etc.) for help on course material. These were roughly held M-F from 9AM to 7PM.

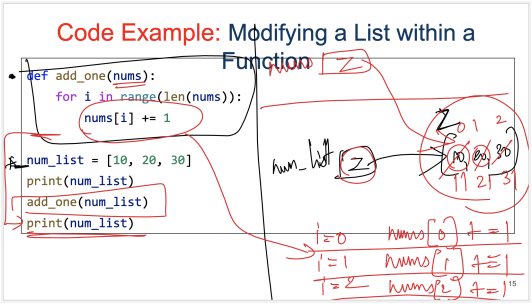
debugging techniques by interpreting the error message, adding a print statement, and fixing the bug. Figure 1(b) displays the end-state of a live-coding example where the instructor executed the Python file and added print statements to track the values of `i` and `nums[i]`.

We also tracked students' lecture attendance to monitor exposure to the treatment condition. Figure 2 displays the high rate of attendance for the 14 required lectures. Notably, we had to halt the course for the last two weeks of the term prior to finals due to an academic strike by teaching assistants. During these two weeks, students did not attend lectures, labs, or discussion sections and there were no programming assignments or worksheets students had to complete. Students were told to review the content until the current point in the course. Fortunately, at this point in the term, the majority of data had been collected. Following the stall, students completed only the final coding challenge, the final exam, and course evaluations. Although we aimed to control for every factor besides the lecture code examples, we certainly acknowledge that there are confounding effects and limitations to our design (see Section 7.3).

4.3 Participants

Per our approved human subjects protocol, students consented to releasing their course data at the start of the term. In total, 115 students (63.1%) of the static-code group and 110 students (62.5%) of the live-coding group consented to our study. We administered a survey at the start of the term to understand our participants' prior experience, demographics, and confidence levels for the course. Table 2 summarizes the participants' information in each group. The populations had nearly identical rates of programming experiences and had similar distributions across the five school years. We also saw no glaring differences between the gender or racial

Figure 1: Static-code vs live-coding example



(a) Static-code example

modifyList.py

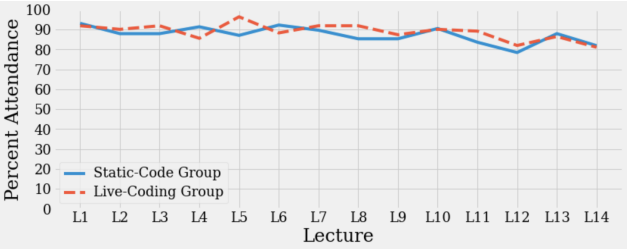
```
1 def add_one(nums):
2     for i in range(len(nums)):
3         nums[i] += 1
4         print(i, nums[i])
5
6 lst = [10, 20, 30]
7 print("Initial list", lst)
8 add_one(lst)
9 print("Final list", lst)
```

user@sahara:~

```
[user@sahara ~]$ python modifyList.py
Initial list [10, 20, 30]
0 11
1 21
2 31
Final list [11, 21, 31]
```

(b) Live-coding example

Figure 2: Lecture attendance throughout the term



makeup of our population groups. In the live-coding group, 52% of the students identified with he/him/his pronouns and 47% identified with she/her/hers pronouns. In the static-code group, the split between he/him/his and she/her/hers pronouns was 55%/44%. Similarly, the self-identified racial makeup of both groups consisted of roughly 50% Asian or Asian-American students, roughly 20% Latinx or Chicanx students, and 10-15% White students. The remaining roughly 15% of students self-identified in racial groups of less than 10 students. Per our human subjects protocol, we do not disclose these groups.

We noticed that the students in the live-coding lecture were slightly more confident in their ability to succeed and wanted a slightly higher grade in the course, on average. Therefore, we compared the average GPAs of the two groups as an additional check for incoming differences between our two groups. A two-sample t-test [34] of the incoming, unweighted GPAs of the live-coding and static-code groups revealed no significant difference between the groups (the live-coding and static-code groups had an average GPA of 3.85 and 3.87, respectively). Due to the relative similarity in prior programming experience, university standing, racial and gender makeup, and incoming GPA, we were ultimately not worried about the slight difference in the students confidence ratings (which tend to be inflated, especially in lower-division classes [32]) or desired final course grade.

5 METHODS

Table 3 summarizes our research questions, data collection, and statistical analyses as a guide to understanding our methods.

5.1 RQ1: Programming Processes

We had access to snapshots of students' code each time they compiled their code by clicking "Run" or "Submit" on our course's online IDE (EdStem) [17]. To gather clean, representative student process data, we conducted a series of coding challenges[1] in which students completed programming tasks within a set timeframe in a proctored environment so that they did not work with partners, get help from tutors, or copy answers from online. We held seven "short" challenges during lectures in which students had 10 minutes to implement one to two functions, one "medium" challenge in which students had 15 minutes to implement a modified "Rainfall Problem" [46], and one "long" challenge in which students had 40 minutes to implement two functions related to image processing. These questions were created by members of the research team to enable students to engage in a programming process. Specifically, three of the seven short challenges included multiple functions, the medium-length challenge—the modified "Rainfall Problem"—included one function that had multiple requirements, and the final, 40-minute coding challenge included two functions that each required students to write nested for-loops. One of the short challenges was a debugging activity, in which students were given an already-written function that had a semantic error (i.e., the code ran without errors, but it outputted an incorrect value from what was required). Students had 10 minutes to find and fix the error. For the seven short coding challenges and the one medium length coding challenge, students were not graded on correctness but rather on participation. The final coding challenge, on the other hand, was part of students' course grade.

Because prior work mentioned a general improvement in the programming process as a perceived benefit of live coding [36, 42], with specific references to incremental development [4, 36] and debugging [25, 33], we focused our analysis on incremental development, debugging, and general programmer productivity. Per our discussion in Section 3.2.1, we applied the Measure of Incremental Development (MID) to our data since it has been tested on Python programming tasks with one to three functions [44]. We separated our analysis between challenges that required only one function and those that required two functions since the program decomposition in one-function tasks may be different from the decomposition in longer tasks that are already decomposed into multiple functions.

For debugging and error-frequency measures, we applied the Repeated Error Density (RED) for TypeErrors, NameErrors, SyntaxErrors, and IndexErrors because of the required input data and its improvement upon previous metrics (see Section 3.2.2). However, the RED alone does not shed light on students' actual debugging techniques. Since using print statements is one of the most common debugging techniques among intermediate programmers [29] and the instructor regularly used print statements during live-coding lectures to show students the output of the code, we also report the proportion of unique print statements added per snapshot to determine if students imitated the instructor's debugging technique.

Finally, we gathered metrics related to programmer productivity, such as the rate of correctness, time until reaching a correct implementation, and the number of compilations used in the development session. We chose these measures in line with software engineering research on *developer productivity*, which broadly is a comparison between the input (time spent working, amount of code) and output (clean and correct code) by a programmer [21]. We focused our productivity analysis on the 40-minute, final coding challenge as it was the only challenge that counted towards students' grade and was the summative programming task for the course. We also replicated this analysis on the debugging challenge, but we note that students were awarded credit for the debugging challenge based on participation and not for correctness. For these coding challenges, we noted down the time that students started working on the task when we administered the coding challenges. We then calculated the time until completion by finding the difference between the time that the students started and the first timestamp in which a student produced the correct solutions (i.e., passed all the test cases).

[1] https://bit.ly/allCodingChallenges

Table 2: Comparison of treatment and control populations

	Prior Programming Experience				
	Yes			No	
Live-coding	48.1%			51.9%	
Static-code	48.7%			51.3%	
	School Year				
	First	Second	Third	Fourth	Fifth
Live-coding	47.7%	23.9%	17.5%	9.2%	1.8%
Static-code	52.6%	21.9%	13.2%	8.8%	3.5%
	Confidence rating in ability to do well in course				
	1	2	3	4	5
Live-coding	1.8%	5.5%	44.5%	37.3%	10.9%
Static-code	4.3%	11.3%	36.5%	38.3%	9.6%
	Minimum final course grade that students would be satisfied with				
	A-range		B-range	C-range	
Live-coding	61.7%		32.2%	6.1%	
Static-code	50.9%		46.4%	2.7%	

Table 3: Summary of data collection and methods

Research Question	Sub-Analysis	Data Source	Metric	Statistical Tests
RQ1: Programming Processes	Incremental Development	All coding challenges	Measure of Incremental Development	Two-sample t-tests
	Error Frequencies and Debugging	All coding challenges	Repeated Error Density for Name, Syntax, and Type Errors; print statements added	Two-sample t-tests
	Programmer Productivity	Final and debugging code challenges	Rate of correctness; time to completion; number of compilations used	Two-sample t-tests/Mann-Whitney U-tests/z-test for proportions
RQ2: Course Performance	Code Tracing, Writing, and Explaining	Worksheets and Exams	Average correctness per question type	Two-sample t-tests
	Overall Performance	PAs, Worksheets, and Exams	Grade out of 100	Two-sample t-tests
RQ3: Lecture Experience	Perceptions of Code Examples (Benefits and Drawbacks)	Open-ended, one-time student feedback	Frequency of themes determined by open-coding	None
	Engagement	End-of-term evaluations	% of students reporting that lectures held their attention	Chi-Square test of trend
			% of students reporting that lectures facilitated note-taking	Chi-Square test of trend

Statistical Analysis Our sample size for these analyses was roughly 110 students per group (some students did not attempt certain code challenges if they were absent from lecture). For the metrics related to incremental development and debugging, we conducted a series of two-sample t-tests [34] to determine the significance and size of the differences between the two groups. T-tests are more powerful than their non-parametric alternative—Mann-Whitney U tests—for equal sample sizes [54] and a sample size of 25 in each group is sufficient for the t-test to perform moderately well even on skewed data [27]. As the sample size increases, the t-test becomes more tolerant of more extreme data distributions and for large sample sizes, the assumption of normality is generally not needed for t-tests [27]. Therefore, we conducted two-sample t-tests despite observing some skew in our data. To compare the rate of correctness on the coding challenges, represented by a proportion, we used a z-test for proportions [41]. For the comparison of the minutes until correct and number of compilations used in reaching the correct solution on the final code challenge, we again applied a t-test because of the larger sample sizes. However, the productivity metrics for the debugging challenge required Mann-Whitney U tests [28] because of the smaller sample sizes of the two groups (since we were comparing only the correct implementations). To

adjust for multiple comparisons, we applied a Holm-Bonferroni correction to hypothesis tests [1] in the same family (i.e., tests related to incremental development, tests related to debugging/error frequencies, etc) when we obtained p-values less than 0.05.

5.2 RQ2: Course Performance

Students' course performance data on programming assignments (PAs), worksheets, and two exams were collected throughout the quarter. Based on the threats to validity in a previous attempt to compare student performance [43], the midterm and final exam were administered in an in-person, proctored, and closed environment. The worksheets and exams asked students to demonstrate code comprehension skills and declarative knowledge of Python. For example, some questions asked students to explain code, some asked students to write a short program, and others asked students to predict code output. To compare student performance across the different types of questions, we grouped worksheet and exam questions into the categories proposed in Venables et al. [50]: Basic Questions, Code Tracing with Loops, Code Tracing without Loops, Code Writing, and Code Explaining. Every question from the worksheets and exams was placed into a single category based on the following descriptions:

- **Code explaining** questions ask the student to explain what a piece of code does or how a piece of code works at a higher level than just asking for the output.
- **Code writing** questions ask the student to either fill in a blank of a pre-written code block or write a short block of code from scratch.
- **Code tracing** questions ask the student to track values of variables or predict the output of a code snippet. The authors distinguished between code tracing with and without loops based on the added difficulty of tracing iterations [50].
- **Basic questions** are questions that don't fall into the above categories, such as, "Do all functions require a return statement?"

The proportions and counts of questions in each question category for each assignment are outlined in Table 4. For each question type, a student's score was calculated by adding the scores on individual questions.

Table 4: Counts of each question type on worksheets and exams

Question Type	Worksheets	Midterm	Final
Basic Questions	3	4	7
Code Writing	4	2	4
Code Tracing (Loops)	5	2	7
Code Tracing (No Loops)	7	7	6
Code Explaining	13	0	0

Statistical Analysis Each comparison made between students performance on question type (basic, code tracing, writing, and explaining questions) and on course component (PAs, worksheets, exams) included roughly 110 students. Therefore, we conducted a series of two-sample t-tests since our learning groups had roughly equal, sufficiently large sample sizes [27, 34, 54].

5.3 RQ3: Lecture Experience

Our data regarding students' self-reported lecture experience comes from two sources: 1) a required survey that we administered to students halfway through the term, and 2) the official, end-of-term course evaluations by students. The required survey accompanied a weekly programming assignment (PA) halfway through the course. Students completed the survey in their own time, which included the following open-ended questions:

- **Benefits:** What are specific things about the *[pre-written]/[live-demo of]* code examples that are helpful for your learning?
- **Drawbacks:** What are some suggestions to improve the code examples so that they may be more beneficial for your learning?

In total, we obtained 97 responses from the static-code group, which is a 84% response rate from consenting students in that group, and 94 responses from the live-coding group, which is 85% of the consenting students in that group. For each question, two members of the research team analyzed the responses using an open-coding ("affinity-diagramming") approach, which involves identifying common themes in the responses and then categorizing the responses based on these themes [20]. The coders followed an identical open-coding process for both open-ended questions.

In the open-coding process, the two coders simultaneously conducted their own analyses of the 30 responses from each lecture group (60 total responses) by constructing a code book with names, descriptions, and examples of the groupings. The coders were instructed to label a single student response with multiple codes if they deemed the response appropriate for more than one theme. After the first set of 60 responses were individually analyzed, the coders met together to compare code books and deliberated until creating a unified code book. Then, using the new unified code book, the coders repeated the process for the next 60 responses (again, 30 for each group), adding themes and descriptions to the code book as they proceeded. After a second round of deliberation, the coders independently labelled a third set of 60 responses and computed the kappa statistic for inter-rater reliability (IRR) [30] at the end of the third round. The IRR between the two raters was 0.70 for the analysis of the perceived benefits and was 0.91 for the analysis on the perceived drawbacks. Even though the two raters never separately rated their own responses (every response was rated by both reviewers), we report the IRR for transparency in the agreement during our open-coding analysis. After creating a final code book after this third round, the two coders sat together to assign a label and resolve any discrepancies for all student responses. Tables 12 and 13 in the Appendix show our final code books after our open-coding approach.

We also leveraged data from the official, end-of-term course evaluations. The evaluations are anonymized when reported to the instructor, so students are encouraged to be truthful when evaluating a class so that the course staff and future students may benefit from the feedback. In total, 107 students from the static-code group and 95 students from the live-coding group provided an evaluation. Though there were other questions about the course and content in general, we report only the lecture-specific items that may be impacted by the style of code examples. Specifically,

we analyze student agreement with following two statements, for which students responded with either Strongly Disagree, Disagree, Neutral, Agree, or Strongly Agree:

- **Attention:** Lectures hold your attention.
- **Note-taking:** Instructor's lecture style facilitates note-taking.

Statistical Analysis We did not conduct any inferential statistics to the results of our open-coding process. However, the response format of the course evaluation items enables a Chi-square test for trend [24]. The Chi-square test of trend is applied to ordered categories (such as a spectrum of agreement) to detect whether there is an association between the condition (live coding) and the outcome (agreement with the statements). A significant result for the test of trend would indicate that there is evidence for a trend towards one "side" of the agreement spectrum based on the treatment condition [24].

6 RESULTS

To interpret our statistical tests, we focus on the significance of the difference represented by the p-value and the direction and magnitude of the effect size. We applied a Holm-Bonferroni correction [1] to the p-values within a specific sub-analysis (incremental development, debugging/testing, worksheet comparisons, etc), though we rarely saw any p-values lower than 0.05. For the two-sample t-tests, our effect sizes are calculated via Cohen's method [10] since our standard deviations were relatively equal. For the Mann-Whitney U-test, our effect size is computed with the rank-biserial correlation, which is typically used for non-parametric tests [13]. We follow the reasoning provided by Funder and Ozer, who point out that Cohen himself eventually regretted proposing the often-used, arbitrary standard for interpreting effect sizes (where 0.1, 0.3, and 0.5 are the thresholds for small, medium, and large effects) [18]. As such, we interpret our effect sizes using the data-driven thresholds proposed by Funder and Ozer: an effect size of 0.05 indicates a *very small effect*, 0.10 is a *small effect*, 0.20 is a *medium effect*, and a 0.30 is a *large effect*. We report the direction of the effect size such that a *positive effect* indicates that the live-coding group had a *more favorable score* (whether that is less error frequency, higher score on exams, a higher rate of correctness, etc).

6.1 Programming Processes

While we replicated the following analyses on students' programming assignment data, we do not report the results because they are more prone to noise [43]. Further, the in-lecture coding challenges were held directly after exposing students to their respective style of lecture code examples. Regardless, we found statistically insignificant differences in the below analyses across programming assignments.

6.1.1 Incremental Development. Table 5 shows the results of the two-sample t-tests between the live-coding and static-code groups on their MID scores for the coding challenges. Across one-function and two-function challenges, there was no statistically significant difference between the two groups.

6.1.2 Debugging and Error Frequency. Table 6 summarizes the two-sample t-tests for the debugging metrics. Although we calculated the Repeated Error Density for more types of errors than are displayed, we only display results for TypeErrors, SyntaxErrors, and NameErrors since these were the three most common error types across the coding challenges. For all the comparisons we made, there were not any statistically significant differences and very small effect sizes. The difference between the groups in terms of the number of print statements added per run was also trivial.

6.1.3 Programmer Productivity. Table 7 displays the results of a z-test of proportions to evaluate differences in rates of correctness on the debugging and the final coding challenges. The rate of correctness in the live-coding group was 11.8% percentage points higher on the debugging challenge with an effect size of 0.299. However, the difference between the groups was not statistically significant.

Table 8 compares the minutes until correct and compilations until correct on the final coding challenge *among students who produced the correct answer*. Though the time until a correct implementation was about 2 minutes lower—an effect size of 0.278—for the live-coding group, on average, we ultimately saw no statistically significant difference based on the p-values.

Table 9 displays the productivity comparisons for the debugging activity, which had a smaller sample size due to the limited number of correct submissions. There was no significant difference in the minutes or compilations until a correct implementation after applying a Holm-Bonferroni correction [1].

6.2 Course Performance

6.2.1 Code Tracing, Writing, and Explaining. Table 10 shows the comparison of student performance on basic, tracing (no loops), tracing (loops), explaining, and code-writing questions over all six worksheets in the course. None of our p-values indicate statistical significance. Similarly, Figure 3 shows the same comparison on exam questions. None of the comparisons showed any significant p-values or effect sizes greater than the threshold for a small effect.

6.2.2 Grades Analysis. Table 11 shows the results of two-sample t-tests between grades on PAs, worksheets, the two exams, and overall course grade. The results suggest that there was no significant difference in academic performance throughout the course. Accordingly, we only see very small or negligible effect sizes.

6.3 Lecture Experience

6.3.1 Perceived Benefits and Suggestions for Improvement. Figure 4 displays the frequency of each label among our open-ended responses related to perceived benefits. The percentages in each column sum to over 100 because some student responses had multiple labels. The percentages in the figure represent the proportion of *responses* that mentioned each label. Note that the labels were determined through an open-coding approach, but the "Categories" column was created after the open-coding process in order to organize the results of our qualitative analysis. A comparison of the relative frequencies of the labels reveals that more responses from the static-code group mentioned a "Code Comprehension" reason than the live-coding group but the live-coding group mentioned benefits related to "Programming Processes" at a higher rate.

Figure 5 shows the same table of frequencies but for the perceived drawbacks. Notably, almost one-fifth (18%) of responses from the

Table 5: Comparison of Measure of Incremental Development (MID) between live-coding and static-code group on coding challenges

Metric	Interpretation	Group	N	Mean	Std dev	t-stat	p	Effect size
MID on Short Challenges (1 function only)	Lower = Better Incr. Dev.	Live	108	1.447	1.374	0.656	0.513	0.088
		Static	114	1.562	1.227			
MID on Longer Challenges (2 functions)	Lower = Better Incr. Dev.	Live	107	1.376	0.986	0.503	0.616	0.068
		Static	111	1.446	1.047			

Table 6: Comparison of error frequencies and debugging metrics between live-coding and static-code group on coding challenges

Metric	Interpretation	Group	N	Mean	Std dev	t-stat	p	Effect size
RED of TypeError	Lower = Less Error Frequency	Live	110	0.145	0.357	0.494	0.622	0.066
		Static	115	0.169	0.382			
RED of SyntaxError	Lower = Less Error Frequency	Live	110	0.177	0.363	-0.021	0.983	-0.003
		Static	115	0.176	0.295			
RED of NameError	Lower = Less Error Frequency	Live	110	0.069	0.143	0.212	0.832	0.028
		Static	115	0.073	0.162			
# of Prints Added per Run	Higher = More Prints Added	Live	109	0.065	0.073	0.048	0.961	0.007
		Static	115	0.064	0.098			

Table 7: Comparison of rate of correctness between live-coding and static-code group on debugging and final coding challenges

Coding Challenge	Group	N	% correct	z-score	p	Effect size
Final Challenge	Live	99	64.6%	1.072	0.284	0.151
	Static	103	57.3%			
Debugging Challenge	Live	81	25.9%	1.859	0.063	0.299
	Static	78	14.1%			

Table 8: Comparison of time and compilations until correct implementation between live-coding and static-code group on final coding challenge

Metric	Interpretation	Group	N	Mean	Std dev	t-stat	p	Effect size
Minutes until correct (final)	Lower = Faster Completion	Live	64	13.715	8.655	1.539	0.126	0.278
		Static	59	16.111	8.379			
Compilations until correct (final)	Lower = Fewer Compilations	Live	64	16.492	13.855	0.521	0.603	0.095
		Static	59	17.745	12.395			

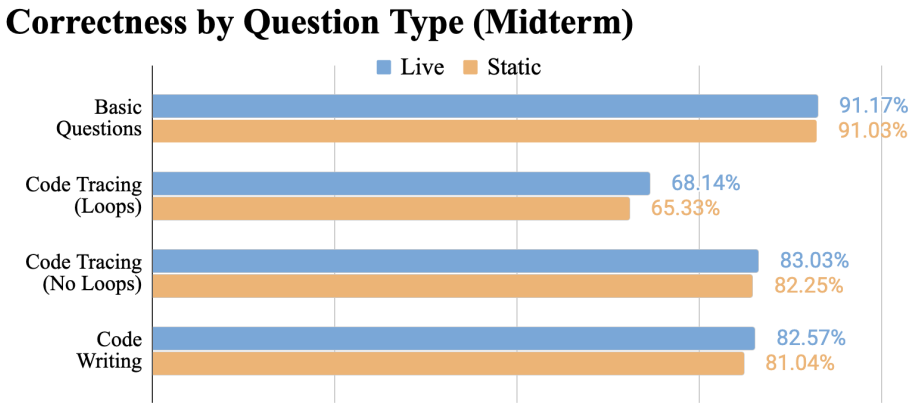
Table 9: Comparison of time and compilations until correct implementation between live-coding and static-code group on debugging challenge

Metric	Interpretation	Group	N	Mean	Median	IQR	U-stat	p	Effect size
Minutes until correct (debugging)	Lower = Faster Completion	Live	21	5.36	5.81	3.73	102	0.606	0.117
		Static	11	6.25	6.31	1.41			
Compilations until correct (debugging)	Lower = Fewer Compilations	Live	21	6.00	4.00	4.00	65	0.046	0.437
		Static	11	8.45	7.00	2.50			

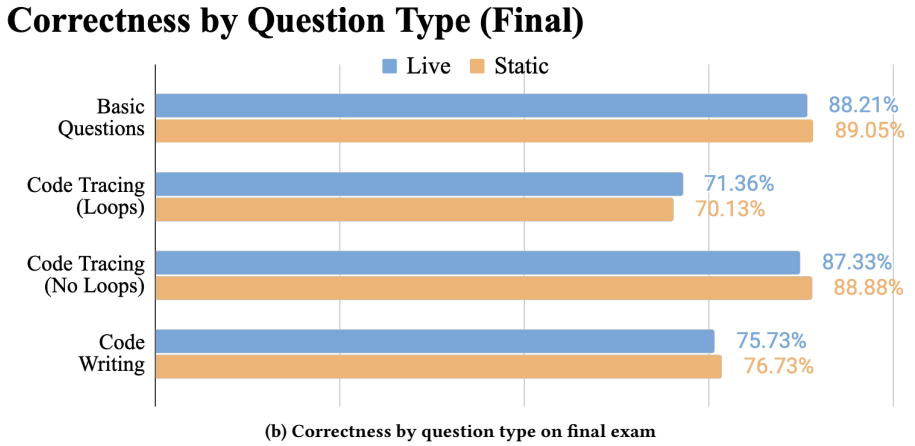
Table 10: Comparison of student performance on basic, code-explaining, code-tracing, and code-writing questions between live-coding and static-code group on worksheets

Question Type	Group	N	Avg % Correct	Std dev	t-stat	p	Effect size
Basic Question	Live	108	0.906	0.147	0.986	0.325	0.132
	Static	115	0.886	0.159			
Code Explaining	Live	108	0.888	0.155	0.196	0.845	0.026
	Static	115	0.884	0.137			
Code Tracing (Loops)	Live	108	0.813	0.215	-0.183	0.855	-0.025
	Static	115	0.819	0.211			
Code Tracing (No Loops)	Live	108	0.877	0.173	0.834	0.405	0.112
	Static	115	0.857	0.169			
Code Writing	Live	108	0.724	0.268	0.260	0.795	0.035
	Static	115	0.715	0.245			

Figure 3: Comparison of student performance on basic, code tracing, and code writing questions between live-coding and static-code groups on exams



(a) Correctness by question type on midterm exam



(b) Correctness by question type on final exam

Table 11: Comparison of overall course performance between live-coding and static-code groups

Item	Group	N	Grade (out of 100) Mean	Std dev	t-stat	p	Effect Size
PAs	Live	110	93.363	12.134	-0.094	0.926	-0.012
	Static	115	93.507	10.865			
Wkshts	Live	110	89.663	15.499	0.514	0.607	0.069
	Static	115	88.664	13.404			
Midterm Exam	Live	110	82.539	16.397	0.429	0.668	0.057
	Static	115	81.632	15.081			
Final Exam	Live	110	78.789	19.993	-0.231	0.817	-0.031
	Static	115	79.385	18.183			
Overall	Live	110	89.094	13.231	0.135	0.892	0.018
	Static	115	88.855	13.274			

Figure 4: Comparison of student responses to: "What are specific things about the [pre-written]/[live-demo of] code examples that are helpful for your learning?" A darker hue indicates more responses with that label.

Category	Label	Frequency in Static-Code Responses	Frequency in Live-Coding Responses
Code Comprehension	part-by-part breakdown	23.71%	14.89%
	reference of correct code	20.62%	9.57%
	general code understanding	16.49%	0.00%
Programming Process	thought process while coding	7.22%	9.57%
	debugging/avoiding errors	6.19%	13.83%
	code writing	4.12%	11.70%
	testing code	0%	7.45%
Features of Code Examples	instructor's explanation	8.25%	13.83%
	variations of code	2.06%	7.45%
	predicting output	2.06%	3.19%
	seeing output	5.15%	0.00%
Lecture Experience	following along with instructor	3.09%	8.51%
	taking notes	0.00%	2.13%
	group learning	1.03%	5.32%
Application	application of abstract concepts	13.40%	2.13%

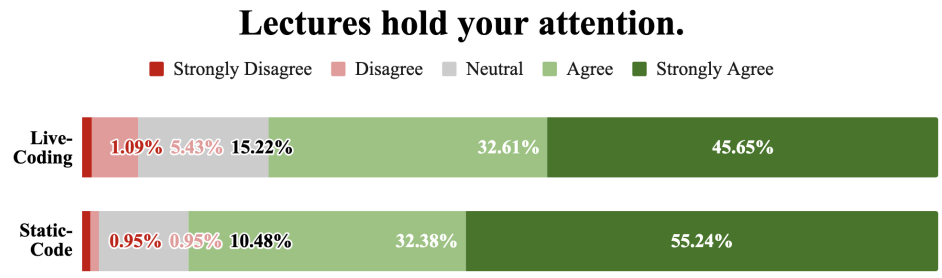
live-coding group mentioned that the instructor should *slow down* in the live-coding group, whereas only 2% of responses from the static-code group mentioned that the instructor should slow down. However, more students in the static-code group suggested the instructor include more documentation, show the process of coding, and include more variations of the code, which were reported at much lower frequencies in the live-coding group.

6.3.2 Lecture Engagement. Figure 6 shows a comparison of students' responses to the questions about lectures holding attention and facilitating note-taking. Note that a larger proportion of red or gray in the figures indicates that more students did not agree that lectures held attention or facilitated note-taking. A Chi-square test of trend for both evaluation items revealed an association between the type of lecture example and students' feeling that lectures held

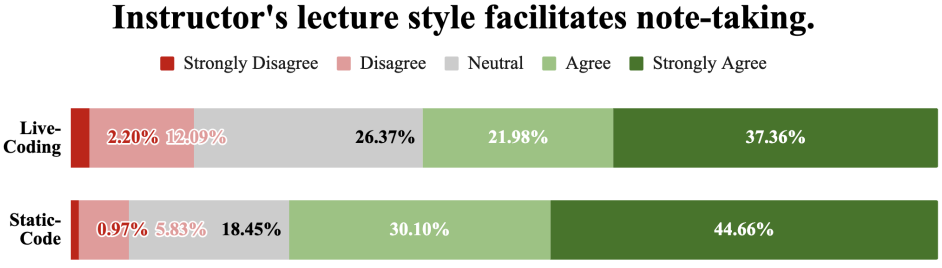
Figure 5: Comparison of student responses to: "What are some suggestions to improve the code examples so that they may be more beneficial for your learning?" A darker hue indicates more responses with that label.

Category	Label	Frequency in Static-Code Responses	Frequency in Live-Coding Responses
Pace	slow down	2.06%	18.09%
	speed up	1.03%	1.06%
Lecture Level Changes	more interactive	6.19%	7.45%
	more examples	4.12%	7.45%
	post examples online	2.06%	5.32%
	more organized	1.03%	1.06%
	use live coding	3.09%	0.00%
Presentation of examples	more explanation	8.25%	12.77%
	show process	5.15%	2.13%
	more documentation	5.15%	2.13%
	show other resources	2.06%	0.00%
	better annotations	10.31%	0.00%
	show output	5.15%	0.00%
Content of examples	more similar to PAs	4.12%	4.26%
	more code variations	9.28%	5.32%
	more difficult examples	6.19%	4.26%
	more bugs	5.15%	1.06%
	less bugs	0.00%	1.06%
No Suggestions	nothing	23.71%	34.04%

Figure 6: Comparison of student responses to survey items about lecture engagement



(a) Responses to question about lecture holding attention



(b) Responses to question about lecture facilitating note-taking

attention and facilitated note-taking. We obtained p-values < 0.001 when applying this test, indicating that students in the live-coding group were more likely to disagree that lectures held attention or facilitated note taking.

7 DISCUSSION

7.1 Key Takeaways

Nearly across the board, we saw a lack of statistically significant differences between the live-coding and static-code groups. Where there was significance, we found that students in the live-coding group:

- more commonly reported that the lectures did not hold their attention (see Section 6.3.2).
- more commonly reported that the lectures did not facilitate note-taking (see Section 6.3.2).

Compared to the static-code group in terms of programming processes, the students in the live-coding lecture:

- demonstrated similar adherence to incremental development according to the MID (see Section 6.1.1).
- experienced similar amounts of struggle with common errors (see Section 6.1.2).
- demonstrated similar use of print statements (see Section 6.1.2).
- had a slightly higher rate of correctness on the final and the debugging coding challenges, though not statistically significant (see Section 6.1.3).

Compared to the static-code group in terms of course performance, the students in the live-coding lecture:

- performed similarly across all types of coding questions (basic, tracing, explaining, writing) on worksheets and exams (see Section 6.2.1).
- earned overall similar grades on assignments, worksheets, and exams (see Section 6.2.2).

We ultimately conclude that live coding neither improved students' programming process skills based on our chosen metrics nor did it detract from students' performance on traditional assessments. Though our findings from RQ1 (programming processes) and RQ2 (course performance) point to a lack of a meaningful difference between the two groups, students' open-ended feedback suggests a difference in what skills students reportedly *observed* during lectures. Specifically, more students mentioned seeing the instructor's thought process, the code writing process, and debugging techniques in the live-coding group (see Section 6.3.1). However, these differences in perceived benefits did not materialize into discernible differences in programming processes or code comprehension according to the metrics we applied.

A potential reason for the lack of meaningful differences between the groups is that lectures were only one part of the learning experience in the course. Each week, students were responsible for attending lectures and labs and for completing required reading activities, programming assignments, and worksheets (summarized in Table 1). A student may pick up domain knowledge or heuristic strategies from any of these required learning activities, not to mention the optional activities such as discussions and office hours. Moreover, within the lectures themselves, code examples make up only half of the time, with other time dedicated to introducing new ideas, reviewing past content, or participation activities. Ultimately, the code examples presented in lecture make up only a fraction of students' learning experience in CS1, so the *impact* of the style of code examples may be limited. Indeed, our work confirms prior work that showed no significant differences in students' course performance between live-coding and static-code pedagogies [35, 40, 43, 45].

Our findings related to the drawbacks of live coding may shed light on why these perceived benefits, and the benefits cited in prior works [4, 5, 25, 36], did not materialize. Our main takeaway from asking students about suggestions for improvement is that far more students in the live-coding lecture—nearly one-fifth of respondents—felt the instructor was going too fast on the examples. Live-coding examples require the instructor to move away from the lecture slides and open a new file in an IDE for each example, which leads to an inherent overhead cost of showing the examples. Though the instructor sometimes started with boilerplate code and generally adhered to recommended live-coding practices [7], our results suggest that the time-consuming nature of live coding still persisted. Indeed, student responses on the two course evaluation items confirms that the live-coding group felt lectures were worse at holding attention and facilitating note-taking. These results may be due to the instructor simultaneously writing code while also explaining their reasoning and strategies. Students could be unsure whether to write down the code or the instructors' explanations, potentially resulting in an inability to focus at all. Moreover, the lack of annotations means that students are not able to copy down memory diagrams or tables to trace variable values during execution, potentially resulting in more live-coding students disagreeing with the statement that lectures facilitated note-taking. Ultimately, students may not have been able to fully absorb the programming processes demonstrated in the live-coding examples because of the pace and difficulty in focusing. However, we posit that improvements and additions to a live-coding pedagogy may offer a path to mitigate the drawbacks of the pedagogy. For example, instructors might consider supplementing their live-coding demos with scaffolded worksheets for students to fill out during class.

7.2 Broader Implications

A key motivation of our work was to determine whether the perception among students and instructors that live coding improves students' programming processes [4, 25, 42] is empirically true. While our analysis of student feedback confirmed that live-coding students reported seeing more "Programming Process" skills in lecture, we ultimately did not find evidence to confirm that students in the live-coding group empirically demonstrate better programming process skills. Through a *modeling* lens set forth in theories of Cognitive Apprenticeship [11] and Bandura's observational learning [2], students in the live-coding group more frequently *identified* the modeling of heuristic strategies for solving programming tasks (code-writing process, debugging, testing), but this did not result in those students more frequently engaging with, or *imitating*, those heuristic strategies according to the process-oriented metrics we applied. Therefore, a key implication of our work is that there may

be an additional step between students observing effective programming processes and being able to apply these processes in their own work. Fortunately, theory offers a potential path forward. Of the 6 teaching methods of Cognitive Apprenticeship [11], live coding only makes use of the modeling method. Therefore, it may be worth exploring live coding *in conjunction with* other Cognitive Apprenticeship methods to help students externalize the programming processes observed via live coding. Collins et al. also describe methods such as *scaffolding* and *coaching* to help students identify and apply the heuristic strategies they observed [11]. For example, a "collaborative" live-coding approach where the instructor asks students to complete sub-tasks of the overall code example may engage these two methods since students would complete small, manageable tasks and get immediate feedback from the instructor.

Through a theoretical lens, our experimental approach to compare static-code examples to live coding is fundamentally a comparison of top-down, program-comprehension-focused examples [6] to bottom-up, program-generation-focused examples [37]. Though static-code emphasizes aspects of Brooks' model of novices' program comprehension, students in the static-code group did not display significantly better performance on code-tracing or code-explaining questions on the exams or worksheets. Similarly, though live coding demonstrates the step-by-step process described by Rist [38], the live-coding group did not perform significantly better on code-writing tasks or exhibit better adherence to incremental development. We speculate that despite the fundamental difference in approach between live-coding and static-code pedagogies, both approaches impart comparable program comprehension and generation skills. Live coding does not *only* demonstrate program generation skills and static-code examples do not *only* demonstrate code comprehension. Furthermore, given the prevalence of static-code examples in other course material, such as the course textbook [47], students in the live-coding group will also pick up comprehension skills from these other sources.

7.3 Threats to Validity and Limitations

Our study includes two key threats to validity related to potentially confounding factors to our analysis. First, we did not randomize assignment to either condition since students self-selected into the lecture groups. Though we compared the two groups' demographics, prior experience, and high school performance, there may have been other differences between students in the two groups that could threaten our findings. Similarly, our analysis of engagement via course evaluations may suffer from selection bias, since students with more extreme perceptions of the class may be more likely to respond. Second, as discussed in Section 7.1, there naturally exist extraneous influences to a students' programming processes or course performance besides the lecture code examples. Many learning experiences may occur outside of lecture, potentially drowning out the effect of lecture examples. This may have had a pronounced impact on our study, especially during the two-week stall in the course. During the stall, students may have spent time reviewing the course content through a variety of means, such as reviewing lecture slides or the textbook, holding group review sessions, or using external resources. Though we collected the majority of the data before the stall, there may have been differences in the way students responded to this stall.

The scope of our study also has limitations to generalizability. Our study was conducted on an introductory, CS1 course in which half of our participants were first-year undergraduate students at a four-year university. Live coding may have different impacts on students in advanced courses in which programming processes are more greatly emphasized. Additionally, there are variations of "live coding" besides an instructor coding in front of a class during lecture. These variations include students writing code in front of the class during lecture or the instructor creating pre-recorded screen recordings of programming to show in lecture [42]. While we evaluated the most common form of live coding according to the literature [42], our analysis is limited to this form of live coding. Other live-coding approaches may have different impacts that we did not uncover.

Readers should also note that live coding may have varying impacts across different subpopulations of learners. For example, our programming process data was typically collected *during* lectures, which means the data typically represents students who attended lectures. On one hand, this is a strength of our design (since the treatment condition involves the lecture examples themselves), but it means we lose information on how live coding may impact more disengaged students (i.e., the ones who did not come to lectures). Similarly, we note that the vast majority of students had a personal computer which they typically brought to lecture. Access to a personal computer may aid in the consumption of live-coding examples, so further studies to evaluate live coding in lower-resource settings will be beneficial.

8 CONCLUSION

Despite live coding being a recommended teaching practice in computing, our findings indicate that live coding might not actually impart the multitude of *perceived* and *intended* benefits of the pedagogy identified in prior work. However, "live coding" has several variations, so future work to evaluate the various live-coding approaches may aid in our understanding of the pedagogy and its empirical impacts on student learning. In the meantime, our findings may motivate educators and researchers to consider methods to realize the perceived learning benefits of live coding while mitigating its drawbacks.

ACKNOWLEDGMENTS

This work was supported in part by NSF award 2044473.

REFERENCES

[1] M Aickin and H Gensler. 1996. Adjusting for multiple testing when reporting research results: The Bonferroni vs Holm methods. *American Journal of Public Health* 86, 5 (1996), 726–728. https://doi.org/10.2105/ajph.86.5.726

[2] Albert Bandura. 1977. *Social Learning theory.* Prentice-Hall, Englewood Cliffs, N.J.

[3] 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

[4] Jens Bennedsen and Michael E. Caspersen. 2005. Revealing the Programming Process. *SIGCSE Bull.* 37, 1 (feb 2005), 186–190. https://doi.org/10.1145/1047124.1047413

[5] 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

[6] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543–554. https://doi.org/10.1016/S0020-7373(83)80031-5

[7] 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

[8] 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

[9] 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

[10] Jacob Cohen. 1977. *Statistical Power Analysis for the behavioral sciences*. Academic Press.

[11] Allan Collins, John Seely Brown, Ann Holum, et al. 1991. Cognitive apprenticeship: Making thinking visible. *American educator* 15, 3 (1991), 6–11.

[12] Allan M. Collins, John Seely Brown, and Susan E. Newman. 1988. Cognitive Apprenticeship: Teaching the Crafts of Reading, Writing, and Mathematics. *Knowing, Learning, and Instruction* 8, 1 (1988), 2–10. https://doi.org/10.5840/thinking19888129

[13] Edward E. Cureton. 1956. Rank-biserial correlation. *Psychometrika* 21 (1956), 287–290.

[14] Leon R. de Bruin. 2019. The use of cognitive apprenticeship in the learning and teaching of improvisation: Teacher and student perspectives. *Research Studies in Music Education* 41, 3 (2019), 261–279. https://doi.org/10.1177/1321103X18773110

[15] Vincenzo Del Fatto, Gabriella Dodero, and Rosella Gennari. 2016. How Measuring Student Performances Allows for Measuring Blended Extreme Apprenticeship for Learning Bash Programming. *Comput. Hum. Behav.* 55, PB (feb 2016), 1231–1240.

[16] Sarah LS Duncan. 1996. Cognitive apprenticeship in classroom instruction: Implications for industrial and technical teacher education. *Journal of Industrial Teacher Education* 33, 3 (1996), 66–86.

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

[18] 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

[19] Chris M. Golde. 2008. Applying lessons from professional education to the preparation of the professoriate. *New Directions for Teaching and Learning* 2008, 113 (2008), 17–25. https://doi.org/10.1002/tl.305 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/tl.305

[20] 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

[21] Adrián Hernández-López, Ricardo Colomo-Palacios, and Ángel García-Crespo. 2012. Productivity in software engineering: A study of its meanings for practitioners: Understanding the concept under their standpoint. In *7th Iberian Conference on Information Systems and Technologies (CISTI 2012)*. 1–6.

[22] 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

[23] Scott D Johnson and Rita McDonough Fischbach. 1992. Teaching Problem Solving and Technical Mathematics through Cognitive Apprenticeship at the Community College Level. https://eric.ed.gov/?id=ED352455

[24] Despina Koletsi and Nikolaos Pandis. 2016. The chi-square test for trend. *American Journal of Orthodontics and Dentofacial Orthopedics* 150, 6 (2016), 1066–1067. https://doi.org/10.1016/j.ajodo.2016.10.001

[25] 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

[26] H. Chad Lane and Kurt VanLehn. 2005. Intention-Based Scoring: An Approach to Measuring Success at Solving the Composition Problem. 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, 373–377. https://doi.org/10.1145/1047344.1047471

[27] 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

[28] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60. https://doi.org/10.1214/aoms/1177730491

[29] Rifat Sabbir Mansur, Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. 2020. Exploring the Bug Investigation Techniques of Intermediate Student Programmers. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '20)*. Association for Computing Machinery, New York, NY, USA, Article 2, 10 pages. https://doi.org/10.1145/3428029.3428040

[30] Mary McHugh. 2012. Interrater reliability: The kappa statistic. *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB* 22 (10 2012), 276–82. https://doi.org/10.11613/BM.2012.031

[31] 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

[32] Clifford Nowell and Richard M. Alston. 2007. I Thought I Got an A! Overconfidence Across the Economics Curriculum. *The Journal of Economic Education* 38, 2 (2007), 131–142. https://doi.org/10.3200/JECE.38.2.131-142 arXiv:https://doi.org/10.3200/JECE.38.2.131-142

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

[34] Harry O. Posten. 1984. Robustness of the Two-Sample T-Test. In *Robustness of Statistical Methods and Nonparametric Statistics*, Dieter Rasch and Moti Lal Tiku (Eds.). Springer Netherlands, Dordrecht, 92–99. https://doi.org/10.1007/978-94-009-6528-7_23

[35] 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

[36] 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

[37] Robert S. Rist. 1990. Variability in program design: the interaction of process with knowledge. *International Journal of Man-Machine Studies* 33, 3 (1990), 305–322. https://doi.org/10.1016/S0020-7373(05)80121-X

[38] Robert S. Rist. 1995. Program structure and design. *Cognitive Science* 19, 4 (1995), 507–561. https://doi.org/10.1016/0364-0213(95)90009-8

[39] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education* 13, 2 (2003), 137–172. https://doi.org/10.1076/csed.13.2.137.14200 arXiv:https://doi.org/10.1076/csed.13.2.137.14200

[40] 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

[41] Randall E. Schumacker. 2023. *Learning Statistics Using R*. SAGE Publications, Inc., 55 City Road, London, Chapter 12. https://doi.org/10.4135/9781506300160

[42] 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

[43] 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* (Turku, Finland) *(ITiCSE '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3587102.3588846

[44] 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 '23)*. Association for Computing Machinery, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569880

[45] Amy Shannon and Valerie Summet. 2015. Live Coding in Introductory Computer Science Courses. *J. Comput. Sci. Coll.* 31, 2 (dec 2015), 158–164.
[46] E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (sep 1986), 850–858. https://doi.org/10.1145/6592.6594
[47] Stepik. 2023. *Stepik*. https://stepik.org/course/125129/
[48] Razieh Tadayon Nabavi. 2012. Bandura's Social Learning Theory & Social Cognitive Learning Theory by Razieh Tadayon Nabavi and Mohammad Sadegh Bijandi. https://www.researchgate.net/publication/267750204_Bandura's_Social_Learning_Theory_Social_Cognitive_Learning_Theory
[49] 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
[50] Anne Venables, Grace Tan, and Raymond Lister. 2009. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (Berkeley, CA, USA) *(ICER '09)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1584322.1584336
[51] Arto Vihavainen, Matti Paksula, and Matti Luukkainen. 2011. Extreme Apprenticeship Method in Teaching Programming for Beginners. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. Association for Computing Machinery, New York, NY, USA, 93–98. https://doi.org/10.1145/1953163.1953196
[52] 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
[53] 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
[54] Donald W. Zimmerman. 1987. Comparative Power of Student T Test and Mann-Whitney U Test for Unequal Sample Sizes and Variances. *The Journal of Experimental Education* 55, 3 (1987), 171–174. http://www.jstor.org/stable/20151691

Table 12: Final code book for perceived benefits

Label	Description
part-by-part breakdown	1. Explaining code line by line 2. Student mentions seeing individual parts of the code 3. Labeling or color coding separate components of a program 4. Breaking down a program 5. Making the program more simple to understand
reference of correct code	1. Student mentions using it to guide other similar activities 2. See what code is "supposed to look like" 3. Thinking about how this code could be modified to do something similar 4. Examples of the correct code for a concept
general code understanding	1. Understanding ""how the code works"" in general 2. Useful to review for understanding
thought process while coding	1. Learning the problem solving process 2. Understanding why the professor writes certain lines of code
debug-ging/avoiding errors	1. Identifying and understanding common errors 2. Process of fixing errors 3. Seeing errors/unexpected output 4. Showing where code can go wrong
code writing	1. Modeling the process of writing code 2. How to approach writing code "from scratch" 3. Seeing the step-by-step process
testing code	1. Learning how to test the correctness of code 2. Understanding why a test passed or failed 3. Seeing examples of test cases
instructor's explanation	1. Live commentary on code 2. Explanation of code 3. Thorough answers to questions from students
variations of code	1. Showing different variations/changes in a code example 2. Showing ""trial and error"" process
predicting output	1. Student mentions enjoying trying to guess the output 2. The instructor asking students to guess what the code will output
seeing output	1. Seeing output of code examples along with them
following along with instructor	1. The ability to follow along with code as it is written live
taking notes	1. Taking notes to reinforce understanding
group learning	1. Suggestions from classmates who have better understanding 2. Students in class giving suggestions for next coding steps
application of concepts	1. Seeing concepts immediately applied during lecture

Table 13: Final code book for suggestions for improvements (drawbacks)

Label	Description
slow down	1. Take more time during the code examples 2. Examples go by too fast to grasp a concept 3. Go into more detail on specific concepts
speed up	1. Make explanations and examples faster
more interactive	1. Add more participation activities, group exercises, or ask more questions
more examples	1. Add more examples in the lecture
make examples easier to see later	1. Make it easier reference everything done in the live demo later
more organized	1. The organization/flow of the lecture is hard to follow
use live coding	1. Show more live-coding examples examples during lecture 2. Show a mix of live coding with the static-code examples
more explanation	1. Instructor should include more or better explanation (ex. of how each line of code works)
show process	1. Write down a step-by-step process during the example
more documentation	1. Write more comments in code examples
show references to textbook chapters	1. Include references to additional resources with code examples (ex. textbook chapter) 2. Review background knowledge required to understand examples
better annotations	1. Show more handwritten annotation 2. Write clearer annotations (handwriting, organization, etc.)
show output	1. Add more print statements to show more output 2. Show output alongside code examples 3. Run code in Edstem
more similar to PAs	1. Examples should be more similar to the content on PAs/assignments 2. Show how the professor would approach PAs 3. Similar to code examples on exams
more variations	1. More variations of code examples for each topic
more difficult examples	1. Lecture examples should be more complicated/difficult 2. Testing more complex inputs/edge cases
more bugs	1. Lecture examples should show more errors to see how they are fixed and how to avoid them
less bugs	1. Show less errors during the code examples
nothing	1. No suggestion provided 2. Suggestion is unrelated to code examples